Evaluation of Bulletproof Implementation

Monero

Ref.	18-06-439-REP
Version	1.2
Date	2018 October 20th
Prepared for	Monero Research Lab
With the support of	OSTIF
Carried out by	Quarkslab

Contents

Project information

Document Status Log			
Version	**Date**	**Status**	**Authors**
0.9	24/07/2018	Delivered to Monero Research Lab	Jean-Baptiste Bédrune Cédric Tessier Marion Videau
1.0	27/07/2018	Reviewed	Sarang Noether
1.1	07/08/2018	Reviewed	Derek Zimmer
1.2	22/10/2018	**Published**	

Quarkslab	
Contact	**Role**
Frédéric Raynal	CEO and Founder
Marion Videau	Chief Scientific Officer
Jean-Baptiste Bédrune	R&D Engineer
Cédric Tessier	R&D Engineer

Monero Research Lab	
Contact	**Role**
Sarang Noether	R&D Engineer

Open Source Technology Improvement Fund	
Contact	**Role**
Derek Zimmer	President and Founder

1. Executive Summary

This report describes the results of the security evaluation by Quarkslab of Monero's implementation of Bulletproof. Three senior engineers reviewed Monero's Bulletproof between April 18 and July 17 for a total of 35 man-days of study. The review target is the C++ code of the https://github.com/moneromooo-monero/bitmonero repository, branch `bp-multi-aggregation`, commit `7f964dfc8f15145e364ae4763c49026a3fab985d`, directory `src/ringct`. The assessment included verifying that the implementation correctly reflected the algorithms described in the original academic paper and looking for vulnerabilities by code review, manual testing and fuzzing.

Bulletproof is a new non-interactive zero-knowledge proof protocol with short proofs and without trusted setup. It is integrated in the Monero project as a replacement for the previous protocol based on ring signatures which generates larger proofs. Bulletproof proves that amounts lie in a given positive interval, which is crucial in validating a transaction. Without this proof, due to the elaborated cryptographic machinery involved, it is possible to create fraudulent coins.

Three categories of concerns emerged from the evaluation work:

Improvements towards a simpler and more robust code. They revolve around adding abstraction layers to the current implementation, building specifications and a few other simplification and performance improvements for a more robust code.

Weaknesses that could turn into vulnerabilities during code evolutions. They coalesce around three topics: insufficient input and boundary validation, lack of edge case and failure case testing and fragilities in the implementation choices. They could induce vulnerabilities in future evolutions of Monero.

Vulnerabilities that should be fixed in priority. In the limited time frame of the assessment, we preferred to focus on a search for weaknesses as wide as possible and did not dig into exploitability. Hence none of the vulnerabilities found has lead to a practical exploit. However, it does not mean it is impossible, especially when considering all the vulnerabilities directly linked to inputs controled by an attacker. We gather in the following the vulnerabilities and the associated recommendations to fix them:

- **Major vulnerabilities that can be triggered by untrusted inputs.** They could be the first steps towards making a verifier accept a false proof.

Arithmetic overflow in the double scalar multiplication							
Recommendation. Always reduce inputs modulo ℓ when calling the double scalar multiplication function.							
Class	**Data validation**	Severity	**Critical**	Difficulty to trigger	**Low**	Difficulty to exploit	**Unknown**

Erroneous identity output in Bos-Coster multi-exponentiation							
Recommendation. Fix the implementation.							
Class	**Arithmetic**	Severity	**Critical**	Difficulty to trigger	**High**	Difficulty to exploit	**Unknown**

Silently discarded element in Pippenger multi-exponentiation							
Recommendation. Fix the implementation.							
Class	**Arithmetic**	Severity	**Critical**	Difficulty to trigger	**High**	Difficulty to exploit	**Unknown**

Invalid input parameters in the function `bulletproof_VERIFY` that verifies a bulletproof							
Recommendation. Add checks that the scalars are reduced, that the points are on the right curve Ed25519 and lie in the main subgroup.							
Class	Data Validation	Severity	High	Difficulty to trigger	Low	Difficulty to exploit	Unknown

- **Minor vulnerabilities in (de)serialization procedures.** Because deserialization occurs on untrusted inputs, the bugs can lead at least to exceptions and denials of service.

Failures in input size validation during deserialization							
Recommendation. Fix the code.							
Class	Data validation	Severity	Medium	Difficulty to trigger	Low	Difficulty to exploit	Unknown

Failures in input type validation during deserialization							
Recommendation. Fix the code.							
Class	Data validation	Severity	Medium	Difficulty to trigger	Low	Difficulty to exploit	Unknown

Failures in input size validation during containers deserialization							
Recommendation. Fix the code.							
Class	Data validation	Severity	Informational	Difficulty to trigger	Low	Difficulty to exploit	Unknown

- **Vulnerabilities that only happen with a negligible probability.** Although highly unlikely to happen by chance, code robustness requires to check for such events.

Zero value challenges can be produced in the bulletproof protocol							
Recommendation. Generate and test hash values such that it is possible to regenerate a new challenge value if null.							
Class	Data validation	Severity	High	Difficulty to trigger	High	Difficulty to exploit	Unknown

Missing checks in Schnorr signature							
Recommendation. Add a check that the random scalar `k` is non null (against key compromise). Add a check that the signature value `sig.c` is non null (against producing signature that does not depend on the private key). In the verification, add a check that the resulting point $r.G + c.\texttt{pub}$ is non null.							
Class	Data validation	Severity	High	Difficulty to trigger	High	Difficulty to exploit	Unknown

- **A vulnerability in the cryptographic implementation of a major setup element** of the proof system in the Java implementation.

Generators of the subgroup are dependant in the Java implementation							
Recommendation. Do not use the Java implementation either for production or as specifications.							
Class	Crypto	Severity	Undef.	Difficulty to trigger	Low	Difficulty to exploit	Unknown

2. Context

In January 2018, Monero Research Lab, through the Open Source Technology Improvement Fund, asked Quarkslab for a statement of work detailing the steps of a security evaluation. The target, in the Monero open-source cryptocurrency, was their implementation of a new mechanism, the Bulletproof.

2.1 Initial request

The Monero project has implemented a new cryptographic proof for Monero (XMR), an open-source cryptocurrency and plans to shift to it.

The Monero project currently uses Borromean-style range proofs [MP15] in their confidential transactions, and plan to replace them with bulletproofs[1]. Their motivation to move from Borromean range proofs to bulletproofs is the size of the proof: bulletproofs would significantly reduce the size of the blockchain, as well as bring down transaction fees on the platform by an estimated 70-80%.

The Monero Research Lab asked for a limited-scope analysis of the bulletproof prove/verify algorithms, answering as many of the following questions as possible:

1. Does their code located at https://github.com/moneromooo-monero/bitmonero/tree/bp-multi/src/ringct accurately represent the prove/verify algorithms from the bulletproof whitepaper located at http://web.stanford.edu/~buenz/pubs/bulletproofs.pdf?
2. Does their implementation allow an attacker to generate a false proof that an honest verifier judges as correct?
3. Does their implementation allow an attacker to examine an honest prover's proof and gain information about the hidden amount or other masks?

In order to test correctness, the original whitepaper prove/verify routines has been translated into Java code located at https://github.com/b-g-goodell/research-lab/blob/master/source-code/StringCT-java/src/how/monero/hodl/bulletproof/MultiBulletproof.java. The code could be used as extra material to help bridge the gap between the paper and the final code.

2.2 Evolution of the request and complementary evaluations

During the process of answering and the selection of evaluators, the academic paper at the origin of Bulletproof and the code to evaluate have evolved, the paper being accepted to an academic conference and the code including several optimizations.

The selected evaluators are:

- Benedikt Bünz, one of the original authors of the bulletproof paper, to check that the Java implementation correctly reflects the paper;
- Kudelski Security and Quarkslab to evaluate that the C++ code reflects the Java code and that the C++ code does not contain vulnerabilities allowing an attacker to either forge a false proof or derive knowledge of hidden amounts from a proof.

[1] Bulletproofs webpage: https://crypto.stanford.edu/bulletproofs/

2.3 Quarkslab's planned work

The evaluation work that Quarkslab planned includes the three following steps:

- Understanding the protocols and isolating the main points of attention regarding implementation. It is important to note that the bulletproofs research results are recent. At the time of the request, they were to be published at IEEE S&P 2018 [BBBPWM18] and the research paper was available at https://eprint.iacr.org/2017/1066.pdf[2].
- Assessing the conformity of the C++ code (`ringct` amounts to around 3500 lines) to the *specifications*[3] (and the reference source code) both from a logical and an implementation standpoint, including the underlying elliptic curve arithmetic used.
- Looking for vulnerabilities and assessing their severity.

2.4 Role of Bulletproof in Monero

Monero is a cryptocurrency whose goal is to provide anonymity to users and confidentiality to transaction amounts. The coin generation happens through a trustless distributed *mining* process relying on a *proof of work*. A transaction registers the spending of *inputs* on *outputs*. The right to spend is afforded by a private signing key corresponding to the public key attached to previous outputs.

Monero relies on three cryptographic mechanisms.

- *One-time keys* generated for each transaction hide the actual recipient of a transaction.
- *Ring signatures* mix the spender's input among other people's inputs (which are hidden, see below). The spender can spend (sign) the amount spent but it is not possible for an external party to link different transactions. A special adaptation of this mechanism detects double spending.
- *Ring confidential transactions* hide the transferred amount.

Ring confidential transactions [NMM16] use zero-knowledge proof techniques (Perdersen commitments) to hide amounts and also keep the verifiability of the blockchain.

In short, a transaction is valid if the total of inputs equals the total of outputs and fees. This means that the total amount of inputs minus the total amount of outputs minus the fees equals zero, which can also be committed to by zero-knowledge techniques.

Such techniques however, relying on group based cryptography, do not differentiate between a small negative amount or a big positive amount (due to modular arithmetic), which could cause, left unchecked, the fraudulent creation of coins.

