Evaluation of Bulletproofs and MLSAG

Réf.	19-04-948-REP
Version	1.0
Date	June 24th, 2019
Prepared for	Particl
Realised by	Quarkslab

Quarkslab

SECURING EVERY BIT OF YOUR DATA

Quarkslab SAS
13 rue Saint Ambroise
75011 Paris, France

Contents

1. Project Information

Changelog			
Version	**Date**	**Details**	**Authors**
1.0	06/24/2019	First version sent to Particl	Lucas Barthelemy, Maxime Peterlin

Quarkslab		
Contact	**Position**	**E-mail address**
Frédéric Raynal	Quarkslab CEO	fraynal@quarkslab.com
Matthieu Duez	Service Manager	mduez@quarkslab.com
Lucas Barthelemy	R&D Engineer	lbarthelemy@quarkslab.com
Maxime Peterlin	R&D Engineer	mpeterlin@quarkslab.com

Particl		
Contact	**Position**	**E-mail address**
Ryno Mathee	Lead & Core Developer	ryno@particl.io

2. Executive Summary

2.1 Context

This document presents the results obtained during the audit of two algorithms added by Particl in their cryptocurrency *Part*. Their main functions are to provide the users with greater confidentiality and anonymity.

The algorithms in question are:

- *Bulletproofs*, to prove that a value is in a given range without ever needing to reveal the actual value, thus preventing money creation;
- *MLSAG*, a ring signature algorithm designed to hide the amount of the transaction and to allow the sender to sign the latter without revealing their identity.

Quarkslab was asked by Particl to assess the security of these two algorithms as they will be at the heart of the transaction verification mechanism. Making sure that Bulletproofs and MLSAG have been correctly implemented is crucial. A vulnerability in either of those algorithms could allow the forging of a proof or a signature, which could result in money theft or creation.

2.2 Mission Conduct

The first few days were dedicated to understanding the general architecture of the project[1], testing the code, etc.

Then the mission was divided into two specific parts. The first one was to audit Bulletproofs. The latest version of the original paper, at the time of the audit[2], was reviewed to understand the details of the algorithm before reviewing the code and looking for vulnerabilities. This part relied heavily on the audit performed by Quarkslab on Monero[3] to make sure that the implementation matches the original paper.

The second part of the audit was similar to the first but focused this time on MLSAG. Up until this point, Quarkslab had not performed an audit on this technology, or a similar one, thus more time was needed to assess its security compared to Bulletproofs.

Ultimately, the main objective was to find vulnerabilities that could allow a rangeproof to be bypassed, the amount of a confidential transaction to be known or a ring signature to be forged.

Apart from testing directly the code in general or just certain functions, the major part of the audit was performed statically. Because of time constraints no fuzzing or differential fuzzing techniques were used. The audit was mostly focused on high level functions such as the proof, signature and verification algorithms rather than the lower-level primitives (operations on scalars, operations on group elements, multi-exponentiation, etc.).

2.3 Results

At the end of the 25 days allotted for this audit, no vulnerability was found in the implementations of Bulletproofs and MLSAG.

[1] https://github.com/particl/particl-core/

[2] https://eprint.iacr.org/eprint-bin/getfile.pl?entry=2017/1066&version=20180701:235657&file=1066.pdf

[3] https://blog.quarkslab.com/resources/2018-10-22-audit-monero-bulletproof/18-06-439-REP-monero-bulletproof-sec-assessmo
pdf

In summary:

- all the vulnerabilities that affected Monero's implementation do not apply to Particl;
- thorough checks ensure that the inputs and outputs are the ones expected by the different algorithms;
- the implementations match the protocols provided in the paper introducing these algorithms.

3. Bulletproofs

This section details the methodology and the results of the Bulletproofs algorithms assessment.

3.1 Code Overview

The implementation chosen by *Particl* is based on Andrew Poelstra's[1] and some improvements have been made to it. These modifications will not be discussed in this report and the implementation used by Particl will be audited as is.

The version of the project audited by Quarkslab was the head of the `master` branch on their *particl-core* GitHub repository when the assessment started, namely commit `b6df3028991a2d721e45571ba38fd896f1162dda`[2].

3.1.1 Project Structure

In the *particl-core* GitHub repository, the implementations for Bulletproofs can be found in the directory `src/secp256k1/src/modules/bulletproofs`.

This folder has the following structure:

```
src/secp256k1/src/modules/bulletproofs/
+-- inner_product_impl.h
+-- main_impl.h
+-- Makefile.am.include
+-- rangeproof_impl.h
+-- tests_impl.h
+-- util.h
```

The main dependencies of the code in `src/secp256k1/src/modules/bulletproofs` are functions defined in this directory or functions handling multi-exponentiations, scalars, group elements and field elements. Most of them are defined in `src/secp256k1/src`.

The files contained in `src/secp256k1/src/modules/bulletproofs` serve the following purposes:

- `main_impl.h`

  Contains the definition and wrappers of the main functions used for bulletproofs (in particular `secp256k1_bulletproof_rangeproof_prove` and `secp256k1_bulletproof_rangeproof_verify`).

- `rangeproof_impl.h`

  Defines callbacks and the core functions of proof and verification forming bulletproofs.

- `inner_product_impl.h`

  Defines the core functions computing the inner product for the proof and the verification.

- `util.h`

[1] https://github.com/apoelstra/secp256k1-mw
[2] https://github.com/particl/particl-core/commit/b6df3028991a2d721e45571ba38fd896f1162dda

Defines utility functions such as Hamming weight computation, dot product, (de)serialization functions, etc.

- `tests_impl.h`

  Tests written to verify if the implementation is correct and works as expected.

The main functions of proof and verification can be included using the header file `src/secp256k1/include/secp256k1_bulletproofs.h`.

3.1.2 Bulletproofs Algorithms

It was agreed with Particl that the verification of the Bulletproofs algorithm would rely on the pseudo-code that was written during the Monero's assessment performed by Quarkslab[3]. The following notations are taken directly from the original paper[4].

Public parameters:

- `l`: cardinality of the subgroup of the elliptic curve used (Ed25519)
- `N`: bitsize of the elements whose range one wants to prove (`N = 64`)
- `M`: number of proofs to aggregate (upper-bounded by `maxM = BULLETPROOF_MAX_OUTPUTS = 16`)
- `G`: the base point of the subgroup of the elliptic curve used
- `H`: another generator of the subgroup of the elliptic curve used whose discrete log w.r.t. `G` is not known and hard to find
- `G_i`: a list of `M*N` generators of the subgroup of the elliptic curve used whose discrete log w.r.t. any other generator is not known and hard to find
- `H_i`: a list of `M*N` generators of the subgroup of the elliptic curve used whose discrete log w.r.t. any other generator is not known and hard to find

Values to commit to, hide, and prove:

- `v`: a list of `M` integers such that for all j, $0 \leq$ v[j] $< 2^N$
- `gamma`: a list of `M` integers such that for all j, $0 \leq$ gamma[j] $< l$

A `bulletproof` is composed of:

- `V`: a vector of curve points, Pedersen commitments to `v[i]` with hiding values `gamma[i]`
- `A`: a curve point, vector commitment to `aL` and `aR` with hiding value `alpha`
- `S`: a curve point, vector commitment to `sL` and `sR` with hiding value `rho`
- `T1`: a curve point, Pedersen commitment to `t1` with hiding value `tau_1`
- `T2`: a curve point, Pedersen commitment to `t2` with hiding value `tau_2`
- `tau_x`: a scalar, hiding value related to `T1`, `T2`, `V` and `t`
- `mu`: a scalar, hiding value related to `A` and `S`
- `L`: a vector of curve points of size `log2(M*N)` computed in the inner product protocol
- `R`: a vector of curve points of size `log2(M*N)` computed in the inner product protocol

[3] https://blog.quarkslab.com/resources/2018-10-22-audit-monero-bulletproof/18-06-439-REP-monero-bulletproof-sec-assessmo
pdf

[4] https://eprint.iacr.org/eprint-bin/getfile.pl?entry=2017/1066&version=20180701:235657&file=1066.pdf

- `a`: a scalar computed in the inner product protocol
- `b`: a scalar computed in the inner product protocol
- `t`: a scalar, inner product value to be verified

The pseudo-code for the *prove* and *verify* algorithms is given in the following sections.

