Monero Bulletproofs+ Security Audit

Version 1.1

February 13, 2021

Prepared for:

Monero Research Lab

Prepared by:

Suyash Bagad Omer Shlomovits Claudio Orlandi

ZenGo X

Contents

1 Executive Summary

The Monero Research Lab announced [11] the implementation of Bulletproofs+ [5], a zero-knowledge proving system set to be used for range proofs in Monero. The Bulletproofs+ framework is planned to replace the existing Bulletproofs zero-knowledge proving system for range proofs. The Bulletproofs+ protocol ensures smaller proof sizes, faster proof generation as well as faster verification with aggregation of multiple proofs. This would result in lighter transactions on the Monero blockchain, faster generation in wallets and also enable faster verification on the end of the network participants.

This report describes the results of the security assessment of Monero's implementation of Bulletproofs+ by ZenGo X. The review of Monero's Bulletproofs+ was conducted between January 17 and February 13, 2021[1] for a total of 40 man-days of study.

1.1 Scope

We perform a cryptographic and security assessment of the Bulletproofs+ protocol specific to the Monero blockchain. The goal of this audit was to assess the readiness of Monero's implementation of Bulletproofs+ as a drop-in replacement to the existing range proof protocol Bulletproofs in Monero. We covered the following points as a part of the audit:

1. A full review of the e-print (url: `https://eprint.iacr.org/2020/735`, version: 17th June, 2020) of the paper with focus on the soundness of the scheme. Note that at the time of writing this report, the paper is not yet published in a peer-reviewed conference or journal.
2. Verifying that the implementation correctly reflected the prover and verifier algorithms described in the original e-print and finding vulnerabilities by code review, manual testing and fuzzing. In particular, we focused on checking if the code:
   (i) allows an attacker to generate a false proof that the verify algorithm deems as correct,
   (ii) leaks any information to an attacker from examining the proof(s) generated by honest prover(s),
   (iii) behaves correctly from a logical and an implementation point of view, including the underlying elliptic curve arithmetic used.

   The details of the review target are:

Language	C++
Repository	`https://github.com/SarangNoether/monero`
Branch	`bp-plus`
Commit	`7f964dfc8f15145e364ae4763c49026a3fab985d`
Files	`src/ringct/bulletproofs.h, src/ringct/bulletproofs.cc`, test modules and other relevant files in the directory `src/ringct`

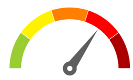
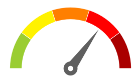
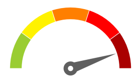
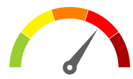
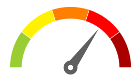
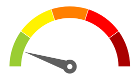
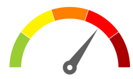
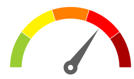
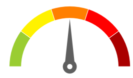
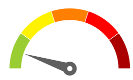
We also used an independent Rust implementation [8] of the Bulletproofs+ protocol from the original authors of the paper to provide an extra layer of validation. Although the Monero implementation greatly differs from the authors' Rust implementation attributing to some key optimisations, the core algorithms in both the implementations are similar. A notable difference, however, is that the Rust implementation's verification only verifies aggregated range proofs while the Monero implementation supports batched verification of multiple aggregated range proofs. [2]

[1] Note that we are releasing the first version of this report on February 10, 2021. Final version is planned to be released on February 17, 2021.

[2] By aggregated range proof, we mean a single `BulletproofPlus` proof for proving that multiple amounts lie in the range $[0, 2^{64} - 1]$. By batch verification, we mean mutliple individual `BulletproofPlus` proofs to be verified using a single multi-sclar multiplication.

1.2 Key Findings

We summarise the issues we found in the following table. Overall, the code is well documented and very closely follows the structure of the Bulletproofs implementation for Monero. Because the formulation of Bulletproofs+ is based on Bulletproofs, there are notable similarities in both of their implementations. We have analysed the Bulletproofs audit reports [13, 15] and ensured that the issues from Bulletproofs relevant to Bulletproofs+ have been taken care of. As an outcome of this audit, we did not find any critical issues and none of the high-severity issues were discovered to be practically exploitable.

Class	**Issue**	**Severity**	**Difficulty to trigger**	**Difficulty to exploit**
Cryptography	Missing 'point at infinity' check on the group elements of a `BulletproofPlus` proof after scalar multiplication by 8. (§ 5.1)			
	Inconsistency in verifier transcript computation: verifier challenges should be computed after multiplication of group elements by 8. (§ 6.1)			Unknown
	Caution on usage of amount commitments $\mathbf{V} \in \mathbb{G}_l^m$ outside Bulletproofs+. (§ 6.5)			Unknown
Data validation	Missing check if challenge 'e' is zero. (§ 5.2)			Unknown
	Incomplete condition for checking 'power of two' in a couple of helper functions. (§ 6.2)			Unknown
	Input parameter validation in `vector_of_scalar_powers()`; an error must be thrown if the paramter $n = 0$. (§ 6.3)			Unknown
	Redundant assertions in some places which should either be changed or removed. (§ 6.4)		NA	Unknown

Index: Critical High Medium Low Informational

We classify the concerns emerged from the evaluation work into three categories:

- Vulnerabilities (§ 5): These are critical or high-severity bugs which should be fixed in priority.
- Weaknesses (§ 6): We refer to issues that would not result in breaking of the system but consist of insufficient input validation or lack of consideration of all edge cases, as weaknesses.
- Improvements (§ 7): These are findings which would lead to a better and modular code base, they consist of some simplifications and some performance improvements.

Note: The references for the Straus, Pippenger and scalar inversion algorithms were absent in the codebase.

2 Context

On December 18, 2020, Dr. Sarang Noether of the Monero Research Lab announced the completion of Bulletproofs+ implementation for Monero in the Monero Research Lab's IRC channel [12]. Following, the authors submitted a Community Crowdfunding System (CCS) proposal on December 22, 2020 to audit the Bulletproofs+ implementation [1], which was accepted on January 15, 2020 by Monero Research Lab. The proposal was subsequently funded by the community in a couple of days and the audit process began on January 17, 2020.

2.1 Role of Bulletproofs+ in Monero

Bulletproofs+ is a zero-knowledge range proof protocol which builds on the Bulletproofs protocol. It does not require any trusted setup similar to that of Bulletproofs. Using Bulletproofs+, we can prove that an integer $a \in \mathbb{Z}_p$ lies in a finite range of the form $[0, 2^n)$, where $\mathbb{Z}_p$ is a finite field with a prime order p. The key idea of both Bulletproofs+ and Bulletproofs is proving that a given integer can be represented in a maximum of n bits and each of those n bits is either 0 or 1. Bulletproofs uses the inner product argument to construct a range proof. An inner product argument proves the knowledge of two vectors $\mathbf{a}, \mathbf{b} \in \mathbb{Z}_p^n$ given the following

(i) vector commitment to them $V = g^\alpha \cdot \mathbf{g}^\mathbf{a} \cdot \mathbf{h}^\mathbf{b} \in \mathbb{G}_l$, $\mathbf{g}, \mathbf{h} \in \mathbb{G}_l^n$ and $\alpha \in \mathbb{Z}_p, g \in \mathbb{G}_l$ [3]

(ii) inner product $c = \langle \mathbf{a}, \mathbf{b} \rangle \in \mathbb{Z}_p$

where $\mathbb{G}_l$ is the prime order group and we assume that the discrete logarithmic relation between the elements of $\mathbf{g}$ and $\mathbf{h}$ is unknown. On the other hand, Bulletproofs+ uses a weighted inner product argument of the form $c = \langle \mathbf{a}, \overrightarrow{y}^n \circ \mathbf{b} \rangle$ where $\overrightarrow{y}^n = \left(y, y^2, \ldots, y^n\right), y \in \mathbb{Z}_p$. Using the weighted inner product argument, Bulletproofs+ succeeds in reducing the proof size of Bulletproofs by 3 elements, i.e. for a 64-bit range proof, a Bulletproofs+ proof is about 15% smaller than that of a Bulletproofs proof.