To ensure that amounts spent are indeed reasonably positive amounts (and not huge amounts equivalent to negative ones) without revealing them, a proof of interval is necessary for each output amount. In Monero, the interval has been fixed to $[0, 2^{64} - 1]$. A first version of a proof of interval implemented in Monero also used ring signature techniques. The size of this proof

[2] The version of the paper corresponding to the line numbering reported in the Java code used for reference is the one of November 2017, 11th available at https://eprint.iacr.org/eprint-bin/getfile.pl?entry=2017/1066&version=20171110:151138&file=1066.pdf

[3] In reality, academic papers are rarely specifications and that is one of the many difficulties in assessing the code derived from them. In some way, the specifications need to be built alongside the evaluation.

was linear in the size of the upper-bound of the interval and the major contributor to the size of a transaction.

Bulletproof is a new proof of interval whose size is only logarithmic in the size of the upper-bound of the interval. It has further optimizations reducing the overall size when several proofs are combined.

3. Code overview

The code studied is in the branch `bp-multi-aggregation` of the repository https://github.com/moneromooo-monero/bitmonero.git[1].

The last commit taken into account in the branch `bp-multi-aggregation` is `7f964dfc8f15145e364ae4763c49026a3fab985d`

To extend the work done on multi-exponentiation during the evaluation, we also studied the code in the branch `bp-multi-aggregation-pippenger`.

The last commit taken in the branch `bp-multi-aggregation-pippenger` is `b7e61db030da8c97b3e82354bfee8caae57d3137` (Wed Jun 20).

The source code snippets and pieces of notation in the report refer to the commit `b7e61db030da8c97b3e82354bfee8caae57d3137` (Wed Jun 20). There were only minor differences on the specific points detailed in the report and because all findings were still present in this latter commit, we opted to refer to it instead. However, it does not mean that our evaluation corresponds to the assessment of this latter commit.

3.1 Structure

The code of Bulletproof is in the directory `src/ringct` (ringct stands for Ring Confidential Transaction). The structure of this directory is :

```
src/ringct/
├── bulletproofs.cc
├── bulletproofs.h
├── CMakeLists.txt
├── multiexp.cc
├── multiexp.h
├── rctCryptoOps.c
├── rctCryptoOps.h
├── rctOps.cpp
├── rctOps.h
├── rctSigs.cpp
├── rctSigs.h
├── rctTypes.cpp
└── rctTypes.h
```

The code in `src/ringct/` mainly depends on:

- cryptographic functions defined in `src/crypto/`,
- utility functions defined in `src/common/`, `src/serialization/` and `contrib/epee/include/`,
- general formats and functions defined in `src/cryptonote_config.h` and in `src/cryptonote_basic/cryptonote_format_utils.h`
- external libraries: `boost` and `openssl/ssl.h`.

At a high level, files in the `src/ringct` directory are organized as follows:

[1] Fork from the main Monero repository https://github.com/monero-project/monero.

- `bulletproofs.h` declares the two main functions `bulletproof_PROVE` and `bulletproof_VERIFY` with variants depending on the input parameters.
- `multiexp.h` declares the structure and the functions used for multi-exponentiation. The three algorithms implemented are Straus, Bos-Coster and Pippenger.
- `rctCryptoOps.h` declares the function `sc_reduce32copy(unsigned char * scopy, const unsigned char *s)` which is a variant of `sc_reduce32(unsigned char *s)` in `src/crypto/crypto-ops.h` providing the result in `scopy`. It is a reduction modulo $\ell = 2^{252} + 27742317777372353535851937790883648493$ (order of the main subgroup of the curve Ed25519) of a 32-byte input.
- `rctOps.h` declares constants and functions related to the manipulation of vectors or points (initialization, random generation, addition, multiplication, commitments, hash-to-point, etc.)
- `rctSigs.h` declares functions related to the Multilayered Spontaneous Anonymous Group Signatures (MLSAG signatures) which allows the confidential transactions. It also contains the former range proof and verification functions relying on ring signatures that should be replaced by bulletproofs.
- `rctTypes` defines all the objects (key, signature, tuple, etc.) in the `rct` namespace and conversion functions.

3.2 The two main algorithms

Both the version of the paper we relied on for the evaluation[2] and the code we evaluated mix different levels of concepts and abstractions with implementation details. Consequently, understanding the role of different operations is more difficult.

We propose a **pseudo-code** description that we derived from our understanding of the paper, at a level of abstraction corresponding to the concepts used. Consequently, it can differ on some points from the implementation in Monero. We named the functions as explicitly as possible and chose to rely on elliptic curve vocabulary instead of a general group one. Some implementation details are discussed in the next chapters.

The range one wants to prove is $[0, 2^{64} - 1]$. We focus on the case of multiple proofs that are aggregated. Important pieces of notations are regrouped below.

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 wrt `G` is not known and hard to find

[2] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille and Greg Maxwell. "Bulletproofs: Short Proofs for Confidential Transactions and More," Version of 21 May 2018, of IACR eprint server https://eprint.iacr.org/2017/1066.pdf

- `Gi`: a list of `M*N` generators of the subgroup of the elliptic curve used whose discrete log wrt any other generator is not known and hard to find
- `Hi`: a list of `M*N` generators of the subgroup of the elliptic curve used whose discrete log wrt 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 <= v[j] < 2^N`
- `gamma`: a list of `M` integers such that for all `j, 0 <= 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 `tau1`
- `T2`: a curve point, Pedersen commitment to `t2` with hiding value `tau2`
- `taux`: 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
- `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

3.2.1 Prove

The function `bulletproof_PROVE` takes as input a `v` and a `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
```

(continues on next page)

(continued from previous page)

```
// 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
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)))

     // 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
```

3.2.2 Verify

The verification algorithm takes as input a list of bulletproofs and rely 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
```

(continues on next page)

(continued from previous page)

```
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
```

4. Evaluation overview

4.1 Hypothesis

Monero and Quarkslab agreed on the following hypothesis:

- The underlying principles coming from the paper are sound: we do not evaluate the paper cryptographic content.
- The Java implementation is sound: we do not evaluate the translation of the paper in Java.
- We are not considering side-channel attacks (local timing attacks, cross VM attacks, etc.) on the device manipulating secret values (the prover's).

4.2 Methodology

Security review for conformity and lack of vulnerabilities on Monero's Bulletproof implementation meant:

- Understanding the general cryptographic principles behind the algorithms.
- Code reviewing by comparison between the Java code and the C++ code and between the paper and the C++ code, alongside the reconstruction of missing specifications.
- Assessing arithmetic operations conformity [CDF].
- Fuzzing multi-exponentiation functions.
- Fuzzing of data serialization.

4.3 Topics covered

In this section we describe the different topics we covered in our review. Vulnerabilities and weaknesses are detailed in the next chapters.

4.3.1 Hash function

Bulletproof depends on a hash function to turn the interactive protocols into non-interactive ones using Fiat-Shamir heuristic. Besides, in Monero, such a hash function is also used as a subroutine in many other functions: hashing (to points for instance), random generation and point derivation.

In Monero's Bulletproof, the underlying hash function is [Keccak]. Keccak is based on a sponge construction, a class of algorithms that produce a pseudorandom bit stream of a chosen length from an input bit stream of arbitrary length. To achieve this, a finite internal state is processed using the Keccak-f[1600] permutation.

Keccak has been chosen by NIST as the basis of its SHA-3 standard. Choosing Keccak as the underlying hash function in Monero is a sound choice from a cryptographic point of view.

A review on this choice is detailed in *Use of near-standards: Keccak vs SHA-3*.

4.3.2 Random generation

In Bulletproof, a prover builds a proof that an amount v is in a given interval. The security of the proof relies on many random values used to hide critical variables in the protocol. For instance, the hiding value γ is used in the Pedersen commitment of v. Knowing γ would allow an attacker to retrieve the value of v through brute force, compromising the confidentiality of the committed amount.

Other random values, α, ρ, τ_1, τ_2, $\mathbf{s}_L$ and $\mathbf{s}_R$ also protect the security of the protocol and should not be easy to guess. A strong PRNG is therefore a critical requirement for Bulletproof.

Random numbers are generated using `rct::skGen`, with a variant `rct::skvGen` producing a vector. Both functions are calling `crypto::rand`, the main random generator of Monero.

Listing 4.1: `src/crypto/crypto.h:152`

```
/* Generate N random bytes
*/
inline void rand(size_t N, uint8_t *bytes) {
  generate_random_bytes_thread_safe(N, bytes);
}
```

The function `generate_random_bytes_thread_safe` is a simple wrapper, calling `generate_random_bytes_not_thread_safe`, making it thread-safe with a lock.

Listing 4.2: `src/crypto/crypto.cpp:89`

```
void generate_random_bytes_thread_safe(size_t N, uint8_t *bytes)
{
  static boost::mutex random_lock;
  boost::lock_guard<boost::mutex> lock(random_lock);
  generate_random_bytes_not_thread_safe(N, bytes);
}
```

The low-level routine `generate_random_bytes_not_thread_safe` uses iterations of the permutation Keccak-f[1600] on a global state. Each iteration produces a maximum of 136 bytes (= 1088 bits), which corresponds to the security parameters of `Keccak [1088,512]`. The production of pseudorandom bits is the *squeezing* phase in the sponge construction vocabulary.

This construction is sound as long as the global state is initialized with true random bits coming from the system.

A weakness in the initialization procedure is detailed in *Random generation initialization*.

4.3.3 Protocol challenges

One of the possibilities to turn an interactive protocol into a non-interactive one is to use the Fiat-Shamir heuristic and replace all random challenges by hashes of the transcript up to that point.

In Monero it is done through the call to various hash variants using at their core the hash function Keccak.