Proof Algorithm

The function `bulletproof_PROVE` takes as input `v` and `gamma` and outputs a proof using an inner product argument of knowledge of two vectors `l` and `r` proving, without revealing it, that for each value `v[i]`, the vector `aL[i]` is indeed its binary representation. It proves that all `v[i]` lie in the interval `[0, 2^N-1]`.

```
bulletproof_PROVE(v, gamma)
// Compute V: a list of curve points, Pedersen commitments to v[i]
// with hiding values gamma[i]
// Compute aL[i] the vector containing the binary representation of v[i]
// Compute aR[i] the opposite of the complementary to one of aL[i]
for (j from 0 to M-1)
     V[j] = pedersen_commitment(gamma[i], H, v[i], G)
     aL[j] = binary_rep(v[j]) // Line 41
     aR[i] = vector_sub(aL[j], one(N)) // Line 42

 // Compute A: a curve point, vector commitment to aL and aR with hiding value alpha
 alpha = random_gen(l) // Line 43
 A = vector_commitment(alpha, H, concat(aL, aR), concat(Gi, Hi)) // Line 44

 // Compute S: a curve point, vector commitment to sL and sR with hiding value rho
 sL = random_gen_vector(M*N, l) // Line 45
 sR = random_gen_vector(M*N, l) // Line 45
 rho = random_gen(l) // Line 46
 S = vector_commitment(rho, H, concat(sL, sR), concat(Gi, Hi)) // Line 47

 // Random challenges to build the inner product to prove the values of aL and aR
 // Line 49 plus non-interactive
 y = hash_to_scalar_non_null(V, A, S)
 z = hash_to_scalar_non_null(V, A, S, y)

 // reconstruct the coefficients of degree 1 and of degree 2 of the
 // range proof inner product polynomial
 (t1,t2) = range_proof_inner_product_poly_coeff(aL, sL, aR, sR, y, z)

 // Compute T1: a curve point, Pedersen commitment to t1 with hiding value tau1
 tau1 = random_gen(l) // Line 52
 T1 = pedersen_commitment(tau1, H, t1, G) // Line 53
 // Compute T2: a curve point, Pedersen commitment to t2 with hiding value tau2
 tau2 = random_gen(l) // Line 52
 T2 = pedersen_commitment(tau2, H, t2, G) // Line 53

 // Random challenge to prove the commitment to t1 and t2
 // Line 55 plus non-interactive
 x = hash_to_scalar_non_null(V, A, S, y, z, T1, T2)

 // Compute t: a scalar, inner product value to be verified
```

(continues on next page)

(continued from previous page)

```
l = range_proof_inner_product_lhs(aL, sL, x, z) // Line 58
r = range_proof_inner_product_rhs(aR, sR, x, y, z) // Line 59
t = inner_product(l, r) // Line 60

// Compute taux: a scalar, hiding value related to x.T1, x^2.T2, z^2.V and t
// Line 61
taux = range_proof_inner_product_poly_hiding_value(tau1, tau2, gamma, x, z)

// Compute mu: a scalar, hiding value related to A and x.S
mu = l_r_vector_commitment_hiding_value(alpha, rho, x) // Line 62

// Adapt Hi, the vector of generators
// to apply an inner product argument of knowledge on l and r
// Line 64
Hiprime = l_r_generators_inner_product_adapt(Hi, y)

// Random challenge
// Line 6 plus non-interactive
x_ip = hash_to_scalar_non_null(V, A, S, y, z, T1, T2, x, taux, mu, t)

Hx = scalar_mul_point(x_ip, H)

// Compute L, R, curve points, and a, b, scalars
// Output of the inner product argument of knowledge
(L, R, a, b) = inner_product_prove(Gi, Hiprime, Hx, l, r)

return (V, A, S, T1, T2, taux, mu, L, R, a, b, t) // Line 63
```

The inner product argument of knowledge corresponds to Protocol 2 in the paper.

```
inner_product_prove(Gi, Hi, U, a, b)
// n is the size of the input vectors
n = M * N
round = 0
while (n > 1)
     n = n / 2 // Line 20
     cL = inner_product(slice(a, 0, n), slice(b, n, 2*n)) // Line 21
 cR = inner_product(slice(a, n, 2*n), slice(b, 0, n)) // Line 22

 // Compute the intermediate commitments L[round], R[round]
 // Line 23-24
     L[round] = vector_commitment(cL, U, concat(slice(a, 0, n), slice(b, n, 2*n)),
                                  concat(slice(Gi, n, 2*n), slice(Hi, 0, n)))
     R[round] = vector_commitment(cR, U, concat(slice(a, n, 2*n), slice(b, 0, n)),
                                 concat(slice(Gi, 0, n), slice(Hi, n, 2*n)))

 // Random challenge Line 26 plus non-interactive
     w = hash_to_scalar_non_null(L[round],R[round])

 // Shrink generator vectors
 // Line 29-30
     Gi = hadamard_points(scalar_mul_vector_points(invert(w), slice(Gi, 0, n)),
                          scalar_mul_vector_points(w,         slice(Gi, n, 2*n)))
     Hi = hadamard_points(scalar_mul_vector_points(w,         slice(Hi, 0, n)),
                          scalar_mul_vector_points(invert(w), slice(Hi, n, 2*n)))
```

(continues on next page)

(continued from previous page)

```
 // Shrink scalar vectors
 // Line 33-34
     a = vector_add(scalar_mul_vector(w,         slice(a, 0, n)),
                    scalar_mul_vector(invert(w), slice(a, n, 2*n)))
     b = vector_add(scalar_mul_vector(invert(w), slice(b, 0, n)),
                    scalar_mul_vector(w,         slice(b, n, 2*n)))
 round = round + 1
return (L, R, a[0], b[0]) // Lines 25 and 15
```

Verification Algorithm

The verification algorithm takes as input a list of bulletproofs and relies on a batch verification optimization.

The simple verification function for the inner product protocol can be optimized by using a multi-exponentiation algorithm.

The batch verification optimization uses a trick involving an additional random scalar for each proof allowing a simultaneous verification of all the proofs with multi-exponentiation.

```
bulletproof_VERIFY(prooflist: a list of bulletproofs)
// Checks that the sizes are coherent,
// that the scalars are reduced,
// that the points are on the right curve
// that the points are on the right subgroup
for (all proof in prooflist)
    if (!bulletproof_early_checks(proof))
        return false

for (all proof in prooflist)
    // Reconstruct the challenges of Lines 49 and 55
    y = hash_to_scalar_non_null(proof.V, proof.A, proof.S)
    y_list = y_list.append(y)
    z = hash_to_scalar_non_null(proof.V, proof.A, proof.S, y)
    z_list = z_list.append(z)
    x = hash_to_scalar_non_null(proof.V, proof.A, proof.S, y, z, proof.T1, proof.T2)
    x_list = x_list.append(x)

    // Check that the commitment to t does indeed correspond to
    // the commitments to t1 (T1) and t2 (T2) and v[i] (V[i])
    // Line 65 (or rather 72)
    if (!check_commitment_inner_product_poly_coeff(proof.t, proof.taux, proof.V,
                                                   proof.T1, proof.T2, x, y, z))
        return false

    // Reconstruct the random challenge, Line 6
    x_ip = hash_to_scalar_non_null(proof.V, proof.A, proof.S, y, z, proof.T1,
                                   proof.T2, x, proof.taux, proof.mu, proof.t)
    x_ip_list = x_ip_list.append(x_ip)

if (!inner_product_batch_verify(Gi, Hi, H, x_ip_list,
                                y_list, z_list, x_list, prooflist)
    return false
return true
```

3.2 Points of Interest

3.2.1 Bulletproofs Scope

The part of the assessment dedicated to Bulletproofs was focused on the following scope:

- Proof algorithm
- Inner product proof algorithm
- Verification algorithm
- Inner product verification algorithm

The secp256k1 primitives (e.g. scalar operations, group and field elements handling, etc.) and the multi-exponentiation algorithms were left out or only partially audited. Also, no fuzzing was performed during this audit and the review was made mostly through static analysis.

3.2.2 Topics Covered

This section describes the different points of interests that were studied during this security assessment.

Random Generation

Bulletproofs are used to verify that a value v is contained in a given interval without ever having to make v public. For this algorithm to work, certain data need to be hidden, or *blinded*, using random values, making random generation in Bulletproofs critical.

An example of such usage of random values can be found in Pedersen Commitments. They use a public group G of order p. For a message m, a random value r and g, h two random generators of G. A Pedersen Commitment C is defined as follows:

$$g, h \in G, m \in \mathbb{Z}_p, r \xleftarrow{\$} \mathbb{Z}_p$$
$$C(m, r) = m \cdot g + r \cdot h$$

If an attacker were to determine the random data r used, the original value could be bruteforced and retrieved relatively easily.

To generate random values, Particl uses the stream cipher *Chacha20* implemented in the function `secp256k1_scalar_chacha20`.

```
/** Generate two scalars from a 32-byte seed and an integer using the chacha20 stream␣
↪cipher */
static void secp256k1_scalar_chacha20(secp256k1_scalar *r1, secp256k1_scalar *r2,␣
↪const unsigned char *seed, uint64_t idx);
```

An example of the generation of such random values is given below.

```
/* [...] */

/* S COMPUTATION */
/* Random generation of the values sL and sR */
secp256k1_scalar_chacha20(&sl, &sr, nonce, i * nbits + j + 2);
```

This stream cipher, however, requires a seed (`nonce` in the example above). This seed can be either set directly, or calculated according to the following snippet of code:

```
uint256 nonce;
if (r.fNonceSet) {
    nonce = r.nonce;
} else {
    if (!r.sEphem.IsValid()) {
        return wserrorN(1, sError, __func__, "Invalid ephemeral key.");
    }
    if (!r.pkTo.IsValid()) {
        return wserrorN(1, sError, __func__, "Invalid recipient pubkey.");
    }
    nonce = r.sEphem.ECDH(r.pkTo);
    CSHA256().Write(nonce.begin(), 32).Finalize(nonce.begin());
    r.nonce = nonce;
}
```

The `ECDH` function is based on `GetStrongRandBytes`, a cryptographically-secure pseudo random number generator used in Bitcoin. Even though this function is slow, it draws entropy from multiple sources (OS, OpenSSL, etc.). Its usage and implementation seem adequate.

Protocol Challenges

The Fiat-Shamir heuristic is used to transform a public-coin interactive proof of knowledge into a non-interactive one. A challenge sent between the prover and the verifier is replaced by a hash of the intermediate values generated by the algorithm at a given time.