In the context of Monero, the input and output amounts are hidden in Pedersen commitments, so its necessary for the owners to prove that the amounts hidden in those commitments are in the range $[0, 2^{64} - 1]$. This ensures that the user cannot create funds out of thin air by wrapping around the amount modulo p and balancing the input-output amounts in a malicious way. Currently, in Monero, a single aggregated Bulletproofs proof is used to prove that every output in a transaction hides an amount in the range $[0, 2^{64} - 1]$. For every transaction, irrespective of the number of outputs created, Bulletproofs+ would require 96 bytes lesser than that of Bulletproofs. For the most common 2-output transactions seen in Monero, the following table shows the proof size improvements.

Spent inputs	Current size	New size	% Reduction
1	1.42 kB	1.33 kB	6.6%
2	1.92 kB	1.83 kB	5.1%

Table 1: Proof size benefits of Bulletproofs+

Moreover, the proof generation and verification of Bulletproofs+ is also faster than that of Bulletproofs. Bulletproofs+ proofs are generated with a 10% speedup as compared to that of Bulletproofs. Proofs are generated typically in a user's wallet and so do not directly affect the computation on the blockchain. In spite of this, faster proof generation reduces computation overhead in the wallets and helps faster transaction creation.

[3] We have used multiplicative notation for group operations here for brevity. In the following chapters, we use the usual additive notation.

Proof verification times are more crucial because the proofs need to be verified on-chain by miners and other network participants. Bulletproofs+ proof verification is marginally faster than Bulletproofs for single amount range proofs. However, similar to Bulletproofs, multiple Bulletproofs+ proofs can be verified in batches much more efficiently than doing so individually. The following table shows the percent reduction in verification time between the Bulletproofs and Bulletproofs+ algorithms for proofs comprising different numbers of outputs [11].

Outputs per proof	Single proofs, % faster	Batched proofs, % faster
2	1.5%	5.3%
4	0.5%	9.2%
8	1.6%	9.2%
16	0.9%	10.8%

Table 2: Verification speedup of Bulletproofs+ over Bulletproofs

The proof size directly impacts the space used on the blockchain while the verification times greatly affect on-chain computation. We see that the proof size reduction as well as verification speedup from using Bulletproofs+ over Bulletproofs clearly motivate the incorporation of Bulletproofs+ in Monero.

3 Paper Review

As a part of this audit, we present a formal review of the e-print of the Bulletproofs+ paper [5] (version: 17 June, 2020) in this section. Note that at the time of writing of this report, the Bulletproofs+ paper is not yet published in a peer-reviewed conference or journal. To this end, we aim to verify that the main protocol and the accompanying proofs are theoretically correct. Moreover, our focus was to check the completeness, soundness and zero-knowledge properties of the argument systems in the paper and spot any anomalies that could cause potential issues in a production-level implementation.

3.1 Summary of the Review

This paper contains an improved version of the Bulletproofs Zero-Knowledge Argument System [4]. The efficiency improvements as claimed appear somewhat marginal compared to the original paper. The technical parts, including the proofs, contain minor errors (e.g. few wrong indices, some wrong but easily fixable mathematical expression, typos, etc.) which do not however seem to undermine the main claims of the paper.

The Zero-Knowledge (ZK) Argument for the weighted-inner product (WIP) relation (Section 3, Figure 1) closely follows the "Improved Inner-Product Argument" (Section 3 in the original Bulletproof paper) with the two main differences.

1. The new protocol already incorporates the "weight" vector y,
2. the WIP protocol is designed to be zero-knowledge.

The protocol in Figure 1 does not appear to have any error, and therefore could be used as a base for an implementation.

Appendix C contains the proof for the protocol in Section 3. We are very confident that the protocol as specified is correct and zero-knowledge as claimed. There are a few typos in the proof of correctness, but nothing that can't be fixed by direct inspection of the protocol.

The proof of soundness or, better, of Witness-Extended Emulation (WEE) contains minor typos, but we are persuaded that the claim holds. The proof itself follows closely, almost step-by-step, the proof of WEE in Theorem 1 in the Bulletproofs paper. The main difference between the two proofs is that Protocol 2 in the original Bulletproofs paper is not ZK, so the base case is trivial (the witness is trivially sent by the prover). Theorem 1 in Bulletproofs+ instead also has to show that the protocol is WEE for $n = 1$, but the technique used is the same which is used in the recursion step of Bulletproofs, and we are persuaded that the claim is correct.

Both Bulletproofs and Bulletproofs+ don't come with explicit proofs of WEE, but only proofs that given enough accepting transcripts you can extract a witness. The notion of WEE is an enhanced version of proof-of-knowledge which essentially requires that there exists a single simulator-extractor that can both extract the witness and produce an indistinguishable transcript. This is needed to compose the protocol within larger protocols. Lindell [10] proves that every PoK is WEE. The definition of WEE used here is the one from [7] which was also used in the original Bulletproof paper, and differs slightly from the definition of Lindell as it allows for some common reference string. To prove that the protocol is WEE both Bulletproofs and Bulletproofs+ rely on a generalised forking Lemma from [3]. The original Lemma there requires an extractor *that always succeeds*. Bulletproofs has (see their Theorem 6) changed this to work also for extractors with negligible error. They don't provide proof that this holds. After a quick check of the proof of the forking Lemma in [3], we believe that their claim is correct, as the proof never explicitly uses the assumption that the extractor works with probability 1.

Bulletproofs only briefly describes (Section 4.4) how to compile the proposed protocols to a non-interactive version and how to extract the public parameters using the Fiat-Shamir heuristic. Bulletproofs+ is even less explicit (Section 2.4). Clearly how this is done has a very big impact on the security of the protocol in practice, and we would recommend thinking carefully about it.

3.2 On Proof Malleability and Transferability

An important concern in practice is the malleability and transferability of the resulting NIZK. The paper does not discuss this, as proofs are only provided for the interactive version of the protocol. Regarding transferability: to avoid "replay attacks" in which someone copies someone else's proof and uses it in their transaction, it could be recommended to hash as much context as necessary when deriving the challenges of the Fiat-Shamir transform, so that the proofs are non-transferable from one context to another (this clearly requires the verifier to be aware of the context and to use it at verification time). For example, if the protocol is run between parties with identities A, B, those identifiers could be included in the hash so that $e = H(\mathtt{tx}, A, B)$ (where $\mathtt{tx}$ is whatever is already hashed by Fiat-Shamir). Now this proof has "context" (A, B) and cannot be therefore used by B towards a verifier C, and so on. The more it is hashed the more "local" the proof is, and the less it can be replayed maliciously. Time, underlying blockchain, type of transaction, etc. could be added to the context if deemed necessary. It would also be worth looking at what others implementations of Bulletproofs have been doing in this regard. We note that these transferability problems are not specific to Bulletproofs+ but appear in any implementation of Fiat-Shamir NIZKs.

Regarding malleability: The only result we are aware of on the non-malleability (or simulation-extractability) of the Fiat-Shamir NIZK (note, "non-malleability" does not include trivial transferability of the entire proof, aka replay attacks as described above) is proven in [6]. The theorem in that paper doesn't apply to Bulletproofs+ for technical reasons. However, it seems unlikely that an adversary should be able to take a proof π for a statement x and should be able to create a proof π' for a related statement x' without actually knowing a witness for x'.