Listing 4.3: `src/ringct/bulletproofs.cc:951`

```
rct::key hash_cache = rct::hash_to_scalar(proof.V);
rct::key y = hash_cache_mash(hash_cache, proof.A, proof.S);
rct::key z = hash_cache = rct::hash_to_scalar(y);
rct::key x = hash_cache_mash(hash_cache, z, proof.T1, proof.T2);
rct::key x_ip = hash_cache_mash(hash_cache, x, proof.taux, proof.mu, proof.t);
w[i] = hash_cache_mash(hash_cache, proof.L[i], proof.R[i]);
```

A security concern is detailed in *Zero value challenges*. Implementation choices are discussed in *Computation of challenges* and *Code factoring*.

4.3.4 Generators of the main subgroup of Ed25519

Bulletproof does not need a trusted setup. As explained in the article:

> *To avoid a trusted setup we can use such a hash function to generate the public parameters* $\mathbf{g}, \mathbf{h}, g, h$ *from a small seed. The hash function needs to map from* $\{0,1\}^*$ *to* $\mathbb{G} \setminus \{1\}$*, which can be built as in [BLS01]*

The choice of the generators is of paramount importance for the security of bulletproofs. All points must be on the main subgroup of the curve Ed25519 and none should have any relation that allows to deduce a discrete log from one another.

The main subgroup of Ed25519 being of prime order, all points generated on this subgroup are generators.

Generators `H` and `G`

Point `G` is the base point of the curve Ed25519.

Point `H` is precomputed and its value is stored in `src/ringct/rctTypes.h`.

Listing 4.4: `src/ringct/rctTypes.h:417`

```
//other basepoint H = toPoint(cn_fast_hash(G)), G the basepoint
 static const key H = { {0x8b, 0x65, 0x59, 0x70, 0x15, 0x37, 0x99, 0xaf, 0x2a, 0xea,␣
↪0xdc, 0x9f, 0xf1, 0xad, 0xd0, 0xea, 0x6c, 0x72, 0x51, 0xd5, 0x41, 0x54, 0xcf, 0xa9,␣
↪0x2c, 0x17, 0x3a, 0x0d, 0xd3, 0x9c, 0x1f, 0x94} };
```

Assessing the creation of `H` is necessary. It could be suspicious, knowing that the function `toPoint()` does not exist and `G` is never stored in an object.

The origin of `H` has been found in the unit tests.

Listing 4.5: `tests/unit_tests/ringct.cpp:814`

```
TEST(ringct, HPow2)
{
   key G = scalarmultBase(d2h(1));

   key H = hashToPointSimple(G);
   for (int j = 0 ; j < ATOMS ; j++) {
      ASSERT_TRUE(equalKeys(H, H2[j]));
```

(continues on next page)

(continued from previous page)

```
      addKeys(H, H, H);
   }
}
```

The first element of `H2` is `H` and its value is actually the value stored in `src/ringct/rctTypes.h`. The representation in an object `key` of `G` is created then hashed with `hashToPointSimple()`.

Listing 4.6: `src/ringct/rctOps.cpp:393`

```
key hashToPointSimple(const key & hh) {
    key pointk;
    ge_p1p1 point2;
    ge_p2 point;
    ge_p3 res;
    key h = cn_fast_hash(hh);
    CHECK_AND_ASSERT_THROW_MES_L1(ge_frombytes_vartime(&res, h.bytes) == 0, "ge_
↪frombytes_vartime failed at "+boost::lexical_cast<std::string>(__LINE__));
    ge_p3_to_p2(&point, &res);
    ge_mul8(&point2, &point);
    ge_p1p1_to_p3(&res, &point2);
    ge_p3_tobytes(pointk.bytes, &res);
    return pointk;
}
```

The function `hashToPointSimple()` hashes `G`'s binary representation then embeds the resulting string into a curve point with `ge_frombytes_vartime()`. The function `ge_frombytes_vartime()` converts bytes into a group element. The resulting element is multiplied by 8 to ensure it is on the main subgroup of the curve Ed25519. Such a construction precludes any easy discrete log relation between `H` and `G`.

On the contrary, the Java implementation which was supposed sound in the hypothesis is not. The vulnerability is described in *Dependent generators in the Java implementation*.

Vectors of generators `Hi` and `Gi`

Besides generators `H` and `G`, two vectors of generators, `Gi` and `Hi`, of size `M*N` (hardcoded constants `M=16` and `N=64`) are needed to prove up to `M` aggregated proofs simultaneously.

These generators are produced according to the following procedure(in **pseudo-code**).

```
for (i from 0 to M*N)
    Hi[i] = hashToPoint(hash(concat(bytes(H), "bulletproof", bytes(2*i))))
    Gi[i] = hashToPoint(hash(concat(bytes(H), "bulletproof", bytes(2*i+1))))
```

Every element generated is unique thanks to the use of an unique index. The parameters used to form the *seed* are simple enough to be harmless. They allow to get rid of any trusted set-up. The use of the hash function ensures there is no discrete log relation between the generators. The multiplication by 8 in the `hashToPoint` function ensures the points all lie on the main subgroup of the curve Ed25519.

We checked that all generators produced are not the point at infinity.

Suggestion to check if points produced are at infinity is described in *Generators at infinity*.

4.3.5 Arithmetic operations

Low-level arithmetic operations originates from the `ref10` implementation of the signature scheme Ed25519. This implementation is part of the SUPERCOP[1] benchmarking framework. It is considered the reference implementation for Ed25519.

In Monero, it has been adapted in the `src/crypto/crypto_ops_builder/ref10CommentedCombined/` directory for build reasons. In RingCT, in `src/ringct/rctOps.cpp`, higher level functions are built on top of `ref10` functions. Functions are simple, amounting to less than 10 lines of code each. Functions specific to bulletproofs, like inner product or Hadamard product, are defined in `src/ringct/bulletproof.cc`.

We have tested or reviewed the three categories of arithmetic functions. We did not find any problems in `ref10`.

We describe a vulnerability in *Overflow in the double scalar multiplication* and discuss a weakness in *Code dependencies: OpenSSL*.

While investigating arithmetic operations, we also reviewed the Schnorr signature implementation. Even though it is not formally part of the evaluation, it depends on `ge_scalarmult_base()` like the double scalar multiplication. A lack of checks in the signature generation is described in *Key compromise in Schnorr signature*.

4.3.6 Multi-exponentiation

Multi-exponentiation computes *simultaneously* multiple exponentiations of different elements of a group with different exponents, in a much faster way than a naive approach.

In Bulletproof, multi-exponentiation is at the heart of the proof verification algorithm. The batch verification of multiple aggregated proofs combines gracefully with a simple verification of aggregated proofs in a single large multi-exponentiation.

In the branch `bp-multi-aggregation` we had to evaluate, the Straus algorithm is used up to a given number of exponentiations, then from this number, Bos-Coster algorithm is preferred.

In the new branch `bp-multi-aggregation-pippenger`, the Straus algorithm is used up to another number of exponentiations, then from this number, Pippenger algorithm is preferred.

The implementation of the Straus and the Bos-Coster algorithms were the only ones included in the formal evaluation we had to perform. Knowing that the implementation of the Pippenger algorithm was available and ready to be integrated, we extended the work done on the multi-exponentiation code in the branch `bp-multi-aggregation-pippenger`.

The Straus algorithm has been reviewed and no vulnerability has been found.

Two major vulnerabilities have been discovered in the other multi-exponentiation functions. They are detailed in *Erroneous identity output in Bos-Coster multi-exponentiation* and *Silently discarded element in Pippenger multi-exponentiation*.

A suggestion of performance improvement when testing points at infinity is described in *Test of points at infinity*.

[1] eBACS: ECRYPT Benchmarking of Cryptographic Systems. https://bench.cr.yp.to/supercop.html. Source code available on a GitHub mirror.

4.3.7 Prove and Verify algorithms

The functions `bulletproof_PROVE` and `bulletproof_VERIFY` have been carefully reviewed in order to assess their conformity to the cryptographic and algorithmic principles of the paper. In *The two main algorithms*, we provide pseudo-code reflecting their behavior.

We considered that the dead code in `bulletproof_VERIFY` related to the inner-product verification is a trace of work in progress and would be removed. We did not investigate its content.

We describe a vulnerability related to the lack of input validation in the function `bulletproof_VERIFY` in *Invalid Verify input parameters*.

We discuss many weaknesses of the current code in *Checks and input validation*, *Edge cases and failure cases*, *Fragilities*, *Missing abstraction layers*, *Lack of specifications* and *Simplifications and performance suggestions*.

4.3.8 Serialization

Serialization is used to transmit and store structured data in the blockchain; therefore its robustness is critical. Monero serialization library is inspired by the boost::serialization framework.

It provides a generic implementation, that can be specialized for various objects. Serializing a high level object reuses all low-level specializations for types it depends on. A complementary goal is to describe how an object needs to be dealt with only once. It means the same code is used in order to generate both serialization and deserialization routines. This nice feature, in theory, comes with various downsides.

First, partially because it is based on C++ template, code readability is questionable. When reading this code, someone may think it will serialize an empty block:

Listing 4.7: `src/cryptonote_basic/cryptonote_format_utils.cpp`

```
binary_archive<false> ba(ss);
cryptonote::block b;
bool r = ::serialization::serialize(ba, b);
```

In fact, because the code uses the `binary_archive<false>` template specialization, the `serialize` function will unserialize the archive to a block.

All common code needs to be written very carefully in order to work both ways. It is a major drawback. As an example, the following code will behave very differently if executed for serialization or deserialization.

Listing 4.8: `src/ringct/rctTypes.h:315`

```
uint32_t nbp = bulletproofs.size();
FIELD(nbp)
```

During serialization, it will initialize a variable with the current size of a vector, and then add this value to the archive stream. But when unserializing, the variable's initial value will be 0 (because vector is yet to be parsed), before being set based on archive data.

If code complexity increases or checks are added, ensuring a correct code becomes more difficult and so is error handling. In case of failure, it could result in partially initialized objects (deserialization) or archives (serialization).

For all those reasons, we briefly audited the serialization, even if it was not directly linked with bulletproof. In addition to code review, related routines have been fuzzed in various ways, using both AFLFast and libFuzzer. Various objects (block, transaction, ...) have been targeted, using *clang* code coverage capabilities to steer the fuzzing campaign. Some issues have been found, and even if none of them is really critical, more attention should be paid to this component.