The hashing function used in Particl is `SHA256`. An example of a challenge generation is given below.

```
/* Update of the commit with A and S to get the challenge `y` */
secp256k1_bulletproof_update_commit(commit, &out_pt[0], &out_pt[1]);
secp256k1_scalar_set_b32(&y, commit, &overflow);
/* Making sure that there is no overflow or that the challenge is not 0 */
if (overflow || secp256k1_scalar_is_zero(&y)) {
    return 0;
}
```

The function `secp256k1_bulletproof_update_commit` is used to update the `commit` hash using the current state of the algorithm. In the example above, `out_pt[0]` corresponds to A and `out_pt[1]` to S.

```
static void secp256k1_bulletproof_update_commit(
        unsigned char *commit,
        const secp256k1_ge *lpt,
        const secp256k1_ge *rpt)
{
    secp256k1_fe pointx;
    secp256k1_sha256 sha256;
    unsigned char lrparity;
    /*
     * 2-bit value:
     *     - bit 0: 1 if `rpt->y` is a quadratic residue, 0 otherwise
     *     - bit 1: 1 if `lpt->y` is a quadratic residue, 0 otherwise
     */
    lrparity = (!secp256k1_fe_is_quad_var(&lpt->y) << 1) + !secp256k1_fe_is_quad_var(&
↪rpt->y);
```

(continues on next page)

(continued from previous page)

```
    secp256k1_sha256_initialize(&sha256);
    /* Add the current commit to the hash */
    secp256k1_sha256_write(&sha256, commit, 32);
    /* Add `lrparity` to the hash */
    secp256k1_sha256_write(&sha256, &lrparity, 1);

    pointx = lpt->x;
    secp256k1_fe_normalize(&pointx);
    secp256k1_fe_get_b32(commit, &pointx);
    /* Add `lpt->x` to the hash */
    secp256k1_sha256_write(&sha256, commit, 32);

    pointx = rpt->x;
    secp256k1_fe_normalize(&pointx);
    secp256k1_fe_get_b32(commit, &pointx);
    /* Add `rpt->x` to the hash */
    secp256k1_sha256_write(&sha256, commit, 32);

    /* Computes the hash and stores it into `commit` */
    secp256k1_sha256_finalize(&sha256, commit);
}
```

Successively hashing the values of `lpt->x`, `rpt->x` and a two-bit value that depends on whether or not `rpt->y` and `lpt->y` are quadratic residues, creates hard-to-guess inputs.

Once the commit has been generated, it is converted and reduced from a 32-byte array into a scalar using `secp256k1_scalar_set_b32`.

Finally, to make sure that the output does not introduce security issues, verifications are made ensuring there is no overflow and that the final value is not zero.

```
/* Making sure that there is no overflow or that the challenge is not 0 */
if (overflow || secp256k1_scalar_is_zero(&y)) {
    return 0;
}
```

This sequence of instructions is called each time a challenge is needed. One of our recommandations would be to create a function for such operations. While verifications are performed after each call to `secp256k1_scalar_set_b32`, it would prevent potential oversights if the code were to be changed.

Generators of the Main Subgroup secp256k1

Bulletproofs is based on two main functions, namely:

- `secp256k1_bulletproof_rangeproof_prove` to create the proof;
- `secp256k1_bulletproof_rangeproof_verify_impl` to verify it.

To perform their various computations on the *secp256k1* elliptic curve, they require generators to be created beforehand, which will be discussed in the following subsections.

These generators cannot be taken randomly, otherwise they could undermine the protocol security. Other than the fact that all points generated should be on the *secp256k1* curve, there can be no discrete log relation between the two. A notable property is that since *secp256k1* is of prime order, all points on this subgroup are generators.

G is the base point of the `secp256k1` curve. It is defined in the file `src/secp256k1/src/modules/bulletproofs/main_impl.h`.

The alternate generator H is defined in a *nothing-up-my-sleeve* way, by hashing the point G with `SHA256` and using the resulting value as x coordinates to lift it into a point. The final value can also be found in `src/secp256k1/src/modules/bulletproofs/main_impl.h`.

Generators are created in different files (some in used production, some only for tests). This report will focus on the generation performed in the file `src/wallet/hdwallet.cpp`.

Generators Creation in `hdwallet.cpp`

In the file `src/wallet/hdwallet.cpp`, the generator used for the value, in the context of a Pedersen commitment, is H.

```
secp256k1_bulletproof_rangeproof_prove(
    secp256k1_ctx_blind,
    m_blind_scratch,
    blind_gens,
    pvRangeproof->data(),
    &nRangeProofLen,
    &nValue,
    nullptr,
    bp,
    1,
    &secp256k1_generator_const_h, /* value_gen */
    64,
    nonce.begin(),
    nullptr,
    0
)
```

The blinding generator used is G and is set in `src/blind.cpp` using the function `secp256k1_bulletproof_generators_create`. The call to this function is also used to generate all the other generators used by bulletproofs, i.e., the vectors $\mathbf{g}$ and $\mathbf{h}$ comprising the elements g_i and h_i.

```
blind_gens = secp256k1_bulletproof_generators_create(
    secp256k1_ctx_blind,          /* Context */
    &secp256k1_generator_const_g, /* Blinding generator */
    128                           /* Number of generators to create */
);
```

`secp256k1_bulletproof_generators_create` uses the coordinates of the point G as a secret key to generate a HMAC based on SHA256. Successive HMACs are generated and provided to the function `secp256k1_generator_generate` as a seed which returns a generator for the *secp256k1* based curve.

The function used for this operation is `secp256k1_generator_generate_internal`. The goal behind this function is to return a random generator. This is achieved using the function `shallue_van_de_woestijne` which implements the algorithm presented in *Indifferentiable Hashing to Barreto-Naehrig Curves*[5]. This algorithm is a hashing function indifferentiable from a random oracle that returns a point on the *secp256k1* curve.

The function `shallue_van_de_woestijne` was not audited, however, if implemented correctly,

[5] https://www.di.ens.fr/~fouque/pub/latincrypt12.pdf

it should not return generators with trivial discrete log relations.

`secp256k1_bulletproof_generators_create` uses `shallue_van_de_woestijne` twice in a row to create a generator which is saved using `secp256k1_generator_save`. This latter function uses `secp256k1_ge_is_infinity` to check if the generator created is at infinity and stops the execution if that's the case.

Arithmetic Operations

While low-level arithmetic operations were not explicitly in-scope, they define the base on which every algorithms are built, making them critical from a security point of view.

The comments from this section apply for both Bulletproof and MLSAG.

The most used primitives are the following:

- `secp256k1_scalar_add`;
- `secp256k1_scalar_mul`;
- `secp256k1_scalar_sqr`;
- `secp256k1_scalar_negate`;
- `secp256k1_scalar_set_b32`;
- `secp256k1_scalar_get_b32`.

The objective was to find a fault in the implementation that would lead to erroneous outputs, overflows, etc. To audit these functions, the code was mostly reviewed statically, no fuzzing or differential fuzzing was performed.

No vulnerability or faulty implementation were observed. These functions usually end with the following instructions, or an equivalent:

```
over = secp256k1_scalar_reduce(r, secp256k1_scalar_check_overflow(r));
if (overflow) {
    *overflow = over;
}
```

This reduces the output and prevents overflow. No method to bypass `secp256k1_scalar_check_overflow` or `secp256k1_scalar_reduce` was found during this audit.

Multi-exponentiation

A mutli-exponentiation is an efficient way to compute simultaneously exponentiations that use different points and scalars. This technique is used by Bulletproofs to verify multiple aggregated proofs in a single computation.

The multi-exponentiation algorithms are regrouped in a high-level function called `secp256k1_ecmult_multi_var`.

```
/**
 * Multi-multiply: R = inp_g_sc * G + sum_i ni * Ai.
 * Chooses the right algorithm for a given number of points and scratch space
 * size. Resets and overwrites the given scratch space. If the points do not
 * fit in the scratch space the algorithm is repeatedly run with batches of
```

(continues on next page)

(continued from previous page)

```
 * points.
 * Returns: 1 on success (including when inp_g_sc is NULL and n is 0)
 *          0 if there is not enough scratch space for a single point or
 *          callback returns 0
 */
static int secp256k1_ecmult_multi_var(
    const secp256k1_ecmult_context *ctx,
    secp256k1_scratch *scratch,
    secp256k1_gej *r,
    const secp256k1_scalar *inp_g_sc,
    secp256k1_ecmult_multi_callback cb,
    void *cbdata,
    size_t n
);
```

The implementation used by Particl relies on two algorithms, namely Straus' and Pippenger's. The use of one algorithm rather than the other is determined by the number of points used. It starts with Straus' algorithm and, after a given threshold is reached (`ECMULT_PIPPENGER_THRESHOLD`), shifts to Pippenger's.

Straus' algorithm is defined in the function `secp256k1_ecmult_strauss_batch` and Pippenger's algorithm is defined in the function `secp256k1_ecmult_pippenger_batch`.

These two algorithms were not reviewed during this audit because of time constraints.

Serialization

In order to transmit information between the prover and the verifier, some data are serialized in a binary blob. Particl uses two primitives to perform these operations: `secp256k1_bulletproof_serialize_points` to serialize data and `secp256k1_bulletproof_deserialize_point` to deserialize them.

These functions are relatively simple. A value is stored or extracted at a given offset. Verifications are made to ensure that these values are valid points on the curve (using `secp256k1_ge_set_xquad`). Additionally, the offsets are not user-controlled, preventing out-of-bound accesses.