3.3 On the correctness of Proofs

Section 4.1 contains a protocol for range proofs which uses the zk-WIP protocol from Section 3 as a building block. The resulting range proof is conceptually much simpler than the analogous in the Bulletproof paper, since the building block already satisfies the zero-knowledge property. Section 4.2 contains an amortized version of the range proof, closely following the technique for amortization in the original Bulletproof paper. Both the protocol for a single instance and the amortized version (Figure 2 and 3) are specified unambiguously, except for the fact that the reference string $\mathbf{g} \in \mathbb{G}_l^{mn}$ and $\mathbf{h} \in \mathbb{G}_l^{mn}$ is provided as part of the relation, while it should be described (as it is) as a part of the reference string. The proof for the single instance protocol is omitted since it is a special case of the amortized protocol. The proof of the amortized protocol is provided in Appendix D. The proofs of correctness and (honest-verifier) zero-knowledge appear sound and we did not spot any mistake.

The proof of WEE for the amortized range proof protocol in Bulletproofs+ has a mistake. Proving WEE involves construction of an extractor for extracting witness values from a number of valid proof transcripts for the same witness. Given a set of valid proof transcripts each using different set of challenges, the first step in proving WEE is to extract the expressions of A and

$V_j,\ j \in [m]$ in terms of the reference string ($\mathbf{g}, \mathbf{h} \in \mathbb{G}_l^{mn}, g, h \in \mathbb{G}_l$), i.e.

$$\begin{aligned} A &= \mathbf{g}^{\mathbf{a}_L} \cdot \mathbf{h}^{\mathbf{a}_R} \cdot g^{\beta} \cdot h^{\alpha}, \\ V_j &= \mathbf{g}^{\mathbf{v}_{L,j}} \cdot \mathbf{h}^{\mathbf{v}_{R,j}} \cdot g^{v_j} \cdot h^{\gamma_j}. \end{aligned}$$

In the WEE proof of the Bulletproofs+ paper, the authors use constant exponent vectors $\mathbf{v}_L, \mathbf{v}_R$ for all $j \in [m]$. We cannot assume this and we need to consider distinct vectors for each $j \in [m]$ and then prove that each of these $\mathbf{v}_{L,j}, \mathbf{v}_{R,j} \in \mathbb{Z}_p^{mn}$ vectors equals the '0' vector. Once we extract these exponents of the elements from the reference string, we need to prove that they satisfy the following desired relations.

$$\begin{aligned} \mathbf{a}_R &= \mathbf{a}_L - \mathbf{1}^{mn} \\ \mathbf{a}_L \circ \mathbf{a}_R &= \mathbf{0}^{mn} \end{aligned}$$

$$\left.\begin{aligned} \langle \mathbf{a}_L, \mathbf{d}_j \rangle &= v_j \\ \mathbf{v}_{L,j} \circ \mathbf{v}_{R,j} &= \mathbf{0}^{mn} \\ \mathbf{v}_{L,j} + \mathbf{v}_{R,j} &= \mathbf{0}^{mn} \end{aligned}\right\} \text{ for all } j \in [m]$$

Now, on substituting the expressions of A, V_j in the expression of $\hat{A}$ below, we can obtain the relation between $\Big(\mathbf{a}_L, \mathbf{a}_R, \{\mathbf{v}_{L,j}, \mathbf{v}_{R,j}\}_{j\in[m]}\Big)$.

$$\begin{aligned} \hat{A} &= \mathbf{g}^{\hat{\mathbf{a}}_L} \cdot \mathbf{h}^{\hat{\mathbf{a}}_R} \cdot g^{\hat{\mathbf{a}}_L \odot_y \hat{\mathbf{a}}_R} \cdot h^{\hat{\alpha}} \\ &= A \cdot \mathbf{g}^{-z\cdot\mathbf{1}^{mn}} \cdot \mathbf{h}^{z\cdot\mathbf{1}^{mn} + \mathbf{d}\circ\overleftarrow{y}^{mn}} \cdot \mathbf{V}^{y^{mn+1}\cdot z^2\cdot \mathbf{z}^m} \cdot g^{(z-z^2)y\cdot\langle \mathbf{1}^{mn}, \mathbf{y}^{nm}\rangle - zy^{mn+1}\cdot\langle \mathbf{1}^{mn}, \mathbf{d}\rangle} \end{aligned}$$

The way it is done in the paper is: we first match the exponents of the generator vectors $\mathbf{g}, \mathbf{h}$ and then use them to cross-check the exponent of g. We have the following correction in the expressions of $\hat{\mathbf{a}}_L, \hat{\mathbf{a}}_R$ from page 36.

$$\begin{aligned} \hat{\mathbf{a}}_L &= \mathbf{a}_L - z\cdot\mathbf{1}^{mn} + \left(\cancel{\mathbf{v}_L \cdot y^{mn+1}}\right) + \left(\sum_{j=1}^{m} z^{2j} y^{mn+1}\cdot \mathbf{v}_{L,j}\right) \\ \hat{\mathbf{a}}_R &= \mathbf{a}_R + \mathbf{d}\circ\overleftarrow{y}^{mn} + z\cdot\mathbf{1}^{mn} + \left(\cancel{\mathbf{v}_R \cdot y^{mn+1}}\right) + \left(\sum_{j=1}^{m} z^{2j} y^{mn+1}\cdot \mathbf{v}_{R,j}\right) \end{aligned}$$

Now, all that remains is computing $\hat{\mathbf{a}}_L \odot_y \hat{\mathbf{a}}_R$ from the above equations and then comparing it to the exponent of g in the expression of $\hat{A}$. Since the two sides of the equation we compare can be thought of as two polynomials in y, z, it is easy to just match the coefficients of the challenge powers. We do not explicitly write out the comparison equations for brevity, but we confirm that the result follows in a way similar to that of the 'Left hand side' and 'Right hand side' comparion table on page 36 of the paper.

In conclusion, apart from the above mistake in the proof of WEE of the range proof protocol, rest all of the proofs seem to be sound and we did not encounter any mistakes. Therefore, inspite of the error in the WEE proof, the conclusion of the proofs still hold (after rightful correction).

4 Code Overview

We studied the code in the `bp-plus` branch of the repository `https://github.com/SarangNoether/monero`. The last commit considered for the review is `2d287f69b79929908f884224c22c58d3bec50b09`. We analyse the code specific to the files:

- `src/ringct/bulletproofs_plus.h`
- `src/ringct/bulletproofs_plus.cc`
- `src/tests/unit_tests/bulletproofs_plus.cpp` (Testing module)

and relevant cryptographic functions in `src/ringct`.

In this section, we explain the implementation of the two main algorithms of Bulletproofs+: Prove and Verify. Specifically, we map the parts of the code to the cryptographic equations for the case of aggregation of multiple Bulletproofs+ proofs. This helps narrow down the domain for testing and analysing the conformity of the code with the equations in the paper. Before we begin, we describe a set of preliminaries and notations used in the code.

4.1 Notation

We want to prove that the amounts in Monero are in the range $[0, 2^{64} - 1]$. We analyse the case of multiple proofs that are aggregated. Important pieces of notation and constants [9] are regrouped below.

