

Audit of Smooth CryptoLib (SCL) for ECDSA Verification

Date	July 11 th , 2024
Version	1.1
Page count	59
Authors	Ryad Benadjila Matthieu Rivain

Contents

1	Introduction	3
1.1	Context	3
1.2	Background and notations	3
1.2.1	Elliptic curves	3
1.2.2	The ECDSA signature algorithm	5
1.3	Scope of the audit	6
1.4	Methodology and summary of findings	8
2	Description and high-level analysis of the SCL library	10
2.1	ModExp precompile contract and ModInv function	10
2.2	Double scalar multiplication (DSM) algorithm	10
2.2.1	Affine and XYZZ coordinates	10
2.2.2	XXZZ point doubling formulas to implement DBLECC	13
2.2.3	XXZZ points addition formulas to implement ADDECC	15
2.2.4	DSM with Shamir's trick and windowing	19
2.2.5	Implementation of $DSM_{i,j}$ in the SCL library	22
2.3	ECDSA verification and weak public keys	26
2.4	Gas cost	30
3	Detailed audit of SCL files	35
3.1	The src/fields/ and src/include/ folders	35
3.1.1	SCL_mask.h.sol	35
3.1.2	SCL_field.h.sol	37
3.1.3	SCL_secp256r1.sol	37
3.1.4	SCL_wei25519.sol	38
3.2	The src/modular/SCL_modular.sol file	38
3.3	The src/elliptic/ folder	39
3.3.1	The SCL_ec0ncurve.sol file	39
3.3.2	The SCL_mulumuladdX_{fullgen_b4/fullgenW}.sol files	41
3.4	The src/lib/ folder	48
3.5	The tests from the test/ folder	50
4	CRYPTOEXPERTS tests on SCL	52

1 Introduction

1.1 Context

Smoo.th offers a unique UX, enabling to onboard users to Web3 technologies, with a familiar experience from classic web by leveraging the FIDO/passkeys framework. At the core of the product, the SmoothCryptoLib (SCL) library implements cryptographic primitives and protocols in Solidity, targeting EVMs.

The SCL ECC implementation is build upon [17]. Used together with Account Abstraction, this library allows to leverage the WebAuthn/Passkeys technology as a way to authenticate transactions on all EVMs. The SCL library is also related to an Ethereum RIP-7696¹ proposal [19] for precompiled contracts implementing signature verification through DSM (Double Scalar Multiplication), allowing to speedup this signature verification over a various range of curves.

The EVM ecosystem already offers opcodes and precompiled contracts that implement operations over elliptic curves, including points addition and scalar multiplication [16, 23]. However, these are specific to dedicated curves `secp256k1` (the bitcoin curve) and `alt_bn128` (pairing friendly curve for Zero Knowledge based applications). Another historical precompiled EVM contract RIP-7212 [25] implements ECDSA signature verification, but specifically over `secp256r1`, also known as the P256 curve. The main advantages of RIP-7696 when compared to all these historical EVM opcodes and precompiles are genericity in terms of underlying fields and curves (for primes that are encoded on a maximum of 256 bits fitting Solidity's `uint256` type), while keeping decent gas costs through optimal DSM computation formulas.

The present report contains the results of the audit of the SCL library conducted between end of May and early July 2024. After the introduction of general notations and algorithms in Section 1.2, we provide the scope of the audit in Section 1.3 and a summary of the audit methodology and findings in Section 1.4. Then, we present an overview of the code architecture and algorithms involved in the SCL library in Section 2. The code review file by file is developed in Section 3. Finally, we provide some information about our testing framework in Section 4.

1.2 Background and notations

The notions and notations defined in this section will be used throughout the whole report, unless stated otherwise.

1.2.1 Elliptic curves

We summarize in this section the notations that will be used across the report. Let $p \in \mathbb{N}$ be a prime and \mathbb{F}_p the associated prime field where $+$ is the additive law and \times the multiplicative law. For all elements in the field $\forall x \in \mathbb{F}_p$, we will indifferently denote x^{-1} or

¹RIP stands for Rollup Improvement Proposals.

$\frac{1}{x}$ the multiplicative inverse of x in \mathbb{F}_p , i.e. the element such that $x \times x^{-1} = 1 \bmod p$ (where \bmod is the modulo operator). Let $a, b \in \mathbb{F}_p$, we will denote $\mathcal{E}(a, b, p)$ the elliptic curve over the prime field \mathbb{F}_p , encompassing all the points $P = (x, y)$ whose affine coordinates (x, y) satisfy the short Weierstraß equation:

$$y^2 = (x^3 + a \times x + b) \bmod p \quad (1)$$

For a given point $P = (x, y)$, we will denote $P_x = x$ and $P_y = y$ in the sequel. $\mathcal{E}(a, b, p)$ is a finite group where we will use an additive notation for the underlying operation. Two elements $P, Q \in \mathcal{E}(a, b, p)$ are points over the curve, and their addition provides a third point $R = P + Q \in \mathcal{E}(a, b, p)$. The number of elements in the group (i.e. number of points on the elliptic curve) is denoted n and is called the order of the curve. The neutral element of the group is called the point at infinity, and will be denoted \mathcal{O} : it is the only point that does not have affine coordinates (x, y) satisfying Equation 1.² For any point $P \in \mathcal{E}(a, b, p)$, we have $P + \mathcal{O} = P$. The opposite (or negative) of a point P is $-P$ such that $P + (-P) = \mathcal{O}$. The addition law for computing the coordinates of $R = P + Q = (R_x, R_y)$ when $P \neq -Q, P \neq \mathcal{O}, Q \neq \mathcal{O}$ is the following (it can be checked to be a group law):

$$\begin{cases} R_x = \lambda^2 - P_x - Q_x \\ R_y = \lambda \times (P_x - R_x) - P_y \end{cases} \quad \text{with } \lambda = \begin{cases} (P_x - Q_x) \times (P_y - Q_y)^{-1} & \text{if } P \neq \pm Q \\ (3 \times P_x^2 + a) \times (2 \times P_y)^{-1} & \text{if } P = Q \end{cases} \quad (2)$$

Under this law, with $P = (P_x, P_y)$ we have the negation defined as

$$-P = (P_x, -P_y) .$$

Let $u \in \mathbb{Z}$ a scalar. We define the scalar multiplication operation, denoted \cdot , by:

$$u \cdot P = \begin{cases} \mathcal{O} & \text{if } u = 0 \\ \underbrace{P + P + \dots + P}_{u \text{ times}} & \text{if } u > 0 \\ -(\underbrace{P + P + \dots + P}_{|u| \text{ times}}) = -(|u| \cdot P) & \text{if } u < 0 \end{cases} \quad (3)$$

Because of the group nature of \mathcal{E} , $\forall P \in \mathcal{E}(a, b, p)$ we have $n \cdot P = \mathcal{O}$ by Lagrange theorem, and hence for any scalar $u \in \mathbb{Z}$ it follows that $u \cdot P = (u \bmod n) \cdot P$.

In the context of the current audit, we will only consider so-called prime curves, i.e. curves where the order n is prime. For such curves, all the points except \mathcal{O} are of order n and generate the curve. More specifically, we will only focus in this report on the P256 NIST curve [9, 10] where:

²This neutral element can be represented in other coordinates systems as we will detail in the sequel of the report.

All the points except \mathcal{O} generate the curve, but a specific point $G = (G_x, G_y)$ is used as the generator for the ECDSA signature algorithm that we will consider:

$$\begin{cases} G_x = 0x6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296 \\ G_y = 0x4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5 \end{cases} \quad (5)$$

1.2.2 The ECDSA signature algorithm

ECDSA (Elliptic Curve Digital Signature Algorithm) [14] is a standardized signature algorithm. It is a classic ECC primitive securing ETH transactions. In the following, we describe the “raw” version of the algorithm where the message input to be signed is a bit string of length $\lceil \log_2(n) \rceil$ bits given an elliptic curve $\mathcal{E}(a, b, p)$ and generator G of order n . For a prime curve, n is also the order of the curve. Here are the steps:

Key generation. A private key d is chosen randomly in $[0, n - 1]$, the corresponding public key $Q = d \cdot G$ is computed.

Signature generation. Given an input message m of size $\lceil \log_2(n) \rceil$ bits and a private key $d \in [0, n-1]$, generate the signature (r, s) with the following steps:

1. Generate a random secret nonce $k \in [1, n - 1]$ and compute $k \cdot G = (x, y)$.
2. Compute $r = x \bmod n$, if $r = 0$ go back to step 1.
3. Compute $s = k^{-1} \times (m + r \times d) \bmod n$. If $s = 0$ go back to step 1.
4. Return (r, s) as the signature – this can be represented as a bit string of size $2 \times \lceil \log_2(n) \rceil$.

Signature verification. Given an input message m of size $\lceil \log_2(n) \rceil$ bits, a signature (r, s) and a public key $Q = d \cdot G$, verify the signature with the following steps:

1. Check that $Q \neq \mathcal{O}$ and that $Q \in \mathcal{E}(a, b, p)$ (i.e. its coordinates satisfy Equation 1). For a non-prime curve, check that $n \cdot Q = \mathcal{O}$ (i.e. the point is in the generator prime subgroup).

2. Check that $r, s \in [1, n - 1]$, reject the signature otherwise.
3. Compute $u = m \times s^{-1} \bmod n$ and $v = r \times s^{-1} \bmod n$.
4. Compute the point $P = u \cdot G + v \cdot Q$. If $P = \mathcal{O}$ reject the signature.
5. Accept the signature if $r = P_x \bmod n$, reject otherwise.

1.3 Scope of the audit


While the SCL library is compatible with any short Weierstraß curve (and by extension with Edwards curves such as Ed25519 using isogenies [24]), and might be used to build complex ECC based cryptographic protocols, the scope of the audit is restricted to the ECDSA signature verification function over the P256 elliptic curve.

As presented in Section 1.2.2, the main costly operation of the signature verification in terms of computation is the Double Scalar Multiplication (DSM) computing $u \cdot P + v \cdot Q$, where u, v are scalars and P, Q are points over the elliptic curve (P256 in our context). The building block of this scalar multiplication is adding two points $P + Q$. As presented on Equation 2, this operation distinguishes between the cases where $P \neq \pm Q$, $P = Q$ and $P = -Q$. In the first case we perform point addition denoted $\text{ADDECC}(P, Q)$ computing $P + Q$, and in the second case this is a point doubling denoted $\text{DBLECC}(P)$ computing $2 \cdot P$.³ In the last case where $P = -Q$, $P + Q = \mathcal{O}$. The library implements two variants of $u \cdot P + v \cdot Q$:

- `libSCL_rip7212.sol`: this variant implements the double scalar multiplication using the so called Shamir's trick with 2 input points and a 2-bit window, thus reducing the number of operations to $2 \times \text{DBLECC}$ and $11 \times \text{ADDECC}$ for precomputation, plus $256 \times \text{DBLECC}$ and an average of $120 \times \text{ADDECC}$ (see the details in Section 2.2.4).
- `libSCL_ecdsab4.sol`: this variant implements the double scalar multiplication using the Shamir's trick with 4 input points and a 1-bit window, thus reducing the number of operations to $11 \times \text{ADDECC}$ for precomputation, plus $128 \times \text{DBLECC}$ and an average of $120 \times \text{ADDECC}$ (see the details in Section 2.2.4).

The code to be reviewed is composed of 10 Solidity source files (581 lines of code without comments) available on the dedicated `audit-cryptoexperts` branch

<https://github.com/get-smooth/crypto-lib/tree/audit-cryptoexperts>

The version of the code to be reviewed corresponds to the commit `810f644`, from May 15, 2024, available [here](#) . More specifically, the audited code is composed of the following source files in the `src/` folder:

³As presented in Equation 2, addition and doubling usually involve different computation formulas explaining this distinction. However, it is to be noted that unified formulas exist depending on the used coordinates system: we will not provide further details as this is not the case in the context of the SCL library.

- Files related to finite fields:
 - `src/fields/SCL_secp256r1.sol`: this file embeds the P256 NIST curve constants.
 - `src/fields/SCL_wei25519.sol`: this file embeds the Wei25519 (isogenic to Ed25519) constants. It has not been reviewed in depth as being out of the audit scope.
- Files related to modular operations:
 - `src/modular/SCL_modular.sol`: this file implements modular exponentiation and modular inverse (modulo primes) with optimizations.
- Files related to elliptic curve operations:
 - `src/elliptic/SCL_ecOncurve.sol`: this implements a routine to test that a point is on the expected curve.
 - `src/elliptic/SCL_mulumuladdX_fullgen_b4.sol`: this file implements $u \cdot P + v \cdot Q$ with Shamir's trick with 4 points by splitting u and v in two 128-bits low and high parts.
 - `src/elliptic/SCL_mulumuladdX_fullgenW.sol`: this file implements $u \cdot P + v \cdot Q$ with windowed Shamir's trick with two points and a 2-bit window size (4 bits in total, 2 for u and 2 for v).
- Files related to ECDSA signature verification (main exported entry points of the audited code):
 - `src/lib/libSCL_rip7212.sol`: this file implements the main entry point of the library for signature verification using the same API as RIP-7212 [25].
 - `src/lib/libSCL_ecdsab4.sol`: this file implements the main entry point of the library for signature verification using an extended API (proposed and explained in RIP-7696 [19]). The API uses additional data extending the RIP-7212 historical API, namely the curves parameters p , a , G and n , as well as the precomputed public key multiple $2^{128} \cdot Q$ and the generator multiple $2^{128} \cdot G$.
- Abstractions and constants defined in:
 - `src/include/SCL_field.h.sol`: this is a field abstraction wrapper that includes the necessary constants from the `src/fields/` folder (in our case with P256 from `src/fields/SCL_secp256r1.sol`, but this is adapted depending on the chosen curve).
 - `src/include/SCL_mask.h.sol`: this is a file containing useful constants, such as curves OIDs, masks, etc.






Beyond the core source files present in the `src/` folder, the repository is composed of:

- Tests of the ECDSA verification APIs (RIP-7212 and RIP-7696 compatible). These tests are written in Solidity and use the `foundry forge` framework [3], and are formed of basic test vectors as well as the more complete Wycheproof test suite [8] (these tests are formatted in the `test/vectors_wycheproof.jsonl` file).
- Deploy contracts in the `script/` folder, with the companion `deploy.sh` shell script to deploy on chains.
- The `foundry.toml` configuration file, containing various flags for the `foundry` toolchain (optimization options, etc.).


The SCL library has one main external dependency: the `ModExp` precompiled EVM contract at address `0x05`, specified in EIP-198, that performs modular exponentiation, used in this context for modular inversion by exploiting Fermat's little theorem in \mathbb{Z}_n . This external contract is called using a `staticcall` opcode as publicly documented [2, 15]. The code of this precompiled contract is out-of-scope of the current audit.

1.4 Methodology and summary of findings

Our observations are categorized as follows:

- Observations that may impact the security or the soundness of SCL library ECC related algorithms, rated as
 - high risk (flagged )
 - medium risk (flagged )
 - low risk (flagged )
- Observations related to coding practices and implementation choices (flagged )
 - These observations do not translate into a direct risk on the security or soundness of SCL, but addressing them would make the code clearer, more efficient and/or less prone to errors.
- Observations related to documentation, comments, variable naming (flagged )
 - These observations do not translate into a direct risk on the security or soundness of SCL, but addressing them would facilitate the understanding of the code by third parties (users, developers, auditors).

Each observation comes with an associated recommendation to fix or improve the underlying issue.

Beyond observations, outstanding remarks (flagged ) are also present in the document: these are used to highlight interesting contextual facts.

Our findings are summarized in the table below:

Category	Number of findings
● High risk	0
● Medium risk	1
● Low risk	2
● Coding practices	8
■ Documentation	12
Total	23
🔍 Remarks	6

The following list of observation is exhaustive:

■ Observation 1: Ambiguous affine representation of point at infinity \mathcal{O} for some curves	12
■ Observation 2: Missing documentation about affine to/from XYZZ transformations for point at infinity \mathcal{O}	12
🔍 Remark 1: Improved computation-memory trade-off for DSM precomputation	22
🔍 Remark 2: Consequences of the x -only output of the DSM algorithm	26
■ Observation 3: Consequences of the x -only output of the DSM algorithm	26
■ Observation 4: Raw ECDSA signature forgeries	28
■ Observation 5: Erroneous blacklist of weak keys	29
🔍 Remark 3: Check of blacklisted weak keys	29
■ Observation 6: ECDSA malleability	30
🔍 Remark 4: ECDSA and composite order curves	30
🔍 Remark 5: Costs for ADDECC and DBLECC DSM	34
● Observation 7: Unused and duplicated constants in SCL_mask.h.sol	35
● Observation 8: Incorrect OIDs for P256 and Ed25519	36
■ Observation 9: Misleading name for SCL_field.h.sol	37
● Observation 10: Unused constants in SCL_secp256r1.sol	38
■ Observation 11: Incorrect comment for ec_isOnCurve	40
● Observation 12: Incorrect checks in ec_isOnCurve	40
● Observation 13: Possible error injection in the DSM algorithms	42
● Observation 14: Suboptimal doubling in DSM precomputation	44
🔍 Remark 6: Potential suboptimal addition in DSM precomputation	45
● Observation 15: Collisions of constants in DSM files	45
● Observation 16: Incomplete / bad code for SCL_mulumuladdX_fullgenW.sol	45
● Observation 17: DSM function names	46
■ Observation 18: Incorrect inline comments in ecGenMulmuladdB4W	46
■ Observation 19: Incorrect or incomplete comments in the DSM files	47
■ Observation 20: Typo in comment and missing comment ECDSA verification files	49
■ Observation 21: Missing rationale about requirements for the inputs	49
● Observation 22: Compilation issues: README.md, case sensitivity for libSCL_RIP7212.sol filename	50
● Observation 23: Duplicate test files	51

2 Description and high-level analysis of the SCL library

2.1 ModExp precompile contract and ModInv function

The SCL library makes use of the `ModExp` precompiled contract from EIP-198 [15] at address `0x5`. The library calls this contract using a `staticcall` in the usual way provided in [2]. We can see in Algorithm 1 a high level description of the implemented logics. The computational cost (and hence gas cost) of this function depends on the size of the inputs: in the context of the SCL library, this cost will be fixed as `ModExp` is only used on fixed 256-bit Solidity `uint256` integers.