Prove Algorithms

This section will explain the inner workings of the different functions implementing the proof algorithm of Bulletproof, compare it to the one described in the reference paper and detail the type of vulnerabilities and weaknesses that were searched during this assessment.

`secp256k1_bulletproof_rangeproof_prove`

The generation of a proof starts in the function `secp256k1_bulletproof_rangeproof_prove`. Its prototype is given below.

```
int secp256k1_bulletproof_rangeproof_prove(const secp256k1_context* ctx,
    secp256k1_scratch_space *scratch,
    const secp256k1_bulletproof_generators *gens,
    unsigned char *proof,
```

(continues on next page)

(continued from previous page)

```
    size_t *plen,
    const uint64_t *value,
    const uint64_t *min_value,
    const unsigned char* const* blind,
    size_t n_commits,
    const secp256k1_generator *value_gen,
    size_t nbits,
    const unsigned char *nonce,
    const unsigned char *extra_commit,
    size_t extra_commit_len
);
```

This function is a wrapper around `secp256k1_bulletproof_rangeproof_prove_impl` and, before calling it, checks if the provided arguments have the expected format and allocates the necessary scratch space for the proofs to be computed.

The performed checks cover all the user-controlled data. No way to bypass them and inject malformed inputs has been found during this audit.

Once the verification process is over, the function creates scratch spaces for the computation of the proof. These spaces are created as shown in the example below:

```
commitp = (secp256k1_ge *)secp256k1_scratch_alloc(scratch, n_commits *␣
↪sizeof(*commitp));
blinds = (secp256k1_scalar *)secp256k1_scratch_alloc(scratch, n_commits *␣
↪sizeof(*blinds));
```

Note: During the creation of the scratch spaces, if `n_commits` were to be user-controlled, and left unchecked in the calling functions, it could lead to a buffer overflows if it were large enough (e.g. `0x276276276276277` or `0x1000000000000000` in the example below).

```
sizeof(size_t) = 8
sizeof(*commitp) = 68
sizeof(*commitp)*0x276276276276277 = 58
sizeof(*blinds) = 20
sizeof(*blinds)*0x1000000000000000 = 0
```

The last part of the initialization is to compute the Pedersen commitments to the values using the aforementioned blinding values. The formula used is:

$$\text{commitp}_i = \text{blind}_i \cdot G + \text{value} \cdot H$$

```
for (i = 0; i < n_commits; i++) {
    int overflow;
    secp256k1_gej commitj;
    /* Converts the 32-bytes blinding factor into 4 64-bit integers and store it into␣
↪the `blinds` scratch space */
    secp256k1_scalar_set_b32(&blinds[i], blind[i], &overflow);
    if (overflow || secp256k1_scalar_is_zero(&blinds[i])) {
        return 0;
    }

    secp256k1_pedersen_ecmult(&commitj, &blinds[i], value[i], &value_genp, &gens->
↪blinding_gen[0]);
    secp256k1_ge_set_gej(&commitp[i], &commitj);
}
```

Finally, the function `secp256k1_bulletproof_rangeproof_prove_impl` is called.

`secp256k1_bulletproof_rangeproof_prove_impl`

This function starts the actual computation of the proof. Its prototype is given below:

```
/* Proof format: t, tau_x, mu, a, b, A, S, T_1, T_2, {L_i}, {R_i}
 *               5 scalar + [4 + 2log(n)] ge
 *
 * The non-bold `h` in the Bulletproofs paper corresponds to our gens->blinding_gen
 * while the non-bold `g` corresponds to the asset type `value_gen`.
 */
static int secp256k1_bulletproof_rangeproof_prove_impl(
    const secp256k1_ecmult_context *ecmult_ctx,
    secp256k1_scratch *scratch,
    unsigned char *proof,
    size_t *plen,
    const size_t nbits,
    const uint64_t *value,
    const uint64_t *min_value,
    const secp256k1_scalar *blind,
    const secp256k1_ge *commitp,
    size_t n_commits,
    const secp256k1_ge *value_gen,
    const secp256k1_bulletproof_generators *gens,
    const unsigned char *nonce,
    const unsigned char *extra_commit,
    size_t extra_commit_len)
```

Similarly to the calling function, `secp256k1_bulletproof_rangeproof_prove_impl` checks the inputs provided (size of the proof buffer, verifying that the values are in the correct range, etc.). Then a commitment to all input data is calculated using SHA256.

After this operation, the commitment has the following format (`extra_commit` is added at the end if there is an extra commit):

```
C = SHA256(commit || len
           || min_value[0] || ... || min_value[n_commits-1]
           || lrparity[0] || commitp[0]->x || gen_value[0]->x
           || ...
           || lrparity[n_commits-1] || commitp[n_commits-1]->x || gen_value[n_commits-
↪1]->x)
```

This hash will be used throughout this process to non-interactively generate scalars using the Fiat-Shamir heuristic.

Random Values Generation

The random values α, ρ, τ_1 and τ_2 are generated using the stream-cipher algorithm *Chacha20* as explained above.

The A and S Commitments

With those values, the function computes A and S (S requires another use of chacha20 to compute s_L and s_R).

```
/*
 * aj <- alpha*h
 * sj <- rho*h
 */
secp256k1_ecmult_const(&aj, &gens->blinding_gen[0], &alpha, 256);
secp256k1_ecmult_const(&sj, &gens->blinding_gen[0], &rho, 256);

/* [...] */

/* A COMPUTATION */

    /* `aterm` contains either -H or +G because of the definitions of aR and aL */
    secp256k1_ge_neg(&aterm, &aterm);
    secp256k1_fe_cmov(&aterm.x, &gens->gens[i * nbits + j].x, al);
    secp256k1_fe_cmov(&aterm.y, &gens->gens[i * nbits + j].y, al);

    /* aj <- alpha*h + G or aj <- alpha*h - H depending on the bit `al` */
    secp256k1_gej_add_ge(&aj, &aj, &aterm);

/* S COMPUTATION */

    /* Random generation of the values sL and sR */
    secp256k1_scalar_chacha20(&sl, &sr, nonce, i * nbits + j + 2);

    /* `stermj` <- sL*G */
    secp256k1_ecmult_const(&stermj, &gens->gens[i * nbits + j], &sl, 256);
    secp256k1_ge_set_gej(&sterm, &stermj);
    secp256k1_gej_add_ge(&sj, &sj, &sterm);

    /* `stermj` <- sR*H */
    secp256k1_ecmult_const(&stermj, &gens->gens[i * nbits + j + gens->n/2], &sr, 256);
    secp256k1_ge_set_gej(&sterm, &stermj);

    /* sj <- rho*h + sL*G + sR*H */
    secp256k1_gej_add_ge(&sj, &sj, &sterm);

/* [...] */
```

Challenges Generation

The SHA256-based commit generated at the beginning of the function is updated with A and S. The challenge y is generated from there.

```
/* get challenges y and z */
secp256k1_ge_set_gej(&out_pt[0], &aj);
secp256k1_ge_set_gej(&out_pt[1], &sj);

secp256k1_bulletproof_update_commit(commit, &out_pt[0], &out_pt[1]);
secp256k1_scalar_set_b32(&y, commit, &overflow);
if (overflow || secp256k1_scalar_is_zero(&y)) {
```

(continues on next page)

(continued from previous page)

```
    return 0;
}
```

The commit is updated again with A and S to generate the challenge z.

Polynomial Coefficients Computation

Then, the function computes the coefficients t_0, t_1 and t_2, of the $< l, r >$ polynomial with an algorithmic trick that is not present as such in the paper but correct nonetheless:

$$
\begin{aligned}
t_0 &=< l(0), r(0) > \\
M &=< l(1), l(1) > \\
N &=< l(-1), r(-1) > \\
t_1 &= \frac{M - N}{2} \\
t_2 &= -(-N + t0) + t1
\end{aligned}
$$

The Commitments T_1, T_2 and the Challenge x

The function commits to t_1 and t_2 by computing the values T_1 and T_2. The SHA256 commit is updated with these values to generate the challenge x.

The Values τ_x and μ

The values τ_x and μ are calculated and then negated as the verifier will use their negation when verifying the proof.

```
/* compute taux and mu */
secp256k1_scalar_mul(&taux, &tau1, &x);
secp256k1_scalar_mul(&tmps, &tau2, &xsq);
secp256k1_scalar_add(&taux, &taux, &tmps);
for (i = 0; i < n_commits; i++) {
    secp256k1_scalar_mul(&tmps, &zsq, &blind[i]);
    secp256k1_scalar_add(&taux, &taux, &tmps);
    secp256k1_scalar_mul(&zsq, &zsq, &z);
}

secp256k1_scalar_mul(&mu, &rho, &x);
secp256k1_scalar_add(&mu, &mu, &alpha);

/* Negate taux and mu so the verifier doesn't have to */
secp256k1_scalar_negate(&taux, &taux);
secp256k1_scalar_negate(&mu, &mu);
```

Most of the values required for the final computations have been calculated and are passed as arguments to the function `secp256k1_bulletproof_inner_product_prove_impl`.

At this point, the serialized proof looks like this:

```
taux || mu || A || S || T1 || T2
```

secp256k1_bulletproof_inner_product_prove_impl

Once again, this function is a simple wrapper around another, namely `secp256k1_bulletproof_inner_product_real_prove_impl`.