Notation:

(i)	$\mathbb{G}_l$	the prime-ordered subgroup of the Ed25519 curve used in Monero
(ii)	$\mathbb{Z}_p$	the scalar field over which the Ed25519 curve is defined, p is a prime
(iii)	m, n	number of proofs to be aggregated and number of bits respectively
(iv)	G	base generator of the subgroup $\mathbb{G}_l$
(v)	H	another generator of the subgroup $\mathbb{G}_l$ such that its discrete log w.r.t G is unknown
(vi)	$\mathbf{G}, \mathbf{H}$	generator vectors in $\mathbb{G}_l$ each of size mn such that the discrete log between their elements as well as G, H is not known
(vii)	$\mathbf{y}^n$	a scalar vector $(1, y, \ldots, y^{n-1}) \in \mathbb{Z}_p^n$
(viii)	$\overrightarrow{y}^n$	a scalar vector $(y, y^2, \ldots, y^n) \in \mathbb{Z}_p^n$
(ix)	$\overleftarrow{y}^n$	a reverse-ordered scalar vector $(y^n, y^{n-1}, \ldots, y^1) \in \mathbb{Z}_p^n$
(x)	$\mathbf{V}$	vector of commitments to the amounts, $\mathbf{V} = \{V_j\} \in \mathbb{G}_l^n$, $V_j = a_j * H + \gamma_j * G$ for all $j \in [m]$, where $a_j, \gamma_j \in \mathbb{Z}_p$ is the amount and the blinding factor respectively

Note that we use additive notation to describe equations in this document as used in the implementation. The paper uses multiplicative notation. Also, the Pedersen commitments in the paper are defined as $V = a * G + \gamma H \in \mathbb{G}_l$ for amount $a \in \mathbb{Z}_p$ and blinding factor $\gamma \in \mathbb{Z}_p$. However, the Monero implementation switches the role of G and H so the commitments take the form $V = a * H + \gamma G$.

Public parameters (in code):

(i) `l` order of the prime-order subgroup of the Ed25519 curve used in Monero, the order of Ed25519 curve is $8 \times \mathtt{l} = 2^{252} + 27742317777372353535851937790883648493$

(ii) `N` number of bits of the elements whose range one wants to prove ($\mathtt{N} = 64$)

(iii) `M` number of proofs to be aggregated ($\mathtt{M} \leq \mathtt{maxM} = 16$)

(iv) `G` the base point of the subgroup of Ed25519 curve used

(v) `H` another generator of the subgroup, the discrete log relation between `H, G` is assumed to be unknown[1]

(vi) `Gi` generator vector of size $\mathtt{maxM} * \mathtt{maxN}$ such that the discrete log between any of its elements as well as `G,H` is not known

(vii) `Hi` generator vector of size $\mathtt{maxM} * \mathtt{maxN}$ such that the discrete log between any of its elements as well as `G,H` is not known[2]

Witnesses (in code):

(i) `v` vector of `M` amounts such that $0 \leq \mathtt{v}[i] < 2^{64}$ for all $i \in [\mathtt{M}]$

(ii) `gamma` vector of `M` scalar field elements known as *blinding* factors

A `BulletproofPlus` proof (in code):

(i) `A` group element which is a Pedersen commitment to the input witness vectors

(ii) `A1` Pedersen commitment (group element) to the witness values at the end of the $\log_2(mn)$ inner-product protocol rounds

(iii) `B` Pedersen commitment (group element) to the randomness used in the final round

(iv) `r1` random scalar used in the final round

(v) `s1` another random scalar used in the final round

(vi) `d1` another random scalar used in the final round

(vii) `L` special Pedersen commitment vector to intermediate witness vectors in the recursive inner-product protocol rounds

(viii) `R` special Pedersen commitment vector to intermediate witness vectors in the recursive inner-product protocol rounds

(ix) `V` a vector in $\mathbb{G}_l$, Pedersen commitments to amount $\mathtt{v}[i]$ and blinding factor $\mathtt{gamma}[i]$ for $i \in [\mathtt{M}]$, although `V` is used in the proof system, it is a part of the transaction and not of a `BulletproofPlus` proof.

Clearly, a `BulletproofPlus` proof consists of $2\log_2(mn) + 3$ elements in $\mathbb{G}_l$ and 3 elements in $\mathbb{Z}_p$. Since the size of compressed group elements as well as scalars for Ed25519 is 32 bytes, the size of a `BulletproofPlus` proof is 96 bytes (or 3 elements) lesser than that of a `Bulletproof` proof.

[1] `H` is generated by hashing `G` using the function `rct::hash_to_p3()`.

[2] All the generators `G, H, Gi, Hi` are the elements of the prime-order subgroup of the curve Ed25519.

4.2 Prover's Algorithm

The `bulletproof_plus_PROVE` function[4] takes as input the vector of amounts and the blinding factors, each of size `M`.

Step 1: A natural first step is checking (i) if the sizes are these vectors are correct, (ii) the vectors contain scalars in the field $\mathbb{Z}_p$.

```
// Given a set of values v [0..2**N) and masks gamma, construct a range proof
BulletproofPlus bulletproof_plus_PROVE(const rct::keyV &sv, const rct::keyV &gamma)
{
    // Sanity check on inputs
    CHECK_AND_ASSERT_THROW_MES(sv.size() == gamma.size(), "Incompatible sizes of sv and gamma");
    CHECK_AND_ASSERT_THROW_MES(!sv.empty(), "sv is empty");
    for (const rct::key &sve: sv)
        CHECK_AND_ASSERT_THROW_MES(is_reduced(sve), "Invalid sv input");
    for (const rct::key &g: gamma)
        CHECK_AND_ASSERT_THROW_MES(is_reduced(g), "Invalid gamma input");

    init_exponents();

    // 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");
```

Listing 1: Sanity checks on inputs

Step 2: The next step in the prover's algorithm is to compute Pedersen commitments to the amount vector. The implementation uses a neat trick here: all the scalars which would be used in the final multi-exponentiation check by the verifier are pre-divided by scalar $8 \in \mathbb{Z}_p$. This is done because in the verification, we have to multiply the group elements (of the proof) by 8 to ensure that they lie in the prime-order subgroup $\mathbb{G}_l$. The net effect of this is that the scalar 8 gets cancelled in final multi-exponentiation step. As an alternative, one could multiply the group elements by $8^{-1} \in \mathbb{Z}_p$ after the check that they lie in $\mathbb{G}_l$. But group operations are a lot more expensive than scalar multiplications. Thus, we do the following:

$$\mathbf{v}_8[i] \coloneqq 8^{-1} \cdot \mathbf{v}[i], \quad \gamma_8[i] \coloneqq 8^{-1} \cdot \gamma[i]$$

```
    // Prepare output commitments and offset by a factor of 8**(-1)
    //
    // This offset is applied to other group elements as well;
    //  it allows us to apply a multiply-by-8 operation in the verifier efficiently
    //  to ensure that the resulting group elements are in the prime-order point subgroup
    //  and avoid much more constly multiply-by-group-order operations.
    for (size_t i = 0; i < sv.size(); ++i)
    {
        rct::key gamma8, sv8;
        sc_mul(gamma8.bytes, gamma[i].bytes, INV_EIGHT.bytes);
        sc_mul(sv8.bytes, sv[i].bytes, INV_EIGHT.bytes);
        rct::addKeys2(V[i], gamma8, sv8, rct::H);
    }
```

Listing 2: Computing Pedersen commitments

[4] The code listed in this report is from the file `src/ringct/bulletproofs_plus.cc`.