We describe three minor vulnerabilities in the serialization/deserialization procedures in *Failures in input size validation during deserialization* and *Failures in input type validation during deserialisation*.

5. Vulnerabilities

Vulnerabilities are actual bugs, errors or omissions in the code that can have a security consequence.

5.1 Zero value challenges

Class	Severity	Difficulty to trigger	Difficulty to exploit
Data validation	High	Extremely High	Unknown

All the challenge values must be non null for the security of the protocol.

In Bulletproof, they are all produced eventually by a call to `rct::hash_to_scalar()` and the resulting values are not checked. Although highly improbable, a zero value challenge would ruin the validity of the proof.

A null `y` or `w[i]` would also be impossible to invert, raising an exception later in the code where their inverses are needed. We did not investigate further the possibilities opened by such failures.

We advise to concatenate an index at the end of the data to hash and increment it until the resulting output is non null. The cost of such an adaptation and check would be negligible.

5.2 Dependent generators in the Java implementation

Class	Severity	Difficulty to trigger	Difficulty to exploit
Cryptography	Undefined	Low	Unknown

The implementation being different from the main code, we rated the severity as undefined. However, one must be very careful with this implementation which was being presented as a basis to implement the main code.

The Java implementation is **not secure**. It must only be used for tests and never in production. It must not be blindly used as a kind of specifications for other implementations, either. Indeed, the discrete logarithm of `H` with respect to `G` is straightforward.

Listing 5.1: `StringCT-java/src/how/monero/hodl/bulletproof/OptimizedLogBulletproof.java:493`

```
// Set the curve base points
G = Curve25519Point.G;
H = Curve25519Point.hashToPoint(G);
```

The function `hashToPoint(G)` in this case hashes the representation of `G`, gets a scalar and multiply it by `G` to get `H`.

Listing 5.2: StringCT-java/src/how/monero/hodl/crypto/Curve25519Point.java:70

```
public static Curve25519Point hashToPoint(byte[] a) {
  return BASE_POINT.scalarMultiply(hashToScalar(a));
}
public static Curve25519Point hashToPoint(Curve25519Point a) {
  return hashToPoint(a.toBytes());
}
```

The discrete logarithm of `H` with respect to `G` is `hashToScalar(G.toBytes())`.

5.3 Overflow in the double scalar multiplication

Class	Severity	Difficulty to trigger	Difficulty to exploit
Data validation	Critical	Low	Unknown

The `rct::addKeys2` function computes, for `a` and `b` scalars, and `H` a curve point, the value of $aG + bH$, with `G` the base point.

The function `rct::addKeys2` is called in `bulletproof_VERIFY` with parameters under direct control of the prover (τ_x and t for example). Prover can send data that will be incorrectly handled. So it is very important that such a function is correctly used.

The function `rct::addKeys2` is also called in the two versions of `bulletproof_PROVE` to compute the Pedersen commitment of an amount (resp. vector of amounts) and a hiding value (resp. vector of hiding values). The function is called with a non-reduced scalar parameter (resp. a vector of non-reduced scalar parameters). One example of such a call is given below:

Listing 5.3: src/ringct/rctSigs.cpp:48

```
Bulletproof proveRangeBulletproof(key &C, key &mask, uint64_t amount)
{
   mask = rct::skGen();
   Bulletproof proof = bulletproof_PROVE(amount, mask);
```

The double scalar multiplication is performed by the `ge_double_scalarmult_base_vartime` function. In order to perform the computations, `a` and `b` are converted into their *non-adjacent form* (or NAF), a unique signed-digit representation of a number which is characterized by the fact that two non-zero digits cannot be adjacent. This can be illustrated with a simple example:

$$7 = (100 - 1)_2 = 2^3 - 2^0 = 8 - 1$$

The implementation uses an optimized version of NAF called w-ary *non-adjacent form* (or w-NAF), with $w = 5$. To accelerate scalar-point multiplications, a scalar n is translated into an array of r_i, such that $\sum_0^t r_i \times 2^i = n$, where $r_i \in [-2^w + 1, 2^w - 1]$ and t is the size of the binary representation of n (the size of the w-ary non-adjacent form of n is $t+1$). The current implementation of the 5-NAF representation, the function `slide` in `src/crypto/crypto_ops_builder/ref10CommentedCombined/ge_double_scalarmult.c`, has a fixed size of 256 signed char.

When a scalar (of type `key`), say `a`, provided as input to the function `rct::addKeys2` is such that `a[31] >= 128`, some of its values will have a w-NAF of size strictly 257 which overflows

the representation on 256 chars. The 5-NAF representation produced by the implementation is the one of $a - 2^{256}$ instead of a. In those cases, the double multiplication result will be wrong.

Other functions calling `ge_double_scalarmult_base_vartime` in the Monero project are also at risk if called with non-reduced inputs.

In order to avoid this problem, `a` and `b` must be reduced in the function prologue, modulo `l`, $\ell = 2^{252} + 27742317777372353535851937790883648493$, the order of the main subgroup of Ed25519. It's the purpose of `sc_reduce32`.

We did not find a way to exploit this vulnerability, but non-trusted inputs should always be handled with great care.

Note: Other functions (`ge_scalarmult`, `ge_scalarmult_base`) have `a[31] <= 127` as a precondition. This is never checked in the bulletproof implementation.

However it seems that it is not possible to call those functions with an invalid scalar, all numbers in input being always reduced modulo `l` by another computation before.

5.4 Erroneous identity output in Bos-Coster multi-exponentiation

Class	Severity	Difficulty to trigger	Difficulty to exploit
Arithmetic	Critical	High	Unknown

Note: We will focus only on the `rct::bos_coster_heap_conv_robust` function, as `rct::bos_coster_heap_conv` is not used anywhere and tend to enter in an infinite loop very easily (and as a consequence was considered broken and "yet to be removed" legacy code).

A major issue has been identified in the robust variant of the Bos-Coster algorithm implementation. It's located in an optimization that has been added to avoid useless computations. Indeed, if an input is either a null scalar or a point at infinity, the result of the exponentiation with this input will be a point at infinity (the identity); therefore, it can be skipped.

If everything is to be skipped (nothing remaining after filtering), `identity()` is simply returned. In case only one scalar / point pair is left, the idea is to return the unique scalar multiplication using directly `ge_scalarmult`.

Listing 5.4: `src/ringct/multiexp.cc:225`

```
if (points < 2)
 {
   ge_p2 p2;
   ge_scalarmult(&p2, data[0].scalar.bytes, &data[0].point);
   rct::key res;
   ge_tobytes(res.bytes, &p2);
   return res;
 }
```

However, the current implementation does not perform the scalar multiplication with the only valid input, but with the first. If it is not the valid one (hence meaning that it was skipped),

identity will be wrongfully returned.

The current implementation is easy to fix, by getting the position from the list of valid exponentiations (`heap`):

Listing 5.5: Fix suggestion for `src/ringct/multiexp.cc:225`

```
if (points < 2)
 {
   ge_p2 p2;
   // perform scalar multiplication with the only input left in heap
   ge_scalarmult(&p2, data[heap[0]].scalar.bytes, &data[heap[0]].point);
   rct::key res;
   ge_tobytes(res.bytes, &p2);
   return res;
 }
```

5.5 Silently discarded element in Pippenger multi-exponentiation

Class	Severity	Difficulty to trigger	Difficulty to exploit
Arithmetic	Critical	High	Unknown

Pippenger is the latest multi-exponentiation algorithm added to Bulletproof, with a higher efficiency when dealing with a lot of inputs (> 64 in the current implementation).

A major issue has been found in the Bulletproof Pippenger implementation, where some selected inputs can return an invalid result.

At the beginning of the algorithm, a value c is chosen in the range $[0, 9]$ (based on the number of inputs). It will be the size of groups of bits during the decomposition of an exponent, one of the essential computations performed by Pippenger.

With $c = 3$, this step can be summarized with an example, where the term a^{85} is decomposed in 3 groups:

$$a^{85} = a^{\left((85)_2 = \underbrace{0\,0\,1}_{1\cdot 2^{2\cdot 3}=1\cdot 64}\ \overbrace{0\,1\,0}^{2\cdot 2^{1\cdot 3}=2\cdot 8}\ \underbrace{1\,0\,1}_{5\cdot 2^{0\cdot 3}=5\cdot 1}\right)} \Rightarrow (a^{64})^1 (a^8)^2 (a)^5$$

But in current code, exponents are not encoded on 9 bits, but on 256 bits.

$$85 = \overbrace{\cdots}^{0\cdot 2^{5\cdot 3}}\ \underbrace{0\,0\,0}_{0\cdot 2^{4\cdot 3}=0\cdot 4096}\ \overbrace{0\,0\,0}^{0\cdot 2^{3\cdot 3}=0\cdot 512}\ \underbrace{0\,0\,1}_{1\cdot 2^{2\cdot 3}=1\cdot 64}\ \overbrace{0\,1\,0}^{2\cdot 2^{1\cdot 3}=2\cdot 8}\ \underbrace{1\,0\,1}_{5\cdot 2^{0\cdot 3}=5\cdot 1}$$

So an optimization is performed, in order to avoid dealing with all leading zeros. The idea is to find the most significant bit in all input exponents, and use its position to limit the number of groups.

This is what is done by the code below (all comments have been added for clarity and are not in the original code):

Listing 5.6: `src/ringct/multiexp.cc:594`

```
// find the greatest exponent
rct::key maxscalar = rct::zero();
for (size_t i = 0; i < data.size(); ++i)
{
  if (maxscalar < data[i].scalar)
    maxscalar = data[i].scalar;
}
// compare it with powers of 2 in order to find the most significant bit position
size_t groups = 0;
while (groups < 256 && pow2(groups) < maxscalar)
  ++groups;
// limit the number of groups needed
groups = (groups + c - 1) / c;
```

Unfortunately, if the largest exponent is itself a power of 2, the most significant bit position will be wrong. Indeed, the condition of the `while` loop should also include powers of 2. As a consequence, the number of groups analyzed during the decomposition could be wrong (depending on the value of `c`), and the most significant bit(s) will be ignored.