`secp256k1_bulletproof_inner_product_prove_impl` has the following prototype:

```
static int secp256k1_bulletproof_inner_product_prove_impl(
  const secp256k1_ecmult_context *ecmult_ctx,
  secp256k1_scratch *scratch,
  unsigned char *proof,
  size_t *proof_len,
  const secp256k1_bulletproof_generators *gens,
  const secp256k1_scalar *yinv,
  const size_t n,
  secp256k1_ecmult_multi_callback *cb,
  void *cb_data,
  const unsigned char *commit_inp
);
```

Note: In this section, there are mentions of the vectors **a** and **b**, which are the generic name given to the argument of the inner-product algorithm. In this case, **a** and **b** are actually **l** and **r**. The two notations will be used interchangeably in the rest of this report.

There are some operations performed at the beginning of the function, but it is mostly initialization and the handling of a special case.

This special case occurs when n is less than 2. The dot product of a and b can be calculated using a simple multiplication and the proof then becomes an explicit list of scalars that contains the result. a and b are extracted using the callback `cb`, which is `secp256k1_bulletproof_abgh_callback`.

```
taux || mu || A || S || T1 || T2 || <l, r> || l[0] || ... || l[n-1] || r[0] || ... ||␣
↪r[n-1]
```

Otherwise the vectors **a** and **b** are extracted (using the same callback) and are passed to the function `secp256k1_bulletproof_inner_product_real_prove_impl` which is an implementation of the *Protocols 1 and 2* defined in the reference paper.

Before jumping to this function, **a** and **b** are added to the commit which is then used to generate u, the random value used in the computation of L and R.

secp256k1_bulletproof_inner_product_real_prove_impl

As explained above, this function is the implementation of the *Protocols 1 and 2*. Its prototype is given below:

```
static int secp256k1_bulletproof_inner_product_real_prove_impl(
    const secp256k1_ecmult_context *ecmult_ctx,
    secp256k1_scratch *scratch,
    secp256k1_ge *out_pt,
    size_t *pt_idx,
```

(continues on next page)

(continued from previous page)

```
    const secp256k1_ge *g,
    secp256k1_ge *geng,
    secp256k1_ge *genh,
    secp256k1_scalar *a_arr,
    secp256k1_scalar *b_arr,
    const secp256k1_scalar *yinv,
    const secp256k1_scalar *ux,
    const size_t n,
    unsigned char *commit
)
```

As expected, the main loop iterates over `n` and reduces it by half each round.

```
/* Protocol 1: Iterate, halving vector size until it is 1 */
for (halfwidth = n / 2, i = 0; halfwidth > IP_AB_SCALARS / 4; halfwidth /= 2, i++)
```

The L and R Computations

The first operations performed are to compute L and R using the following formula:

$$
\begin{aligned}
L &= \mathbf{a}_{[:n']}\mathbf{g}_{[n':]} + \mathbf{b}_{[n':]}\mathbf{h}_{[:n']} + c_L u \\
R &= \mathbf{a}_{[n':]}\mathbf{g}_{[:n']} + \mathbf{b}_{[:n']}\mathbf{h}_{[n':]} + c_R u \\
c_L &=< \mathbf{a}_{[:n']}, \mathbf{b}_{[n':]} > \\
c_R &=< \mathbf{a}_{[n':]}, \mathbf{b}_{[:n']} >
\end{aligned}
$$

An example of this implementation is given below for L. Similar calculations are done for R.

Note: The paper splits the vectors at the middle, while the implementation used by Particl splits the vector by picking elements at odd or even indexes depending on the value computed. However, to simplify the notations, the ones from the paper will be used in this section.

```
/* L */
/* fdata.g_sc = <a, b> = sum(a_arr[0]*b_arr[1], a_arr[2]*b_arr[3], ..., a_
↪arr[2*(halfwidth-1)]*b_arr[2*(halfwidth-1)+1]) */
secp256k1_scalar_clear(&pfdata.g_sc);
for (j = 0; j < halfwidth; j++) {
    secp256k1_scalar prod;
    secp256k1_scalar_mul(&prod, &a_arr[2*j], &b_arr[2*j + 1]);
    secp256k1_scalar_add(&pfdata.g_sc, &pfdata.g_sc, &prod);
}
/* <a, b> * u */
secp256k1_scalar_mul(&pfdata.g_sc, &pfdata.g_sc, ux);

secp256k1_scalar_set_int(&pfdata.yinvn, 1);
/* L = a_even . G_odd + b_odd . H_even + <a, b> * u */
secp256k1_ecmult_multi_var(ecmult_ctx, scratch, &tmplj, NULL, &secp256k1_bulletproof_
↪innerproduct_pf_ecmult_callback_l, (void *) &pfdata, n + 1);
```

L and R are added to the proof sequentially using similar instructions:

```
secp256k1_ge_set_gej(&out_pt[(*pt_idx)++], &tmplj);
```

Then, x and x^{-1} are computed. To retrieve the challenge x, the following instruction is used:

```
secp256k1_bulletproof_update_commit(
    commit,
    &out_pt[*pt_idx - 2],
    &out_pt[*pt_idx] - 1
);
```

However, it should be:

```
secp256k1_bulletproof_update_commit(
    commit,
    &out_pt[*pt_idx - 2],
    &out_pt[*pt_idx - 1]
);
```

This call works because the value is a pointer on an array of `char` (which means elements have a size of 1 byte). If it was, for example, an array of 32-bit integers (elements with a size of 4 bytes), the wrong value would have been picked to generate the challenge.

Afterwards, the vectors **a'** and **b'** are calculated following the formula given below:

$$\mathbf{a'} = \mathbf{a}_{[:n']}x + \mathbf{a}_{[n':]}x^{-1}$$
$$\mathbf{b'} = \mathbf{b}_{[:n']}x^{-1} + \mathbf{b}_{[n':]}x$$

Then, the vectors **g'** and **h'** are computed using the formula:

$$\mathbf{g'} = x^{-1}\mathbf{g}_{[:n']} + x\mathbf{g}_{[n':]}$$
$$\mathbf{h'} = x\mathbf{h}_{[:n']} + x^{-1}\mathbf{h}_{[n':]}$$

The function is then called recursively on the newly generated arguments:

```
secp256k1_bulletproof_inner_product_real_prove_impl(
    ecmult_ctx, scratch, out_pt, &pt_idx, gens->blinding_gen,
    geng, genh, a_arr, b_arr, yinv, &ux, n, commit
);
```

Once this function has returned and the recursion is finished, the final serialized proof looks as follows (there are four scalars instead of two because two points L and R were traded against scalars):

$$\left(\tau_x, -\mu, A, S, T_1, T_2, < \mathbf{l}, \mathbf{r} >, a_0, a_1, b_0, b_1, L_1, R_1, \ldots, L_{log(n)-1}, R_{log(n)-1}\right)$$

During this audit, no deviation from the reference paper was observed and no vulnerability that could lead to information leakage was found. As far as the audit went, the implementation seems sound and correct.

Verify Algorithms

This section will explain the inner workings of the different functions implementing the verification algorithm of Bulletproof, compare it to the one described in the reference paper and detail the type of vulnerabilities and weaknesses that were researched during this assessment.

secp256k1_bulletproof_rangeproof_verify

The verification of the bulletproof proofs starts in the function `secp256k1_bulletproof_rangeproof_verify`. Its prototype is given below:

```
int secp256k1_bulletproof_rangeproof_verify(
    const secp256k1_context* ctx,
    secp256k1_scratch_space *scratch,
    const secp256k1_bulletproof_generators *gens,
    const unsigned char *proof,
    size_t plen,
    const uint64_t *min_value,
    const secp256k1_pedersen_commitment* commit,
    size_t n_commits,
    size_t nbits,
    const secp256k1_generator *value_gen,
    const unsigned char *extra_commit,
    size_t extra_commit_len
);
```

This function is a wrapper around `secp256k1_bulletproof_inner_product_verify_impl` and, before calling it, checks if the arguments provided have the expected format and allocates the necessary scratch space for the proof to be computed.

The performed checks cover all the user-controlled data. No way to bypass them and inject malformed inputs has been found during this audit.

Afterwards, for each proof, the function regenerates the SHA256-based commitment used to generate the Fiat-Shamir challenges. The operations performed are the same as in the proof algorithm.

In order to prepare the call to `secp256k1_bulletproof_inner_product_verify_impl` different values have to be deserialized from the proof. The function will retrieve:

- the challenges x, y and z;
- the commitments T_1 and T_2;
- the values τ_x and $-\mu$;
- the values $< \mathbf{l}, \mathbf{r} >$.