Algorithm 1 Modular exponentiation as implemented in the precompiled contract specified in EIP-198 [15]: from a , b and $p \in \mathbb{N}$ compute $a^b \bmod p$

```

1: procedure MODEXP( $a, b, p$ )
2:   return  $a^b \bmod p$      $\triangleright$  Usually implemented by square and multiply on big integers

```

This `ModExp` contract is mostly used for modular inversion using Fermat's little theorem: given $p \in \mathbb{N}$ prime, we have $\forall a \in \mathbb{N}$ the fact that $\text{ModExp}(a, p-2, p) = a^{p-2} \bmod p = a^{-1} \bmod p$, such that $a^{-1} \times a = 1 \bmod p$. Algorithm 2 provides this description. Since the exponentiation of 0 is 0, by default the inverse of 0 will produce 0 with this implementation of `ModInv`.

Algorithm 2 Modular inversion: from a and $p \in \mathbb{N}$ prime compute $a^{-1} \bmod p$ such that $a^{-1} \times a \bmod p = 1$. a can be 0 in which case the function returns 0.

Require: p is prime to apply Fermat's little theorem

```

1: procedure MODINV( $a, p$ )
2:   return ModExp( $a, p-2, p$ )     $\triangleright$  By convention, inverse of 0 is 0

```

2.2 Double scalar multiplication (DSM) algorithm

In this section, we provide background about how DSM is implemented in SCL, and the rationale behind the implementation choices. First, we introduce the XYZZ coordinates system in Section 2.2.1 used in the library to implement EC doubling `DBLECC` and addition `ADDECC` described in Section 2.2.2 and Section 2.2.3. Then, we provide the DSM implementation strategy using Shamir's trick and windowing in Section 2.2.4.

2.2.1 Affine and XYZZ coordinates

Points on Elliptic Curves are represented using affine coordinates (x, y) that satisfy the Weierstraß Equation 1 as summarized in Section 1.2. However, other extended coordinates systems exist that add at least one dimension. Such systems usually allow to naturally represent the point at infinity \mathcal{O} , optimize addition and doubling formulas, get compact

hardware implementations, bring some side-channel protections, etc. Popular examples used in wide-spread cryptographic libraries are projective, Jacobian or Co-Z coordinates.

The SCL library uses a less wide-spread Elliptic Curve system of coordinates named XYZZ for Weierstraß curves, introduced by Sutherland in 2008 and summarized in [13]. The main advantage of these formulas in the context of the EVM machine is the fact that they use the best compromise of additions and multiplications in \mathbb{F}_p when considering an equal cost for these operations. Indeed, the `addmod` and `mulmod` opcodes have the same gas cost of 8 units. Also, ECDSA verification does not require advanced side-channel resistant formulas. Further rationale and comparison with other formulas are provided in [17], thus we will not expand in more details and mainly focus on the formulas themselves as they will be useful for the current document.

The XYZZ system represents points on the short Weierstraß Elliptic Curve using 4 coordinates (X, Y, ZZ, ZZZ) such that $ZZZ^2 = ZZ^3$ (hence the notation). When injecting the new variables $x = \frac{X}{ZZ} = X \times ZZ^{-1} \bmod p$ and $y = \frac{Y}{ZZZ} = Y \times ZZZ^{-1} \bmod p$ in the short Weierstraß equation, we get in \mathbb{F}_p :

$$Y^2 = X^3 + a \times X \times ZZ^2 + b \times ZZZ^2 \quad (6)$$

When $ZZ = ZZZ = 1$, the affine and XYZZ coordinates coincide as the original equation is recovered. All the affine points can be represented with $(x, y, 1, 1)$, and the point $(0, 0, 0, 0)$ satisfies Equation 6 while not being represented as an affine point. By extension (this will be clearer when analyzing the formulas), any XYZZ point with $ZZ = ZZZ = 0$ is considered to be the point at infinity. From these definitions and notations, we describe the two implemented procedures `AffineToXYZZ` (from affine point (x, y) to its XYZZ representation $(x, y, 1, 1)$) in Algorithm 3, and the reverse procedure `XYZZToAffine` (from XYZZ (X, Y, ZZ, ZZZ) to $(x = \frac{X}{ZZ}, y = \frac{Y}{ZZZ})$ affine) in Algorithm 4. The `XYZZToAffine` algorithm actually only computes the x coordinate of the (x, y) affine point for reasons that will be explained in Section 2.2.5. It is also to be noted that `AffineToXYZZ` and `XYZZToAffine` are not dedicated functions in the SCL code, but are rather inlined in the source of the DSM procedures (transferring the input points from affine to XYZZ, and transferring back the output point from XYZZ to affine). Examples of inlined `AffineToXYZZ` and `XYZZToAffine` in Solidity assembly are presented on Listing 1.

```

1  ...
2  // [Audit] Example of inlined AffineToXYZZ handling input points of DSM
3  mstore4(mload(0x40), 128, mload(add(Q, _gx)), mload(add(Q, _gy)), 1, 1)
   //G the base point [1]
4  mstore4(mload(0x40), 256, mload(add(Q, _gpow2p128_x)),
   mload(add(Q, _gpow2p128_y)), 1, 1) //G'=2^128.G [2]
5  ...
6  // [Audit] Example of inlined XYZZToAffine handling the output point
7  // of DSM (x-only coordinate)
8  // [Audit] The precompiled call computes ZZ^-1
9  // Call the precompiled contract 0x05 = ModExp
10 if iszero(staticcall(not(0), 0x05, T, 0xc0, T, 0x20)) { revert(0, 0) }
11 X := mulmod(X, mload(T), _p) //X/zz

```

12

13

...

Listing 1: Inline code for `AffineToXYZZ` and `XYZZToAffine` (extracted from the `ecGenMulmuladdX_store` function in `SCL_mulmuladdX_fullgen_b4.sol`)

Equality of two XYZZ points (X_1, Y_1, ZZ_1, ZZZ_1) and (X_2, Y_2, ZZ_2, ZZZ_2) can be tested with the two equalities $X_1 \times ZZ_2 = X_2 \times ZZ_1$ and $Y_1 \times ZZZ_2 = Y_2 \times ZZZ_1$. As for affine coordinates, in XYZZ point negation is simply obtained by negating the Y coordinate: $-(X, Y, ZZ, ZZZ) = (X, -Y, ZZ, ZZZ)$. It follows that to test if two XYZZ points (X_1, Y_1, ZZ_1, ZZZ_1) and (X_2, Y_2, ZZ_2, ZZZ_2) are equal or opposite, it is enough to check if $X_1 \times ZZ_2 = X_2 \times ZZ_1$ (i.e. they share the same x normalized coordinates).

■ Observation 1: Ambiguous affine representation of point at infinity \mathcal{O} for some curves

In the SCL library, the point at infinity is by convention represented as $(0,0)$ in affine coordinates: the rationale is that these coordinates do not satisfy the curve equation and cannot be confused with other points on the curve. While this is generally true, this becomes false whenever $b = 0$ in the short Weierstraß Equation 1, hence bringing confusion for $(0,0)$.

This is rated as ■ since we only consider the P256 curve in the scope of the audit, where $b \neq 0$ and $(0,0)$ does not satisfy the equation, but this should be documented for a more general usage of the SCL library.

Recommendation:

Document (in the code and in the SCL documentation) the $(0,0)$ point at infinity representation, and the issues raised for short Weierstraß with $b = 0$.

■ Observation 2: Missing documentation about affine to/from XYZZ transformations for point at infinity \mathcal{O}

In the SCL library, the point at infinity is by convention represented as $(0,0)$ in affine coordinates: the rationale is that these coordinates do not satisfy the curve equation and cannot be confused with other points on the curve. However, this brings ambiguity when applying the transformations:

- In Algorithm 3: the specific case of point at infinity $(0,0)$ cannot be translated to XYZZ with $ZZ = ZZZ = 0$ with the current implementation $((0,0,1,1)$ is not the point at infinity). This is not a problem per se as safeguards prevent this situation (see Section 2.3); the ECDSA algorithm only allows (x,y) coordinates that satisfy the curve equation (with a dedicated explicit check $\neq (0,0)$) as

inputs of the DSM procedures.

- In Algorithm 4, the point at infinity in affine representation will be produced whenever $ZZ = 0$ thanks to the fact that $ZZ^{-1} = 0$. In the general case, this is not sufficient as $y = 0$ is also needed (which can be produced with $ZZZ^{-1} = 0$). However, since the ECDSA algorithm uses a x -only coordinate logic (see the discussion in Section 2.2.5 about this), this has no consequences.

Recommendation:

The representation and handling of the point at infinity $((0,0)$ in affine coordinates and $ZZ = ZZZ = 0$ in XYZZ coordinates) should be properly documented. Also, document why the `AffineToXYZZ` inline implementation does not handle this point at infinity and why `XYZZToAffine` x -only output is not an issue.

Algorithm 3 Affine point of coordinates $(x, y) \in \mathcal{E}(a, b, p)$ to XYZZ coordinates as implemented in SCL

Require: Point is supposed to be on the curve and not point at infinity \mathcal{O}

```
1: procedure AFFINETOXYZZ( $x, y$ )
2:   return  $(x, y, 1, 1)$ 
```

Algorithm 4 XYZZ point of coordinates $(X, Y, ZZ, ZZZ) \in \mathcal{E}(a, b, p)$ to affine representation $P = (x, y)$ (point \mathcal{O} , when $ZZ = 0$ accepted)

Require: Point is supposed to be on the curve

```
1: procedure XYZZTOAFFINE( $X, Y, ZZ, ZZZ$ )
2:    $ZZ_{\text{inv}} = \text{MODINV}(ZZ, p)$   $\triangleright$  Compute  $ZZ^{-1} \bmod p$ 
3:    $\triangleleft$   $\triangleright$  Below,  $ZZ_{\text{inv}} = 0$  when input is  $(X, Y, 0, 0) = \mathcal{O}$ 
4:    $x = X \times ZZ_{\text{inv}}$ 
5:   return  $x$   $\triangleright$  Point with  $x = 0$  is considered as  $\mathcal{O}$ 
```

The XYZZ coordinates do not enjoy a unified addition and doubling formula. We now review the XYZZ formulas for doubling then for addition in the following sections.

2.2.2 XYZZ point doubling formulas to implement DBLECC

We present hereafter the XYZZ doubling formula as described in [13] as well as its variants that are used in SCL. The different variants shall be referred to as **ECDBL**, **ECDBL2**, **ECDBLNEG** and **ECNEGDBL** in this report (this terminology is not introduced in the code and is used here for the sake of the presentation). In the scope of the SCL implementation considered here, only 3 variants are used: **ECDBL**, **ECDBLNEG** and **ECNEGDBL**.

The most generic doubling formula for XYZZ is **ECDBL** as presented in Algorithm 5: the input point $P = (X, Y, ZZ, ZZZ)$ is supposed to be on the curve (i.e. satisfy Equation 6 and satisfy $ZZZ^2 = ZZ^3$) and the output provides $2 \cdot P$. The formula accepts point at infinity \mathcal{O} as input since $ZZ = ZZZ = 0$ provides $ZZ' = ZZZ' = 0$ as output, i.e. the expected \mathcal{O} .

Algorithm 5 Doubling formula **ECDBL** for XYZZ point of coordinates $P = (X, Y, ZZ, ZZZ) \in \mathcal{E}(a, b, p)$, return $2 \cdot P$ (point \mathcal{O} accepted)

Require: Point is supposed to be on the curve

```

1: procedure ECDBL( $X, Y, ZZ, ZZZ$ )
2:    $U = 2 \times Y \bmod p$ 
3:    $V = U^2 \bmod p$ 
4:    $W = U \times V \bmod p$ 
5:    $S = X \times V \bmod p$ 
6:    $M = 3 \times X^2 + a \times ZZ^2 \bmod p$ 
7:    $X' = M^2 - 2 \times S \bmod p$ 
8:    $Y' = M \times (S - X') - W \times Y \bmod p$ 
9:    $ZZ' = V \times ZZ \bmod p$ 
10:   $ZZZ' = W \times ZZZ \bmod p$ 
11: return ( $X', Y', ZZ', ZZZ'$ )

```

A variant for doubling is when we consider an input point in its XYZZ affine form, i.e. $(X, Y, 1, 1)$. In this formula, the number of operations can be reduced for optimization as presented in Algorithm 6 with **ECDBL2**. The point at infinity is not accepted as it cannot be represented as $(X, Y, 1, 1)$ (see the discussion in Observation 2 (■)).

Algorithm 6 Doubling formula **ECDBL2** for point of normalized coordinates $P = (X, Y, 1, 1) \in \mathcal{E}(a, b, p)$, return $2 \cdot P$ (point \mathcal{O} **not** accepted)

Require: Point is supposed to be on the curve

```

1: procedure ECDBL2( $X, Y$ )
2:    $U = 2 \times Y \bmod p$ 
3:    $V = U^2 \bmod p$ 
4:    $W = U \times V \bmod p$ 
5:    $S = X \times V \bmod p$ 
6:    $M = 3 \times X^2 + a \bmod p$ 
7:    $X' = M^2 - 2 \times S \bmod p$ 
8:    $Y' = M \times (S - X') - W \times Y \bmod p$ 
9:    $ZZ' = V$ 
10:   $ZZZ' = W$ 
11: return ( $X', Y', ZZ', ZZZ'$ )

```

Beyond these two variants for doubling, SCL makes use of two other variants including negation. This is used in conjunction with addition formulas that include negation

to optimize some arithmetic operations (as we will expose later in Section 2.2.5). First, **ECDBLNEG** presented in Algorithm 7 performs doubling and then negation of the output, i.e. it computes for input P the point $-(2 \cdot P)$. Second, **ECNEGDBL** presented in Algorithm 8 performs negation of the input and then doubling, i.e. computes $2 \cdot (-P)$ – recall that in $XYZZ$, point negation is simply obtained by negating the Y coordinate: $-(X, Y, ZZ, ZZZ) = (X, -Y, ZZ, ZZZ)$. The point at infinity is accepted for both **ECDBLNEG** and **ECNEGDBL** as they are mere variations of **ECDBL**.

Algorithm 7 Doubling and negation formula **ECDBLNEG** for $XYZZ$ point of coordinates $P_1 = (X, Y, ZZ, ZZZ) \in \mathcal{E}(a, b, p)$, return $-(2 \cdot P) = -2 \cdot P$ (point \mathcal{O} accepted)

Require: Points are supposed to be on the curve

```

1: procedure ECDBLNEG( $X, Y, ZZ, ZZZ$ )
2:    $U = 2 \times Y \bmod p$ 
3:    $V = U^2 \bmod p$ 
4:    $W = U \times V \bmod p$ 
5:    $S = X \times V \bmod p$ 
6:    $M = 3 \times X^2 + a \times ZZ^2 \bmod p$ 
7:    $X' = M^2 - 2 \times S \bmod p$ 
8:    $Y' = M \times (X' - S) + W \times Y \bmod p$  ▷ Negation of output here
9:    $ZZ' = V \times ZZ \bmod p$ 
10:   $ZZZ' = W \times ZZZ \bmod p$ 
11:  return ( $X', Y', ZZ', ZZZ'$ )

```

Algorithm 8 Negation and doubling formula **ECNEGDBL** for $XYZZ$ point of coordinates $P_1 = (X, Y, ZZ, ZZZ) \in \mathcal{E}(a, b, p)$, return $2 \cdot (-P) = -2 \cdot P$ (point \mathcal{O} accepted)

Require: Points are supposed to be on the curve

```

1: procedure ECNEGDBL( $X, Y, ZZ, ZZZ$ )
2:    $U = -2 \times Y \bmod p$  ▷ Negation of input here
3:    $V = U^2 \bmod p$ 
4:    $W = U \times V \bmod p$ 
5:    $S = X \times V \bmod p$ 
6:    $M = 3 \times X^2 + a \times ZZ^2 \bmod p$ 
7:    $X' = M^2 - 2 \times S \bmod p$ 
8:    $Y' = M \times (S - X') - W \times Y \bmod p$ 
9:    $ZZ' = V \times ZZ \bmod p$ 
10:   $ZZZ' = W \times ZZZ \bmod p$ 
11:  return ( $X', Y', ZZ', ZZZ'$ )

```

2.2.3 $XYZZ$ points addition formulas to implement ADDECC

We present hereafter the $XYZZ$ addition as described in [13] as well as its variants that are used in SCL. The different variants shall be referred to as **ECADD**, **ECADD2**, **ECADD3** and

Algorithm 9 Addition formula **ECADD** for XYZZ points of coordinates $P_1 = (X_1, Y_1, ZZ_1, ZZZ_1), P_2 = (X_2, Y_2, ZZ_2, ZZZ_2) \in \mathcal{E}(a, b, p)$, return $P_1 + P_2$ (point \mathcal{O} **not** accepted, unless $P_1 = P_2 = \mathcal{O}$)

Require: Points are supposed to be on the curve, $P_1 \neq \mathcal{O}$ and $P_2 \neq \mathcal{O}$

```

1: procedure ECADD( $X_1, Y_1, ZZ_1, ZZZ_1, X_2, Y_2, ZZ_2, ZZZ_2$ )
2:    $U_1 = X_1 \times ZZ_2 \bmod p$ 
3:    $U_2 = X_2 \times ZZ_1 \bmod p$ 
4:    $S_1 = Y_1 \times ZZZ_2 \bmod p$ 
5:    $S_2 = Y_2 \times ZZZ_1 \bmod p$ 
6:    $P = U_2 - U_1 \bmod p$ 
7:    $R = S_2 - S_1 \bmod p$ 
8:    $PP = P^2 \bmod p$ 
9:    $PPP = P \times PP \bmod p$ 
10:   $Q = U_1 \times PP \bmod p$ 
11:   $X' = R^2 - PPP - 2 \times Q \bmod p$ 
12:   $Y' = R \times (Q - X') - S_1 \times PPP \bmod p$ 
13:   $ZZ' = ZZ_1 \times ZZ_2 \times PP \bmod p$ 
14:   $ZZZ' = ZZZ_1 \times ZZZ_2 \times PPP \bmod p$ 
15:  return ( $X', Y', ZZ', ZZZ'$ )

```

Algorithm 10 Addition formula **ECADD2** for XYZZ points of coordinates $P_1 = (X_1, Y_1, ZZ_1, ZZZ_1), P_2 = (X_2, Y_2, 1, 1) \in \mathcal{E}(a, b, p)$, return $P_1 + P_2$ (point \mathcal{O} **not** accepted). Note that P_2 must be normalized.