This bug will obviously lead to invalid results, but the consequences can be even worse. As we said, the condition to trigger the bug is having our largest exponent be a power of 2, for instance:

$$2^{32} \Rightarrow 10000000000000000000000000000000$$

And that the bug will make the exponentiation ignore the *MSB*:

$$2^{32} \Rightarrow 00000000000000000000000000000000$$

It means setting an exponent value to zero, hence forcing the exponentiation to return the identity for the corresponding point.

Moreover, because all other exponents are smaller than this power of 2, their MSB will be in a lower position. In the case where the ignored MSB is positioned at the beginning of a group, all other exponents are located in a previous group.

$$2^6 = \underbrace{0\,0\,1}_{1\cdot 2^{2\cdot 3}=1\cdot 64} \quad \overbrace{0\,0\,0}^{0\cdot 2^{1\cdot 3}=0\cdot 8} \quad \underbrace{0\,0\,0}_{0\cdot 2^{0\cdot 3}=0\cdot 1}$$

$$37 = \underbrace{0\,0\,0}_{0\cdot 2^{2\cdot 3}=0\cdot 64} \quad \overbrace{1\,0\,0}^{4\cdot 2^{1\cdot 3}=4\cdot 8} \quad \underbrace{1\,0\,1}_{0\cdot 2^{0\cdot 3}=0\cdot 1}$$

If this is true, all other exponentiations will not be impacted, and a point will be silently discarded from the computation.

Implementation can be fixed using the same trick done in the `straus` one:

```
while (groups < 256 && !(maxscalar < pow2(groups)))
    ++groups;
```

But we are advising to refactor the code and add properly tested functions (like a `msb` one for `rct::key`) in order to perform those operations.

5.6 Invalid Verify input parameters

Class	Severity	Difficulty to trigger	Difficulty to exploit
Data validation	High	Low	Unknown

The checks of the verifier's input parameters at the beginning `bulletproof_VERIFY` concentrate on the length of the proofs:

- check that the vector of aggregated `V` commitments is not empty,
- check that the vectors of inner-product argument points `L` and `R` have the same sizes,
- check that the vector `L` is not empty,
- check that the maximal size for a vector `L` is not greater than 32.

The maximal size for a list `L` is $\log_2(\texttt{MaxM} * \texttt{N}) = 10$. We do not know where the value `32` is coming from.

Besides size questions, a Bulletproof is composed of points and scalar. There are no checks that the scalars are reduced, which should not be a problem *mathematically*, but we saw in the case of the failing `addKeys2` that it can have severe consequences.

There are no checks that the points provided are indeed points of Ed25519 and that they are on the main subgroup. We did not manage to find a way to exploit the missing checks in the current protocol. However, we suggest to add them anyway to make the code more robust, former attacks in different contexts showing their importance (see [LiLe97] and [ABMSV03]).

5.7 Key compromise in Schnorr signature

Class	Severity	Difficulty to trigger	Difficulty to exploit
Data validation	High	Extremely High	Unknown

To generate a Schnorr signature, a random scalar `k` is generated such that `0 <= k <= l-1`. For security reasons, the scalar `k` must be non null. Otherwise, an attacker can detect that `sig.c == H(0)` and derive the secret key value. The signature value `sig.c` must be also checked for being non zero, otherwise the signature does not depend on the signer's key. These checks are described for instance in [TR-03111]. Comments in the source-code below are our own.

Listing 5.7: `src/crypto/crypto.cpp:244`

```
void crypto_ops::generate_signature(const hash &prefix_hash, const public_key &pub,␣
↪const secret_key &sec, signature &sig) {
   ge_p3 tmp3;
   ec_scalar k;
   s_comm buf;
   // [...]
   buf.h = prefix_hash;
   buf.key = pub;
   random_scalar(k); // no check on k != 0
   ge_scalarmult_base(&tmp3, &k);
   ge_p3_tobytes(&buf.comm, &tmp3);
```

(continues on next page)

(continued from previous page)

```
    hash_to_scalar(&buf, sizeof(s_comm), sig.c); // no check on sig.c != 0
    sc_mulsub(&sig.r, &sig.c, &unwrap(sec), &k);
}
```

The probability of such events to occur is negligible but their occurrence would lead to a compromise of the private key or a signature which does not depend on the signer's key. Therefore, we recommend adding these checks to ensure code robustness and security.

In the verification function, we also advise to check that the scalar `sig.c` is not null (the check present in the code with `sc_ckeck()` checks if the scalars are already reduced) and check that the resulting point $r.G + c.\texttt{pub}$ is not the point at infinity (case where the scalar `k` used by the signer is zero, meaning that the signer's key is compromised). Comments in the source-code below are our own.

Listing 5.8: `src/crypto/crypto.cpp:267`

```
bool crypto_ops::check_signature(const hash &prefix_hash, const public_key &pub,␣
↪const signature &sig) {
  ge_p2 tmp2;
  ge_p3 tmp3;
  ec_scalar c;
  s_comm buf;
  assert(check_key(pub));
  buf.h = prefix_hash;
  buf.key = pub;
  if (ge_frombytes_vartime(&tmp3, &pub) != 0) {
    return false;
  }
  if (sc_check(&sig.c) != 0 || sc_check(&sig.r) != 0) {
    return false; // test if the scalars are reduced
  } // no check that the scalar sig.c != 0
  ge_double_scalarmult_base_vartime(&tmp2, &sig.c, &tmp3, &sig.r); // no check that␣
↪tmp2 is not the point at infinity
  ge_tobytes(&buf.comm, &tmp2);
  hash_to_scalar(&buf, sizeof(s_comm), c);
  sc_sub(&c, &c, &sig.c);
  return sc_isnonzero(&c) == 0;
}
```

5.8 Failures in input size validation during deserialization

Class	Severity	Difficulty to trigger	Difficulty to exploit
Data validation	Medium	Low	Unknown

An error has been found during fuzzing using `AddressSanitizer` (it is only present in the `bp-multi-aggregation` branch):

```
==31931==WARNING: AddressSanitizer failed to allocate 0x13b02f5f800 bytes
```

The issue is located in the `rct::rctSigPrunable::serialize_rctsig_prunable` function:

Listing 5.9: `src/ringct/rctTypes.h:315`

```
uint32_t nbp = bulletproofs.size();
FIELD(nbp)
PREPARE_CUSTOM_VECTOR_SERIALIZATION(nbp, bulletproofs);
if (bulletproofs.size() > outputs)
    return false;
```

It looks like a direct consequence of the serialization library downsides.

Serialization works. The number of proofs is packed in the stream, then `PREPARE_CUSTOM_VECTOR_SERIALIZATION` marks the beginning of a vector. If the size is invalid, serialization fails as expected.

Deserialization is broken. The variable `nbp` is initialized to `0`, then parsed from raw data to a value (by `FIELD`), that is used unchecked in `PREPARE_CUSTOM_VECTOR_SERIALIZATION` to reserve the memory of a vector by resizing it. If allocation succeeds, the next check is still valid, because `vector::resize` modifies the actual size of the vector (contrary to `vector::reserve`).

An attacker can use this bug to perform a vector pre-allocation with an arbitrary size (up to 4 GB). If memory is lacking, an exception (`std::bad_alloc`) will be raised and has to be caught. We patched the code locally (in order to continue the fuzzing), by performing the check before anything else:

Listing 5.10: Patch suggestion for `src/ringct/rctTypes.h:315`

```
uint32_t nbp = bulletproofs.size();
FIELD(nbp)
if (nbp > outputs)
    return false;
PREPARE_CUSTOM_VECTOR_SERIALIZATION(nbp, bulletproofs);
assert(bulletproofs.size() <= outputs);
```

5.9 Failures in input size validation during containers deserialization

Class	Severity	Difficulty to trigger	Difficulty to exploit
Data validation	Informational	Low	Unknown

Containers (de)serialization routines have been reviewed. A minor issue has been found in the following code:

Listing 5.11: `src/serialization/container.h:65`

```
bool do_serialize_container(Archive<false> &ar, C &v)
{
    // [...]
    // very basic sanity check
    if (ar.remaining_bytes() < cnt) {
        ar.stream().setstate(std::ios::failbit);
        return false;
    }
```

The sanity check ensures a fast detection if the remaining bytes in the stream are not enough to unserialize the container (marking the stream as broken to ensure nothing will be done with it).

The check is invalid as only the number of elements are taken into account and not their sizes.

This failure does not seem critical, because the routine fails during the deserialization process. However, if the returned boolean is not verified properly, a partially initialized container object can be created.

A correct implementation could be something like:

Listing 5.12: Patch suggestion for `src/serialization/container.h:65`

```
bool do_serialize_container(Archive<false> &ar, C &v)
{
    // [...]
    // very basic sanity check
    if ((cnt > SIZE_MAX / sizeof(typename C::value_type)) ||
        (ar.remaining_bytes() < cnt * sizeof(typename C::value_type))) {
        ar.stream().setstate(std::ios::failbit);
        return false;
    }
```

5.10 Failures in input type validation during deserialisation

Class	Severity	Difficulty to trigger	Difficulty to exploit
Data validation	Medium	Low	Unknown

Parsing a transaction can fail with a `boost::bad_get` exception —the exception does not seem to be caught in every code paths— due to an invalid variant type:

Listing 5.13: `src/cryptonote_basic/cryptonote_format_utils.cpp`

```
bool expand_transaction_1(transaction &tx, bool base_only) {
   // [...]
   for (size_t n = 0; n < tx.rct_signatures.outPk.size(); ++n) {
       const auto& target = tx.vout[n].target;
       // this can raise boost::bad_get if target is not a txout_to_key
       rv.outPk[n].dest = rct::pk2rct(boost::get<txout_to_key>(target).key);
   }
```

Variants are a C++17 generalization of *unions* in C, for non-POD types. Non-POD (non-Plain-Old-Data) types can be approximately viewed as types that cannot be mapped directly to C types. An example is given below:

Listing 5.14: Example of variant

```
std::variant<int, std::string, std::vector<uint64_t>> myobj;
```

The stored type is runtime dependent. Therefore it needs to be checked dynamically. If this is not performed, then using the associated value may fail and an exception be raised.