The structure `innp_ctx` is filled with different parameters that will be used later on to verify the proof.

```
ecmult_data[i].a = age;
ecmult_data[i].s = sge;
ecmult_data[i].n = nbits * n_commits;
ecmult_data[i].count = 0;
ecmult_data[i].asset = &value_gen[i];
ecmult_data[i].min_value = min_value == NULL ? NULL : min_value[i];
ecmult_data[i].commit = commitp[i];
ecmult_data[i].n_commits = n_commits;

secp256k1_scalar_mul(&taux, &taux, &ecmult_data[i].randomizer61);
secp256k1_scalar_add(&mu, &mu, &taux);
innp_ctx[i].proof = &proof[i][64 + 128 + 1];
innp_ctx[i].p_offs = mu; /* mu + tau_x * randomizer61 */
memcpy(innp_ctx[i].commit, commit, 32);
```

(continues on next page)

(continued from previous page)

```
innp_ctx[i].yinv = ecmult_data[i].yinv;
innp_ctx[i].rangeproof_cb = secp256k1_bulletproof_rangeproof_vfy_callback;
innp_ctx[i].rangeproof_cb_data = (void *) &ecmult_data[i];
innp_ctx[i].n_extra_rangeproof_points = 5 + n_commits;
```

Finally, the function `secp256k1_bulletproof_inner_product_verify_impl` is called:

```
secp256k1_bulletproof_inner_product_verify_impl(
    ecmult_ctx, scratch, gens, nbits * n_commits, innp_ctx,
    n_proofs, plen - (64 + 128 + 1), same_generators
);
```

secp256k1_bulletproof_inner_product_verify_impl

This function will do all the necessary computations before finally calling the single multi-exponentiation that verifies the proof. Its prototype is given below:

```
static int secp256k1_bulletproof_inner_product_verify_impl(
    const secp256k1_ecmult_context *ecmult_ctx,
    secp256k1_scratch *scratch,
    const secp256k1_bulletproof_generators *gens,
    size_t vec_len,
    const secp256k1_bulletproof_innerproduct_context *proof,
    size_t n_proofs,
    size_t plen,
    int shared_g
)
```

Initialization

The function starts by performing different checks and initilizating the structure `ecmult_data`.

```
ecmult_data.n_proofs = n_proofs;
ecmult_data.g = gens->blinding_gen;
ecmult_data.geng = gens->gens;
ecmult_data.genh = gens->gens + gens->n / 2;
ecmult_data.vec_len = vec_len;
ecmult_data.lg_vec_len = secp256k1_floor_lg(2 * vec_len / IP_AB_SCALARS);
ecmult_data.shared_g = shared_g;
ecmult_data.randomizer = (secp256k1_scalar *)secp256k1_scratch_alloc(scratch, n_
↪proofs * sizeof(*ecmult_data.randomizer));
ecmult_data.proof = (secp256k1_bulletproof_innerproduct_vfy_data *)secp256k1_scratch_
↪alloc(scratch, n_proofs * sizeof(*ecmult_data.proof));
```

It also generates a random seed by hashing information from all the proofs:

```
SHA256(proof[0].proof || proof[0].commit || proof[0].p_offs || ... || proof[n_proofs -
↪ 1].proof || proof[n_proofs - 1].commit || proof[n_proofs - 1].p_offs))
```

After this operation starts the aggregation of all the proofs.

The following explanations will be a process applied for each individual proof (and iterated over `n_proofs`).

Preparation of the Multi-Exponentiation

The function retrieves the dot product $< \mathbf{l}, \mathbf{r} >$, commits to it and then adds the remaining values making up the proof:

$$a_0, a_1, b_0, b_1, (L_1, R_1), \ldots, (L_{log_2(n)-1}, R_{log_2(n)-1})$$

Afterwards, different complex operations are performed and their use is not explicit. These operations will be detailed from what was understood during the time allotted for the audit.

These operations are listed below.

- Computation of the dot product between the a's (respectively a0 and a1) and b's (respectively b0 and b1).
- Generation of the challenges x_i.
- Computation of x_i^2 into `ecmult_data.proof[i].xsq[j]`.
- Computation of the value in `ecmult_data.p_offs` which is $\sum_{i=1}^{\text{n\_proofs}} \text{rand}_i \left[x_i(< \mathbf{l}, \mathbf{r} > -ab) - \mu + \tau_x c\right]$, where `c` is `randomizer61`.
- Generation of the randomizers rand_i for each proof, which are calculated using a hash of the current commit.
- Computation of $rx_1x_2 \ldots x_n$ where `r` is `randomizer61`.
- Storage of $-a_0$, $-a_1$, $-b_0$ and $-\text{rx}_1\text{x}_2 \ldots \text{x}_\text{n}$ into `ecmult_data.proof[i].abinv`.
- Computation of $(-a_0 r x_1 x_2 \ldots x_n)^{-1}$ into `ecmult_data.proof[i].xsqinv_mask`
- Computation of $-a_0 r (x_1 x_2 \ldots x_n)^{-1}$ into `ecmult_data.proof[i].xcache[0]`

As explained above, all these operations are performed for each proof. Once done, the multi-exponentiation can be calculated.

```
secp256k1_ecmult_multi_var(
    ecmult_ctx, scratch, &r, NULL,
    secp256k1_bulletproof_innerproduct_vfy_ecmult_callback,
    (void *) &ecmult_data, total_n_points
)
```

Multi-Exponentiation

The multi-exponentiation is performed using the function `secp256k1_ecmult_multi_var`. The goal is to compute the verification of the proof in a single multi-exponentiation. Not much detail will be given on the underlying algorithm that performs it. However, what is interesting is what is being computed.

In the section *An Optimized Verifier Using Multi-Exponentiation and Batch Verification* of the reference paper, there are two relations that need to be equal to infinity in order for the proof to be verified:

$$A = P + (\hat{t} - a \cdot b) x_u \cdot g + \sum_{i=1}^{n} s'_i g_i + s_i h_i + \sum_{j=1}^{\log_2(n)} x_j^2 \cdot L_j + x_j^{-2} \cdot R_j$$

$$B = (\delta(y, z) - \hat{t}) \cdot g - \tau_x \cdot h + z^2 \cdot V + x \cdot T_1 + x^2 T_2$$

In the implementation used by Particl, they use the fact that for a random value c, if $c \cdot A + B = 0$, then with high probability, both A and B are equal to 0. In this case, c is `randomizer61` which is multiplied with the scalars of the relation A.

Part of this process is implemented in `secp256k1_bulletproof_innerproduct_vfy_ecmult_callback`.

The prototype of this function is given below:

```
static int secp256k1_bulletproof_innerproduct_vfy_ecmult_callback(
    secp256k1_scalar *sc, secp256k1_ge *pt, size_t idx, void *data
)
```

This function returns different results depending on the `idx` parameter. When it does return, it provides the multi-exponentiation function with a point and scalar to multiply.

- The first n points are (s'_i, g_i): an aggregation of all the scalars multiplied by the generator **g**.
- The next n points are (s_i, h_i): an aggregation of all the scalars multiplied by the generator **h**.
- The next $2log(n)$ are an alternation between (x_i^{-2}, L_i) and (x_i^2, R_i).
- All other points after that are a special case of the function that makes a call to the callback `secp256k1_bulletproof_rangeproof_vfy_callback` (this function will be detailed below). It will mostly be responsible for giving the tuple (scalar, point) of the elements making up P.

Finally, the single multi-exponentiation is computed and the function returns whether or not the point obtained is at infinity:

```
return secp256k1_gej_is_infinity(&r);
```

No deviation from the reference paper was observed and no vulnerability that could lead to the crafting of a proof was found. As far as the audit went, the implementation seems sound and correct.

4. MLSAG

This section details the methodology and the results of the MLSAG algorithms assessment.

4.1 Code Overview

Particl uses their own implementation of MLSAG based on the algorithm described in the paper *Ring Confidential Transactions*[1].

The version of the project audited by Quarkslab was the head of the `master` branch on their *particl-core* GitHub repository when the assessment started, namely commit `b6df3028991a2d721e45571ba38fd896f1162dda`[2].

4.1.1 Project Structure

In the *particl-core* GitHub repository, the implementations for MLSAG can be found in the directory `src/secp256k1/src/modules/mlsag`.

This folder has the following structure:

```
src/secp256k1/src/modules/bulletproofs/
+-- main_impl.h
+-- Makefile.am.include
+-- tests_impl.h
```

The main dependencies of the code in `src/secp256k1/src/modules/mlsag` are functions defined in this directory or functions handling scalars, group elements and field elements. Most of them are defined in `src/secp256k1/src`.

The files contained in `src/secp256k1/src/modules/mlsag` serve the following purposes:

- `main_impl.h`

  Contains the definition and wrappers of the main functions used for MLSAG (in particular `secp256k1_prepare_mlsag`, `secp256k1_generate_mlsag` and `secp256k1_verify_mlsag`).

- `tests_impl.h`

  Contains tests written to verify if the implementation is correct and works as expected.

The main functions of signature and verification can be included using the header file `src/secp256k1/include/secp256k1_mlsag.h`.

4.1.2 MLSAG Algorithms

Public parameters

- `n` a number of columns.
- `m` a number of rows.
- `pk`: an `n*m` public key matrix.

[1] https://lab.getmonero.org/pubs/MRL-0005.pdf
[2] https://github.com/particl/particl-core/commit/b6df3028991a2d721e45571ba38fd896f1162dda

Private parameters:

- `sk`: a secret key vector.
- `index` : a secret index, secret key will be used at this index.

Elements of the signature to verify:

- `I`: a key image vector.
- `c1` : a scalar, starting point for the computation of `L'` and `R'` and successive c_i.
- `ps`: a set of `n*m` random values used to compute `L'` and `R'`.

The pseudo-code for the *signature* and *verify* algorithms is given in the following sections.

Signature Algorithm

the function `secp256k1_generate_mlsag` takes as input:

- a number of public key vectors `n`,
- a size of public key vectors `m`,
- an `n*m` public key matrix `pk`,
- a secret key `sk`,
- a secret index.

The last row of the public key matrix is composed of the sum of input/output commitments. The function outputs an mlsag signature that will be verified by the function `secp256k1_verify_mlsag`. The signature is composed of a scalar `c1`, a key image vector `I` and a set of random values `ps`.