Require: Points are supposed to be on the curve, $P_1 \neq \mathcal{O}$ and $P_2 \neq \mathcal{O}$

```

1: procedure ECADD2( $X_1, Y_1, ZZ_1, ZZZ_1, X_2, Y_2$ )
2:    $U_2 = X_2 \times ZZ_1 \bmod p$ 
3:    $S_2 = Y_2 \times ZZZ_1 \bmod p$ 
4:    $P = U_2 - X_1 \bmod p$ 
5:    $R = S_2 - Y_1 \bmod p$ 
6:    $PP = P^2 \bmod p$ 
7:    $PPP = P \times PP \bmod p$ 
8:    $Q = X_1 \times PP \bmod p$ 
9:    $X' = R^2 - PPP - 2 \times Q \bmod p$ 
10:   $Y' = R \times (Q - X') - Y_1 \times PPP \bmod p$ 
11:   $ZZ' = ZZ_1 \times PP \bmod p$ 
12:   $ZZZ' = ZZZ_1 \times PPP \bmod p$ 
13:  return ( $X', Y', ZZ', ZZZ'$ )

```

Algorithm 11 Addition formula **ECADD3** for normalized XYZZ points of coordinates $P_1 = (X_1, Y_1, 1, 1), P_2 = (X_2, Y_2, 1, 1) \in \mathcal{E}(a, b, p)$, return $P_1 + P_2$ (point \mathcal{O} **not** accepted). Note that P_1 and P_2 must be normalized.

Require: Points are supposed to be on the curve, $P_1 \neq \mathcal{O}$ and $P_2 \neq \mathcal{O}$

```

1: procedure ECADD3( $X_1, Y_1, X_2, Y_2$ )
2:    $P = X_2 - X_1 \bmod p$ 
3:    $R = Y_2 - Y_1 \bmod p$ 
4:    $PP = P^2 \bmod p$ 
5:    $PPP = P \times PP \bmod p$ 
6:    $Q = X_1 \times PP \bmod p$ 
7:    $X' = R^2 - PPP - 2 \times Q \bmod p$ 
8:    $Y' = R \times (Q - X') - Y_1 \times PPP \bmod p$ 
9:    $ZZ' = PP \bmod p$ 
10:   $ZZZ' = PPP \bmod p$ 
11:  return ( $X', Y', ZZ', ZZZ'$ )

```

Algorithm 12 Negation and addition formula **ECNEGADD** for XYZZ points of coordinates $P_1 = (X_1, Y_1, ZZ_1, ZZZ_1), P_2 = (X_2, Y_2, ZZ_2, ZZZ_2) \in \mathcal{E}(a, b, p)$, return $-P_1 + P_2$ (point \mathcal{O} **not** accepted, unless $P_1 = P_2 = \mathcal{O}$)

Require: Points are supposed to be on the curve

```

1: procedure ECNEGADD( $X_1, Y_1, ZZ_1, ZZZ_1, X_2, Y_2, ZZ_2, ZZZ_2$ )
2:    $U_1 = X_1 \times ZZ_2 \bmod p$ 
3:    $U_2 = X_2 \times ZZ_1 \bmod p$ 
4:    $S_1 = Y_1 \times ZZZ_2 \bmod p$ 
5:    $S_2 = Y_2 \times ZZZ_1 \bmod p$ 
6:    $P = U_2 - U_1 \bmod p$ 
7:    $R = S_2 + S_1 \bmod p$  ▷ Negation of input  $P_1$  here
8:    $PP = P^2 \bmod p$ 
9:    $PPP = P \times PP \bmod p$ 
10:   $Q = U_1 \times PP \bmod p$ 
11:   $X' = R^2 - PPP - 2 \times Q \bmod p$ 
12:   $Y' = R \times (Q - X') - S_1 \times PPP \bmod p$ 
13:   $ZZ' = ZZ_1 \times ZZ_2 \times PP \bmod p$ 
14:   $ZZZ' = ZZZ_1 \times ZZZ_2 \times PPP \bmod p$ 
15:  return ( $X', Y', ZZ', ZZZ'$ )

```

ECNEGADD in this report (this terminology is not introduced in the code and is used here for the sake of the presentation). In the scope of the SCL implementation considered here, only 3 variants are used: **ECADD**, **ECADD2** and **ECNEGADD**.

The most general addition formula **ECADD** is provided with Algorithm 9: from two points P_1 and P_2 the addition $P_1 + P_2$ is computed. The point at infinity \mathcal{O} is not accepted as input unless $P_1 = P_2 = \mathcal{O}$: when ZZ_1 or ZZ_2 is zero, we have $ZZ' = 0$ which does not allow for the output to be equal to the other point (unless both points are the point at infinity, in which case the output is also \mathcal{O} and conforms to $ZZ' = 0$). The addition formula hence produces the point at infinity whenever $P = Q = \mathcal{O}$ or $P = -Q$ (in this case $U_1 = U_2$ in the formula of Algorithm 9, implying $PPP = PP = P = 0$ and then $ZZ' = ZZZ' = 0$).

More optimized addition formulas can take advantage of the normalized affine $XYZZ$ form $(X, Y, 1, 1)$ of at least one of the points. We can see **ECADD2** in Algorithm 10 where the second input P_2 is supposed to be normalized (the point at infinity is not accepted for P_1 and is inherently not accepted for P_2). Further optimized, **ECADD3** in Algorithm 11 supposes that both P_1 and P_2 are normalized (and inherently not equal to \mathcal{O}).

Finally, we describe **ECNEGADD** in Algorithm 12 where negation of the first input P_1 is performed before the addition, yielding $-P_1 + P_2$ as output. This combined negation and addition formula is used in SCL in conjunction with the double negation formulas to save some arithmetic operations.

Summary of the rules and exceptions for the formulas used in SCL:

- **ECDBL**, **ECDBLNEG** and **ECNEGDBL**: non-normalized $XYZZ$ points as input, \mathcal{O} where $ZZ = ZZZ = 0$ accepted as input and produced as output.
- **ECDBL2**: only normalized $XYZZ$ points as inputs (affine equivalent), \mathcal{O} not accepted and not produced.
- **ECADD**, **ECNEGADD**: non-normalized $XYZZ$ points as input, \mathcal{O} is not accepted unless the two points are both the point at infinity, \mathcal{O} is produced whenever inputs are opposite for **ECADD** and equal for **ECNEGADD**. Equal points are not accepted as inputs for **ECADD**, opposite points are not accepted as inputs for **ECNEGADD** (doubling formulas must be used in this case).
- **ECADD2**: non-normalized $XYZZ$ for the first point and normalized $XYZZ$ for the second point, \mathcal{O} not accepted as input, \mathcal{O} produced whenever the two input points are opposite. Equal input points are not accepted as inputs (doubling formulas must be used in this case).
- **ECADD3**: the two input points are normalized $XYZZ$, \mathcal{O} not accepted as input, \mathcal{O} produced whenever the two input points are opposite. Equal input points are not accepted as inputs (doubling formulas must be used in this case).

Doubling and addition formulas in SCL and the point at infinity case:

It is to be noted that when the formulas are used with the $ZZ = ZZZ = 0$ point at infinity convention, inputs following the requirements provided above, and input points $\neq \mathcal{O}$ in XYZZ coordinates that satisfy the extended Equation 6, then if the output point is \mathcal{O} , its XYZZ coordinates satisfy $ZZ = ZZZ = 0$:

- For the **ECDBL** family of formulas (with the negation variants), this is straightforward as ZZ' and ZZZ' are multiples of respectively ZZ and ZZZ .
- For the **ECADD** family of formulas, this is also straightforward as ZZ' and ZZZ' are multiple of ZZ_i and respectively ZZZ_i with $i \in \{1, 2\}$ (depending on the variant).

This means that as long as the input requirement rules of these formulas are respected, the point at infinity with $ZZ = ZZZ = 0$ will be respected, and it is enough to check if an XYZZ point is \mathcal{O} with only one $ZZ = 0$ check (which is what is performed in the DSM implementation of SCL).

2.2.4 DSM with Shamir's trick and windowing

The Shamir's trick (a.k.a. Strauss-Shamir's trick) is a method to compute a double scalar multiplication $u \cdot P + v \cdot Q$ by merging the point doublings arising in $u \cdot P$ and those arising in $v \cdot Q$ to reduce the overall number of operations. This is a particular case of the Pippenger (bucket) algorithm for multi-base scalar multiplication (or exponentiation). This trick can be combined with *windowing*, a classical optimisation that takes advantage of precomputed points (small multiple of the input base points) to reduce the number of additions in the main loop, yielding a so-called bidimensional Shamir's trick as explained in [17].

In the context of the SCL library, $u \cdot P + v \cdot Q$ is computed from input points which are power-of-two multiples of P and Q . Specifically, the scalars u and v are split in i parts of $\lceil \log_2(n) \rceil / i$ bits and the input points of the Shamir's trick multi-base scalar multiplication are defined as:

$$P_k := 2^{k \frac{\lceil \log_2(n) \rceil}{i}} \cdot P, \quad Q_k = 2^{k \frac{\lceil \log_2(n) \rceil}{i}} \cdot Q, \quad \text{for } k \in [0, i-1]$$

As a complementary optimization, these i parts are further split into j -bit windows. This bidimensional split is represented on Figure 1. One point is precomputed for each possible combination of $2i$ j -bit window values (i.e., one j -bit value per point P_k, Q_k), which makes a total of $(2^j)^{2i} - 1 = 2^{2ij} - 1$ precomputed points (discarding the all-0 combination which corresponds to the point at infinity \mathcal{O}). The most general algorithmic description of this Shamir's trick with windowing is provided in Algorithm 13: this will be referred to as $\text{DSM}_{i,j}$ for splitting of the scalars in i parts with window of size j bits.

In terms of computation, this roughly results in the following. For now, we use the generic DBLECC and ADDECC notations without the dedicated implementations from Section 2.2.2 and Section 2.2.3 as we use a high-level algorithmic overview:

- **Precomputation:** $2^{2ij} - 1$ points precomputed in Step 6 of Algorithm 13 given the $P_k, Q_k, k \in [0, i - 1]$ as inputs.⁴ Each point requires either one DBLECC or one ADDECC to be computed from input and/or previously precomputed points.
- **Core loop computation:** the main loop in Step 14 of Algorithm 13 uses $\frac{\lceil \log_2(n) \rceil}{i \times j}$ iterations. This represents $\frac{\lceil \log_2(n) \rceil}{i}$ DBLECC (each iteration performs j doublings) plus an average of $\frac{\lceil \log_2(n) \rceil}{i \times j} \times (1 - \frac{1}{2^{2ij}})$ ADDECC (each iteration performs at most 1 addition). This additions number averaging is due to the specific case of the point at infinity in Step 20 when $s = 0$, happening with a probability of $1 - \frac{1}{2^{2ij}}$.

The two variants implemented in the SCL library are the following for the P256 curve (with the value of n provided in Section 1.2, and $\lceil \log_2(n) \rceil = 256$ bits):

- **DSM_{2,1}:** this makes use of a Shamir's trick with a 2 split providing 4 base points: $P, Q, 2^{128} \cdot P$ and $2^{128} \cdot Q$, and a window size of 1 bit. Applying the previous generic formulas with $i = 2, j = 1$, 15 points are precomputed with 11 ADDECC: $T_0 = \mathcal{O}, T_1 = P, T_2 = 2^{128} \cdot P, T_3 = P + 2^{128} \cdot P, T_4 = Q, T_5 = P + Q, T_6 = 2^{128} \cdot P + Q, T_7 = 2^{128} \cdot P + Q + P = T_6 + P, T_8 = 2^{128} \cdot Q, T_9 = P + 2^{128} \cdot Q, T_{10} = 2^{128} \cdot P + 2^{128} \cdot Q, T_{11} = Q + 2^{128} \cdot P + 2^{128} \cdot Q = T_{10} + Q, T_{12} = Q + 2^{128} \cdot Q, T_{13} = Q + 2^{128} \cdot Q + P = T_{12} + P, T_{14} = 2^{128} \cdot P + Q + 2^{128} \cdot Q = T_6 + 2^{128} \cdot Q$, and $T_{15} = P + 2^{128} \cdot P + Q + 2^{128} \cdot Q = T_{14} + P$ (without computation for T_0 and the input points T_1, T_2, T_4, T_8 , and one addition for the other points).

The core loop of the DSM then makes use of 128 DBLECC and an average of 120 ADDECC.

- **DSM_{1,2}:** in this case, the Shamir's trick uses a split of $i = 1$ meaning only the two points P and Q as inputs. The windowing is of size $j = 2$, and applying the previous formulas lead to a precomputation of 15 points using 2 DBLECC and 11 ADDECC: $T_0 = \mathcal{O}, T_1 = P, T_2 = 2 \cdot P, T_3 = 3 \cdot P = T_2 + P, T_4 = Q, T_5 = P + Q, T_6 = 2 \cdot P + Q = T_5 + P, T_7 = 3 \cdot P + Q = T_6 + P, T_8 = 2 \cdot Q, T_9 = P + 2 \cdot Q = T_8 + P, T_{10} = 2 \cdot P + 2 \cdot Q = T_9 + Q, T_{11} = 3 \cdot P + 2 \cdot Q = T_{10} + P, T_{12} = 3 \cdot Q = T_8 + Q, T_{13} = 3 \cdot Q + P = T_{12} + P, T_{14} = 3 \cdot Q + 2 \cdot P = T_{13} + P$, and $T_{15} = 3 \cdot Q + 3 \cdot P = T_{14} + P$ (without computation for T_0 and the input points T_1, T_4 , one doubling for T_2, T_8 and one addition for the other points).

The core loop of the DSM then makes use of 256 DBLECC and an average of 120 ADDECC.

⁴The point at infinity for $z = 0$ is not accounted as it can be implicitly computed.

Algorithm 13 Double scalar multiplication (DSM) with Shamir's trick and windowing: given two points $P, Q \in \mathcal{E}(a, b, p)$ and scalars $u, v \in [0, n - 1]$ compute $u \cdot P + v \cdot Q$. Parameters are $i \geq 1$ the split of the scalars and $j \geq 1$ the window size. We denote $P_k = 2^{k \frac{\lceil \log_2(n) \rceil}{i}} \cdot P$ and $Q_k = 2^{k \frac{\lceil \log_2(n) \rceil}{i}} \cdot Q$ for $k \in [0, i - 1]$, which are supposed to be precomputed and provided as inputs (it should be noticed that $P_0 = P$ and $Q_0 = Q$).

Require: Input points are supposed to be on the curve (not \mathcal{O})

```

1: procedure DSMi,j( $\{P_k\}_{k \in [0, i-1]}$ ,  $\{Q_k\}_{k \in [0, i-1]}$ ,  $u, v$ )
2:   if  $u = 0$  and  $v = 0$  then
3:     return  $\mathcal{O}$ 
4:   Split the  $u$  and  $v$  viewed as  $\lceil \log_2(n) \rceil$  bit strings in  $i$  parts:  $x = \sum_{k=0}^{i-1} 2^{k \frac{\lceil \log_2(n) \rceil}{i}} \times x_k$ 
   for  $x \in \{u, v\}$ .  $u_k$  and  $v_k$  are strings of  $\frac{\lceil \log_2(n) \rceil}{i}$  bits

   The DSM can then be viewed as:
   
$$u \cdot P + v \cdot Q = \sum_{k=0}^{i-1} (u_k \cdot P_k + v_k \cdot Q_k)$$

5:   We define the following  $j$  bits extraction at position  $k$  function from the scalar  $x$ ,
   where  $\gg$  is the logical right shift,  $\ll$  is the logical left shift, and  $\&$  is the logical
   AND:
   
$$\text{Ext}(x, k, j) = (x \gg k) \& (2^j - 1)$$

6:   Precomputation: Precompute the  $2^{2ij} - 1$  points,  $\forall z \in [0, 2^{2ij} - 1]$ :
   
$$T_z = \sum_{k=0}^{i-1} \text{Ext}(z, k \times j, j) \cdot P_k + \text{Ext}(z \gg (i \times j), k \times j, j) \cdot Q_k$$

    $\triangleright$  We have  $T_0 = \mathcal{O}$ 
7:   Find first non-zero MSBs of scalars  $u, v$ :
8:    $\text{shift} = \frac{\lceil \log_2(n) \rceil}{i} - j$ 
9:   repeat
10:     $s = \sum_{k=0}^{i-1} (\text{Ext}(u_k, \text{shift}, j) + (\text{Ext}(v_k, \text{shift}, j) \ll j)) \ll (k \times 2 \times j)$ 
11:     $\text{shift} = \text{shift} - j$ 
12:  until  $s \neq 0$ 
13:  Set initial value point:  $R = T_s$ 
14:  DSM left-to-right scalars parsing loop:
15:  repeat
16:     $R = 2^j R$   $\triangleright$  Perform  $j$  EC doublings
17:     $s = \sum_{k=0}^{i-1} (\text{Ext}(u_k, \text{shift}, j) + (\text{Ext}(v_k, \text{shift}, j) \ll j)) \ll (k \times 2 \times j)$ 
18:     $\text{shift} = \text{shift} - j$ 
19:    if  $s = 0$  then
20:      continue  $\triangleright$  Precomputed is  $T_0 = \mathcal{O}$ , skip the addition
21:    else
22:       $R = R + T_s$   $\triangleright$  Perform EC addition with precomputed point
23:  until  $\text{shift} = 0$ 
24:  return  $R$ 

```

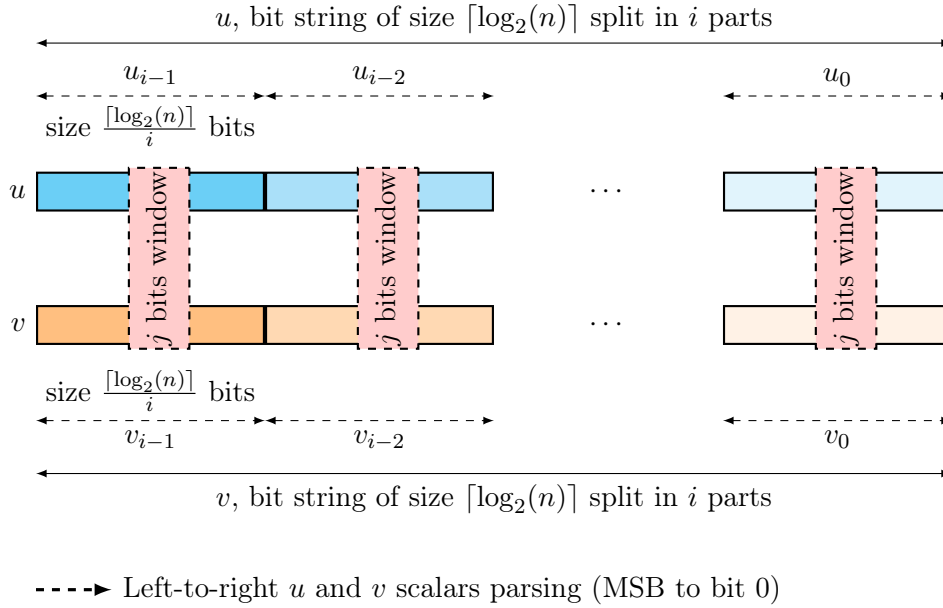


Figure 1: Bidimensional windowed Shamir's trick optimization for Double Scalar Multiplication $\text{DSM}_{i,j}$: this uses a i parts split of u and v , with a j bits window size, i.e. j bits are extracted from each part to form each double and add iteration local scalar bits.