Monero is using the `boost::variant` library, which inspired C++17 standard library. A variant type should always be checked, but it is especially true with distrusted data. The previous function is supposed to return false if an unexpected event occurs, so an additional check is needed:

```
bool expand_transaction_1(transaction &tx, bool base_only) {
    // [...]

    for (size_t n = 0; n < tx.rct_signatures.outPk.size(); ++n) {
        const auto& target = tx.vout[n].target;
        CHECK_AND_ASSERT_MES(target.type() == typeid(txout_to_key), false,
↪"unexpected type id in transaction");
        rv.outPk[n].dest = rct::pk2rct(boost::get<txout_to_key>(target).key);
    }
```

A quick code overview revealed some more locations with a missing check, but this one is the only one that failed during the fuzzing of unserialized inputs.

6. Weaknesses

We call *weaknesses* characteristics of the code that are not currently bugs but could easily induce errors during code maintenance and evolution.

6.1 Checks and input validation

6.1.1 Function preconditions

As we described in *Vulnerabilities* (*Overflow in the double scalar multiplication*, *Invalid Verify input parameters* and *Key compromise in Schnorr signature*) all parameters depending on the prover can be considered *hostile*. Therefore, they require extensive checks along all the chain of calls of functions and sub-functions. Any function applied on a proof is at risk of getting malformed parameters.

One way to help crucial preconditions to be ensured is to apply a more systematic approach to commenting the function requirements and intended output. This could have helped avoid the vulnerability on the double scalar multiplication if the function computing the w-NAF had had such a comment.

6.1.2 Shift bounds

We found a few shifts without bound checking. For instance, in `bulletproof_VERIFY`, the global variable `maxM` should be garanteed to be kept small enough so that the shift never overflows.

Listing 6.1: `src/ringct/bulletproofs.cc:944`

```
size_t M, logM;
for (logM = 0; (M = 1<<logM) <= maxM && M < proof.V.size(); ++logM);
```

A minor issue has been also found in the function checking that the size of `L` in a proof is indeed at least `logN`, fixed here to the literal value 6.

Listing 6.2: `src/ringct/rctTypes.cpp:236`

```
size_t n_bulletproof_amounts(const Bulletproof &proof)
{
    CHECK_AND_ASSERT_MES(proof.L.size() >= 6, 0, "Invalid bulletproof L size");
    return 1 << (proof.L.size() - 6);
}
```

This code is very simple, with first a boundary check to avoid negative values. Then the number of elements in the `proof.L` array is used in a bit shift operation. The issue here is that the upper bound is not checked, and 38 elements (or more) will result in an undefined result.

> *The result is undefined if the right operand is negative, or greater than or equal to the number of bits in the left expression's type.*[1]

It's a good practice to also add an upper bound validation, instead of relying on the CPU instruction applying a mask or returning zero. We propose the simple patch below:

[1] A7.8 Shift Operators, Appendix A. Reference Manual, The C Programming Language

```
size_t n_bulletproof_amounts(const Bulletproof &proof)
{
    size_t count = proof.L.size();
    CHECK_AND_ASSERT_MES(count >= 6 && count < 38, 0, "Invalid bulletproof L size");
    return 1 << (count - 6);
}
```

6.2 Edge cases and failure cases

6.2.1 Empty proof

While studying the aggregated bulletproof verification steps, we realized that the way `rct::bulletproof_VERIFY` is implemented makes it return `true` if an empty proof vector is provided.

Listing 6.3: `src/ringct/bulletproofs.cc:934`

```
rct::key z0 = rct::identity();
rct::key z1 = rct::zero();
rct::key z2 = rct::identity();
rct::key z3 = rct::zero();
rct::keyV z4(maxMN, rct::zero()), z5(maxMN, rct::zero());
// [...]
rct::key Y = z0;
sc_sub(tmp.bytes, rct::zero().bytes, z1.bytes);
rct::addKeys(Y, Y, rct::scalarmultBase(tmp));
rct::addKeys(Y, Y, z2);
rct::addKeys(Y, Y, rct::scalarmultH(z3));

std::vector<MultiexpData> multiexp_data;
multiexp_data.reserve(2 * maxMN);
for (size_t i = 0; i < maxMN; ++i)
{
    sc_sub(tmp.bytes, rct::zero().bytes, z4[i].bytes);
    multiexp_data.emplace_back(tmp, Gi_p3[i]);
    sc_sub(tmp.bytes, rct::zero().bytes, z5[i].bytes);
    multiexp_data.emplace_back(tmp, Hi_p3[i]);
}
rct::addKeys(Y, Y, multiexp(multiexp_data, true));

if (!(Y == rct::identity()))
{
    return false;
}
```

Which gives us:

$$Y = I - 0 \cdot G + I + 0 \cdot H + \prod_{i=0}^{\texttt{maxNM}} G_i^0 H_i^0 \Rightarrow I \stackrel{?}{=} I \Rightarrow \texttt{true}$$

We did not find this corner case in bulletproof tests, so we do not know if this is expected or not. It is logically sound. However taking into account that the input to the function `rct::bulletproof_VERIFY` is controled by the prover, it seems risky to be able to get `true` just by providing an empty proof.

When this function is used in `rct::rctSigs`, the absence of proof has to be handled specifically in the verification process.

```
if (!proofs.empty() && !verBulletproof(proofs))
    LOG_PRINT_L1("Aggregate range proof verified failed");
    return false;
}
```

In order to avoid future issues, we advise rejecting the empty proof case at the beginning of `rct::bulletproof_VERIFY`.

6.2.2 Tests on malformed proofs

All the unit tests of the `bulletproof_VERIFY()` function test well-formed proofs produced by `bulletproof_PROVE()`. No tests are made directly on malformed inputs provided to the verification algorithm. In order to cover more realistic pathological cases, we advise including different edge cases and failure cases in addition to testing the happy paths.

For the two last tests `TEST(bulletproofs, invalid_gamma_0)` and `TEST(bulletproofs, invalid_gamma_ff)` it is unclear whether what is tested is invalid values of `gamma`, `0x00...00` and `0xff...ff` or an invalid amount. The test defines `invalid_amount[8] = 1`, case where the input is 2^{64}, out of the intended interval $[0, 2^{64} - 1]$.

The test could be on an invalid amount. Indeed `0x00...00` is not an invalid value for `gamma` as long as such a value cannot be distinguished from all other hiding values. In this case, the test would need renaming.

The test could also be on an invalid `gamma` as `0xff...ff` as a scalar is greater than the order `l` of the main subgroup of Ed25519. In this case, the function `rct::addKeys2` outputs wrong values, providing a false proof. The test fails for the wrong reason (computation error in `rct::addKeys2` and not detection of an invalid value for `gamma` or for an invalid amount).

Listing 6.4: `tests/unit_tests/bulletproofs.cpp:187`

```
TEST(bulletproofs, invalid_gamma_ff)
{
   rct::key invalid_amount = rct::zero();
   invalid_amount[8] = 1;
   rct::key gamma = rct::zero();
   memset(&gamma, 0xff, sizeof(gamma));
   rct::Bulletproof proof = bulletproof_PROVE(invalid_amount, gamma);
   ASSERT_FALSE(rct::bulletproof_VERIFY(proof));
}
```

This last test could have detected the computation error in `rct::addKeys2` if designed in a better way.

6.2.3 Generators at infinity

In the implemented bulletproof protocol, the functions `rct::hashToPointSimple()` or `rct::hashToPoint()` are eventually called to create generators.

None of the calls to these functions is followed by a check that the output is not the point at infinity. The case is highly improbable, but such checks are not costly and ensure a more robust

code.

The computation being completely deterministic and identical for all version of the software, adding checks in unit tests might be a good idea in order to secure any future modifications.

6.3 Fragilities

Fragile characteristics are the contrary of robust ones. Even though the code is valid and works as expected, small changes or mistakes could lead to catastrophic security consequences.

6.3.1 Random generation initialization

The global state of the PRNG is initialized during software launch, with random data coming from the system generator:

- on Windows, 32 bytes are returned from `CryptGenRandom`. The cryptographic context is acquired using the `CRYPT_VERIFYCONTEXT` flag. It is a good practice that ensures no persistent key is generated by the CryptoAPI.
- on other operating systems, 32 bytes are read from `/dev/urandom`.

The function responsible for this critical step is called using a dynamic initializer called through the CRT library if the code is compiled with Visual C++, and with a constructor if `gcc` or `clang` are used.

Listing 6.5: `src/crypto/random.c:112`

```
INITIALIZER(init_random) {
  generate_system_random_bytes(32, &state);
  REGISTER_FINALIZER(deinit_random);
#if !defined(NDEBUG)
  assert(curstate == 0);
  curstate = 1;
#endif
```

In case of a failure in `generate_system_random_bytes`, the software will abort. If this function is never called (which should never happen in theory), then the execution continues and the output of the PRNG is predictable. This corner case is covered by an `assert` which forces the function `generate_random_bytes_not_thread_safe` to abort, but it is only enabled on debug builds.

Listing 6.6: `src/crypto/random.c:121`

```
void generate_random_bytes_not_thread_safe(size_t n, void *result) {
#if !defined(NDEBUG)
  assert(curstate == 1);
```

We strongly advise to keep this check in release builds too. The call to the initialization function of the PRNG is critical and there are scenarios where it could be skipped. For example, initializers of Visual C++ CRT are only called if the module is compiled in C. Simply renaming `random.c` to `random.cpp` will break the PRNG initialization on a Windows production build.