For each public key that is not of the secret index, the algorithm computes a pair of points `L` and `R` from public parameters and random values `ps`. Then a hash is computed with those elements and the key image vector to produce the hash c_i. For the secret index, specific values of `L` and `R` are computed with a random scalar `alpha` and a specific value of `ps` is computed from the secret key and the previous hash $c_{index-1}$. The key image vector `I` is also computed. Once this iteration is over, the signature can be computed and returned as output. Because `I` is necessary to the computation of all c_i except c_{index} and the hash $c_{index-1}$ is necessary for the computation of the specific value of `ps` at the secret index, the algorithm is split in three parts :

- compute the hash at secret index c_{index},
- compute all other hashes from $c_{index+1}$ to $c_{index-1}$
- compute specific value of `ps` at the secret index

```
secp256k1_generate_mlsag(I, n, m, index, pk, sk)

    // first part of the algorithm
    // for all elements of the secret key vector, compute the following values :
    // a random value alpha[k]
    // the values L[k][index] and R[k][index] for the secret index
    // an element of the key image vector I[k]
    for (k from 0 to m)
```

(continues on next page)

(continued from previous page)

```
        // see below for random generation
        alpha[k] <- generate a random number
        L[k][index] = G*alpha[k]
        R[k][index] = H(pk[k][index]) * alpha[k]
        I[k][index] = sk * H(pk[k][index])

    // compute L for the last row of pubkey matrix, this is a special row that
    // contains sum of input/output commitments
    alpha[m+1] <- generate a random number
    L[m+1][index] = G*alpha[m+1]

    // generate the value c[index] from computed values
    c = sha256(pk[0][index], L[0][index], R[0][index], ..., pk[m-1][index], L[m-
↪1][index], R[m-1][index], pk[m][index], L[m][index])
    // store c_index as current c
    clast = c

    // for all other indexes, generate L and R values
    for (i from index+1 to index-1 modulo n)
        for (k from 0 to m)
          // see below for random generation
          // generate random number ss and add it to the set of
          // random numbers ps which will be part of the signature and are
          // required to compute L' and R' in the verify algorithm
          ss <- generate a random number
          ps[k][i] = ss

          L[k][i] = G * ss + pk[k][i] * clast
          R[k][i] = H(pk[k][i]) * ss + I[k] * clast

    // compute L for the last row of pubkey matrix, this is a special row that
    // contains sum of input/output commitments
    ss <- generate a random number
    L[m+1][i] = G * ss + pk[m+1][i] * clast

    // generate c for next iteration
    clast = sha256(pk[0][i], L[0][i], R[0][i], ..., pk[m-1][i], L[m-1][i], R[m-1][i],␣
↪pk[m][i], L[m][i])

  // computation of a value derived from the secret key,
  // clast, which is c_{index-1}, and the random values ``\alpha``
  // used in the computation of L_{index} and R{index}.
  // This value will  be part of the final signature
  // as another "random" value for ps^k_{index}
  for (k from 0 to m)
    ss = alpha[k] - clast * sk[k]

    // finalize the signature composed of the key image vector I,
    // the first hash :math:`\mathrm{c_1}` and the "random" values ps
    // (a subset of them is derived from the secret key)
    finalise_signature()
```

Verification Algorithm

The verification algorithm `secp256k1_verify_mlsag` takes a signature composed of a key image vector `I`, a hash c_1 and a set of seemingly random values `ps`. It also has access to the public key matrix `pk`

The algorithm computes the values `L'`, `R'` and the hashes c'_1 to c'_n. If $c_1 == c_n$ the signature is accepted, otherwise it is rejected. If one of the secret key vectors is used as shown in the signature algorithm, then this equality will be verified and the signature accepted.

```
secp256k1_verify_mlsag(I, n, m, pk, c1, ps)

    // set clast as c1
    clast = c1
    // compute all values of L', R' and c'
    for (i from 0 to n)
        for (k from 0 to m)

            // compute L[k][i]', R[k][i]'
            ss = ps[k][i]
            L[k][i] = G*ss + pk[k][i]*clast
            R[k][i] = H(pk[k][i])*ss + I[k]*clast

        // compute L' for the last row of pubkey matrix, this is a special row that
        // contains sum of input/output commitments
        ss = ps[k][m]
        L[k][m] = G * ss + pk[k][m] * clast

        // generate c for next iteration
        clast = sha256(pk[0][i], L[0][i], R[0][i], \dots, pk[m-1][i], L[m-1][i], R[m-
↪1][i], pk[m][i], L[m][i])

    // check that the last hash is equal to the first (input hash)
    if (c1 == clast )
        return success
    else
        return failure
```

4.2 Points of Interest

4.2.1 MLSAG Scope

The part of the assessment dedicated to MLSAG was focused on the following scope:

- signature algorithm and
- verification algorithm.

4.2.2 Topics Covered

This section describes the different points of interests that were studied during this security assessment.

Random Generation

In MLSAG, random values such as α and s_j are generated in the function `secp256k1_generate_mlsag`. This function takes `nonce` as argument which is then used to initialize random-generating functions.

Below is the example of the generation of the random value α. Using `preimage` and `nonce`, `secp256k1_rfc6979_hmac_sha256_initialize` is initialized and then used afterwards to generate a random α using `secp256k1_rfc6979_hmac_sha256_generate`.

```
memcpy(tmp, nonce, 32);
memcpy(tmp+32, preimage, 32);

/* seed the random no. generator */
secp256k1_rfc6979_hmac_sha256_initialize(&rng, tmp, 32 + 32);

secp256k1_sha256_initialize(&sha256_m);
/* Initialize the SHA256 hash with `preimage` */
secp256k1_sha256_write(&sha256_m, preimage, 32);
sha256_pre = sha256_m;

for (k = 0; k < dsRows; ++k)
{
    /* Generation of a random alpha */
    do {
        secp256k1_rfc6979_hmac_sha256_generate(&rng, tmp, 32);
        secp256k1_scalar_set_b32(&alpha[k], tmp, &overflow);
    } while (overflow || secp256k1_scalar_is_zero(&alpha[k]));
}
```

To understand how this value is generated, the code of `src/wallet/hdwallet.cpp` was analyzed. In this file, the function `secp256k1_generate_mlsag` is coded as follows:

```
if (0 != (rv = secp256k1_generate_mlsag(secp256k1_ctx_blind, &vKeyImages[0],
    &vDL[0], &vDL[32], randSeed, txhash.begin(), nCols, nRows,
    vSecretColumns[l], &vpsk[0], &vm[0]))) {
    return wserrorN(1, sError, __func__, _("secp256k1_generate_mlsag failed %d"), rv);
}
```

In this case, the parameter `nonce` is `randSeed` which is initialized using `GetStrongRandBytes`.

```
uint8_t randSeed[32];
GetStrongRandBytes(randSeed, 32);
```

This function has already been analyzed in the part of this report dedicated to Bulletproofs and no weakness was found.

Sign and Verify Algorithms

This section will discuss and analyze the implementation done by Particl of the MLSAG algorithm to show that it is adequate and matches the original paper.

It will focus on the three main functions on which this MLSAG implementation is based on, namely:

- `secp256k1_prepare_mlsag`,
- `secp256k1_generate_mlsag` and

- `secp256k1_verify_mlsag`.

MLSAG Computation Preparations

The first function called before generating a signature is `secp256k1_prepare_mlsag`. The role of this function is to add the commitments to the matrix containing the public keys (the messages to sign) and the blinding values to the vector containing the private keys.

The prototype of this function is:

```
int secp256k1_prepare_mlsag(uint8_t *m, uint8_t *sk,
    size_t nOuts, size_t nBlinded, size_t nCols, size_t nRows,
    const uint8_t **pcm_in, const uint8_t **pcm_out, const uint8_t **blinds);
```

This function is pretty straightforward and relies almost exclusively on secp256k1 primitives to do operations on group elements and scalars in order to add them, multiply them, etc.

As hinted above, two variables are provided to this function:

- `m` a pointer to the matrix containing the public keys;
- `sk` a pointer to the vector containing the secret keys.

To prevent any confusion, from this point on, this report will use the notations of the paper *Ring Confidential Transactions*[3].

Let l be the number of output commitments, n the number of members of the ring, m the number of keys per member and π the column from which the secret keys are known by the signer.

In this implementation, the matrix of public keys $\left\{P_i^j\right\}_{i=1,\dots,n}^{j=1,\dots,m+1}$ has the following format:

$$\left\{P_i^j\right\} = \begin{pmatrix} P_0^0 & \dots & P_\pi^0 & \dots & P_n^0 \\ \vdots & & \vdots & & \vdots \\ P_0^j & \dots & P_\pi^j & \dots & P_n^j \\ \vdots & & \vdots & & \vdots \\ P_0^m & \dots & P_\pi^m & \dots & P_n^m \\ 0 & \dots & 0 & \dots & 0 \end{pmatrix}$$

Similarly, the vector of secret keys $\bar{x}$ has the following format:

$$\bar{x} = \begin{pmatrix} x_0 \\ \vdots \\ x_j \\ \vdots \\ x_m \\ 0 \end{pmatrix}$$

The goal of `secp256k1_prepare_mlsag` is to fill the last line of both elements.

