Monero Bulletproofs Implementation Review

Benedikt Bünz

August 3, 2018

1 Intorduction

Monero[1] is a decentralised cryptocurrency with a focus on providing strong privacy for it's users. It implements the CryptoNote[3] protocol along with several improvements to it. Transactions use a protocol called Ring-Confidential Transactions. It hides the sender within an anonymity set and the amount using a cryptographic commitments. In order to ensure the validity of the transaction a ring signature is used to prove that one of the senders' private key is known and a zero-knowledge range proof is used to ensure that all outputs are positive. Recently a new zero-knowledge proof system, Bulletproofs[2] was published. Bulletproofs enable shorter ring signatures which could potentially lead to lower fees and better scalability for Monero. In order to ensure the safety of Monero's Bulletproof implementation, three independent reviewers were tasked with reviewing it. The reviews are funded through a community effort

2 Reviewed Code

I was tasked with reviewing the Java implementation of Bulletproofs by the Monero Research Lab (MRL). This was later ported to a C++ implementation. I reviewed git commit 6a9b3fbce0ace724ba30a454cc0df5f82f82257a of the implementation from Github `https://github.com/monero-project/research-lab`. I reviewed the files LinearBulletproofs.java, LogBulletproofs.java MultiBulletproofs.java and OptimizedLogBulletproof in the directory (sourcecode/StringCT-java/src/how/monero/hodl/bulletproof).

I did not review any other source code. In particular, I did not review the cryptographic libraries used or any code from the C++ production implementation. I give no guarantees as to the correctness and security of any

[1] `https://www.getmonero.org`

code that I did not review. As for the code that I did review, I spent significant time evaluating it's correctness/security and efficiency. I present the results in this document. Nevertheless, it is possible that I missed significant implementation and protocol bugs and the review was done to the best of my abilities and efforts.

3 Overview

In general the Java implementation of Bulletproofs follows the description of the paper correctly and is correct and secure. There is, however, a significant vulnerability in the generation of the public parameters which if not fixed can be used to break the proof system. It is easy to fix and I give specific recommendations for how to do this in Section 4. Additionally the code even outside of the cryptographic library is not constant time and is therefore possibly vulnerable against timing attacks. I will describe this in detail in Section 5. Comments regarding the aggregate range proof implementation are in Section 6. Further, there are several optimizations in OptimizedLogBulletproofs that were not taken and that could significantly improve the verification time. I point them out in Section 7. A few minor comments on implementation choices are mentioned in Section 8.

4 Public Paramenter

The Bulletproof paper only says that the public parameters which consists of a list of generators for a prime order group which are generated through a setup algorithm. This setup algorithm, however, requires a trusted party which is a cumbersome requirement that better be avoided. It turns out that the only requirement is that the discrete logs between all generators is unknown. The parameters, can therefore either be derived from a common random string, like the digits of π, a set of block headers or through a hash function that is modeled as a random function (oracle). The MRL implementation generates the parameters as follows. Let $H : \mathbb{G} \to \mathbb{G}$ be a hash function that maps a point on the curve Ed25519 to a point on

Ed25519.

$$g := (1, 1, 2) \in \mathbb{G} \tag{1}$$
$$h := H(g) \tag{2}$$
$$g_i := H(g^{2 \cdot i}) \quad \forall i \in [0, n-1] \tag{3}$$
$$h_i := H(g^{2 \cdot i+1}) \quad \forall i \in [0, n-1] \tag{4}$$

This, however, means that $h_0 = H(g^1) = H(g) = h$. This violates the necessary requirement that the discrete log between h_0 and h is unknown and can be used to break the soundness of the implementation. Furthermore the generation of the parameters is unnecessarily complex and expensive. A simpler scheme would be to generate them from a strings rather than curve points. Let $H' : \{0,1\}^* \to \mathbb{G}$ be a hash function. One can define the parameters as follows:

$$g := (1, 1, 2) \in \mathbb{G} \tag{5}$$
$$h := H('BP'||'h') \tag{6}$$
$$g_i := H('BP'||'g'||i) \quad \forall i \in [0, n-1] \tag{7}$$
$$h_i := H('BP'||'h'||i) \quad \forall i \in [0, n-1] \tag{8}$$

It is additionally important that clients generate these parameters themselves or check that the parameters were generated correctly as otherwise they might be vulnerable to a trapdoor.