6.3.2 Manual type isolation

All the main elements manipulated in the code of Bulletproof —scalars or points or hash outputs— have the same type in the main code: the type `key`. This lack of strong typing allows reusing a variable for different purposes.

In the code below, `aG` begins as a scalar by receiving the value of the modular reduction of `a`. Then it is used as a point when receiving the value of the multiplication.

Listing 6.7: `src/ringct/rctOps.cpp:157`

```
//does a * G where a is a scalar and G is the curve basepoint
void scalarmultBase(key &aG,const key &a) {
    ge_p3 point;
    sc_reduce32copy(aG.bytes, a.bytes); //do this beforehand!
    ge_scalarmult_base(&point, aG.bytes);
    ge_p3_tobytes(aG.bytes, &point);
}
```

For the same reasons, constants are used for different purposes. The function `identity()` returns the point at infinity (the zero of the elliptic curve).

Listing 6.8: `src/rctOps.h:70`

```
//Creates a zero elliptic curve point
inline key identity() { return I; }
```

The same function `identity()` is also used as a vector having its least significant bit set to 1 (and the others to zero) as in `src/ringct/bulletproofs.cc:439` where `aL[i] = rct::identity()`

The lack of strong typing added to the lack of meaningful naming can quickly lead to security issues. For instance, swapping a scalar and a point can go completely unnoticed.

We suggest creating different types for (reduced) scalars and for points and adapt a specific hash function that returns a scalar. This would reduce the cognitive load of developers reading the code and leverage the C++ type system to help error detection.

6.3.3 Code duplication

While reviewing Bulletproof and Schnorr signature, we found that the functions `hash_to_scalar()` used in each context had different prototypes.

Listing 6.9: `src/crypto/crypto.h:101`

```
void hash_to_scalar(const void *data, size_t length, ec_scalar &res)
```

Listing 6.10: `src/ringct/rctOps.h:148`

```
void hash_to_scalar(key &hash, const void * data, const size_t l);
```

A closer look reveals that these two functions are indeed almost identical.

Listing 6.11: src/crypto/crypto.cpp:110

```
void hash_to_scalar(const void *data, size_t length, ec_scalar &res) {
cn_fast_hash(data, length, reinterpret_cast<hash &>(res));
sc_reduce32(&res);
}
```

Listing 6.12: src/crypto/hash.c:42

```
void hash_process(union hash_state *state, const uint8_t *buf, size_t count) {
   keccak1600(buf, count, (uint8_t*)state);
}

void cn_fast_hash(const void *data, size_t length, char *hash) {
   union hash_state state;
   hash_process(&state, data, length);
   memcpy(hash, &state, HASH_SIZE);
}
```

The comment even indicates the existence of the same function in a different namespace.

Listing 6.13: src/ringct/rctOps.h:298

```
//Hashing - cn_fast_hash
//be careful these are also in crypto namespace
//cn_fast_hash for arbitrary multiples of 32 bytes
void cn_fast_hash(key &hash, const void * data, const std::size_t l) {
   keccak((const uint8_t *)data, l, hash.bytes, 32);
}

void hash_to_scalar(key &hash, const void * data, const std::size_t l) {
   cn_fast_hash(hash, data, l);
   sc_reduce32(hash.bytes);
}

//cn_fast_hash for a 32 byte key
void cn_fast_hash(key & hash, const key & in) {
   keccak((const uint8_t *)in.bytes, 32, hash.bytes, 32);
}
```

Avoiding code duplication is a good security practice ensuring that corrections are reported on a single location.

6.3.4 Hard coded literals

In Bulletproof, many values are hardcoded in various places in the code. For instance, the maximal value for the interval to be proved is 2^{64}, a variable of size `key` is a vector of 32 bytes. The upper-bound for the interval is the reason why in many places the operation `1 << 6` is present.

Such reliance on hardcoded literals could lead to future potential troubles when updating types or upper-bounds. Making sure that all modifications are reported could be an error-prone operation. Besides, there is a loss of meaning when using literals. We suggest to rely on global constants gathered at meaningful places.

An example of such weaknesses is given by the function `estimate_rct_tx_size()`.

Listing 6.14: `src/wallet/wallet2.cpp:531`

```
size_t estimate_rct_tx_size(int n_inputs, int mixin, int n_outputs, size_t extra_size,
↪ bool bulletproof)
{
  size_t size = 0;

  // tx prefix

  // first few bytes
  size += 1 + 6;

  // vin
  size += n_inputs * (1+6+(mixin+1)*2+32);

  // vout
  size += n_outputs * (6+32);

  // extra
  size += extra_size;

  // rct signatures

  // type
  size += 1;

  // rangeSigs
  if (bulletproof)
    size += ((2*6 + 4 + 5)*32 + 3) * n_outputs;
  else
    size += (2*64*32+32+64*32) * n_outputs;

  // MGs
  size += n_inputs * (64 * (mixin+1) + 32);

  // mixRing - not serialized, can be reconstructed
  /* size += 2 * 32 * (mixin+1) * n_inputs; */

  // pseudoOuts
  size += 32 * n_inputs;
  // ecdhInfo
  size += 2 * 32 * n_outputs;
  // outPk - only commitment is saved
  size += 32 * n_outputs;
  // txnFee
  size += 4;

  LOG_PRINT_L2("estimated rct tx size for " << n_inputs << " with ring size " <<␣
↪(mixin+1) << " and " << n_outputs << ": " << size << " (" << ((32 * n_inputs/*+1*/
↪) + 2 * 32 * (mixin+1) * n_inputs + 32 * n_outputs) << " saved)");
  return size;
}
```

7. Improvements

7.1 Code dependencies: OpenSSL

In RingCT, a dependency on OpenSSL is required for a unique function, `rct::invert`. The function `rct::invert` computes the inverse of an integer modulo a fixed number, $\ell = 2^{252} + 27742317777372353535851937790883648493$, the order of the main subgroup.

Adding a heavy dependency raises issues both in terms of security and compatibility. OpenSSL is famous for its size and lengthy list of vulnerabilities. Considering the current use of the library in Bulletproof, the burden far outweights the benefits in terms of compatibility testing and vulnerabilities.

An inversion function modulo ℓ adapted to Bulletproof would allow to get rid of the dependency on OpenSSL and, furthermore, benefit from specific optimizations.

In the Monero project, the only other dependency on OpenSSL is `openssl/sha.h` allowing access to the `sha256` hash function. It does not warrant the linking with such a heavy dependency.

7.2 Use of near-standards: Keccak vs SHA-3

Monero's current implementation of its hash function uses the draft 3.0 of Keccak, the latest before NIST standardized a subset of the family into SHA-3. Monero's Keccak only differs from SHA-3 in the message padding.

```
--- keccak.c    2018-05-24 17:29:19.668521100 +0200
+++ sha3.c      2018-06-20 15:55:13.940522200 +0200
@@ -116,7 +116,7 @@
     }

     memcpy(temp, in, inlen);
-    temp[inlen++] = 1;
+    temp[inlen++] = 6;
     memset(temp + inlen, 0, rsiz - inlen);
     temp[rsiz - 1] |= 0x80;
```

For practical reasons, we advise replacing `Keccak[1088, 512]` with `SHA-3 256`. Indeed, a standardized algorithm means an easier evaluation of the impact of potential attacks, the compatibility with existing tools and test suites, etc.

7.3 Missing abstraction layers

Proper abstraction layers and type isolation help improve the code clarity and reduce the risk of missed vulnerabilities during a security review process.

7.3.1 Code factoring

The `bulletproof_PROVE` and the `bulletproof_VERIFY` functions are two monolithic functions mixing different layers of abstractions, from Pedersen commitment computations to lower level

arithmetic operations like multi-exponentiation, vector sums and multiplications to an unrolled inner-product protocol.

There are two `bulletproof_PROVE` functions (164 lines and 229 lines), one for proving a single value, the other one for proving `M` aggregated values. No subroutines are shared between these two functions although it appears possible to consider the single value case as the case `M=1` of the multi-value case.

The function `bulletproof_VERIFY` (249 lines) does not have any separated subroutines although the inner-product proof of knowledge argument augmented with the batch verification trick seems a good candidate.

We have proposed in Section *The two main algorithms* two pseudo-codes to simplify the two main algorithms through the use of subroutines. It exposed little flaws or issues in the current code.

For instance, the operation `sc_muladd(tmp.bytes, z.bytes, ip1y.bytes, k.bytes)` is grouped with multi-exponentiation related computations (and taken into account in timings) and not grouped with the other instructions computing the value of $\delta(y, z)$.

Listing 7.1: `src/ringct/bulletproofs.cc:975`

```
// bos coster is slower for small numbers of calcs, straus seems not
if (1)
{
  PERF_TIMER_START_BP(VERIFY_line_61rl_new);
  sc_muladd(tmp.bytes, z.bytes, ip1y.bytes, k.bytes);
  std::vector<MultiexpData> multiexp_data;
```

The computation of the challenges `w[i]` in `bulletproof_PROVE` and `bulletproof_VERIFY` raises a different issue. In the current, code the functions directly include all the inner-product prove and verify operations. Hence `w[i]` can be computed based on all former elements of the transcript. If the code were to be refactored and had to keep the current behavior, this dependency should be provided as input to the sub-function. From a specification point of view, this choice is not obvious. Besides, from a cryptographic point of view, it is unclear whether providing this dependency has a security advantage. This aspect is not considered in the paper where it seems enough for `w[i]` to depend on `L[i]` and `R[i]` alone. In the pseudo-code we proposed, we translated directly the paper subroutine organization.

7.3.2 Variable and function names

Variables names are often not explicit in the code. This is aggravated by the lack of strong typing.

For instance, all four variables in the code below have the same naming scheme and the same type. However, two of them are scalars and two of them are points.

Listing 7.2: `src/ringct/bulletproofs.cc:933`

```
// setup weighted aggregates
rct::key z0 = rct::identity();
rct::key z1 = rct::zero();
rct::key z2 = rct::identity();
rct::key z3 = rct::zero();
```

This increase the chances of unnoticed swapping of variables.