The last line of the matrix $\left\{P_i^j\right\}_{i=1,\dots,n}^{j=1,\dots,m+1}$ will contain differences of the input and output commitments, as shown below:

[3] https://lab.getmonero.org/pubs/MRL-0005.pdf

$$\left\{P_i^j\right\} = \begin{pmatrix} P_0^0 & \cdots & P_\pi^0 & \cdots & P_n^0 \\ \vdots & & \vdots & & \vdots \\ P_0^j & \cdots & P_\pi^j & \cdots & P_n^j \\ \vdots & & \vdots & & \vdots \\ P_0^m & \cdots & P_\pi^m & \cdots & P_n^m \\ \sum_{j=1}^m c_{in,0,j} - \sum_{k=1}^l c_{out,k} & \cdots & \sum_{j=1}^m c_{in,\pi,j} - \sum_{k=1}^l c_{out,k} & \cdots & \sum_{j=1}^m c_{in,n,j} - \sum_{k=1}^l c_{out,k} \end{pmatrix}$$

The last element of $\bar{x}$ will contain the difference between the input and output blind values:

$$\bar{x} = \begin{pmatrix} x_0 \\ \vdots \\ x_j \\ \vdots \\ x_m \\ \sum_{j=1}^m b_{in,j} - \sum_{k=1}^l b_{out,k} \end{pmatrix}$$

These values will be important for the other two functions detailed in the following parts of this section.

Signature Algorithm

The generation of the MLSAG signature is performed by the function `secp256k1_generate_mlsag`.

The prototype of this function is given below:

```
int secp256k1_generate_mlsag(
    const secp256k1_context *ctx,
    uint8_t *ki, /* I */
    uint8_t *pc, /* pc = c1 */
    uint8_t *ps, /* s */
    const uint8_t *nonce, /* Seed for the random generator */
    const uint8_t *preimage, /* Preimage used in the SHA256 hash */
    size_t nCols, /* number of columns*/
    size_t nRows, /* number of rows */
    size_t index, /* real index of the transaction */
    const uint8_t **sk /* secret key vector */,
    const uint8_t *pk /* public key matrix */
);
```

The correspondences between the inputs of the function and the values in the paper are:

- `ki`, the values I_j, with $j = 1, \ldots, m$,
- `pc`, a pointer to the value c_1,
- `s`, a pointer to the matrix of random values $\left(s_i^j\right)_{i=1,\ldots,n}^{j=1,\ldots,m}$,
- `nCols`, the value n,
- `nRows`, the value $m + 1$,
- `index`, the value π,

- `sk`, the vector of secret keys $\bar{x}$,
- `pk`, the matrix of public keys $\left\{P_i^j\right\}_{i=1,\ldots,n}^{j=1,\ldots,m+1}$.

This section will explain the inner-workings of the function and compare it to the algorithm provided by the reference paper.

`secp256k1_generate_mlsag` starts by checking that the input values respect the appropriate format.

```
if (!pk
    || nRows < 2
    || nCols < 1
    || nRows > MLSAG_MAX_ROWS)
    return 1;
```

It then initializes the random number generator using the `preimage` and the `nonce` discussed in the previous section.

```
memcpy(tmp, nonce, 32);
memcpy(tmp+32, preimage, 32);

/* seed the random no. generator */
secp256k1_rfc6979_hmac_sha256_initialize(&rng, tmp, 32 + 32);

secp256k1_sha256_initialize(&sha256_m);
/* Initialize the SHA256 hash with `preimage` */
secp256k1_sha256_write(&sha256_m, preimage, 32);
sha256_pre = sha256_m;
```

The generation of the signature is then split into two parts and follows the generation performed in the paper.

The algorithm will first compute the values L_π^j, R_π^j and I_j, for $j = 1, \ldots, m$ and with H_p a hash function returning a point, using the following formulae:

$$\begin{aligned} L_\pi^j &= \alpha_j G \\ R_\pi^j &= \alpha_j H_p\left(P_i^j\right) \\ I_j &= x H_p\left(P_i^j\right) \end{aligned}$$

It also updates the SHA256 hash which will be used to compute $c_{\pi+1}$ at the end. Then, one last separate iteration is used to compute L_π^{m+1} and update the SHA256 hash:

$$L_\pi^{m+1} = \alpha_j G$$

Finally, $c_{\pi+1}$ is computed using the SHA256 hash:

$$c_{\pi+1} = \text{SHA256}(P_\pi^0, L_\pi^0, R_\pi^0, P_\pi^1, L_\pi^1, R_\pi^1, \ldots, P_\pi^m, L_\pi^m, R_\pi^m, P_\pi^{m+1}, L_\pi^{m+1})$$

P_π^{m+1} and L_π^{m+1} being the messages signed by the algorithm, with:

$$P_\pi^{m+1} = \sum_{j=1}^{m} c_{in,\pi,j} - \sum_{k=1}^{l} c_{out,k}$$

Note: It was noticed that functions such as `secp256k1_ecmult` and `secp256k1_ec_pubkey_create` are used interchangeably to compute the multiplication between a scalar and a curve point. It seems the operations performed are similar and it is therefore recommended to use only one of the two in order to simplify the code and its understanding.

The second part of the algorithm computes the L_i^j, R_i^j and c_i for all other indices. The formulae change from the ones used for the index π and are given below:

$$\begin{aligned} L_i^j &= s_i^j G + c_i P_i^j \\ R_i^j &= s_i^j H_p\left(P_i^j\right) + c_i I_j \\ c_i &= \text{SHA256}(P_i^0, L_i^0, R_i^0, P_i^1, L_i^1, R_i^1, \ldots, P_i^m, L_i^m, R_i^m, P_i^{m+1}, L_i^{m+1}) \end{aligned}$$

During these computations, when the index is 0, it stores the value c_1 into `clast`.

Finally, the function calculates the remaining s_π^j to make the signature verification algorithm work. These values are computed as follows (with `l` being the order of the *secp256k1* curve):

$$s_\pi^j = \alpha_j - c_\pi x_j \text{mod } l$$

No deviation from the original algorithm was observed and the primitives are used adequately. No security issue affecting the signature generation was found during this audit.

Verification Algorithm

The verification of the MLSAG signature is performed by the function `secp256k1_verify_mlsag`.

An MLSAG signature Σ comprises the following elements:

$$\Sigma = (I_1, \ldots, I_m, c_1, s_0^0, \ldots, s_0^m, s_i^1, \ldots, s_1^m, \ldots, s_n^0, \ldots, s_n^m)$$

The prototype for the function `secp256k1_verify_mlsag` is given below and one can observe that these values are provided to it:

```
int secp256k1_verify_mlsag(
    const secp256k1_context *ctx, /* EC multiplication context */
    const uint8_t *preimage,      /* Preimage used in the SHA256 hash */
    size_t nCols,                 /* n, the number of columns */
    size_t nRows,                 /* m, the number of rows */
    const uint8_t *pk,            /* The matrix of public keys */
    const uint8_t *ki,            /* Vector of I */
    const uint8_t *pc,            /* Pointer to c1 */
    const uint8_t *ps             /* Matrix of s */
);
```

The only goal of this function is to compute all c_i, with $i = 1, \ldots, n+1$, and to verify that $c_1 = c_{n+1}$.

First the function retrieves c_1 and initializes the SHA256 which will be used to compute the subsequent c_i.

```
secp256k1_scalar_set_b32(&clast, pc, &overflow);
if (overflow || secp256k1_scalar_is_zero(&clast))
    return 1;

/* cSig contains c1 */
cSig = clast;

secp256k1_sha256_initialize(&sha256_m);
secp256k1_sha256_write(&sha256_m, preimage, 32);
sha256_pre = sha256_m;
```

Then it iterates over n and m to compute L, R and c_i using the following formulae:

$$
\begin{aligned}
L_i^j &= s_i^j G + c_i P_i^j \\
R_i^j &= s_i^j H\left(P_i^j\right) + c_i I_j \\
c_i &= \mathrm{SHA256}(P_i^0, L_i^0, R_i^0, P_i^1, L_i^1, R_i^1, \ldots, P_i^m, L_i^m, R_i^m, P_i^{m+1}, L_i^{m+1})
\end{aligned}
$$

All user-controlled values are verified within the function and cannot lead to overflows or similar defaults.

Finally, the function verifies that the equality $c_1 = c_{n+1}$ holds true and returns accordingly.

```
/* -c_{1} */
secp256k1_scalar_negate(&cSig, &cSig);
/* Computes c_{n+1} - c_{1} */
secp256k1_scalar_add(&zero, &clast, &cSig);

/* c_{n+1} - c_{1} == 0 ? */
/* Returns 0 on success, 2 on failure */
return secp256k1_scalar_is_zero(&zero) ? 0 : 2;
```

No deviation from the original algorithm was observed and the primitives are used adequately. No security issue affecting the signature generation was found during this audit.

5. Conclusion

The main objective of this audit was to assess the security of two protocols implemented in the Part cryptocurrency by the company Particl and the audit was, as such, divided into two parts.

The first part was to audit Bulletproof, a short non-interactive zero-knowledge proof that allows to verify that a value is in a given range without ever revealing it. The algorithms of proof and verification were assessed to make sure that there was no vulnerability that could allow the creation of a fake proof or a way to retrieve information from it. During the audit, no vulnerability of the sort was found and the report tried as best as possible to highlight that the implementation matches the reference paper.

The second part of the audit focused on MLSAG, a signature scheme that allows the hiding of the amounts, origin and destination of a transaction. Similarly to Bulletproofs, the goal was to identify vulnerabilities that could be exploited to craft a fake signature or to retrieve sensitive information from one. Again, no such vulnerability was found, and the report explains the process applied and provides key elements to show that the implementation is sound.