Q Remark 1: Improved computation-memory trade-off for DSM precomputation

The computation-memory trade-off of the DSM precomputation could be improved using signed digits in the scalar representation. Specifically, using techniques proposed in [21, 22], one can leverage signed digit representations to halve the number of pre-computed points.

2.2.5 Implementation of $\text{DSM}_{i,j}$ in the SCL library

The files `SCL_mulumuladdX_fullgen_b4.sol` `SCL_mulumuladdX_fullgenW.sol` implement respectively $\text{DSM}_{1,2}$ and $\text{DSM}_{2,1}$ with DBLECC and ADDECC using XYZZ coordinates as described in Section 2.2.1. In this section, we provide the algorithmic description of these routines. Algorithm 14 depicts the routine from `SCL_mulumuladdX_fullgen_b4.sol` using Shamir's trick with 4 base points and a 1-bit window. We shall refer to this routine as `SCL_DSM_B4` in this report. Algorithm 15 depicts the routine using Shamir's trick with 2 base points and a 2-bit window in `SCL_mulumuladdX_fullgenW.sol`. We shall refer to this routine as `SCL_DSM_W` in this report. We note that the terminology `SCL_DSM_B4` and `SCL_DSM_W` are introduced here for the sake of the presentation but are not used in the audited code. The building blocks of these algorithms are `ECDBL`, `ECDBLNEG` and `ECNEGBL`.

as presented in Section 2.2.2, and **ECADD2** and **ECNEGADD** as presented in Section 2.2.3. The other variants **ECDBL2** and **ECADD3** have been introduced for completeness, and to exhibit potential implementation optimizations discussed in Section 3 – see Observation 14 (●) and Remark 6 (Q).

Some elements must be noticed:

- In the precomputation phase, it is possible to use the specific **ECADD2** addition variant because the second point is always normalized, as it is one of the function inputs (i.e. coming from affine coordinates).
- The usage of the negation variants **ECDBLNEG** and **ECNEGADD** in the core loop of the scalars parsing allows to save operations due to the fact that EVM opcodes offer modular additions with **addmod**, but do not offer modular subtraction per se (a regular **sub** opcode is used and negation modulo p of a value x is computed with $\text{sub}(p, x) = p - x = -x \bmod p$).
- Because of the usage of addition **ECADD2** in the precomputation step, some constraints exist on the input points P and Q . These constraints are of two natures:
 - Addition formulas suppose that the two input points are different (otherwise the dedicated doubling formula must be used). While the **ECNEGADD** in the main loop is protected by specific conditions that avoid this, this is not the case of **ECADD2** usage during the precomputation phase. This induces some constraints on the two input points P and Q each time **ECADD2** is used.
 - Addition formulas also suppose that neither of the two input points is the point at infinity \mathcal{O} . The main consequence is that all the elements in the precomputed table T_s must not equal \mathcal{O} (except T_0) so that the additions in the main loop avoid an input T_s equal to \mathcal{O} . As a result, all the **ECADD2** involved during precomputation (each of them yielding a precomputed point) should take input points which are not opposite. Whenever fulfilled, these constraints further ensure that the point at infinity does not occur in input of an **ECADD2** involved in the precomputation.

This brings 15 constraints for Algorithm 14 and 9 constraints for Algorithm 15 summarized in the **Require** field of the description. These constraints will naturally provide a blacklist of forbidden ECDSA weak public keys as detailed in Section 2.3.

- Compared to the general high-level algorithm provided in Algorithm 13, a coordinate normalization step is added at the end of **SCL_DSM_B4** and **SCL_DSM_W** to translate the $XYZZ$ coordinates into affine coordinates. In fact, only the x coordinate of the output affine point is returned, due to the way the ECDSA verification is performed (see Section 1.2.2). This normalization step consists in computing $R_X \times R_{ZZ}^{-1} \bmod p$ by using the **ModExp** external precompile for the modular inversion. By returning an x -only output, these algorithms save the normalization of the y coordinates (which would cost a few additional modular multiplications assuming the use of Montgomery's trick to avoid an additional modular inversion).

Algorithm 14 Double scalar multiplication (DSM) with Shamir's trick and windowing. This is the specific implementation of $\text{DSM}_{2,1}$ (see Algorithm 13 for the notations) as implemented in the file `libSCL_ecdsab4.sol`. Inputs are $P_0 = P$, $P_1 = 2^{128} \cdot P$, $Q_0 = Q$, $Q_1 = 2^{128} \cdot Q$, as well as scalars $u, v \in [0, n-1]$. The function returns only the X coordinate of the output point.

Require: Input points are supposed to be on the curve (not \mathcal{O}), and they respect the following 15 constraints:

$$P_0 \neq \pm Q_0, P_1 \neq \pm Q_0, P_1 + Q_0 \neq \pm P_0, Q_1 \neq \pm P_0, Q_1 + P_1 \neq Q_0, Q_0 + Q_1 \neq \pm P_0, P_1 + Q_0 \neq \pm Q_1, \\ P_1 + Q_0 + Q_1 \neq \pm P_0$$

```

1: procedure SCL_DSM_B4( $P_{0x}, P_{0y}, P_{1x}, P_{1y}, Q_{0x}, Q_{0y}, Q_{1x}, Q_{1y}, u, v$ )
2:   if  $u = 0$  and  $v = 0$  then
3:     return  $\mathcal{O}$   $\triangleright (0,0)$  represents the point at infinity  $\mathcal{O}$  by convention
4:   Precomputation: Precompute 15 points in memory table  $T$ . This table is made of 15 cells of 4 uint256
    values representing  $(X, Y, ZZ, ZZZ)$ .  $T_0 = \mathcal{O}$  implicitly.
5:    $\triangleright$  Note in the following affine to  $XYZZ$  implicit transfer with  $ZZ = ZZZ = 1$ 
6:    $T_1 = (P_{0x}, P_{0y}, 1, 1)$   $\triangleright T_1 = P_0$ 
7:    $T_2 = (P_{1x}, P_{1y}, 1, 1)$   $\triangleright T_2 = P_1$ 
8:    $T_3 = \text{ECADD2}(P_{0x}, P_{0y}, 1, 1, P_{1x}, P_{1y})$   $\triangleright T_3 = P_0 + P_1$ 
9:    $T_4 = (Q_{0x}, Q_{0y}, 1, 1)$   $\triangleright T_4 = Q_0$ 
10:   $T_5 = \text{ECADD2}(P_{0x}, P_{0y}, 1, 1, Q_{0x}, Q_{0y})$   $\triangleright T_5 = P_0 + Q_0$ 
11:   $T_6 = \text{ECADD2}(P_{1x}, P_{1y}, 1, 1, Q_{0x}, Q_{0y})$   $\triangleright T_6 = P_1 + Q_0$ 
12:   $T_7 = \text{ECADD2}(T_{6x}, T_{6y}, T_{6zz}, T_{6zzz}, P_{0x}, P_{0y})$   $\triangleright T_7 = (P_1 + Q_0) + P_0$ 
13:   $T_8 = (Q_{1x}, Q_{1y}, 1, 1)$   $\triangleright T_8 = Q_1$ 
14:   $T_9 = \text{ECADD2}(Q_{1x}, Q_{1y}, 1, 1, P_{0x}, P_{0y})$   $\triangleright T_9 = Q_1 + P_0$ 
15:   $T_{10} = \text{ECADD2}(Q_{1x}, Q_{1y}, 1, 1, P_{1x}, P_{1y})$   $\triangleright T_{10} = Q_1 + P_1$ 
16:   $T_{11} = \text{ECADD2}(T_{10x}, T_{10y}, T_{10zz}, T_{10zzz}, Q_{0x}, Q_{0y})$   $\triangleright T_{11} = (Q_1 + P_1) + Q_0$ 
17:   $T_{12} = \text{ECADD2}(Q_{0x}, Q_{0y}, 1, 1, Q_{1x}, Q_{1y})$   $\triangleright T_{12} = Q_0 + Q_1$ 
18:   $T_{13} = \text{ECADD2}(T_{12x}, T_{12y}, T_{12zz}, T_{12zzz}, P_{0x}, P_{0y})$   $\triangleright T_{13} = (Q_0 + Q_1) + P_0$ 
19:   $T_{14} = \text{ECADD2}(T_{6x}, T_{6y}, T_{6zz}, T_{6zzz}, Q_{1x}, Q_{1y})$   $\triangleright T_{14} = (P_1 + Q_0) + Q_1$ 
20:   $T_{15} = \text{ECADD2}(T_{14x}, T_{14y}, T_{14zz}, T_{14zzz}, P_{0x}, P_{0y})$   $\triangleright T_{15} = (P_1 + Q_0 + Q_1) + P_0$ 

  Denote  $\text{MSK}(x, m, \text{sr}, \text{sl}) = \text{Z}((x \gg \text{sl}) \& m) \ll \text{sl}$ ,  $\text{Z}(x) = 0$  if  $x = 0$  and  $\text{Z}(x) = 1$  otherwise.

21:  Find first non-zero MSBs of scalars  $u, v$ :
22:   $m = 1 \ll 127$   $\triangleright$  Mask to parse the scalars
23:  repeat
24:     $s = \text{MSK}(u, m, 0, 0) + \text{MSK}(u, m, 128, 1) + \text{MSK}(v, m, 0, 2) + \text{MSK}(v, m, 128, 3)$ 
25:     $m = m \gg 1$ 
26:  until  $s \neq 0$ 

27:  Set initial value point:  $R = T_s$ 

28:  DSM left-to-right scalars parsing loop:
29:  repeat
30:     $R = \text{ECDBLNEG}(R_x, R_y, R_{zz}, R_{zzz})$   $\triangleright R = -2 \cdot R$ 
31:     $s = \text{MSK}(u, m, 0, 0) + \text{MSK}(u, m, 128, 1) + \text{MSK}(v, m, 0, 2) + \text{MSK}(v, m, 128, 3)$ 
32:    if  $s = 0$  then
33:       $R_y = -R_y$   $\triangleright$  Negate for the next round
34:      continue  $\triangleright$  Precomputed is  $T_0 = \mathcal{O}$ , skip the addition
35:    if  $R_{zz} = 0$  then  $\triangleright R = \mathcal{O}$ , result is  $T_s$ 
36:       $R = T_s$ 
37:      continue
38:    if  $R_x \times T_{szz} = T_{sx} \times R_{zz}$  and  $R_y \times T_{szzz} = T_{sy} \times R_{zzz}$  then
39:       $\triangleright$  Case  $R = T_s$ , must double with pre-negation:  $R = 2 \cdot (-R)$ 
40:       $R = \text{ECNEGDBL}(R_x, R_y, R_{zz}, R_{zzz})$ 
41:    else
42:       $\triangleright$  Nominal addition case:  $R = -R + T_s$ 
43:       $R = \text{ECNEGADD}(R_x, R_y, R_{zz}, R_{zzz}, T_{sx}, T_{sy}, T_{szz}, T_{szzz})$ 
44:       $m = m \gg 1$ 
45:    until  $m = 0$ 
46:   $x = R_x \times \text{ModInv}(R_{zz}, p)$   $\triangleright$   $X$  coordinate normalization  $x = R_x \times R_{zz}^{-1} \bmod p$  ( $R_{zz} = 0 \Rightarrow R = \mathcal{O}$ ,  $x = 0$ )
47:  return  $x$ 

```


Algorithm 15 Double scalar multiplication (DSM) with Shamir's trick and windowing. This is the specific implementation of $\text{DSM}_{1,2}$ (see Algorithm 13 for the notations) as implemented in the file `SCL_mulumuladdX_fullgenW.sol`. Inputs are P , Q , and scalars $u, v \in [0, n-1]$. The function returns only the X coordinate of the output point.

Require: Input points are supposed to be on the curve (not \mathcal{O}), and they respect the following 9 constraints: $Q \neq -P$, $Q \neq -2 \cdot P$, $Q \neq -3 \cdot P$, $2 \cdot Q \neq \pm P$, $2 \cdot Q \neq -3 \cdot P$, $3 \cdot Q \neq \pm P$, $3 \cdot Q \neq -2 \cdot P$.

```

1: procedure SCL_DSM_W( $P_x, P_y, Q_x, Q_y, u, v$ )
2:   if  $u = 0$  and  $v = 0$  then
3:     return  $\mathcal{O}$   $\triangleright (0,0)$  represents the point at infinity  $\mathcal{O}$  by convention
4:   Precomputation: Precompute 15 points in memory table  $T$ . This table is made of 15 cells of 4 uint256
    values representing  $(X, Y, ZZ, ZZZ)$ .  $T_0 = \mathcal{O}$  implicitly.
5:    $\triangleright$  Note in the following affine to  $XYZZ$  implicit transfer with  $ZZ = ZZZ = 1$ 
6:    $T_1 = (P_x, P_y, 1, 1)$   $\triangleright T_1 = P$ 
7:    $T_2 = \text{ECDBL}(P_x, P_y, 1, 1)$   $\triangleright T_2 = 2 \cdot P$ 
8:    $T_3 = \text{ECADD2}(T_{2X}, T_{2Y}, T_{2ZZ}, T_{2ZZZ}, P_x, P_y)$   $\triangleright T_3 = 2 \cdot P + P = 3 \cdot P$ 
9:    $T_4 = (Q_x, Q_y, 1, 1)$   $\triangleright T_4 = Q$ 
10:   $T_5 = \text{ECADD2}(P_x, P_y, 1, 1, Q_x, Q_y)$   $\triangleright T_5 = P + Q$ 
11:   $T_6 = \text{ECADD2}(T_{5X}, T_{5Y}, T_{5ZZ}, T_{5ZZZ}, P_x, P_y)$   $\triangleright T_6 = (P + Q) + P = 2 \cdot P + Q$ 
12:   $T_7 = \text{ECADD2}(T_{6X}, T_{6Y}, T_{6ZZ}, T_{6ZZZ}, P_x, P_y)$   $\triangleright T_7 = (2 \cdot P + Q) + P = 3 \cdot P + Q$ 
13:   $T_8 = \text{ECDBL}(Q_x, Q_y, 1, 1)$   $\triangleright T_8 = 2 \cdot Q$ 
14:   $T_9 = \text{ECADD2}(T_{8X}, T_{8Y}, T_{8ZZ}, T_{8ZZZ}, P_x, P_y)$   $\triangleright T_9 = 2 \cdot Q + P$ 
15:   $T_{10} = \text{ECADD2}(T_{9X}, T_{9Y}, T_{9ZZ}, T_{9ZZZ}, P_x, P_y)$   $\triangleright T_{10} = (2 \cdot Q + P) + P = 2 \cdot Q + 2 \cdot P$ 
16:   $T_{11} = \text{ECADD2}(T_{10X}, T_{10Y}, T_{10ZZ}, T_{10ZZZ}, P_x, P_y)$   $\triangleright T_{11} = (2 \cdot Q + 2 \cdot P) + P = 2 \cdot Q + 3 \cdot P$ 
17:   $T_{12} = \text{ECADD2}(T_{8X}, T_{8Y}, T_{8ZZ}, T_{8ZZZ}, Q_x, Q_y)$   $\triangleright T_{12} = 2 \cdot Q + Q = 3 \cdot Q$ 
18:   $T_{13} = \text{ECADD2}(T_{12X}, T_{12Y}, T_{12ZZ}, T_{12ZZZ}, P_x, P_y)$   $\triangleright T_{13} = 3 \cdot Q + P$ 
19:   $T_{14} = \text{ECADD2}(T_{13X}, T_{13Y}, T_{13ZZ}, T_{13ZZZ}, P_x, P_y)$   $\triangleright T_{14} = (3 \cdot Q + P) + P = 3 \cdot Q + 2 \cdot P$ 
20:   $T_{15} = \text{ECADD2}(T_{14X}, T_{14Y}, T_{14ZZ}, T_{14ZZZ}, P_x, P_y)$   $\triangleright T_{15} = (3 \cdot Q + 2 \cdot P) + P = 3 \cdot Q + 3 \cdot P$ 

  Denote  $\text{MSK}(x, m, sr, sl) = Z((x \gg sl) \& m) \ll sl$ ,  $Z(x) = 0$  if  $x = 0$  and  $Z(x) = 1$  otherwise.

21: Find first non-zero MSBs of scalars  $u, v$ :
22:  $m = 1 \ll 255$   $\triangleright$  Mask to parse the scalars
23: repeat
24:    $s = \text{MSK}(u, m \gg 1, 0, 0) + \text{MSK}(u, m, 0, 1) + \text{MSK}(v, m \gg 1, 0, 2) + \text{MSK}(v, m, 0, 3)$ 
25:    $m = m \gg 2$ 
26: until  $s \neq 0$ 

27: Set initial value point:  $R = T_s$ 

28: DSM left-to-right scalars parsing loop:
29: repeat
30:    $R = \text{ECDBL}(R_X, R_Y, R_{ZZ}, R_{ZZZ})$ 
31:    $R = \text{ECDBLNEG}(R_X, R_Y, R_{ZZ}, R_{ZZZ})$   $\triangleright$  We get  $R = -2^2 \cdot R$ 
32:    $s = \text{MSK}(u, m \gg 1, 0, 0) + \text{MSK}(u, m, 0, 1) + \text{MSK}(v, m \gg 1, 0, 2) + \text{MSK}(v, m, 0, 3)$ 
33:   if  $s = 0$  then
34:      $R_Y = -R_Y$   $\triangleright$  Negate for the next round
35:     continue  $\triangleright$  Precomputed is  $T_0 = \mathcal{O}$ , skip the addition
36:   if  $R_{ZZ} = 0$  then  $\triangleright R = \mathcal{O}$ , result is  $T_s$ 
37:      $R = T_s$ 
38:     continue
39:   if  $R_X \times T_{sZZ} = T_{sX} \times R_{ZZ}$  and  $R_Y \times T_{sZZZ} = T_{sY} \times R_{ZZZ}$  then
40:      $\triangleright$  Case  $R = T_s$ , must double with pre-negation:  $R = 2 \cdot (-R)$ 
41:      $R = \text{ECNEGDBL}(R_X, R_Y, R_{ZZ}, R_{ZZZ})$ 
42:   else
43:      $\triangleright$  Nominal addition case:  $R = -R + T_s$ 
44:      $R = \text{ECNEGADD}(R_X, R_Y, R_{ZZ}, R_{ZZZ}, T_{sX}, T_{sY}, T_{sZZ}, T_{sZZZ})$ 
45:      $m = m \gg 2$ 
46: until  $m = 0$ 

47:  $x = R_X \times \text{ModInv}(R_{ZZ}, p)$   $\triangleright$   $X$  coordinate normalization  $x = R_X \times R_{ZZ}^{-1} \bmod p$  ( $R_{ZZ} = 0 \Rightarrow R = \mathcal{O}$ ,  $x = 0$ )
48: return  $x$ 

```

Q Remark 2: Consequences of the x -only output of the DSM algorithm

As explained, the SCL DSM formulas use a x -only output. This brings an output collision on possible opposite points on the curve that share the same x and opposite y .