Step 3: Next, the vectors $\mathbf{a}_L, \mathbf{a}_R$ are computed as: $\mathbf{a}_L$ is the concatenation of binary (denoted as $\texttt{bin}(v)$) representations of the amounts.

$$\mathbf{a}_L = (\texttt{bin}(v_1) \parallel \texttt{bin}(v_2) \parallel \ldots \parallel \texttt{bin}(v_m))$$
$$\mathbf{a}_R \coloneqq \mathbf{a}_L - \mathbf{1}^{mn}$$

```
    // Decompose values
    //
    // Note that this effectively pads the set to a power of 2, which is required for the
    inner-product argument later.
    for (size_t j = 0; j < M; ++j)
    {
        for (size_t i = N; i-- > 0; )
        {
            if (j < sv.size() && (sv[j][i/8] & (((uint64_t)1)<<(i%8))))
            {
                aL[j*N+i] = rct::identity();
                aL8[j*N+i] = INV_EIGHT;
                aR[j*N+i] = aR8[j*N+i] = rct::zero();
            }
            else
            {
                aL[j*N+i] = aL8[j*N+i] = rct::zero();
                aR[j*N+i] = MINUS_ONE;
                aR8[j*N+i] = MINUS_INV_EIGHT;
            }
        }
    }
```

Listing 3: Decomposition of input amounts

Step 4: Note that the vectors $\mathbf{a}_L, \mathbf{a}_R$ too are multipled by 8^{-1} for the same reason as mentioned above. Next, we initialise the transcript and compute the Pedersen commitment to $\mathbf{a}_L, \mathbf{a}_R$ as

$$A = \left(\frac{\alpha}{8}\right) * G + \sum_{i \in [n]} \left(\left(\frac{\mathbf{a}_L[i]}{8}\right) * \mathbf{G}[i] + \left(\frac{\mathbf{a}_R[i]}{8}\right) * \mathbf{H}[i] \right).$$

```
    // This is a Fiat-Shamir transcript
    rct::key transcript = copy(initial_transcript);
    transcript = transcript_update(transcript, rct::hash_to_scalar(V));

    // A
    rct::key alpha = rct::skGen();
    rct::key pre_A = vector_exponent(aL8, aR8);
    rct::key A;
    sc_mul(temp.bytes, alpha.bytes, INV_EIGHT.bytes);
    rct::addKeys(A, pre_A, rct::scalarmultBase(temp));
```

Listing 4: Initialise transcript and compute A

Step 5: We then draw up the challenges $y, z \in \mathbb{Z}_p$ from the transcript and compute constant vectors depending only on the challenges. Namely, we compute:

$$\mathbf{d} = \left(z^2 \cdot \mathbf{2}^n, z^4 \cdot \mathbf{2}^n, \ldots, z^{2m} \cdot \mathbf{2}^n\right) \in \mathbb{Z}_p^{mn}$$
$$\mathbf{y}^{mn+2} = \left(1, y, y^2, \ldots, y^{mn+1}\right)$$

where $\mathbf{w}^k = (1, w, \ldots, w^{k-1})$ for any scalar $w \in \mathbb{Z}_p$. Note that while drawing up challenges, we need to ensure that we do not encounter `rct::zero()`. If we do encounter a 0, we must re-compute the challenges from the beginning.

```
    // Challenges
    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;
    }
    rct::key z_squared;
    sc_mul(z_squared.bytes, z.bytes, z.bytes);

    // Windowed vector
    // d[j*N+i] = z**(2*(j+1)) * 2**i
    //
    // We compute this iteratively in order to reduce scalar operations.
    rct::keyV d(MN, rct::zero());
    d[0] = z_squared;
    for (size_t i = 1; i < N; i++)
    {
        sc_mul(d[i].bytes, d[i-1].bytes, TWO.bytes);
    }

    for (size_t j = 1; j < M; j++)
    {
        for (size_t i = 0; i < N; i++)
        {
            sc_mul(d[j*N+i].bytes, d[(j-1)*N+i].bytes, z_squared.bytes);
        }
    }

    rct::keyV y_powers = vector_of_scalar_powers(y, MN+2);
```

Listing 5: Computing challenges

Step 6: After all the above setup, the prover is ready to compute the witnesses before building the weighted inner product relations.

$$\begin{aligned}
\mathbf{a}'_L &= \mathbf{a}_L - z \cdot \mathbf{1}^{mn}, \\
\mathbf{a}'_R &= \mathbf{a}_R + \mathbf{d} \circ \overleftarrow{y}^{mn} + z \cdot \mathbf{1}^{mn}, \\
\alpha' &= \alpha + \sum_{j=1}^{m} z^{2j} y^{mn+1} \cdot \gamma_j.
\end{aligned}$$

```
    // Prepare inner product terms
    rct::keyV aL1 = vector_subtract(aL, z);

    rct::keyV aR1 = vector_add(aR, z);
    rct::keyV d_y(MN);
    for (size_t i = 0; i < MN; i++)
    {
        sc_mul(d_y[i].bytes, d[i].bytes, y_powers[MN-i].bytes);
    }
    aR1 = vector_add(aR1, d_y);

    rct::key alpha1 = alpha;
    temp = ONE;
    for (size_t j = 0; j < sv.size(); j++)
    {
        sc_mul(temp.bytes, temp.bytes, z_squared.bytes);
        sc_mul(temp2.bytes, y_powers[MN+1].bytes, temp.bytes);
        sc_mul(temp2.bytes, temp2.bytes, gamma[j].bytes);
        sc_add(alpha1.bytes, alpha1.bytes, temp2.bytes);
    }
```

Listing 6: Computing effective witness vectors

Step 7: The last step in the prover's algorithm is computing the weighted inner product argument. Prover runs the following recursive protocol:

Inputs: $(\mathbf{G}', \mathbf{H}' \in \mathbb{G}_l^n, G, H;\ \mathbf{a}', \mathbf{b}' \in \mathbb{Z}_p^n, \alpha')$

- Set $n' = \frac{n}{2}$
- Computing weighted inner products:

$$c_L = \langle \mathbf{a}'[0:n'],\ \overrightarrow{y}^{n'} \circ \mathbf{b}'[n':n] \rangle$$
$$c_R = \langle y^{n'} \cdot \mathbf{a}'[n':n],\ \overrightarrow{y}^{n'} \circ \mathbf{b}'[0:n'] \rangle$$

- Compute L, R (note the division of scalars by 8):

$$L = \left(\frac{y^{-n'}}{8} \cdot \mathbf{a}'[0:n']\right) * \mathbf{G}'[n':n] + \left(\frac{1}{8} \cdot \mathbf{b}'[n':n]\right) * \mathbf{H}'[0:n'] + \frac{c_L}{8} * H + \frac{d_L}{8} * G,$$
$$R = \left(\frac{y^{n'}}{8} \cdot \mathbf{a}'[n':n]\right) * \mathbf{G}'[0:n'] + \left(\frac{1}{8} \cdot \mathbf{b}'[0:n']\right) * \mathbf{H}'[n':n] + \frac{c_R}{8} * H + \frac{d_R}{8} * G.$$

  for $d_L, d_R \leftarrow \mathbb{Z}_p$.

- Update the generator vectors:

$$\mathbf{G}' \leftarrow e^{-1} * \mathbf{G}'[0:n'] + (ey^{-n'}) * \mathbf{G}'[n':n]$$
$$\mathbf{H}' \leftarrow e * \mathbf{H}'[0:n'] + e^{-n'} * \mathbf{H}'[n':n]$$

