Monero

Audit of Bulletproofs+ code

JP Aumasson

20210323

Contents

1

Introduction

Monero solicited our services to review the security of the implementation of the Bulletproofs+ (BP+) range proof system, as integrated in Monero to make transactions smaller and faster to generate.

A first audit from ZenGo X covered the protocol, the accuracy of the implementation, and to some extent the code's security.

This new audit focuses on the code's security, and potential for abuse or misuse in the context of Monero. This is a smaller audit than the previous one (5 day-equivalent vs. 40), which complements the research from ZenGo.

We reviewed the BP+ implementation in terms of:

- Validation of inputs in proving and verification
- Risk of abuse of the cryptographic logic
- DoS-like scenarios (infinite loops, etc.)
- Software bugs jeopardizing BP+' security
- Weak set of parameters of usage scenarios
- Abuse of batch/aggregated verification
- Unsafe or insufficient error handling
- Primitives and parameters safety
- Dependencies' reliability
- General security assurance

1.1 Scope

We review the code at commit `0c6dfc9` (Feb 11,2021), focusing on the files

- `src/ringct/bulletproofs_plus.h` and
- `src/ringct/bulletproofs_plus.h`,

and using test routines in `tests/unit_tests/bulletproofs_plus.cpp` to test various usage scenarios.

The main functions (on which we spent the most review and testing time) are

- `bulletproof_plus_PROVE(const rct::keyV &sv, const rct::keyV &gamma)`
- `bulletproof_plus_VERIFY(const std::vector<const BulletproofPlus*> &proofs)`

We also reviewed the supporting arithmetic functions, for multi-exponentiation, exponents initialization, inner product, vector-wise operations, sum of powers, and so on. These (`static`) functions were evaluated in terms of correctness, and with respect to the pre-conditions imposed by their callers; they were thus not evaluted in a full adversarial setting, wherein an attacker could arbitrarily feed malformed input.

Although BP+ is just a "+" sign away from BP and bears a lot of structural similarity, its core is quite different as it uses a new zero-knowledge inner product argument. We carefully reviewed the input arguments' sanitization, the attack surface available to an attacker (quite narrow), and the potential edge cases' reachability and detection.

Finally, we reviewed the changes related to the full integration in Monero, done in a separate PR. We believe that these changes don't jeopardize the security of the BP+ implementation, and does not expose more attack surface than needed.

1.2 Summary

We didn't find anything that we believe qualifies as a security issue. We only report 6 potential in terms of "defense-in-depth", performance, and general quality, and which are clearly not about exploitable defects.

The absence of finding doesn't imply the absence of bugs, but just reduces their likelihood. We could have missed bugs, because of lack of knowledge about certain attack classes, because of our insufficient testing, or because of misunderstanding of BP+ or of the code's logic.

We nonetheless believe we gained a good understanding of the BP+ logic, and of its implementation mechanism, notably thanks to the preliminary study in the ZenGo X report that documented the role of each variable with respect to the BP+ specification. (Note that we did not notice errors in ZenGo X's description of the code, and are quite confident in its accuracy.)

Note that we reviewed the implementation's security in the context of its use in Monero, with the given hardcoded parameters. When needed, we reviewed part of the internal dependencies (such as random objects generation and multi-exponentiation), where didn't notice anything scary.

The audit took approximately 6 day-equivalent of work, of which most of the time was devoted to code analysis, and a smaller part to protocol analysis.

We would like to thank the Monero community for their renewed trust.

2

Observations (not security issues)

These are potential improvements, but not security risks:

2.1 Avoid timing leak of incorrect proof index?

The batch verification finally runs an aggregated check using (Pippenger algorithm's) multi-exponentiation, however if a proof is malformed then the verification algorithm will directly abort while looping over all the proofs, thus directly revealing the index of an incorrect proof by timing measurement.

We could not think of an adversarial scenario leveraging this property. However, unable to exclude such scenario, we report this property nonetheless.

A simple (but potentially incomplete) mitigation would be the mark the proof/verification as a failure, and only return after the loop during which the error was observed. (This would still leak the type of error to a timing attacker though.)

Likewise, the proving function can leak the index of an invalid element in `sv` or `gamma`.

2.2 Add `maxN` consistency checks

The `rct` namespace declares

```
static constexpr size_t maxN = 64; // maximum number of bits in range
```

and then some arrays are statically allocated dependending on this value. In the provign function, however, `maxN` isn't used but a local `logN` variable is declared and set to 6:

```
// Useful proof bounds
//
// N: number of bits in each range (here, 64)
// logN: base-2 logarithm
// M: first power of 2 greater than or equal to the number of range proofs to aggregate
// logM: base-2 logarithm
constexpr size_t logN = 6; // log2(64)
constexpr size_t N = 1<<logN;
size_t M, logM;
for (logM = 0; (M = 1<<logM) <= maxM && M < sv.size(); ++logM);
CHECK_AND_ASSERT_THROW_MES(M <= maxM, "sv/gamma are too large");
const size_t logMN = logM + logN;
const size_t MN = M * N;
```

If `logN` were to be inconsistent with `maxN` then things would go wrong. That is unlikely as long as both values are hardcoded but if a modification modifies one without the other, we recommend adding a consistency check that log of `maxN` is indeed `logN`.

Likewise in the verification function.

2.3 Omit superfluous non-zeroness check

In the verification:

```
// Random weighting factor must be nonzero, which is exceptionally unlikely!
rct::key weight = ZERO;
while (weight == ZERO)
{
    weight = rct::skGen();
}
```

This check isn't necessary, because `rct::skGen()` guarantees a non-zero input, as it runs through `sc_isnonzero()`.

2.4 Omit comparison of hardcoded values

The multi-exponentiation compares 232 with `STRAUS_SIZE_LIMIT`, also equal to 232, so this comparison isn't really needed (and likely optimized out by the compiler):

```
static inline rct::key multiexp(const std::vector<MultiexpData> &data, size_t HiGi_size)
{
    if (HiGi_size > 0)
    {
        static_assert(232 <= STRAUS_SIZE_LIMIT, "Straus in precalc mode can only be calculated \
            till STRAUS_SIZE_LIMIT");
        return HiGi_size <= 232 && data.size() == HiGi_size ?  straus(data, straus_HiGi_cache, 0)
            : pippenger(data, pippenger_HiGi_cache, HiGi_size, get_pippenger_c(data.size()));
    }
```

2.5 Save computation by early abort upon too large proof

In verification, the proof maximal length verification is done once after preprocessing all the proofs:

```
for (const BulletproofPlus *p: proofs)
{
    (...)
}
CHECK_AND_ASSERT_MES(max_length < 32, false, "At least one proof is too large");
```

Doing the check within the loop for each proof would save some computations (but would reveal the faulty proof index via time measurements).

2.6 Detect software failures by failing upon statistically unlikely events

Hash of the transcript to the zero scalar is checked in the proving routine, for example in these lines:

```
rct::key y = transcript_update(transcript, A);
if (y == rct::zero())
{
    MINFO("y is 0, trying again");
    goto try_again;
}
rct::key z = transcript = rct::hash_to_scalar(y);
if (z == rct::zero())
{
    MINFO("z is 0, trying again");
    goto try_again;
}
```

If zero is found, the operation is repeated.

However, a failure (that is, a zero) is much more likely to be caused by some software failure than actually hitting the zero value (which has probability around 2^{-252} per test).

Failing with a detailed error message thus seems to be a better engineering choice, as it would allow to detect such software errors, and prevent infinite loops.