5 Timing Vulnerabilities

Side channel attacks have both been studied extensively in the literature and have been deployed in practice. Cryptographic libraries are especially vulnerable to so called timing attacks. In these attacks an adversary measures the time it takes a victim to execute a certain procedure. If that time is dependent on secret information, such as a key, then the attacker can derive information about that secret and with repeated trials potentially even learn it completely. It is, therefore, considered best practice to make cryptographic implementations 'constant time', i.e. have their runtime be independent of any secrets. The MRL implementation is not constant time even if one assumes that the underlying curve operations are implemented as constant time. In Bulletproofs the first three rounds of the protocol (the inner product range proof) need to be executed in constant time. It is ok

if the InnerProduct argument is not executed in constant time as the argument is only used for efficiency and is allowed to leak secrets.

The following code does an additional operation if $a_{L,i} = 1$, i.e. the $i+1$st most significant bit of v is 1.

```
238             BigInteger basePow = BigInteger.valueOf(2).pow(i);
239             if (tempV.divide(basePow).equals(BigInteger.ZERO))
240             {
241                 aL[i] = Scalar.ZERO;
242             }
243             else
244             {
245                 aL[i] = Scalar.ONE;
246                 tempV = tempV.subtract(basePow);
247             }
```

A more efficient and constant time implementation of the code would be:

```
238         BigInteger vBigInt=v.toBigInteger();
239         for (int i = 0; i < N; ++i)
240         {
241                 aL[i] = vBigInt.testBit(i) ? Scalar.ONE : Scalar.ZERO;
242                 aR[i] = aL[i].sub(Scalar.ONE);
243         }
```

For the cryptographic operations the computation of A is particularly vulnerable to timing attacks as $a_{L,i}$ is either 0 or $p-1$. A naive implementation would take either no or $\log_2(p)$ squarings based on whether the bit i of v is 1 or 0. Alternatively exponentiation to $p-1$ can be implemented as an inversion or h_i^{-1} can be precomputed. Both techniques are more efficient and are less vulnerable to timing attacks then the current implementation. In order to demonstrate the effectiveness of this timing attack, I measured the time it takes to generate a range proof with value 0 and one with value $2^{64}-1$. Here the difference in the computation of A should be the largest. Using a JMH[2] microbenchmark I took 25 measurements of 10 seconds each. The benchmarks were run on a MacBook Pro (13-inch, Late 2016) with an i5-6360U CPU. From the results in Table 1 we can see that computing the range proof for $v = 2^{64}-1$ is about 5% faster than for $v = 0$. This means that an adversary who can estimate the time it takes for a prover to create

[2] http://openjdk.java.net/projects/code-tools/jmh/

v	Time per proof	Error (99.9% CI)
0	0.213s	0.005s
$2^{64}-1$	0.201s	0.011s

Table 1: Timing attack

a proof can distinguish between those two cases. This is problematic as some of the privacy guarantees of Monero/Confidential Transactions would be weekend. Note that this particular attack mostly takes advantage of the non constant time cryptography. If the C implementation does use constant time operations then the vulnerability might be less severe.