- Update witness vectors:

$$\mathbf{a}' \leftarrow e \cdot \mathbf{a}'[0:n'] + (e^{-1}y^{n'}) \cdot \mathbf{a}'[n':n]$$
$$\mathbf{b}' \leftarrow e^{-1} \cdot \mathbf{b}'[0:n'] + e \cdot \mathbf{b}'[n':n]$$
$$\alpha' \leftarrow \alpha' + e^2 d_L + e^{-2} d_R$$

```
    // Inner-product rounds
    while (nprime > 1)
    {
        nprime /= 2;

        rct::key cL = weighted_inner_product(slice(aprime, 0, nprime), slice(bprime, nprime, bprime.size()), y);
        rct::key cR = weighted_inner_product(vector_scalar(slice(aprime, nprime, aprime.size()), y_powers[nprime]), slice(bprime, 0, nprime), y);

        rct::key dL = rct::skGen();
        rct::key dR = rct::skGen();

        L[round] = compute_LR(nprime, yinvpow[nprime], Gprime, nprime, Hprime, 0, aprime, 0, bprime, nprime, cL, dL);
        R[round] = compute_LR(nprime, y_powers[nprime], Gprime, 0, Hprime, nprime, aprime, nprime, bprime, 0, cR, dR);

        const rct::key challenge = transcript_update(transcript, L[round], R[round]);
        if (challenge == rct::zero())
        {
            MINFO("challenge is 0, trying again");
            goto try_again;
        }

        const rct::key challenge_inv = invert(challenge);

        sc_mul(temp.bytes, yinvpow[nprime].bytes, challenge.bytes);
        hadamard_fold(Gprime, challenge_inv, temp);
        hadamard_fold(Hprime, challenge, challenge_inv);

        sc_mul(temp.bytes, challenge_inv.bytes, y_powers[nprime].bytes);
        aprime = vector_add(vector_scalar(slice(aprime, 0, nprime), challenge), vector_scalar(slice(aprime, nprime, aprime.size()), temp));
```

```
        bprime = vector_add(vector_scalar(slice(bprime, 0, nprime), challenge_inv),
    vector_scalar(slice(bprime, nprime, bprime.size()), challenge));

        rct::key challenge_squared;
        sc_mul(challenge_squared.bytes, challenge.bytes, challenge.bytes);
        rct::key challenge_squared_inv = invert(challenge_squared);
        sc_muladd(alpha1.bytes, dL.bytes, challenge_squared.bytes, alpha1.bytes);
        sc_muladd(alpha1.bytes, dR.bytes, challenge_squared_inv.bytes, alpha1.bytes);

        ++round;
    }
```

Listing 7: Weighted inner product argument

Step 8: The final round of the inner product argument consists of the following:

- Compute commitments to the 1-sized $\mathbf{a}', \mathbf{b}'$ vectors after $\log_2(mn)$ inner-product rounds:

$$A = \left(\frac{r'}{8}\right) * \mathbf{G}'[0] + \left(\frac{s'}{8}\right) * \mathbf{H}'[0] + \left(\frac{yr' \cdot \mathbf{b}'[0] + ys' \cdot \mathbf{a}'[0]}{8}\right) * H + \left(\frac{\delta'}{8}\right) * G,$$
$$B = \left(\frac{yr's'}{8}\right) * G + \left(\frac{\eta'}{8}\right) * H,$$

  where $r', s', \delta', \eta' \leftarrow \mathbb{Z}_p$.

- Update scalar proof elements:

$$\begin{aligned} r' &\leftarrow r' + e\mathbf{a}'[0], \\ r' &\leftarrow s' + e\mathbf{b}'[0], \\ r' &\leftarrow \eta' + e\delta' + e^2\alpha'. \end{aligned}$$

```
    // Final round computations
    rct::key r = rct::skGen();
    rct::key s = rct::skGen();
    rct::key d_ = rct::skGen();
    rct::key eta = rct::skGen();

    std::vector<MultiexpData> A1_data;
    A1_data.reserve(4);
    A1_data.resize(4);

    sc_mul(A1_data[0].scalar.bytes, r.bytes, INV_EIGHT.bytes);
    A1_data[0].point = Gprime[0];

    sc_mul(A1_data[1].scalar.bytes, s.bytes, INV_EIGHT.bytes);
    A1_data[1].point = Hprime[0];

    sc_mul(A1_data[2].scalar.bytes, d_.bytes, INV_EIGHT.bytes);
    ge_p3 G_p3;
    ge_frombytes_vartime(&G_p3, rct::G.bytes);
    A1_data[2].point = G_p3;

    sc_mul(temp.bytes, r.bytes, y.bytes);
    sc_mul(temp.bytes, temp.bytes, bprime[0].bytes);
    sc_mul(temp2.bytes, s.bytes, y.bytes);
    sc_mul(temp2.bytes, temp2.bytes, aprime[0].bytes);
    sc_add(temp.bytes, temp.bytes, temp2.bytes);
    sc_mul(A1_data[3].scalar.bytes, temp.bytes, INV_EIGHT.bytes);
    ge_p3 H_p3;
    ge_frombytes_vartime(&H_p3, rct::H.bytes);
    A1_data[3].point = H_p3;

    rct::key A1 = multiexp(A1_data, 0);

    sc_mul(temp.bytes, r.bytes, y.bytes);
    sc_mul(temp.bytes, temp.bytes, s.bytes);
    sc_mul(temp.bytes, temp.bytes, INV_EIGHT.bytes);
    sc_mul(temp2.bytes, eta.bytes, INV_EIGHT.bytes);
```

```
    rct::key B;
    rct::addKeys2(B, temp2, temp, rct::H);

    rct::key e = transcript_update(transcript, A1, B);
    rct::key e_squared;
    sc_mul(e_squared.bytes, e.bytes, e.bytes);

    rct::key r1;
    sc_muladd(r1.bytes, aprime[0].bytes, e.bytes, r.bytes);

    rct::key s1;
    sc_muladd(s1.bytes, bprime[0].bytes, e.bytes, s.bytes);

    rct::key d1;
    sc_muladd(d1.bytes, d_.bytes, e.bytes, eta.bytes);
    sc_muladd(d1.bytes, alpha1.bytes, e_squared.bytes, d1.bytes);

    return BulletproofPlus(std::move(V), A, A1, B, r1, s1, d1, std::move(L), std::move(R));
```

Listing 8: Final round

4.3 Verifier's Algorithm

The verification function takes input multiple `BulletproofPlus` proofs and verifies them in an aggregated manner.

Step 1: The first step is a basic setup.

```
// Given a batch of range proofs, determine if they are all valid
bool bulletproof_plus_VERIFY(const std::vector<const BulletproofPlus*> &proofs)
{
    init_exponents();

    const size_t logN = 6;
    const size_t N = 1 << logN;

    // Set up
    size_t max_length = 0; // size of each of the longest proof's inner-product vectors
    size_t nV = 0; // number of output commitments across all proofs
    size_t inv_offset = 0;
    size_t max_logM = 0;

    std::vector<bp_plus_proof_data_t> proof_data;
    proof_data.reserve(proofs.size());
```

Listing 9: Verification setup

Step 2: Iterate over each proof and run sanity checks on the proof elements, reconstruct the challenges $(y_j, z_j, \mathbf{x}_j, e_j) \in \mathbb{Z}_p$ for all $j \in [m]$ and batch invert the $\{y_j\}_{j\in[m]}$ challenges. Note that the inner-product challenges are denoted by $\mathbf{x}_j = (x_{j,1}, x_{j,2}, \ldots, x_{j,\log_2(n)})$.

```
    for (const BulletproofPlus *p: proofs)
    {
        const BulletproofPlus &proof = *p;

        // Sanity checks
        CHECK_AND_ASSERT_MES(is_reduced(proof.r1), false, "Input scalar not in range");
        CHECK_AND_ASSERT_MES(is_reduced(proof.s1), false, "Input scalar not in range");
        CHECK_AND_ASSERT_MES(is_reduced(proof.d1), false, "Input scalar not in range");

        CHECK_AND_ASSERT_MES(proof.V.size() >= 1, false, "V does not have at least one
    element");
        CHECK_AND_ASSERT_MES(proof.L.size() == proof.R.size(), false, "Mismatched L and R
    sizes");
        CHECK_AND_ASSERT_MES(proof.L.size() > 0, false, "Empty proof");

        max_length = std::max(max_length, proof.L.size());
        nV += proof.V.size();

        bp_plus_proof_data_t pd;
```

```
        // Reconstruct the challenges
        rct::key transcript = copy(initial_transcript);
        transcript = transcript_update(transcript, rct::hash_to_scalar(proof.V));
        pd.y = transcript_update(transcript, proof.A);
        CHECK_AND_ASSERT_MES(!(pd.y == rct::zero()), false, "y == 0");
        pd.z = transcript = rct::hash_to_scalar(pd.y);
        CHECK_AND_ASSERT_MES(!(pd.z == rct::zero()), false, "z == 0");

        // Determine the number of inner-product rounds based on proof size
        size_t M;
        for (pd.logM = 0; (M = 1<<pd.logM) <= maxM && M < proof.V.size(); ++pd.logM);
        CHECK_AND_ASSERT_MES(proof.L.size() == 6+pd.logM, false, "Proof is not the expected
     size");
        max_logM = std::max(pd.logM, max_logM);

        const size_t rounds = pd.logM+logN;
        CHECK_AND_ASSERT_MES(rounds > 0, false, "Zero rounds");

        // The inner-product challenges are computed per round
        pd.challenges.resize(rounds);
        for (size_t j = 0; j < rounds; ++j)
        {
            pd.challenges[j] = transcript_update(transcript, proof.L[j], proof.R[j]);
            CHECK_AND_ASSERT_MES(!(pd.challenges[j] == rct::zero()), false, "challenges[j]
    == 0");
        }

        // Final challenge
        pd.e = transcript_update(transcript,proof.A1,proof.B);
        CHECK_AND_ASSERT_MES(!(pd.e == rct::zero()), false, "e == 0");

        // Batch scalar inversions
        pd.inv_offset = inv_offset;
        for (size_t j = 0; j < rounds; ++j)
            to_invert.push_back(pd.challenges[j]);
        to_invert.push_back(pd.y);
        inv_offset += rounds + 1;
        proof_data.push_back(pd);
    }
```