Variables called `L61Left` or `L61Right` are too closely related to an ephemeral early draft of the paper. They do not convey a real meaning (besides the line number in the last version of the paper is 72). We suggest isolating them in a specific function with an explicit naming like `check_commitment_inner_product_poly_coeff()`.

In the paper, a commitment to v with a hiding random exponent γ is computed as $h^\gamma \cdot g^v$. The notations are swapped in the code where `H` is used to commit on the secret values and `G` is used to blind the commitment: $\gamma \cdot \mathtt{G} + v \cdot \mathtt{H}$. But the use of **g** and **h** is not swapped and `Hi` and `Gi` follow the use in the paper. This lead to a risk of wrongly implementing the protocol.

For clarity, we suggest naming more explicitly variables, like `hiding_G` for the generator used for hiding and `binding_G` for the generator used for binding.

Many function names can be misleading. For instance, the function producing vector commitments (output is a point) is called `vector_exponent()`, the function initializing vectors of generators (points) are named `init_exponents()`, `vector_scalar2()` produces a vector of points although `vector_scalar()` produces a vector of scalars.

Coupled with the already-mentioned problem of a single type `key` the risk of an unnoticed mistake is increased.

7.4 Lack of specifications

Having an academic paper at the basis of an implementation adds a layer of difficulty for the development of the code and for its evaluation. Indeed, it is very seldom enough for a specification. A specification would allow no ambiguity, no implementation dependent choices and no prior deep knowledge of the underlying theories to be able to understand what needs to be implemented.

7.4.1 Input parameters

In the paper, input parameters of the prove and verify algorithms are not entirely specified for the bulletproofs as the two functions are interleaved in an interactive protocol.

The paper can also be misleading: for instance, in the last algorithm presented in the optimization section, for the combination of two multi-exponentiations, the commitment `V` to the value is not part of the input parameters. For the algorithms of the inner-product protocol, there is an unused parameter in the prover's input parameters in the paper: the prover never needs to use the point P, which is only necessary to the verifier. Hence the input parameters in the implementation are different from the paper.

7.4.2 Computation of challenges

The computation of the challenges in `src/ringct/bulletproof.cc` depends on the transcript up to the point of their computation as suggested in the original paper. However, the rule used to insert some dependency on preceding values is non-obvious. For instance, computing `x` includes `z` twice.

Listing 7.3: `src/ringct/bulletproofs.cc:951`

```
rct::key hash_cache = rct::hash_to_scalar(proof.V);
rct::key y = hash_cache_mash(hash_cache, proof.A, proof.S);
rct::key z = hash_cache = rct::hash_to_scalar(y);
rct::key x = hash_cache_mash(hash_cache, z, proof.T1, proof.T2);
rct::key x_ip = hash_cache_mash(hash_cache, x, proof.taux, proof.mu, proof.t);
w[i] = hash_cache_mash(hash_cache, proof.L[i], proof.R[i]);
```

It is important to have this rule specified: protocol security evaluation often implies developing compatible entities (prover or verifier).

7.5 Simplifications and performance suggestions

7.5.1 *Ugly* sum computation

In the `bulletproof_PROVE()` function in the aggregated proof case, a piece of code implements the following sum:

$$\sum_{j=1}^{m} z^{j+1} \cdot \left(\mathbf{0}^{(j-1)\cdot n} \parallel \mathbf{2}^n \parallel \mathbf{0}^{(m-j)\cdot n}\right)$$

Comments below are from the original source code:

Listing 7.4: `src/ringct/bulleproof.cc:747`

```
// This computes the ugly sum/concatenation from PAPER LINE 65
rct::keyV zero_twos(MN);
const rct::keyV zpow = vector_powers(z, M+2);
for (size_t i = 0; i < MN; ++i)
{
  zero_twos[i] = rct::zero();
  for (size_t j = 1; j <= M; ++j)
  {
    if (i >= (j-1)*N && i < j*N)
    {
      CHECK_AND_ASSERT_THROW_MES(1+j < zpow.size(), "invalid zpow index");
      CHECK_AND_ASSERT_THROW_MES(i-(j-1)*N < twoN.size(), "invalid twoN index");
      sc_muladd(zero_twos[i].bytes, zpow[1+j].bytes, twoN[i-(j-1)*N].bytes, zero_
↪twos[i].bytes);
    }
  }
}
```

In fact the *ugly* sum can be simplified. With a little abuse of notations, we could write it as:

$$\mathbf{2}^n \cdot \left(z^2 \cdot \mathbf{z}^m\right)$$

where each coefficient z^{j+2} is multiplying a vector $\mathbf{2}^n$, resulting in a vector of size $m \times n$. It provides a simpler and more efficient implementation of the sum.

Listing 7.5: simplification suggestion for `src/ringct/bulleproof.cc:747`

```
//  Last term (sum) in r(X), PAPER LINE 71
rct::keyV zero_twos(MN);
const rct::keyV zpow = vector_powers(z, M+2);
for (size_t j = 0; j < M; ++j){
  for (size_t i = 0; i < N; ++i){
    zero_twos[i] = rct::zero();
    sc_mul(zero_twos[i+j*N].bytes, zpow[2+j].bytes, twoN[i].bytes);
    }
  }
}
```

7.5.2 Inner-product challenge for the prover

In the `bulletproof_PROVE()` functions, the final challenge list `rct::keyV w(logN)` (resp. `rct::keyV w(logMN)`) can be replaced by a simple scalar, `rct::key w`, as the complete list of all challenges is never used (contrary to the `bulletproof_VERIFY()` function).

7.5.3 Test of points at infinity

Both Bos-Coster and Pippenger multi-exponentiation algorithms can be optimized on specific cases, when dealing with points at infinity.

To check if a given point `result` is the point at infinity, the code below is used:

Listing 7.6: `src/ringct/multiexp.cc:607`

```
if (memcmp(&result, &ge_p3_identity, sizeof(ge_p3)))
```

with `ge_p3_identity` defined as:

Listing 7.7: `src/crypto/crypto-ops-data.c:873`

```
const ge_p3 ge_p3_identity = { {0}, {1, 0}, {1, 0}, {0} };
```

This structure is made of 4 elements X, Y, Z and T, such as the Cartesian coordinates (x, y) of a point P would be:

$$x = X/Z$$
$$y = Y/Z$$
$$XY = ZT$$

In Cartesian coordinates, $(0, 1)$ represents the point at infinity, which means that $X = 0$ and $Y = Z$ in `ge_p3`. The current test for points at infinity in multi-exponentiation algorithms is only true if $X = 0$, $Y = 1$, $Z = 1$ and $T = 0$. We checked that it is very common to have $Y = Z$ with $Y \neq 1$: all these points at infinity are not captured by the current test.

The code can be improved in order to match more points at infinity. It can also be factorized as it seems to be a common check.

8. Conclusion

Bulletproof is a very recent protocol allowing to dramatically reduce the size of proofs of interval used in Monero to ensure every value manipulated is positive without revealing it.

During our security assessment of the Bulletproof implementation, we found:

- a structural cryptographic weakness in the Java implementation of the subgroup generators,
- two bugs that correspond to extremely improbable events triggered by functions internal computations (values hashing to zero or null random values) but whose consequences could be catastrophic,
- an absence of proper security checks on the inputs of the main proof verification function although these inputs are all controlled by a potential attacker,
- a direct consequence of this lack of input checking is the call of subfunctions with improper inputs causing an overflow in an arithmetic function,
- two major vulnerabilities that correspond to algorithmic mistakes in two multi-exponentiation algorithms whose inputs are also directly deduced from attacker inputs,
- three minor vulnerabilities in the serialization/deserialization procedures also fed with attacker inputs.

In the timeframe of the assessment, none of the vulnerabilities found has led to a practical exploit allowing to either produce a false proof accepted by a verifier or to reveal information on a proof. However, it does not mean it is impossible, especially when considering all the vulnerabilities in the verification directly linked to inputs controlled by an attacker.

Our code review also revealed several weaknesses: aspects that can be improved in order to strengthen code security or characteristics that are not currently bugs but could induce bugs during code maintenance and evolution.

We also proposed several simplification and performance improvements.

Monero is well-known for being at the forefront of cryptographic innovations in cryptocurrencies, incorporating the latest cutting-edge improvements. It made the review particularly challenging and interesting and we hope we helped improve the project's overall security.

8. Bibliography

[MP15] Gregory Maxwell and Andrew Poelstra. “Borromean ring signatures”. 2015. Available at https://github.com/Blockstream/borromean_paper/blob/master/borromean_draft_0.01_9ade1e49.pdf

[BBBPWM18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille and Greg Maxwell. “Bulletproofs: Short Proofs for Confidential Transactions and More,” 2018 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, US, pp. 319-338. doi:10.1109/SP.2018.00020

[NMM16] Shen Noether, Adam Mackenzie, and the Monero Research Lab. “Ring Confidential Transactions.” Ledger, 1 (2016): 1-18. Available at: https://ledger.pitt.edu/ojs/index.php/ledger/article/view/34

[CDF] Crypto Differential Fuzzing https://github.com/kudelskisecurity/cdf

[Keccak] Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche. The Keccak sponge functions. https://keccak.team/keccak.html

[LiLe97] Chae Hoon Lim and Pil Joong Lee. “A key recovery attack on discrete log-based schemes using a prime order subgroup”, Advances in Cryptology—CRYPTO ’97, Lecture Notes in Computer Science, vol. 1294.

[ABMSV03] Adrian Antipa, Daniel Brown, Alfred Menezes, René Struik, and Scott Vanstone. “Validation of Elliptic Curve Public Keys”. In Public Key Cryptography — PKC 2003. PKC 2003. Lecture Notes in Computer Science, vol 2567.

[TR-03111] Federal Office for Information Security. Technical Guideline BSI TR-03111, Elliptic Curve Cryptography. Version 2.10, 2018-06-01.