Secondly, point at infinity is represented by convention in SCL using $(0, 0)$ in affine coordinates. As shown in the normalization step of Algorithm 14 and Algorithm 15, when the output point is the point at infinity we have $R_{ZZ} = 0$, yielding the output $x = 0$ which is indeed compatible with the representation $(0, 0)$. However, it is to be noted that the Weierstraß Equation 1 accepts points of zero coordinate $x = 0$ whenever b is a quadratic residue in \mathbb{F}_p , which is the case for P256. Indeed, the following two points $(0, \pm\sqrt{b})$ are on P256:

```
(0, 0x66485c780e2f83d72433bd5d84a06bb6541c2af31dae871728bf856a174f93f4)
(0, 0x99b7a386f1d07c29dbcc42a27b5f9449abe3d50de25178e8d7407a95e8b06c0b)
```

This means that whenever $u \cdot P + v \cdot Q$ is equal to one of these two points (which can happen e.g. when $u = 0$, $v = 1$, and Q is one of the two previous points), the output of the DSM formulas will be $x = 0$ and can be confused with the point at infinity since these formulas use a x -only coordinate output.

■ Observation 3: Consequences of the x -only output of the DSM algorithm

As explained in Remark 2 (Q), the x -only output of the DSM algorithm together with the handling of the point at infinity can induce collisions and/or confusion between the points $(0, \pm\sqrt{b})$ and \mathcal{O} . Another consequence is also observed hereafter (see Observation 6 (■)). This should be documented in the library.

Recommendation:

Document the observations raised in Remark 2 (Q).

2.3 ECDSA verification and weak public keys

Building upon the DSM implementations described in Section 2.2.5, the SCL library implements two variants of the ECDSA verification procedure described in Section 1.2.2:

- The `ECDSA_VERIFY_RIP7212` procedure as shown in Algorithm 16 implements the RIP-7212 API, taking as input the hash m , the signature r, s , and the public key affine coordinates Q_x and Q_y . After checking that r and s are in $[1, n-1]$, the modular

Algorithm 16 ECDSA signature verification in the SCL library, based on the SCL_DSM_W formula described in Algorithm 15, over curve P256 $\mathcal{E}(a, b, p)$ of order n and generator $G = (G_x, G_y)$. Input scalars $u, v \in [0, 2^{256} - 1]$.

Require: Public key $Q = (Q_x, Q_y)$ is supposed to be on the curve and not \mathcal{O} , and Q is not in a set of 9 blacklisted weak keys:

$$\{-G, -2 \cdot G, -3 \cdot G, \frac{1}{2} \cdot G, -\frac{1}{2} \cdot G, -\frac{3}{2} \cdot G, \frac{1}{3} \cdot G, -\frac{1}{3} \cdot G, -\frac{2}{3} \cdot G\}$$

```

1: procedure ECDSA_VERIFY_RIP7212( $m, r, s, Q_x, Q_y$ )
2:   if  $s = 0$  or  $r = 0$  or  $s > n$  or  $r > n$  then
3:     return False  $\triangleright$  Signature not OK
4:    $s^{-1} \bmod n = \text{ModInv}(s, n)$   $\triangleright s$  cannot be 0
5:    $u = m \times s^{-1} \bmod n$ 
6:    $v = r \times s^{-1} \bmod n$ 
7:    $O_x = \text{SCL\_DSM\_W}(G_x, G_y, Q_x, Q_y, u, v)$   $\triangleright O_y$  not needed
8:    $\delta = (O_x - r) \bmod n$ 
9:   if  $\delta = 0$  then
10:    return True  $\triangleright$  Signature OK
11:  else
12:    return False  $\triangleright$  Signature not OK

```

Algorithm 17 ECDSA signature verification in the SCL library, based on the SCL_DSM_B4 formula described in Algorithm 14, over curve P256 $\mathcal{E}(a, b, p)$ of order n and generator $G = (G_x, G_y)$. As inputs, beyond $Q = (Q_x, Q_y)$ and $G = (G_x, G_y)$ we consider precomputed elements $Q' = 2^{128} \cdot Q$ and $G' = 2^{128} \cdot G$. Input scalars $u, v \in [0, 2^{256} - 1]$.

Require: Inputs Q, G, Q', G' are supposed to be on the curve and not \mathcal{O} , and Q is not in a set of 14 blacklisted weak keys:

$$\{\pm G, \pm 2^{128} \cdot G, \pm \frac{1}{2^{128}} \cdot G, (1 - 2^{128}) \cdot G, -(1 + 2^{128}) \cdot G, \pm \frac{2^{128}}{1 - 2^{128}} \cdot G, \pm \frac{1}{1 + 2^{128}} \cdot G, -\frac{2^{128}}{1 + 2^{128}} \cdot G, \frac{1 - 2^{128}}{1 + 2^{128}} \cdot G\}$$

```

1: procedure ECDSA_VERIFY_B4( $m, r, s, Q_x, Q_y, Q'_x, Q'_y, G_x, G_y, G'_x, G'_y$ )
2:   if  $s = 0$  or  $r = 0$  or  $s > n$  or  $r > n$  then
3:     return False  $\triangleright$  Signature not OK
4:    $s^{-1} \bmod n = \text{ModInv}(s, n)$   $\triangleright s$  cannot be 0
5:    $u = m \times s^{-1} \bmod n$ 
6:    $v = r \times s^{-1} \bmod n$ 
7:    $O_x = \text{SCL\_DSM\_B4}(G_x, G_y, G'_x, G'_y, Q_x, Q_y, Q'_x, Q'_y, u, v)$   $\triangleright O_y$  not needed
8:    $\delta = (O_x - r) \bmod n$ 
9:   if  $\delta = 0$  then
10:    return True  $\triangleright$  Signature OK
11:  else
12:    return False  $\triangleright$  Signature not OK

```

inverse $s^{-1} \bmod n$ is computed using `ModInv`. The two scalar u, v are computed with modular multiplications, and `SCL_DSM_W` (Algorithm 15) is called to get the result of $u \times G + v \times Q$ x affine coordinate. The generator point $G = (G_x, G_y)$ is considered to be implicitly known as the verification function is dedicated to the P256 curve. The list of 9 constraints in the **Require** of Algorithm 15 induces a blacklist of 9 weak public keys summarized in the **Require** of Algorithm 16.

- The `ECDSA_VERIFY_B4` procedure as shown in Algorithm 17 implements the extended API of the proposed RIP-7696. It takes as inputs the hash m , the signature r, s , the public key affine coordinates Q_x and Q_y , the public key multiple $Q' = 2^{128} \cdot Q$ affine coordinates Q'_x and Q'_y , the curve generator affine coordinates G_x and G_y , the generator multiple $G' = 2^{128} \cdot G$ affine coordinates G'_x and G'_y . The ECDSA verification algorithm is the same as for `ECDSA_VERIFY_RIP7212` except that `SCL_DSM_B4` is called for DSM. The list of 15 constraints in the **Require** of Algorithm 14 induces a blacklist of 14 weak public keys summarized in the **Require** of Algorithm 17.

■ Observation 4: Raw ECDSA signature forgeries

This Observation is inherent to the choice of **raw ECDSA** as described in Section 1.2.2.

- The particular value for the input message $m = 0 \bmod n$ (e.g. $m = 0$ or $m = n$) leads to $u = m \times s^{-1} \bmod n = 0$. This allows, given any public key Q , to forge a valid signature **without knowing the private key**. Here is the forgery scenario. Given a known public key $Q \in \mathcal{E}(a, b, p)$ on the curve, for any random $\alpha \in [1, n - 1]$, let us denote $F = \alpha \cdot Q$, $\tilde{r} = F_x \bmod n$, $\tilde{s} = \tilde{r} \times \alpha^{-1} \bmod n = F_x \times \alpha^{-1} \bmod n$. The signature (\tilde{r}, \tilde{s}) with the null message $m = 0$ satisfies the ECDSA verification algorithm. Indeed, we have $u = 0$, and $v = \tilde{r} \times \tilde{s}^{-1} = \alpha$, then $u \cdot G + v \cdot Q = \alpha \cdot Q = F$, and the relation $F_x = \tilde{r} \bmod n$ is satisfied by construction.
- One can also obtain signature forgeries for any public key Q and non-zero values of m . Simply pick random values of u and v , compute $F = u \cdot G + v \cdot Q$ and get $r = F_x$. Then define $s = r \times v^{-1}$ and $m = u \times s$. This yields a valid signature according to the raw verification formulas.

The above forgery attacks are avoided in regular ECDSA by using a cryptographic hash function: m is the result of the hash function applied to the input message and cannot be chosen to be equal to 0 or as the value obtained while performing the second attack. We assume that in the context of the SCL library, the input m will always be the result of a 256-bit hash function, hence the ■ rating of the Observation.

Recommendation:

Document the raw ECDSA issue and the fact that the exposed API must take as input a hash digest for m .

■ Observation 5: Erroneous blacklist of weak keys

The list of weak keys is summarized in [1] for `ECDSA_VERIFY_B4`, and it is a work in progress for `ECDSA_VERIFY_RIP7212`. For `ECDSA_VERIFY_B4`, the list is incomplete and contains a few errors. The missing weak keys are mostly due to the fact that only the doubling as inputs in `ECADD2` are accounted, and not the point at infinity production (with opposite inputs) as outputs of `ECADD2` to the precomputed table.

Recommendation:

Fix the weak keys of `ECDSA_VERIFY_B4` (see the **Require** of Algorithm 17) and complete the weak keys of `ECDSA_VERIFY_RIP7212` (see the **Require** of Algorithm 16).

It is to be noted that the SCL library does not explicitly check that the public key Q is on the curve and that it is not one of the blacklisted weak keys. This check is expected to be performed “off-chain” for the sake of gas saving, or is sometimes considered to be unnecessary. This is for instance the case in the presence of “known inputs” as, e.g., for a WebAuthn/Passkeys use case where the public keys are usually generated, checked and stored in a trusted environment. This explains why Observation 5 (■) is rated as low risk, also mitigated by the presence of the code referenced in Remark 3 (Q) (although out of scope of the present audit).

Q Remark 3: Check of blacklisted weak keys

A routine `ecCheckPrecompute` has been implemented and is available outside the audited branch ([here](#) 🔗 🔍). This function is a helper to be called off-chain for checking for weak keys. It consists in performing the precomputation with a `revert` inside `ECADD2` whenever it is detected that input points are equal or opposite. This is done by checking if $ZZ' = 0$ in `ECADD2` ([here](#) 🔗 🔍), meaning that the two input points have similar x coordinates (hence are equal or opposite). This implementation is specific to `ECDSA_VERIFY_B4`, and a similar function should be implemented for `ECDSA_VERIFY_RIP7212`. The availability of this routines for the two variants of the DSM should mitigate Observation 5 (■).

As already discussed in Section 2.2.5 and shown in Algorithm 14 and Algorithm 15, the DSM implementations of the SCL library outputs only the x coordinate of the computed

point. This is performed because the ECDSA algorithm (for which DSM is used for in the context of the library) indeed only needs this x coordinate for signature verification. This x -only coordinate output usage in ECDSA, which is inherent to this signature algorithm, brings Observation 6 (■).

Finally, we consider the special case of a point at infinity $u \cdot G + v \cdot Q = \mathcal{O}$ output of the DSM called during the ECDSA algorithm: as explained in step 1 of Signature verification from Section 1.2.2, this specific case must reject the input signature. Whenever $u \cdot G + v \cdot Q = \mathcal{O}$, we have $O_x = 0$ in step 7 of Algorithm 16 and Algorithm 17, which leads to $\delta = -r \bmod n$. The check $\delta = 0$ that validates the signature can then only be true if and only if $r = 0 \bmod n$, which is not possible since $r > 0$ and $r < n$ from the checks at the beginning of the algorithm. Hence, the case of a DSM output of \mathcal{O} is naturally prevented by the $x = 0$ convention for the point at infinity affine representation.

■ Observation 6: ECDSA malleability

The output of DSM does not distinguish between opposite points on the curve since these two share the same x coordinate and opposite y coordinates. This is a known fact for ECDSA bringing malleability [12] of the signature, and the consequences of such a malleability depends on the context where the signature verification is used. For WebAuthn/Passkeys, this has no drastic consequences since signatures cannot be replayed. For a network where transactions can be replayed and where only the exact value of the signature is checked, this issue might be critical. Classical mitigations for this use a canonical encoding for the s component of the signature (e.g. ensure that $s < \frac{p}{2}$).

Recommendation:

Check and document possible ECDSA signature malleability in the various contexts of SCL usage.

Q Remark 4: ECDSA and composite order curves

This is out of scope of the audit as it concerns non-prime order curves (e.g. Wei25519 of order multiple of 8), but the current implementation of ECDSA verification does not check if the public key is in the prime subgroup of the curve (i.e. multiple of G) as expected in step 1 of Signature verification described in Section 1.2.2.

This check should probably be added to the weak keys and on-curve checks.

2.4 Gas cost

In this section, we will provide a brief overview of the gas costs of the two DSM implementations considered in the SCL library. Since the implementations make use of aggressive

Inlining of Yul assembly and variables allocation, we will have a high-level approach for the gas cost that is a theoretical approximation of the real cost (not accounting for some memory accesses, memory expansion, and function calls). Our estimation has been confirmed by benchmarks using the **forge** framework helpers for gas measurement.

First of all, we recall the cost of the basic EVM opcodes and precompiles that are used in the inline Yul assembly in SCL's Solidity code. As we can see on Table 1, modular addition **addmod** and multiplication **mulmod** on words of 256 bits take 8 gas units. Regular addition **add** and subtraction **sub** on 256 bits take 3 gas units, as well as explicit memory load **mload** and memory store **mstore**. Finally, the **ModExp** precompile on 256 bits presented in Section 2.1 takes $\approx 13,200$ gas units⁵, explained by the assembly exposed in Listing 2: this is a rough approximation supposing a warm access to the **ModExp** precompile (this yields a 100 gas units for **staticcall** instead of 2,600 in case of cold access), as well as a cost of 13,056 units for the precompile itself (see [2, 15]).

```

1  ...
2  //T[0] = inverseModp_Hard(T[0], p); //1/zzz, inline modular inversion
   using Memmpile:
3  // Define length of base, exponent and modulus. 0x20 == 32 bytes
4  mstore(T, 0x20)
5  mstore(add(T, 0x20), 0x20)
6  mstore(add(T, 0x40), 0x20)
7  // Define variables base, exponent and modulus
8  //mstore(add(pointer, 0x60), u)
9  mstore(add(T, 0x80), sub(_p,2))
10 mstore(add(T, 0xa0), _p)
11 // Call the precompiled contract 0x05 = ModExp
12 if iszero(staticcall(not(0), 0x05, T, 0xc0, T, 0x20)) { revert(0, 0) }
13 ...

```

Listing 2: Code of the inline assembly **staticcall** to the **ModExp** precompile in DSM, ensuring the **ModInv** function

```

1  function ecDbl(x, y, zz, zzz, _p,a) -> _x, _y, _zz, _zzz{
2      let T1 := mulmod(2, y, _p) //U = 2*Y1, y free
3      let T2 := mulmod(T1, T1, _p) // V=U^2
4      let T3 := mulmod(x, T2, _p) // S = X1*V
5      T1 := mulmod(T1, T2, _p) // W=UV
6      _y:= addmod(mulmod(3,
   mulmod(x,x,_p),_p),mulmod(a,mulmod(zz,zz,_p),_p),_p)//M=3*X12+aZZ12
7      _zzz := mulmod(T1, zzz, _p) // zzz3=W*zzz1
8      _zz := mulmod(T2, zz, _p) //zz3=V*ZZ1
9      _x := addmod(mulmod(_y, _y, _p), mulmod(sub(_p,2), T3, _p), _p)
   //X3=M^2-2S
10     T2 := mulmod(_y, addmod(_x, sub(_p, T3), _p), _p) //-M(S-X3)=M(X3-S)
11     _y := addmod(mulmod(T1, y, _p), T2, _p) //-Y3= W*Y1-M(S-X3), we
   replace Y by -Y to avoid a sub in ecAdd

```

⁵As a matter of fact, classical extended gcd algorithm for modular inversion, such as the implementation from OpenZeppelin [7], takes $\approx 20,000$ gas units: using **ModExp** precompile is cheaper.