Listing 10: Reconstructing challenges across multiple proofs

Step 3: The idea of aggregating verification of multiple `BulletproofPlus` proofs is that we can combine the single multi-scalar multiplication checks of each proof using random weights and verify if it evaluates to identity. Let's say the j-th `BulletproofPlus` proof for m_j different amounts is given as:

$$\Pi_j = \left\{ A_j, A_{1,j}, B_j \in \mathbb{G}_l, (\mathbf{L}_j, \mathbf{R}_j) = \left(L_{i,j}, R_{i,j}\right)_{i=1}^{\log_2(nm_j)} \in \mathbb{G}_l^{2\times\log_2(nm_j)}, r'_j, s'_j, \delta'_j \in \mathbb{Z}_p \right\}$$

This proof can be verified using a single multi-scalar multiplication of the form [2]:

$$\begin{aligned} M_j = \quad & \left(e_j \cdot r'_j \cdot \mathbf{s}_j + z_j e_j^2 \mathbf{1}^{m_j n}\right) * \mathbf{G}[0:m_j] + \\ & \left(e_j \cdot s'_j \cdot \mathbf{s}'_j - z_j e_j^2 \cdot \mathbf{1}^{m_j n} - e_j^2 \cdot \mathbf{d}_j \circ \overleftarrow{y}_j^{mn}\right) * \mathbf{H}[0:m_j] + \\ & \left(r'_j \odot s'_j - e_j^2 \zeta(y_j, z_j)\right) * H + \\ & \left(\delta'_j\right) * G + \\ & \left(-e_j^2\right) * A_j + \\ & \left(-e_j^2 y_j^{mn+1} \cdot z_j^2 \cdot \mathbf{z}_j^m\right) * \mathbf{V}_j + \\ & (\mathbf{x}_{L,j}) * \mathbf{L}_j \ + (\mathbf{x}_{R,j}) * \mathbf{R}_j + \\ & (-e_j) * A_{1,j} - B_j \end{aligned} \tag{1}$$

where we have

$$\begin{aligned}
\zeta(y_j, z_j) &= (z_j - z_j^2)y_j \cdot \langle \mathbf{1}^{m_j n}, \mathbf{y}_j^{nm_j} \rangle - z_j y_j^{mn+1} \cdot \langle \mathbf{1}^{m_j n}, \mathbf{d}_j \rangle, \\
\mathbf{x}_{L,j} &= e_j^2 \cdot (-x_{1,j}^2, -x_{2,j}^2, \ldots, -x_{\log_2(nm_j)}^2), \\
\mathbf{x}_{R,j} &= e_j^2 \cdot (-x_{1,j}^{-2}, -x_{2,j}^{-2}, \ldots, -x_{\log_2(nm_j)}^{-2}),
\end{aligned}$$

and $\mathbf{s}_j = (s_1, s_2, \ldots, s_{m_j n}), \mathbf{s}'_j = (s_1^{-1}, s_2^{-1}, \ldots, s_{m_j n}^{-1}) \in \mathbb{Z}_p^{m_j n}$ such that for all $i \in [nm_j]$

$$s_i = \prod_{k=1}^{\log_2(nm_j)} x_{k,j}^{b(i,k)} \quad \text{where} \quad b(i,k) = \begin{cases} 1 & \text{if } k\text{-th bit of } (i-1) \text{ is } 1 \\ -1 & \text{otherwise.} \end{cases}$$

Note that the generator vectors $\mathbf{G}, \mathbf{H}$ are of size `maxMN` of which only the first m_j terms would be used for verifying the proof Π_j. The proof Π_j is considered to be a valid `BulletproofPlus` proof if M_j equals to the identity $\mathcal{O}$ (point at infinity).

The key thing to note is that we can combine multiple of such proofs and verify all of them at once using random weights as

$$\sum_{j=1}^{\text{num\_proofs}} w_j * M_j \stackrel{?}{=} \mathcal{O}.$$

This greatly speeds up batched aggregated verification because of the Pippenger's multi-exponentiation algorithm. Lastly, while computing the proof, recall that the quantities $A, A_1, B, \mathbf{L}, \mathbf{R}, \mathbf{V} \in \mathbb{G}_l$ were computed using scalar witnesses which were divided by 8. In the verification of the proof, the proof elements $A, A_1, B, \mathbf{L}, \mathbf{R}, \mathbf{V} \in \mathbb{G}_l$ are first multiplied by 8 so as to ensure that all of them lie in the prime order subgroup [16]. The net effect of this is that it cancels out the 8 in the final multi-scalar multiplication check in equation (1). We explain the parts of the verifier code for completeness.

Step 3.1: We first need to initialise the scalars of the common generators $(\mathbf{G}, \mathbf{H}, G, H)$ with 0. Note that the scalar multiplicands of $(\mathbf{G}, \mathbf{H}, G, H)$ across different proofs (which are to be verified) are linearly combined using random weights.

```
    // Weights and aggregates
    //
    // The idea is to take the single multiscalar multiplication used in the verification
    //  of each proof in the batch and weight it using a random weighting factor, resulting
    //  in just one multiscalar multiplication check to zero for the entire batch.
    // We can further simplify the verifier complexity by including common group elements
    //  only once in this single multiscalar multiplication.
    // Common group elements' weighted scalar sums are tracked across proofs for this
    reason.
    //
    // To build a multiscalar multiplication for each proof, we use the method described in
    //  Section 6.1 of the preprint. Note that the result given there does not account for
    //  the construction of the inner-product inputs that are produced in the range proof
    //  verifier algorithm; we have done so here.
    rct::key G_scalar = rct::zero();
    rct::key H_scalar = rct::zero();
    rct::keyV Gi_scalars(maxMN, rct::zero());
    rct::keyV Hi_scalars(maxMN, rct::zero());
```

Listing 11: Initialise scalars multiplicands of ($\mathbf{G}$, $\mathbf{H}$, G, H) with 0