6 Aggregate Rangeproofs

Bulletproofs allows a prover to prove that m commitments are in a range. This function is implemented in the MultiBulletproofs class which additionally also implements batch verification (see Section 7). In general the functionality is implemented correctly and securely but only works for m being a power of two. This might be a limitation if a user has a transaction with 5 outputs. There are two general ways to handle this. The first involves padding, where the statement is padded such that it becomes a power of two. This can be done without changing the code by assuming that the additional commitments ($V_j s$) are the identity in the group. That is they are commitments to 0 with randomness 0. Without additional communication the verifier can verify the proof by assuming that the missing commitments are exactly the identity in the group. The downside is that this increases the prover and verifier computation from roughly $2n$ to $2 \cdot 2^{\lceil \log_2(n) \rceil}$ (at most a factor of 2). Another approach is changing the inner product argument. The protocol would be run as before but the input to the inner product argument now isn't a power of 2. To handle this in every round where $n' \mod 2 = 1$ the prover sends $a_{n'}$ and $b_{n'}$ and then sets $n' = n' - 1$. The verifier if $n' \mod 2 = 1$ computes $P = P \cdot g_{n'}^{-a'} h_{n'}^{-b'} \cdot h^{-a' \cdot b'}$ and $n' = n - 1$. This protocol has up to $4 \log_2(n)$ communication but does not require additional computational effort. It is important to note that running the inner product argument as is when n is not a power of 2 and setting $n' = \lfloor \frac{n}{2} \rfloor$ is not secure!

7 Optimizations

There are several places in which the prover and verifier computation can be optimized. The verifier's optimization concern the OptimizedLogBulletproofs.java code which seems to be the most efficient of the implementations. The optimization regarding the computation of A was already described in Section 5. Both the prover and the verifier would profit from Multi-exponentiation algorithms like the Pippenger algorithm [1]. These algorithms can perform a multi-exponenation $\prod_{i=1}^{n} g_i^{x_i}$ using $\tilde{O}(\frac{n}{\log n})$ elliptic curve operations. The MRL implementation does not use fast multi-exponentiation algorithms.
In the inner product argument the generators are updated in each round ($g'_i = g_i^{x^{-1}} \cdot g_{n/2+i}^{x}$). This is, however, not necessary as an alternative implementation would keep the generators the same and only update the exponents. The update to the generators costs $2n$ exponentiations and $2n$ point multiplications. By not updating the generators this cost can be avoided. On the other hand updating the generators will half the size of the witness and thus cheapen future computations. A careful analysis, however, reveals that it can be advantageous to only update the g values every k rounds and in the intermediary rounds simply update the exponents. The concrete values for k and the efficiency depends for example on the concrete choice of multi-exponentiation algorithms.

Another optimization concerns the verifier. In the MRL implementation the verifier computes the final exponents using $\log_2(n)$ operations per exponent:

```
// Convert the index to binary IN REVERSE
//and construct the scalar exponent
int index = i;
Scalar gScalar = proof.a;
Scalar hScalar = proof.b.mul(Invert(y).pow(i));

for (int j = rounds-1; j >= 0; j--)
{
// because this is done in reverse bit order
    int J = w.length - j - 1;
     // assumes we don't get too big
    int basePow = (int) Math.pow(2,j);
    if (index / basePow == 0) // bit is zero
    {
```

```
15          gScalar = gScalar.mul(Invert(w[J]));
16          hScalar = hScalar.mul(w[J]);
17      }
18      else // bit is one
19      {
20          gScalar = gScalar.mul(w[J]);
21          hScalar = hScalar.mul(Invert(w[J]));
22          index -= basePow;
23      }
24  }
25
26  // Adjust the scalars using the exponents
27  //from PAPER LINE 62
28  gScalar = gScalar.add(z);
29  hScalar = hScalar.sub(z.mul(y.pow(i))
30  .add(z.sq().mul(Scalar.TWO.pow(i))).mul(Invert(y).pow(i)));
```