EVM operations costs	
EVM opcode or precompile	Gas cost
addmod	8
mulmod	8
add	3
sub	3
mload	3
mstore	3
ModExp precompile	≈13,200

Table 1: Gas cost of EVM opcodes (see [20, 15])

```

12     _y := sub(_p, _y)
13 }
14 //normalized addition of two point, must not be neutral input
15 function ecAddn2(x1, y1, zz1, zzz1, x2, y2, _p) -> _x, _y, _zz, _zzz {
16     y1 := sub(_p, y1)
17     y2 := addmod(mulmod(y2, zzz1, _p), y1, _p)
18     x2 := addmod(mulmod(x2, zz1, _p), sub(_p, x1), _p)
19     _x := mulmod(x2, x2, _p) //PP = P^2
20     _y := mulmod(_x, x2, _p) //PPP = P*PP
21     _zz := mulmod(zz1, _x, _p) //ZZ3 = ZZ1*PP
22     _zzz := mulmod(zzz1, _y, _p) //ZZZ3 = ZZZ1*PPP
23     zz1 := mulmod(x1, _x, _p) //Q = X1*PP
24     _x := addmod(addmod(mulmod(y2, y2, _p), sub(_p, _y), _p),
25         mulmod(sub(_p, 2), zz1, _p), _p) //R^2-PPP-2*Q
26     x1:=mulmod(addmod(zz1, sub(_p, _x), _p), y2, _p)//necessary split
27     not to explode stack
28     _y := addmod(x1, mulmod(y1, _y, _p), _p) //R*(Q-X3)
29 }

```

Listing 3: Code of ECDBL and ECADD2 Yul functions used during the precomputation

```

1 let T1 := mulmod(2, Y, _p) //U = 2*Y1, y free
2 let T2 := mulmod(T1, T1, _p) // V=U^2
3 let T3 := mulmod(X, T2, _p) // S = X1*V
4 T1 := mulmod(T1, T2, _p) // W=UV
5 let T4:=mulmod(mload(add(Q,_a)),mulmod(ZZ,ZZ,_p),_p) // a * ZZ1**2
6 T4 := addmod(mulmod(3, mulmod(X,X,_p),_p),T4,_p)//M=3*X12+aZZ1**2
7 ZZZ := mulmod(T1, ZZZ, _p) //zzz3=W*zzz1
8 ZZ := mulmod(T2, ZZ, _p) //zz3=V*ZZ1
9 X:=sub(_p,2)//-2
10 X := addmod(mulmod(T4, T4, _p), mulmod(X, T3, _p), _p) //X3=M^2-2S
11 T2 := mulmod(T4, addmod(X, sub(_p, T3), _p), _p) //-M(S-X3)=M(X3-S)
12 Y := addmod(mulmod(T1, Y, _p), T2, _p) //-Y3= W*Y1-M(S-X3), -Y3=
13 //Y:=sub(p,Y) => this is commented for ecDblNeg, must be uncommented for
    ecDbl

```

Listing 4: Code of inlined assembly for ECDBLNEG and ECDBL used in the core loop

ECC operations gas costs in SCL								
Operation	#addmod	#mulmod	#add	#sub	#mload	#mstore	#Extra ⁶	Gas cost
ECDBL (function ⁷)	4	14	0	3	0	0	≈243	≈396
ECDBL (inline)	4	14	1	3	1	0	≈252	≈411
ECDBLNEG (inline)	4	14	1	2	1	0	≈243	≈400
ECADD2 (function ⁷)	6	11	1	5	0	0	≈258	≈412
ECNEGADD (inline)	6	18	14	4	10	2	≈450	≈732

Table 2: Cost of ECC doubling and addition operations in the SCL library

```

1 let T4:=mload(add(Mem,T1))//X2
2 mstore(add(Mem, _zzz2), mload(add(Mem,add(96,T1))))//ZZZ2
3 ...
4 mstore(add(Mem,_y2), addmod(mulmod( mload(add(Mem,add(32,T1))), ZZZ, _p),
    mulmod(Y,mload(add(Mem, _zzz2)), _p), _p))//R=S2-S1, sub avoided
5 T1:=mload(add(Mem,add(64,T1)))//zz2
6 let T2 := addmod(mulmod(T4, ZZ, _p), sub(_p, mulmod(X,T1,_p)), _p)//P=U2-U1
7 ...
8 T4 := mulmod(T2, T2, _p) //PP
9 T2 := mulmod(T4, T2, _p) //PPP
10 ZZ := mulmod(mulmod(ZZ, T4,_p), T1 ,_p)//zz3=zz1*zz2*PP
11 T1:= mulmod(X,T1, _p)
12 ZZZ := mulmod(mulmod(ZZZ, T2, _p), mload(add(Mem, _zzz2)),_p) //
    zzz3=zzz1*zzz2*PPP
13 X := addmod(addmod(mulmod(mload(add(Mem, _y2)), mload(add(Mem, _y2)), _p),
    sub(_p, T2), _p), mulmod( T1 ,mulmod(sub(_p,2), T4, _p),_p ), _p)//
    R2-PPP-2*U1*PP
14 T4 := mulmod(T1, T4, _p)//Q=U1*PP
15 Y := addmod(mulmod(addmod(T4, sub(_p, X), _p), mload(add(Mem, _y2)), _p),
    mulmod(mulmod(Y,mload(add(Mem, _zzz2)), _p), T2, _p), _p)//
    R*(Q-X3)-S1*PPP

```

Listing 5: Code of inlined assembly for ECNEGADD used in the core loop

Now that we have presented the cost of EVM opcodes, we present on Listing 3 the assembly code for ECDBL and ECADD2 used during the points precomputation step of Algorithm 14 and Algorithm 15, where these building blocks are used to implement the formulas presented in Section 2.2.2 and Section 2.2.3. These are regular functions called from Yul. The implementation of ECDBL, ECDBLNEG and ECNEGADD are used in the core loop of the DSM in the form of inline assembly as this allows for aggressive optimizations in gas consumption: these are presented in Listing 4 and Listing 5. We can see here many optimizations in action: how `sub` is used to compute opposites in the field \mathbb{F}_p , why the negation forms for doubling and additions allow to save some operations, etc. The sum-

⁶Approximate extra costs due to memory movements such as `push`, `pop`, etc. Opcodes with i inputs and j outputs account for $3(i+j)$ gas units (3 units for loading and 3 units for storing any value). This is a very rough approximation.

⁷Call cost and local variables memory access not included.

⁸Average value (the loop iteration performs an addition except with probability $1/2^{2ij} = 1/16$).

DSM gas costs in SCL								
Operation	#ECDBL function	#ECDBL inline	#ECDBLNEG inline	#ECADD2 function	#ECNEGADD inline	#ModExp staticcall	Gas cost Estimated	Gas cost Measured
SCL_DSM_B4	0	0	128	11	$\approx 120^8$	1	$\approx 156K$	$\approx 159K$
SCL_DSM_W	2	128	128	11	$\approx 120^8$	1	$\approx 208K$	$\approx 199K$

Table 3: DSM Yul implementations gas costs theoretical and measured estimates. This is a very rough estimation that sometimes overestimates and sometimes underestimates `mload` and `mstore` costs, memory expansion issues, function calls costs, looping, scalars parsing, possible compiler optimizations, etc. The estimated theoretical gas cost is nonetheless close to the one measured from `forge`.

many of the EVM opcodes usage and gas costs per assembly implementation is presented in Table 2, where extra gas costs⁶ are added to account for hidden memory operations (`push` and `pop` for high-level EVM opcode usage in Yul). Based on these figures and on the ECC doubling and addition usage of DSM, a rough theoretical estimation of gas costs for `SCL_DSM_B4` and `SCL_DSM_W` are provided in Table 3, and we can see that the estimations fit the measurements using the `forge` framework: we get respectively $\approx 159,000$ and $\approx 199,000$ gas units. Regarding the ECDSA verification functions, the gas costs mostly add up one `ModExp staticcall` for `ModInv` (and marginal `uint256` modular arithmetic opcodes), leading to a median cost of $\approx 162K$ when using `SCL_DSM_B4` for DSM, and $\approx 208K$ when using `SCL_DSM_W` as DSM.

Q Remark 5: Costs for ADDECC and DBLECC DSM

The costs provided in [18] are not completely exact, specifically for the ADDECC cost in `libSCL_rip7212.sol` (where the number of ADDECC is advertised as an average of 60 while it is 120). This is not considered as an Observation since [18] is not part of the audited repository and is hence out of the audit scope.

3 Detailed audit of SCL files

In the current section, we will review the main Solidity files of the SCL library in detail, presented by category: files related to fields and definitions, files related to modular arithmetic, files related to elliptic curves and DSM, files related to ECDSA verification, files related to API, and files related to tests.

3.1 The src/fields/ and src/include/ folders

These files mostly contain constants to bring abstractions over finite fields and curves.

3.1.1 SCL_mask.h.sol

Constants are defined as represented in Listing 6: 128-bit mask `_MASK128`, some OIDs for known curves P256 (`secp256r1`), Ed25519, `secp256k1` the “bitcoin curve”, etc.

```

1 // prime field modulus of the ed25519 curve
2 uint256 constant _MASK128 = 0xffffffffffffffffffffffffffff;
3 uint256 constant _HI_SCALAR=128;
4 uint256 constant _HIBIT_CURVE=255;
5
6 // prime field modulus of the secp256r1 curve
7 uint256 constant MODEXP_PRECOMPILE=0x05;
8
9 /* curves are identified by their OID */
10 uint256 constant _SECP256R1=0x06082A648CE3D030107;
11 uint256 constant _ED25519= 0x060a2b060104019755010501;
12 uint256 constant _SECP256K1=0x06052B8104000A;
13 uint256 constant _STARKCURVE=0x01;//OID doesn't exist for stark curve
14 uint256 constant _BABYJJ=0x02;
15 uint256 constant _ZERO_U256 =
    0x0000000000000000000000000000000000000000000000000000000000000000;
16
17 uint256 constant _UNUSED =
    0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff1;
18 //when a constant shall be defined
19 uint256 constant _TODO =
    0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff3;

```

Listing 6: Constant definitions inside `SCL_mask.h.sol`

● Observation 7: Unused and duplicated constants in `SCL_mask.h.sol`

Some constants are defined in `SCL_mask.h.sol` which are not used in the library. Some constants are used but not always. For instance, `MODEXP_PRECOMPILE = 0x05`:

- It is used in the `SCL_modular.sol` file:

...

```
if iszero(staticcall(not(0), MODEXP_PRECOMPILE, pointer, 0xc0,
    pointer, 0x20)) { revert(0, 0) }
...
```

- It is not used in the DSM files SCL_mulumuladdX_fullgen_b4.sol and SCL_mulumuladdX_fullgenW.sol:

```
...
// Call the precompiled contract 0x05 = ModExp
if iszero(staticcall(not(0), 0x05, T, 0xc0, T, 0x20)) { revert(0,
    0) }
...
```

Also, some constants are duplicated between SCL_mask.h.sol and the curves files. For instance, the ModExp precompile is defined in both SCL_mask.h.sol and src/fields/SCL_secp256r1.sol.

Recommendation:

Cleanup the constants and keep only the ones used in the library, or comment their future usage. Do not duplicate constants. Use the defined constants consistently in the library, and for the exceptions provide the possible reasons (optimization, etc.).

● Observation 8: Incorrect OIDs for P256 and Ed25519

The OIDs for P256 and Ed25519 defined in hexadecimal in SCL_mask.h.sol are wrong, they should be:

```
uint256 constant _SECP256R1 = 0x06082A8648CE3D030107 //
    1.2.840.10045.3.1.7
uint256 constant _ED25519 = 0x06092B06010401DA470F01 //
    1.3.6.1.4.1.11591.15.1
```

The OID for P256 is missing a hexadecimal digit, while the one for Ed25519 is obsolete (it is an old one 1.3.6.1.4.1.3029.1.5.1 used by OpenPGP, but this has been properly standardized since 2016, see [6, 5]).

Recommendation:

Fix the OIDs for P256 and Ed25519 in SCL_mask.h.sol.

3.1.2 SCL_field.h.sol

This file is an abstraction layer to import constants for the underlying field and curve. As we can see on Listing 7, an `import` of the necessary constants used in the upper layers is made from the proper file, `"@solidity/fields/SCL_secp256r1.sol"` in our case. Other curves are commented and only one must be uncommented to import the desired curve. It should be noticed that the files `fields/SCL_ecstark.sol` for the Stark curve and `fields/SCL_babyjubub.sol` for the pairing curve are not present in the audit branch of the repository, although they are referenced in the current header file.

```

1 //choose the field to import
2 import {deux_d, unscaling_factor, d, scaling_factor, p, pp1div4, a,b,gx,
  gy, gpow2p128_x, gpow2p128_y, n, pMINUS_2, nMINUS_2, MINUS_1,
  _HIBIT_CURVE, _MODEXP_PRECOMPILE, FIELD_OID } from
  "@solidity/fields/SCL_secp256r1.sol";
3 //import { p, gx, gy, gpow2p128_x, gpow2p128_y, n, pMINUS_2, nMINUS_2,
  MINUS_1, _HIBIT_CURVE, FIELD_OID } from
  "@solidity/fields/SCL_ed25519.sol";
4 //import { deux_d, p, pp1div4, a,b,gx, gy, gpow2p128_x, gpow2p128_y, n,
  pMINUS_2, nMINUS_2, MINUS_1, _HIBIT_CURVE, _MODEXP_PRECOMPILE,
  FIELD_OID } from "@solidity/fields/SCL_ecstark.sol";
5
6 //import {unscaling_factor, d, scaling_factor, FIELD_OID, MINUS_1, p,n,
  gx, gy, pMINUS_2, nMINUS_2, deux_d,a, _HIBIT_CURVE,b , gpow2p128_x,
  gpow2p128_y, _MODEXP_PRECOMPILE , pp1div4 } from
  "@solidity/fields/SCL_babyjubub.sol";

```

Listing 7: Abstraction layer for field and curve in SCL_field.h.sol

■ Observation 9: Misleading name for SCL_field.h.sol

From the name of the file, `SCL_field.h.sol`, we expect only field related constants to be defined. This is not the case as curve related constants are also defined in this file.

Recommendation:

Use a more general name for the file `SCL_field.h.sol` to account for curve related constants.

3.1.3 SCL_secp256r1.sol

This file contains the `uint256` constants for the P256 curve as defined in Equation 4 and Equation 5, notably: the prime p of the field, the elliptic curve coefficients a and b , its order n , and the affine coordinates of the generator $G = (G_x, G_y)$. In addition, the affine coordinates of $2^{128} \cdot G$ are also provided as constants. Other constants are present, such as

$p - 2$, $n - 2$, $-1 \bmod 2^{256}$, as well as the affine coordinates of multiples of G ($2 \cdot G$, $3 \cdot G$, \dots , $15 \cdot G$).

● Observation 10: Unused constants in SCL_secp256r1.sol

Many constants defined in `SCL_secp256r1.sol` are not used. The rationale behind their presence seems to be saving computation:

- $p - 2 = -2 \bmod p$ can be used in the ECC addition and doubling formulas to avoid using a `sub(p, 2)` opcode (see Listing 3, Listing 4 and Listing 5).
- The $i \cdot G$ with $i \in [2, 15]$ allow to save some point additions in the windowing point precomputation of `DSMi,j`.

However, since the APIs for `SCL_DSM_B4` and `SCL_DSM_W` expect **generic input curve and points**, these precomputed constants cannot be used without loss of generality.

Recommendation:

Clarify the usage of precompiled constants in `SCL_secp256r1.sol`, and remove the ones that are not used in the library.

3.1.4 SCL_wei25519.sol

As for P256 constants described in Section 3.1.3, the `SCL_wei25519.sol` file contains `uint256` constants for the prime p of the Wei25519 curve, $p - 1 = -1 \bmod p$, $p - 2 = -2 \bmod p$, a , b , etc. The base point $G = (G_x, G_y)$ and its power $2^{128} \cdot G$ are also defined. Finally, other constants are present such as A , the Montgomery curve coefficient. These seem to be useful for the isogenies formulas between Ed25519 and Wei25519 (see [24]). Since Wei25519 is out of scope and not used in the audited branch of the SCL library, we will not further analyze this file.

3.2 The src/modular/SCL_modular.sol file

In this section, we analyze the `SCL_modular.sol` file. The main purpose of this file is to provide various functions to implement `ModExp` and `ModInv` with optimizations. First of all, `ModExp` is implemented in its most generic form with `MODEXP_PRECOMPILE` as:

```
1 function ModExp(uint256 g, uint256 e, uint256 modulus)
```

Then, `ModInv` is implemented using `MODEXP_PRECOMPILE` in a generic way using Fermat's little theorem for inversion modulo a prime:

```
1 function ModInv(uint256 u, uint256 m) view returns (uint256 result)
```

Then, two optimized variants are implemented: `nModInv` and `pModInv` which hardcode the order n of the curve and the prime modulus is p respectively. These variants save a subtraction operation by using the constants `nMINUS_2` and `pMINUS_2` taken from the curve header (see Section 3.1.2). However, as reported in Observation 10 (●) although the `nModInv` function is called in the ECDSA layer, the `pModInv` does not seem to be used in the SCL library.

3.3 The src/elliptic/ folder

These files concern the ECC layer, where the DSM formulas are implemented, as well as a helper to check if points are on the curve.

3.3.1 The SCL_ecOncurve.sol file

This file contains one main function, `ec_isOnCurve`, whose code is shown on Listing 8. After some checks on the coordinates of the input point $qx = Q_x$ and $qy = Q_y$, the left hand side of the Weierstraß Equation 1 is computed with $LHS = Q_y^2 \bmod p$, and the left hand side with $RHS = Q_x^3 + a \times Q_x + b \bmod p$, and then checks if $LHS = RHS$.

```
1  /// @notice Check the validity of a Public keys
2  /// @param qx The x value of the public key Q used for the signature
3  /// @param qy The y value of the public key Q used for the signature
4  /// @dev Note The public key is assumed to belong to the curve and not
   neutral, additional weak keys are rejected
5
6  function ec_isOnCurve(uint256 p, uint256 a, uint256 b, uint256 qx, uint256
   qy)
7  pure returns (bool)
8  {
9      // check the validity of the range related to prime field characteristic
10     if (qx == 0 || qx >= p || qy == 0 || qy >= p) {
11         return false;
12     }
13     // check the curve equation
14     uint256 LHS = mulmod(qy, qy, p); // y^2
15     uint256 RHS = addmod(mulmod(mulmod(qx, qx, p), qx, p), mulmod(qx,
   a, p), p); // x^3+ax
16     RHS = addmod(RHS, b, p); // x^3 + a*x + b
17
18     return LHS == RHS;
19 }
```

Listing 8: The `ec_isOnCurve` function

■ Observation 11: Incorrect comment for `ec_isOnCurve`

The comment for the function `ec_isOnCurve` is incorrect:

- The parameters p , a and b are missing.
- The comment “additional weak keys are rejected” is wrong: weak keys are not rejected. This could be done by calling the routine `ecCheckPrecompute` (see Remark 3 (Q)).

Recommendation:

Fix comment.

● Observation 12: Incorrect checks in `ec_isOnCurve`

The following check is incorrect:

```
// check the validity of the range related to prime field
characteristic
if (qx == 0 || qx >= p || qy == 0 || qy >= p) {
    return false;
}
```

Indeed, the checks `qx == 0` and `qy == 0` can filter some legitimate points on the curve in the general Weierstraß case, and in the specific P256 case (see Observation 1 (■) and Observation 3 (■)).

Recommendation:

The check should only filter the point at infinity, represented as $(0,0)$, (on short Weierstraß curves with $b \neq 0$), by patching the test with the following:

```
// check the validity of the range related to prime field
characteristic
if (qx >= p || qy >= p) {
    return false;
}
// Check for point at infinity
if (qx == 0 && qy == 0) {
    return false;
}
```

Also, the point at infinity is by definition on the curve, even though it should be filtered in this function for specific reasons related to its usage in SCL (weak keys,

etc.). This is not explicit given the function's name, `ec_isOnCurve`, and should be clarified by a specific comment (or changing the name).

3.3.2 The `SCL_mulumuladdX_{fullgen_b4/fullgenW}.sol` files

These two files implement respectively Algorithm 14 and Algorithm 15 with the addition and doubling formulas described in Section 2.4. The main part of these files is Yul inline assembly, with many optimizations that make these functions challenging to read. Since the algorithms have been previously detailed and commented, we will not provide further details on the code here. The main element to keep in mind is that the assembly uses a constant and deterministic memory layout since the inputs are of fixed size. This means that the code does not allow for an attacker to play with variable-length inputs involving dangerous parsing or so. This static layout of the algorithms discards many attack vectors.

In this section, we will hence mostly focus on the algorithmic level attacks which injects inputs producing errors in the intermediate results of the DSM. These attacks take advantage of the various control flow branches in the algorithm to produce, e.g., the point at infinity where not expected, forbidden inputs to the addition formulas, etc. Here is the list of the possible attack vectors we have identified and how they are prevented in the library:

- **Attacks on the DSM precomputation step:** These attacks try to inject faults in the `ECADD2` and `ECDBL` formulas involved in the point precomputation. To do so, an attacker will aim at having the same point or the point at infinity in `ECADD2` since the formula does not accept them, or try to produce a point at infinity as output so that the precomputed table contains \mathcal{O} other than T_0 . Regarding `ECDBL`, the formula is resilient to all inputs as it accepts the point at infinity. The 15 constraints for `SCL_DSM_B4` and the 9 constraints for `SCL_DSM_W` on the two input points (derived to 14 and 9 weak public keys when the first point is G the generator of P256) cover all the possible bad points injections, mitigating this class of attacks on the precomputation step.
- **Attacks on the DSM core loop:** These attacks try to inject faults in the core loop of the DSM. Let us go through the different control flow branches of Algorithm 14 and Algorithm 15, and check that the marginal cases are properly addressed.
 - The `ECDBLNEG` call at step 30 of Algorithm 14 accepts non-normalized points and \mathcal{O} as input. For `SCL_DSM_W`: the `ECDBL` call at step 30 and `ECDBLNEG` call at step 31 in Algorithm 15 accept non-normalized points and \mathcal{O} as inputs. No possible input can make these steps fail.

In the following, the analysis focuses on the `SCL_DSM_B4` variant, namely Algorithm 14. The exact same reasoning holds for `SCL_DSM_W` (Algorithm 15).

- The output of the doubling negation provides the new R . In step 32, the extracted scalar bits are checked to be 0: if this is the case R is negated and the loop continues. This detects \mathcal{O} as precomputed input, and since $R + \mathcal{O} = R$ nothing is done: this also prevents putting a \mathcal{O} in the addition formulas.
 - In step 35, there is a check to test if $R = \mathcal{O}$: in this case we also avoid using the point at infinity in the addition formula by setting $R = T_s$ (where $s \neq 0$ because of the previous test). The implementation of $R = T_s$ is **not sound** in `SCL_mulumuladdX_fullgen_b4.sol` and `SCL_mulumuladdX_fullgenW.sol` as explained in Observation 13 (●).
 - At this point, we are sure that $R \neq \mathcal{O}$ and $T_s \neq \mathcal{O}$. In condition in step 38 holds, this means that $R = T_s$ and we must use doubling instead of addition formula: this explains the usage of `ECNEGDBL`.
 - Finally, the nominal case where $R \neq \mathcal{O}$, $T_s \neq \mathcal{O}$ and $R \neq T_s$ is executed in step 41, using a `ECNEGADD` (which hence can be used with the insurance of no bad input points).
- **Attacks on the DSM final normalization step:** attacks exploiting this step could involve marginal cases of the modular inversion with `ModInv`. This mostly concerns the case where $R_{ZZ} = 0$ (i.e. $R = \mathcal{O}$ at the end of the DSM core loop): in this case, $R_{ZZ}^{-1} = 0$ and the output is $x = 0$. This specific case can bring some confusion as explained in Observation 3 (■), but this does not bring critical issues.

Summary: to the extent of our analysis, except for the case described in Observation 13 (●), all the marginal cases should be covered by the two DSM implementations of the SCL library.

● Observation 13: Possible error injection in the DSM algorithms

The step 36 in Algorithm 14, and the equivalent step 37 in Algorithm 15, are implemented with the following in the `SCL_mulumuladdX_fullgen_b4.sol` and `SCL_mulumuladdX_fullgenW.sol` files:

```
...
if iszero(ZZ) {
    X := T4//X2
    Y := mload(add(Mem,add(32,T1)))//Y2
    ZZ := 1
    ZZZ := 1
    continue
}
...
```

The main issue here is that T_s is supposed to be normalized so that $ZZ := 1$ and $ZZZ := 1$. While this is the case for some precomputed points, this is not the case

for the ones that are outputs of `ECADD2` or `ECDBL` formulas where usually $ZZ \neq 1$ and $ZZZ \neq 1$. This means that an attacker can trigger a bad point injection that will produce an undetected bad computation for the DSM. The strategy is to reach this marginal case with a value of T_s that is not normalized. There are many ways of performing this, we provide a simple one for `SCL_mulumuladdX_fullgen_b4.sol` (a very similar example can be adapted for `SCL_mulumuladdX_fullgenW.sol`): we take input points $P = G$, $Q = -2 \cdot G$, and the following scalars:

$u = 0x20000000000000000000000000000000a000000000000000000000000000000$,
 $v = 0x60000000000000000000000000000000$.

These scalars can be split in bits as: $u = (1 \ll 253) \mid (1 \ll 127) \mid (1 \ll 125)$ and $v = (1 \ll 126) \mid (1 \ll 125)$. Some of the precomputations (that are of interest for us) are the following: $T_0 = \mathcal{O}$, $T_1 = G$, $T_4 = Q = -2 \cdot G$, $T_7 = 2^{128} \cdot G - 2 \cdot G + G = (2^{128} - 1) \cdot G$ (T_0 and T_1 are normalized while T_7 is not).

Here is the unrolling of the `SCL_DSM_B4` DSM core loop:

- The first scalars bits MSB from step 21 provide $s = 1$, hence R is initialized to $R = T_1 = G$.
- In step 30, $R = -2 \cdot R = -2 \cdot G$ is computed.
- Then the scalars bits extracted at step 31 provide $s = 4$, and we go to step 41 as no other condition is satisfied. This computes $R = -R + T_4 = 2 \cdot G - 2 \cdot G = \mathcal{O}$, and the point at infinity is reached.
- Entering the second iteration of the loop, $R = -2 \cdot R = \mathcal{O}$ is computed at step 30.
- Then the scalars bits extracted at step 31 provide $s = 7$, and we go to step 36 since $R = \mathcal{O}$: this computes $R = T_7$, **which fails with the current implementation** since T_7 is not normalized.

Using Observation 4 (■), the current Observation on DSM can immediately be translated to forging false negative ECDSA signatures: from scalars u , v and relations between P and Q that make DSM fail, it is possible to create a legitimate r , s signature with a public key leading to a failing ECDSA verification. The previous exhibited values of u , v and Q lead to the following failing raw ECDSA verification:

$r = 0x34f87673c7484c8e8886a54dad431b330e1cad445d32013423fce765d497f87a$,
 $s = 0x8f2280ee8a32f1f813d72a377ef41072acc943e78a26ed4a26e295d4969c9b56$,
 $m = 0x47492e075b24d4cfc7f82a6bb90decdb09311928f2e05badf165d4316756d917$,
 $Q_x = 0x7cf27b188d034f7e8a52380304b51ac3c08969e277f21b35a60b48fc47669978$,
 $Q_y = 0xf888aaee24712fc0d6c26539608bcf244582521ac3167dd661fb4862dd878c2e$.

It is to be noted that $Q = -2 \cdot G$ is a weak key for SCL_DSM_W but it is not for SCL_DSM_B4 (see Algorithm 16 and Algorithm 17). This means that the previous vector will make fail both DSM formulas: SCL_DSM_W for a genuine reason (weak key), and SCL_DSM_B4 because of the currently addressed issue. The corresponding Whycheproof json formatted test vector for ECDSA, see Section 3.5, is the following (the message msg is empty because of raw ECDSA):

```
{
  "x": "7cf27b188d034f7e8a52380304b51ac3c08969e277f21b35a60b48fc47669978",
  "y": "f888aaee24712fc0d6c26539608bcf244582521ac3167dd661fb4862dd878c2e",
  "r": "34f87673c7484c8e8886a54dad431b330e1cad445d32013423fce765d497f87a",
  "s": "8f2280ee8a32f1f813d72a377ef41072acc943e78a26ed4a26e295d4969c9b56",
  "hash": "47492e075b24d4cfc7f82a6bb90decdb09311928f2e05badf165d4316756d917",
  "valid": true,
  "msg": "",
  "comment": "DSM B4 failure in special case"
}
```

The current Observation is rated as ● since two elements limit its scope when considering ECDSA:

- False negative ECDSA signatures are not considered as critical in the current SCL ECDSA usage context (contrary to e.g. forging true positive signatures without knowing the private key)
- Observation 4 (■) supposes raw ECDSA, the usage of a hash function prevents the forgery.

However, this unsoundness of the DSM formula could lead to more serious issues in other contexts (hence this rather high rating despite the previous two points about ECDSA).

Recommendation:

Fix the affectation in step 36 for SCL_DSM_B4 (and equivalent for SCL_DSM_W) with the proper ZZ and ZZZ precomputed values. This will add more instructions for this specific case, but should not degrade the average gas consumption of the DSM as this marginal execution should not happen for regular inputs with overwhelming probability.

● Observation 14: Suboptimal doubling in DSM precomputation

Algorithm 15 implemented in SCL_mulumuladdX_fullgenW.sol makes use of ECDBL during the precomputations, but only normalized points are used as inputs. Using ECDBL2 from Algorithm 6 allows to save some operations.

Recommendation:

Use `ECDBL2` instead of `ECDBL` in `SCL_mulumuladdX_fullgenW.sol`.

Q Remark 6: Potential suboptimal addition in DSM precomputation

Algorithm 14 implemented in `SCL_mulumuladdX_fullgen_B4.sol` uses `ECADD2` during the precomputations, but some additions (steps 8, 10, 11, 14, 15, 17) use a normalized `XYZZ` point as input. Using `ECADD3` from Algorithm 11 allows to save some operations in these steps (at the expense of using a new function, hence a larger footprint and a somehow potential limited benefit in terms of gas).

Since it is not clear if using `ECADD3` has positive impacts on gas, this is not an Observation.

● Observation 15: Collisions of constants in DSM files

Many constants share the same names in `SCL_mulumuladdX_fullgen_B4.sol` and `SCL_mulumuladdX_fullgenW.sol`, preventing from importing them both to use the two DSM implementations in the same Solidity file.

Recommendation:

Rename the constants in one of the files to avoid name collisions and allow to use both DSM formulas jointly.

● Observation 16: Incomplete / bad code for `SCL_mulumuladdX_fullgenW.sol`

The return value of the `ecGenMulmuladdB4W` function (corresponding to algorithm `SCL_DSM_W`) is a pair of coordinates (x, y) :

```
function ecGenMulmuladdB4W(  
    uint256 [6] memory Q, //store Qx, Qy, p, a, gx, gy,  
    uint256 scalar_u,  
    uint256 scalar_v  
) view returns (uint256 X, uint Y)  
...  
    X := mulmod(X, mload(T), _p) //X/zz  
    Y:=0 //todo  
} //end assembly  
}
```

This does not correspond to the x -only spirit of the API as used in the ECDSA verification. Moreover, the code for computing y is **missing** (forced to 0 for now). It is not clear why this y coordinate is needed (which might be related to the RIP-7212 API).

Also, there seems to be a typo in the return type of Y in the “view returns (uint256 X, uint Y)” statement: it should be a `uint256` (although in Solidity `uint` is an alias for `uint256`, it should be explicit as for X here).

Recommendation:

Remove the y coordinate from the return value and the code associated to it, or properly complete the code to return (x, y) . Document this choice. If needed, fix the type of this output.

● Observation 17: DSM function names

The name of the function implementing the 4 base points Shamir `SCL_DSM_B4` is `ecGenMulmuladdX_store`, which is not very explicit. The name of the function implementing the 4-bit windowing `SCL_DSM_W` is `ecGenMulmuladdB4W` which is confusing as the `B4` suffix seems to indicate 4 base points.

Recommendation:

Rename the functions with more explicit and appropriate names.

■ Observation 18: Incorrect inline comments in `ecGenMulmuladdB4W`

Some inline comments describing the inner loop of the `ecGenMulmuladdB4W` function (in `SCL_mulmuladdX_fullgenW.sol` file) are wrong.

First of all, the following comment “`//inline DblNeg`” is not exact as a regular `ECDBL` is implemented (and not `ECDBLNEG` as commented). This corresponds to step 30 of Algorithm 15.

```
...
{ //inline DblNeg
  //X,Y,ZZ,ZZZ:=ecDblNeg(X,Y,ZZ,ZZZ), not having it inplace increase
    by 12K the cost of the function

  let T1 := mulmod(2, Y, _p) //U = 2*Y1, y free
  let T2 := mulmod(T1, T1, _p) // V=U^2
  let T3 := mulmod(X, T2, _p) // S = X1*V
  T1 := mulmod(T1, T2, _p) // W=UV
  let T4:=mulmod(mload(add(Q,_a_)),mulmod(ZZ,ZZ,_p),_p)//aZZ1^2
```

```

T4 := addmod(mulmod(3, mulmod(X,X,_p),_p),T4,_p)//M=3*X12+aZZ12
ZZZ := mulmod(T1, ZZZ, _p) //zzz3=W*zzz1
ZZ := mulmod(T2, ZZ, _p) //zz3=V*ZZ1
X:=sub(_p,2)//-2
X := addmod(mulmod(T4, T4, _p), mulmod(X, T3, _p), _p) //X3=M^2-2S
T2 := mulmod(T4, addmod(X, sub(_p, T3), _p), _p) //-M(S-X3)=M(X3-S)
Y := addmod(mulmod(T1, Y, _p), T2, _p) //-Y3= W*Y1-M(S-X3), we
      replace Y by -Y to avoid a sub in ecAdd
//Y:=sub(p,Y)
...

```

Secondly, later in the same core loop the following comment corresponding to step 41 of Algorithm 15 is also incorrect as it should be a **ECNEGDBL** (and not a **ECDBL** as commented):

```

//special case ecAdd(P,P)=Ecdbl
if iszero(mload(add(Mem,_y2_))) {
    if iszero(T2) {
        T1 := mulmod(sub(_p,2), Y, _p) //U = 2*Y1, y free
        T2 := mulmod(T1, T1, _p) // V=U^2
    }
    ...
}

```

Recommendation:

Fix the comments in the core loop of **ecGenMulmuladdB4W**.

■ Observation 19: Incorrect or incomplete comments in the DSM files

This observation applies to both DSM files, **SCL_mulmuladdX_fullgen_B4.sol** and **SCL_mulmuladdX_fullgenW.sol**

The comments for the **ecGenMulmuladdX_store** function are the following:

```

//this function is for use only after validation of the Q input:
//Q shall belongs to the curve, and different from -P, -P128,
  -(P+P128), ...
//those 16 values are tested by the ValidateKey function
//due to handling of Neutral element, this function will not work for
  16 specific weak keys
//those value are excluded from the
function ecGenMulmuladdX_store(
    uint256 [10] memory Q, //store Qx, Qy, Q'x, Q'y p, a, gx, gy,
    gx2pow128, gy2pow128
    uint256 scalar_u,
    uint256 scalar_v
) view returns (uint256 X) {
    ...
}

```

This comment is not complete (unfinished last sentence).

The comments for the `ecGenMulmuladdB4W` function are the following:

```
/* It is a custom 4 dimensional version of Shamir's trick (tis not a
   window)*/
/* (gen= any curve, sw=short weierstrass) */
/* b4=Four dimensional multiexponentiation */
...
//this function is for use only after validation of the Q input:
//Q shall belongs to the curve, and different from -P, -P128,
   -(P+P128), ...
//those 16 values are tested by the ValidateKey function
//due to handling of Neutral element, this function will not work for
   16 specific weak keys
//those value are excluded from the
function ecGenMulmuladdB4W(
    uint256 [6] memory Q, //store Qx, Qy,  p, a, gx, gy,
    uint256 scalar_u,
    uint256 scalar_v
) view returns (uint256 X, uint Y) {
}
...
```

The comment describing the algorithm “It is a custom 4 dimensional version of Shamir’s trick (tis not a window)” is wrong. The comment of the function is incomplete (last sentence unfinished), and the description of the weak keys is wrong (a copy paste of the one from `ecGenMulmuladdX_store`).

Recommendation:

Fix the comments.

3.4 The `src/lib/` folder

The two files `libSCL_ecdsab4.sol` and `libSCL_rip7212.sol` implement the ECDSA verification functions, each one respectively using `SCL_DSM_B4` and `SCL_DSM_W` variants of the DSM. The `verify` function is the main exported function of the SCL library as shown on Listing 9. The inputs depend on the variant: for `SCL_DSM_W` it fits the RIP-7212 API with as inputs the message, r , s , Q_x , and Q_y (the P256 curve parameters are implicit from the included headers). The RIP-7696 proposal API is more elaborate as it takes in addition the curve parameters (allowing flexibility for the curve choice) as well as the precomputations for $2^{128} \cdot Q$ and $2^{128} \cdot G$. The implemented algorithms are almost a straightforward Solidity translation of Algorithm 17 and Algorithm 16.

```
1 //the name of the library will be modified to fit RIP number
2 library SCL_ECDSAB4{
3
```



```

4      /// @notice Verifies an ECDSA signature on the secp256r1 curve given
      the message, signature, curve parameters and extended public key.
5
6      /// @param message The original message that was signed
7      /// @param r uint256 The r value of the ECDSA signature.
8      /// @param s uint256 The s value of the ECDSA signature.
9      /// @param Qpa [qx, qy, q2p128_x, q2p128_y, p, a, gx, gy, gpow2p128_x,
      gpow2p128_y] where
10     /// qx The x value of the public key Q used for the signature where
11     /// qy The y value of the public key Q used for the signature
12     /// q2p128_x The x value of precomputed 2**128.Q
13     /// q2p128_y The x value of precomputed 2**128.Q
14     /// @param n The order of the curve
15     /// @return bool True if the signature is valid, false otherwise
16     /// @dev Note The public key is assumed to belong to the curve and not
      neutral, additional weak keys are rejected as described in
      ecdsa_checkpub
17
18 function verify(bytes32 message, uint256 r, uint256 s, uint256[10] memory
      Qpa, uint256 n) public
19 view returns (bool)
20 {
21
22 ...
23 library SCL_RIP7212{
24
25 function verify(bytes32 message, uint256 r, uint256 s, uint256 qx, uint256
      qy) public view returns (bool) {
26 ...