This is unnecessary as all exponents can be computed in $n + \log(n)$ steps using batch inversion [4].
The following code computes *gScalar* and *hScalar* in just $O(n)$ operations by iteratively computing gScalars and hScalars:

```
1  Scalar yInvert = Invert(y);
2  Scalar[] wSquared = Arrays.stream(w).map(Scalar::sq)
3  .toArray(Scalar[]::new);
4  Scalar[] gScalars = new Scalar[N];
5  Scalar[] hScalars = new Scalar[N];
6  gScalars[0] = Invert(Arrays.stream(w).reduce(Scalar.ONE, Scalar::mul));
7  hScalars[N - 1] = gScalars[0];
8  for (int i = 1; i < N; ++i) {
9      int lower = i - Integer.lowestOneBit(i);
10     int wIndex = wSquared.length - Integer.numberOfTrailingZeros(i) - 1;
11     gScalars[i] = gScalars[lower].mul(wSquared[wIndex]);
12     hScalars[N - i - 1] = gScalars[i];
13 }
14 for (int i = 0; i < N; ++i) {
15     gScalars[i] = gScalars[i].mul(proof.a).add(z);
16     hScalars[i] = hScalars[i].mul(proof.b)
17     .sub(zSquared.mul(Scalar.TWO.pow(i))).mul(yInvert.pow(i));
18
19     hScalars[i] = hScalars[i].sub(z);
```

```
20      // Now compute the basepoint's scalar multiplication
21      // Each of these could be written as a multiexp operation instead
22      InnerProdG = InnerProdG.add(Gi[i].scalarMultiply(gScalars[i]));
23      InnerProdH = InnerProdH.add(Hi[i].scalarMultiply(hScalars[i]));
24  }
```

Another optimization for both the prover and the verifier concerns the inner product argument. In the paper and in the implementation the protocol terminates at $n = 1$. It is equally efficient in terms of proof size to terminate the protocol at $n = 2$, i.e. send 2 a and b values. The advantage of this is that the prover and verifier have to do two fewer exponentiations. Batch verification which allows a verifier to check multiple proofs at once is implemented in the MultiBulletproofs class. A few minor optimizations to the code could be made. The implementation uses 256 bit random weights. It is fine to use much smaller weights (like 2^40) as the security is statistical rather than computational. Additionally the first and second check can be batched using another random weight. This is particularly useful if multi-exponentiation is used.

8 Other notes

As Henry DeValence has pointed out the treatment of co-factors in the MRL implementation is dangerous. Bulletproofs is designed for prime-order curves but Monero uses Ed25519 which has a co-factor of 8. Monero chooses to work in the subgroup of Ed25519 which is of prime order. This is a fine choice but it is easy to make mistakes when implementing this. As I did not audit the C++ implementation and also was not tasked with auditing the underlying crypto libraries of the MRL Java implementation, I can not comment on whether this is handled correctly. However, I can give a few general comments on how to work with a prime order subgroup of a composite order curve. It is in general very important that all user-provided inputs (i.e. the proof elements) are checked to be actual points in the subgroup. The way to do this is by either raising to the power of p (the size of the subgroup) and ensuring that this is the point at infinity. Another way to handle this in general is to use the Decaf scheme by Mike Hamburg. This directly implements these checks into the decoding and encoding algorithms. Additional care also needs to be taken when hashing onto the curve, by ensuring that the hash actually lands in the subgroup. This can be easily implemented by hashing onto the curve and then raising the element to the co-factor.
Another small comment concerns the use of the Fiat-Shamir heuristic. The

MRL implementation correctly not hashes the previous challenge into each new challenge in order to commit to the whole transcript in each round. It would be a good idea to additionally commit to the public parameters by hashing them into the first challenge. The hash of the public parameters can be precomputed and reused. This ensures that the right parameters are used.
A final small note is that the EdCurve25519 class overwrites the equal function but not the hashCode function. This will lead to problems if objects like HashMap or HashSet are being used.

References

[1] Jonathan Bootle. Efficient multi-exponentiation.

[2] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *Bulletproofs: Short Proofs for Confidential Transactions and More*, page 0. IEEE, 2018.

[3] Nicolas van Saberhagen. Cryptonote v 2.0. *HYPERLINK https://cryptonote. org/whitepaper. pdf*, 2013.

[4] Hovav Shacham and Dan Boneh. Improving ssl handshake performance via batching. In *Cryptographers' Track at the RSA Conference*, pages 28–43. Springer, 2001.