```

Listing 9: The top level entry points of the SCL library: ECDSA verification function

■ Observation 20: Typo in comment and missing comment ECDSA verification files

In libSCL_ecdsab4.sol, the comment “q2p128_y The x value of precomputed 2**128.Q” contains a typo.

In libSCL_rip7212.sol, the verify routine does not have a comment providing the API.

Recommendation:

Fix the typo and add the missing comments.

■ Observation 21: Missing rationale about requirements for the inputs

In the verify of the SCL_ECDSAB4 export, we suppose the following:

```
/// @dev Note The public key is assumed to belong to the curve and
    not neutral, additional weak keys are rejected as described in
    ecdsa_checkpub
```

This explains why no `ec_isOnCurve` is called. However, the `verify` in `SCL_RIP7212` uses such a `ec_isOnCurve` to check the public key Q , but as discussed in Observation 11 (■) it is missing the weak keys check.

All-in-all, the difference between these two ECDSA verification APIs regarding the inputs requirements and the inner checks misses explanations and rationale.

Recommendation:

Add explanations and documentation about the inputs requirements and inner checks of inputs for the two `verify` APIs.

● Observation 22: Compilation issues: README.md, case sensitivity for `libSCL_RIP7212.sol` filename

There are some compilations issues related to:

- The compilation directives in the `README.md` file tell to “*Clone the repository and forge test*”. Actually, the forge environment with its standard library must first be created, which can be done with `'forge init --force'`: this should be added.
- The file `libSCL_RIP7212.sol` should use lowercase on case sensitive file systems (such as ext4 on Linux) for the RIP7212 part, which prevents a proper `forge` compilation with an error.

Recommendation:

Rename `libSCL_RIP7212.sol` to `libSCL_rip7212.sol`, fix the compilation directives in `README.md`.

3.5 The tests from the `test/` folder

The `test` folder contains 3 solidity files that use the `forge` testing framework:

- The two files `libSCL_ecdsaP256_B4.t.sol` and `libSCL_ecdsa.t.sol` contain tests for the `SCL_ECDSAB4` verification API: the `test_secp256r1` function performs 1000 tests on the same hardcoded test vector, and the `test_ecdsaB4_wycheproof` function

reads the Wycheproof test vectors from the external `json` file while using a Solidity helper function `ecPow128` (that implements `ECDBL` in a loop) to compute $2^{128} \cdot Q$.

● Observation 23: Duplicate test files

The two test files `libSCL_ecdsaP256_B4.t.sol` and `libSCL_ecdsa.t.sol` seem to be duplicates of the same tests for `SCL_ECDSAB4` verification API.

Recommendation:

Keep only one test file.

- The file `libSCL_rip7212.t.sol` contains tests for the `SCL_RIP7212` verification API: the `test_secp256r1` function performs 1000 tests on the same hardcoded test vector, and the `test_rip7212_wycheproof` function reads the Wycheproof test vectors from the external `json` file.

The `json` file `vectors_wycheproof.jsonl` contains the Wycheproof test vectors formatted with the following information: the values Q_x and Q_y , the signature r and s , the hash value (which is in fact the raw input of the SCL ECDSA verification API), the message whose SHA-256 is the hash value, the validity field providing the `true` or `false` outcome of the signature verification, and finally a comment field containing a free text explaining the rationale of the test. An example of test vector is provided in Listing 10. It is to be noted that these tests are a reformatting of [11] (the same repository contains various `json` files for other curves than P256, and other hash functions which might not be compatible with SCL input message size of 256 bits).

```

1 {
2   "x": "2927b10512bae3eddcfe467828128bad2903269919f7086069c8c4df6c732838",
3   "y": "c7787964eaac00e5921fb1498a60f4606766b3d9685001558d1a974e7341513e",
4   "r": "2ba3a8be6b94d5ec80a6d9d1190a436effe50d85a1eee859b8cc6af9bd5c2e18",
5   "s": "4cd60b855d442f5b3c7b11eb6c4e0ae7525fe710fab9aa7c77a67f79e6fadd76",
6   "hash": "bb5a52f42f9c9261ed4361f59422a1e30036e7c32b270c8807a419feca605023",
7   "valid": true,
8   "msg": "313233343030",
9   "comment": "wycheproof/ecdsa_secp256r1_sha256_p1363_test.json
10  EcdsaP1363Verify SHA-256 #1: signature malleability"
11 }
12 ...

```

Listing 10: Example of test in the `vectors_wycheproof.jsonl` file

4 CRYPTOEXPERTS tests on SCL

In this section we provide an overview of the testing approach adopted for SCL. Inspired from the tests presented in Section 3.5, we have used the same **forge** framework to implement our testing scripts. The main advantage of **forge** is that:

- Tests can be written in Solidity.
- The Solidity code can interact with external files, which is how the Wycheproof json file is parsed. As we can see on Listing 11, the `test_rip7212_wycheproof` reads the external file with the dedicated `vm.readLine` opcode implemented in the **forge** EVM instance, and then the **forge** standard library is used for the Json parsing of the fields. Finally, once the elements have been formatted in memory, the SCL library API `SCL_RIP7212.verify` can be called.

```

1  ...
2  function test_rip7212_wycheproof() public view{
3  ...
4      string memory file = "./test/vectors_wycheproof.jsonl";
5      while (true) {
6
7          string memory vector = vm.readLine(file);
8          if (bytes(vector).length == 0) {
9              break;
10         }
11         cpt=cpt+1;
12         // console.log("\n -----s",vector);//display all wycheproof
           vectors
13
14         uint256 x = uint256(stdJson.readBytes32(vector, ".x"));
15         uint256 y = uint256(stdJson.readBytes32(vector, ".y"));
16         uint256 r = uint256(stdJson.readBytes32(vector, ".r"));
17         uint256 s = uint256(stdJson.readBytes32(vector, ".s"));
18         bytes32 hash = stdJson.readBytes32(vector, ".hash");
19         bool expected = stdJson.readBool(vector, ".valid");
20         string memory comment = stdJson.readString(vector, ".comment");
21
22         bool result = SCL_RIP7212.verify(hash, r, s, x, y);
23     ...

```

Listing 11: Reading an external file with the **forge** testing framework

In order to make tests interaction easier, we have exploited another feature of **forge**: `ffi` [4] allows to call any external arbitrary command. The communication between the Solidity test function and the external command uses the EVM ABI encoding rules. In the context of SCL, we have developed two Python scripts based on an internal ECC computation engine:

- A first script `gen_ecmuladd.py` to generate test vectors for the two DSM variants implemented in the SCL library. This script produces an EVM ABI encoded hexadecimal string containing the public key Q coordinates, $2^{128} \cdot Q$ coordinates, the

scalars u and v , and the expected result of $u \cdot G + v \cdot Q$ on the P256 curve. We present on Listing 13 the Solidity test function interacting in `ffi` with the Python script: we use the `console.log` helper to log and print the exchanges, and `assert` the results sent by the script. We have patched the DSM files as recommended in Observation 15 (●) to deal with the constants names collisions and test both `ecGenMulmuladdX_store` and `ecGenMulmuladdB4W` in the same function. The console output result of a passing test is provided on Figure 2, while a failing test output for `ecGenMulmuladdX_store` is provided on Figure 4 (corresponding to the test case exhibited in Observation 13 (●)). By adding the json ECDSA test exhibited in Listing 12 to the `vectors_wycheproof.jsonl` Wycheproof tests file, we can observe the failure in the original test framework as shown on Figure 3. Beware that this test vector will make fail both ECDSA implementations tests from `test_rip7212_wycheproof` and `test_ecdsaB4_wycheproof`: the first failure is expected because the public key $Q = -2 \cdot G$ is one of the weak keys (see Algorithm 16), but the second failure should not happen as it exploits the uncovered `SCL_DSM_B4` issue.

```

1 {
2   "x": "7cf27b188d034f7e8a52380304b51ac3c08969e277f21b35a60b48fc47669978",
3   "y": "f888aaee24712fc0d6c26539608bcf244582521ac3167dd661fb4862dd878c2e",
4   "r": "34f87673c7484c8e8886a54dad431b330e1cad445d32013423fce765d497f87a",
5   "s": "8f2280ee8a32f1f813d72a377ef41072acc943e78a26ed4a26e295d4969c9b56",
6   "hash": "47492e075b24d4cfc7f82a6bb90decdb09311928f2e05badf165d4316756d9
7   17",
8   "valid": true,
9   "msg": "",
10  "comment": "DSM B4 failure in special case"
11 }

```

Listing 12: Test case for failing ECDSA verification exploiting Observation 13 (●)

- A second script `gen_sign.py` to generate test vectors for the ECDSA verification algorithms: this script generates a public key Q , a message h , the signature r and s (with the associated private key), and the expected result of the signature (`true` or `false` when an erroneous signature is generated). The Solidity code interfacing with the Python script is provided in Listing 14.

Another advantage of `forge` is the gas cost estimation: the figures provided in Section 2.4 have been gathered using the framework measurement abilities.

```

1 // Test scalar formulas
2 function test_ecmuladd() public {
3   console.log("[+] Testing the ecmuladd");
4
5   uint256[10] memory Qpa = [0, 0, 0, 0, p, a, gx, gy, gpow2p128_x,
6     gpow2p128_y];
7   // Prepare the external command to get the values
8   string[] memory cmds = new string[](1);
9   cmds[0] = "./gen_ecmuladd.py";

```

```

10 bytes memory cmd_result = vm.ffi(cmds);
11
12 (uint256[2] memory Q, uint256[2] memory Q128, uint256 u, uint256 v,
    uint256[2] memory Res) = abi.decode(cmd_result, (uint256[2],
    uint256[2], uint256, uint256, uint256[2]));
13 console.log("=====");
14 console.log("Q      : (%x, %x)", Q[0], Q[1]);
15 console.log("Q128   : (%x, %x)", Q128[0], Q128[1]);
16 console.log("u       : %x", u);
17 console.log("v       : %x", v);
18 console.log("Res    : (%x, %x)", Res[0], Res[1]);
19
20 (Qpa[0], Qpa[1]) = (Q[0], Q[1]);
21
22 //(Qpa[2], Qpa[3]) = (Q128[0], Q128[1]);
23
24 (Qpa[2], Qpa[3]) = ecPow128(Q[0], Q[1], 1, 1); //compute Q^128
25 console.log("=====");
26 console.log("Q128_ : (%x, %x)", Qpa[2], Qpa[3]);
27
28 // Check the ecPow128 function
29 assertTrue(Q128[0] == Qpa[2], "[-] ecPow128 x bad");
30 assertTrue(Q128[1] == Qpa[3], "[-] ecPow128 y bad");
31
32 // Check the ecmuladd versions
33 uint256 x1 = ecGenMulmuladdX_store(Qpa, u, v);
34 console.log("===== ecGenMulmuladdX_store");
35 console.log("x1      : %x", x1);
36
37 assertTrue(x1 == Res[0], "[-] ecGenMulmuladdX_store bad");
38
39 uint256 x; uint256 y;
40 uint256[6] memory QpaW = [0, 0, p, a, gx, gy];
41 (QpaW[0], QpaW[1]) = (Q[0], Q[1]);
42 (x, y) = ecGenMulmuladdB4W(QpaW, u, v);
43 console.log("===== ecGenMulmuladdB4W");
44 console.log("x, y : (%x, %x)", x, y);
45
46 assertTrue(x == Res[0], "[-] ecGenMulmuladdB4W x bad");
47 // XXX: this is not implemented in SCL for now
48 //assertTrue(y == Res[1], "[-] ecGenMulmuladdB4W y bad");
49
50 return;
51 }

```

Listing 13: Solidity test of DSM using forge ffi and the external gen_ecmuladd.py script

```

1 // Test verify
2 function test_verify() public {
3     console.log("[+] Testing the verification function");
4     uint256[10] memory Qpa = [0, 0, 0, 0, p, a, gx, gy, gpow2p128_x,
        gpow2p128_y];
5
6     // Prepare the external command that provides public key, (possibly

```

```

    wrong) signature and the expected result
7   string[] memory cmds = new string[](1);
8
9   for(uint i=0;i < 10;i++){
10      cmds[0] = "./gen_sign.py";
11
12      bytes memory cmd_result = vm.ffi(cmds);
13
14      (uint256 Qx, uint256 Qy, bytes32 h, uint256 r, uint256 s, bool
    res) = abi.decode(cmd_result, (uint256, uint256, bytes32, uint256,
    uint256, bool));
15
16      console.log("=====");
17      console.log("PubKey: (%x, %x)", Qx, Qy);
18      console.log("H      : %x", uint256(h));
19      console.log("Sig    : (%x, %x)", r, s);
20      console.log("Res     : %d", res);
21
22      // Check the signature
23      (Qpa[0], Qpa[1]) = (Qx, Qy);
24      (Qpa[2], Qpa[3]) = ecPow128(Qx, Qy, 1, 1); //compute Q^128
25
26      if(ec_isOnCurve(p,a,b,Qx,Qy) == false){
27          revert();
28      }
29
30      bool result = SCL_ECDSAB4.verify(h, r, s, Qpa, n);
31
32      assertTrue(result == res, "[-] Signature verification test
    failed!");
33  }
34
35  return;
36 }
```

Listing 14: Solidity test of ECDSA using forge ffi and the external gen_sign.py script

Figure 2: DSM formulas testing with external Python script: example of a passing test

Figure 3: ECDSA failure using Wycheproof `json` formatted test vector from Observation 13 (●)

[illegible]

Figure 4: DSM formulas testing with external Python script: example of a failing test using Observation 13 (●)

References

- [1] Avoiding Edge Cases for Double Scalar Multiplication. https://github.com/get-smooth/crypto-lib/blob/main/doc/weak_ecdsa_keys.md.
- [2] Ethereum Precompiled Contracts. <https://docs.bifrostnetwork.com/bifrost-network/developer-documentations/ethereum-api/ethereum-precompiled-contracts>.
- [3] Foundry Book. <https://book.getfoundry.sh>.
- [4] Foundry Book (ffi). <https://book.getfoundry.sh/cheatcodes/ffi>.
- [5] OID Description: 1.3.6.1.4.1.11591.15.1. <http://oid-info.com/get/1.3.6.1.4.1.11591.15.1>.
- [6] OID Description: 1.3.6.1.4.1.3029.1.5.1. <http://oid-info.com/get/1.3.6.1.4.1.3029.1.5.1>.
- [7] OpenZeppelin invMod implementation. <https://github.com/OpenZeppelin/openzeppelin-contracts/blob/442886ed5ff8a0b9ab477b191f5238541ee6d772/contracts/utils/math/Math.sol#L243>.
- [8] Project Wycheproof. <https://github.com/c2sp/wycheproof>.
- [9] SEC 1. Standards for Efficient Cryptography Group: Elliptic Curve Cryptography. American National Standards Institute.
- [10] SEC 2. Standards for Efficient Cryptography Group: Recommended Elliptic Curve Domain Parameters. American National Standards Institute.
- [11] Test vectors of type EcdsaVerify are meant for the verification of IEEE P1363 encoded ECDSA signatures. https://github.com/C2SP/wycheproof/blob/master/testvectors/ecdsa_secp256r1_sha256_p1363_test.json.
- [12] Transaction malleability. https://en.bitcoin.it/wiki/Transaction_malleability.
- [13] XYZZ coordinates for short Weierstrass curves. <https://www.hyperelliptic.org/EFD/g1p/auto-shortw-xyzz.html>.
- [14] ANSI X9.62, Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA), September 1998. American National Standards Institute, X9-Financial Services.
- [15] Vitalik Buterin. EIP-198: Big integer modular exponentiation, January 2017. <https://github.com/ethereum/EIPs/blob/master/EIPS/eip-198.md>.

- [16] Vitalik Buterin. You can *kinda* abuse ECRECOVER to do ECMUL in secp256k1 today, July 2018. <https://ethresear.ch/t/you-can-kind-a-abuse-ecrecover-to-do-ecmul-in-secp256k1-today/2384>.
- [17] Renaud Dubois. Speeding up elliptic computations for Ethereum Account Abstraction. Cryptology ePrint Archive, Paper 2023/939, 2023. <https://eprint.iacr.org/2023/939>.
- [18] Renaud Dubois. Ethereum Zürich: Even Faster secp256r1 for Passkeys and SGX, April 2024. https://github.com/get-smooth/crypto-lib/blob/main/doc/EthZurich_7_4_24.pdf.
- [19] Renaud Dubois. RIP-7696: generic Double Scalar Multiplication (DSM) for all curves, April 2024. <https://ethereum-magicians.org/t/rip-7696-generic-double-scalar-multiplication-dsm-for-all-curves/19798>.
- [20] Ethereum Foundation. An Ethereum Virtual Machine Opcodes Interactive Reference. <https://www.evm.codes/?fork=grayGlacier>.
- [21] Bodo Möller. Securing elliptic curve point multiplication against side-channel attacks. In George I. Davida and Yair Frankel, editors, *Information Security, 4th International Conference, ISC 2001, Malaga, Spain, October 1-3, 2001, Proceedings*, volume 2200 of *Lecture Notes in Computer Science*, pages 324–334. Springer, 2001.
- [22] Bodo Möller. Parallelizable elliptic curve point multiplication method with resistance against side-channel attacks. In Agnes Hui Chan and Virgil D. Gligor, editors, *Information Security, 5th International Conference, ISC 2002 Sao Paulo, Brazil, September 30 - October 2, 2002, Proceedings*, volume 2433 of *Lecture Notes in Computer Science*, pages 402–413. Springer, 2002.
- [23] Christian Reitwiessner. EIP-196: Precompiled contracts for addition and scalar multiplication on the elliptic curve alt_bn128, February 2017. <https://github.com/ethereum/EIPs/blob/master/EIPS/eip-196.md>.
- [24] IETF (R. Struik). Alternative Elliptic Curve Representations (draft-ietf-lwig-curve-representations-19), 2020. <https://datatracker.ietf.org/doc/html/draft-ietf-lwig-curve-representations-19>.
- [25] Doğan Alpaslan Ulaş Erdoğan. RIP-7212: Precompile for secp256r1 Curve Support, June 2023. <https://github.com/ethereum/RIPs/blob/master/RIPS/rip-7212.md>.