Step 3.2: When we start to process each `BulletproofPlus` proof, we first do some sanity checks on the proof size and its elements. Specifically it is checked that the size of the vector $\mathbf{L}$ is equal to $\log_2(mn) = 6 + \log_2(m)$ since $n = 64$. Furthermore, we generate random scalars (called as weights) to linearly combine the scalar multiplicands of the reference string elements. Lastly, all the group elements are multiplied by scalar $8 \in \mathbb{Z}_p$ so as to ensure that they lie in the prime order subgroup $\mathbb{G}_l$ and to zeroise the torsion elements (if any). Note that while generating an honest proof, we divide our scalars by $8 \in \mathbb{Z}_p$ before computing group elements of the proof. For example,

$$A = \left(\frac{\alpha}{8}\right) * G + \sum_{i \in [n]} \left(\left(\frac{\mathbf{a}_L[i]}{8}\right) * \mathbf{G}[i] + \left(\frac{\mathbf{a}_R[i]}{8}\right) * \mathbf{H}[i] \right).$$

During the proof verification, this element A is multiplied by $8 \in \mathbb{Z}_p$ before proceeding to the core verification equation. Therefore, what we get is:

$$8 * A = \alpha * G + \sum_{i \in [n]} \left(\mathbf{a}_L[i] * \mathbf{G}[i] + \mathbf{a}_R[i] * \mathbf{H}[i] \right),$$

which is the correct expression of the Pedersen commitment to the vectors $\mathbf{a}_L, \mathbf{a}_R \in \mathbb{Z}_p^{mn}$. This is how the pre-multiplied $\frac{1}{8} \in \mathbb{Z}_p$ to the scalars while proof generation are cancelled off at the beginning of verification processing.

```
    // Process each proof and add to the weighted batch
    for (const BulletproofPlus *p: proofs)
    {
        const BulletproofPlus &proof = *p;
        const bp_plus_proof_data_t &pd = proof_data[proof_data_index++];

        CHECK_AND_ASSERT_MES(proof.L.size() == 6+pd.logM, false, "Proof is not the expected size");
        const size_t M = 1 << pd.logM;
        const size_t MN = M*N;

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

        // Rescale previously offset proof elements
        //
        // This ensures that all such group elements are in the prime-order subgroup.
        proof8_V.resize(proof.V.size()); for (size_t i = 0; i < proof.V.size(); ++i) rct::scalarmult8(proof8_V[i], proof.V[i]);
        proof8_L.resize(proof.L.size()); for (size_t i = 0; i < proof.L.size(); ++i) rct::scalarmult8(proof8_L[i], proof.L[i]);
        proof8_R.resize(proof.R.size()); for (size_t i = 0; i < proof.R.size(); ++i) rct::scalarmult8(proof8_R[i], proof.R[i]);
        ge_p3 proof8_A1;
        ge_p3 proof8_B;
        ge_p3 proof8_A;
        rct::scalarmult8(proof8_A1, proof.A1);
        rct::scalarmult8(proof8_B, proof.B);
        rct::scalarmult8(proof8_A, proof.A);
```

Listing 12: Sanity checks on each proof

Step 3.3: Compute the scalar multiplicand of the commitment vector $\mathbf{V}_j$ for the j-th proof:

$$w_j \cdot \left(-e_j^2 y_j^{mn+1} \cdot z_j^2 \cdot \mathbf{z}_j^m \right)$$

```
        // V_j: -e**2 * z**(2*j+1) * y**(MN+1) * weight
        rct::key e_squared;
        sc_mul(e_squared.bytes, pd.e.bytes, pd.e.bytes);

        rct::key z_squared;
        sc_mul(z_squared.bytes, pd.z.bytes, pd.z.bytes);

        sc_sub(temp.bytes, ZERO.bytes, e_squared.bytes);
        sc_mul(temp.bytes, temp.bytes, y_MN_1.bytes);
        sc_mul(temp.bytes, temp.bytes, weight.bytes);
        for (size_t j = 0; j < proof8_V.size(); j++)
        {
```

```
            sc_mul(temp.bytes, temp.bytes, z_squared.bytes);
            multiexp_data.emplace_back(temp, proof8_V[j]);
        }
```

Listing 13: Scalar multiplicand of $\mathbf{V}_j$

Step 3.3: Compute the scalar multiplicand of the elements $B_j, A_j, A_{1,j}, G$ for the j-th proof:

$$\begin{aligned} B_j: &\quad w_j \cdot (-1) \\ A_j: &\quad w_j \cdot \left(-e_j^2\right) \\ A_{1,j}: &\quad w_j \cdot (-e_j) \\ G_{\text{scalar}} \mathrel{+}= &\quad w_j \cdot \left(\delta_j'\right) \end{aligned}$$

```
        // B: -weight
        sc_mul(temp.bytes, MINUS_ONE.bytes, weight.bytes);
        multiexp_data.emplace_back(temp, proof8_B);

        // A1: -weight*e
        sc_mul(temp.bytes, temp.bytes, pd.e.bytes);
        multiexp_data.emplace_back(temp, proof8_A1);

        // A: -weight*e*e
        rct::key minus_weight_e_squared;
        sc_mul(minus_weight_e_squared.bytes, temp.bytes, pd.e.bytes);
        multiexp_data.emplace_back(minus_weight_e_squared, proof8_A);

        // G: weight*d1
        sc_muladd(G_scalar.bytes, weight.bytes, proof.d1.bytes, G_scalar.bytes);
```

Listing 14: Scalar multiplicand of B_j, A_j, $A_{1,j}$, G

Step 3.4: To compute the scalar multiplicands of H, we first compute the following scalar vector:

$$\mathbf{d}_j = \left(z_j^2 \cdot \mathbf{2}^n, z_j^4 \cdot \mathbf{2}^n, \ldots, z_j^{2m} \cdot \mathbf{2}^n\right) \in \mathbb{Z}_p^{mn}$$

$$H_{\text{scalar}} \mathrel{+}= w_j \cdot \left(r_j' s_j' y_j + e_j^2 (z_j - z_j^2) \sum_{k=1}^{mn} y_j^k + e_j^2 z_j y_j^{mn+1} \sum_{k=1}^{mn} \mathbf{d}_j[k] \right)$$

```
        // Windowed vector
        // d[j*N+i] = z**(2*(j+1)) * 2**i
        rct::keyV d(MN, rct::zero());
        d[0] = z_squared;
        for (size_t i = 1; i < N; i++)
        {
            sc_add(d[i].bytes, d[i-1].bytes, d[i-1].bytes);
        }

        for (size_t j = 1; j < M; j++)
        {
            for (size_t i = 0; i < N; i++)
            {
                sc_mul(d[j*N+i].bytes, d[(j-1)*N+i].bytes, z_squared.bytes);
            }
        }

        // More efficient computation of sum(d)
        rct::key sum_d;
        sc_mul(sum_d.bytes, TWO_SIXTY_FOUR_MINUS_ONE.bytes, sum_of_even_powers(pd.z, 2*M).
    bytes);

        // H: weight*( r1*y*s1 + e**2*( y**(MN+1)*z*sum(d) + (z**2-z)*sum(y) ) )
        rct::key sum_y = sum_of_scalar_powers(pd.y, MN);
        sc_sub(temp.bytes, z_squared.bytes, pd.z.bytes);
        sc_mul(temp.bytes, temp.bytes, sum_y.bytes);

        sc_mul(temp2.bytes, y_MN_1.bytes, pd.z.bytes);
        sc_mul(temp2.bytes, temp2.bytes, sum_d.bytes);
        sc_add(temp.bytes, temp.bytes, temp2.bytes);
```

```
        sc_mul(temp.bytes, temp.bytes, e_squared.bytes);
        sc_mul(temp2.bytes, proof.r1.bytes, pd.y.bytes);
        sc_mul(temp2.bytes, temp2.bytes, proof.s1.bytes);
        sc_add(temp.bytes, temp.bytes, temp2.bytes);
        sc_muladd(H_scalar.bytes, temp.bytes, weight.bytes, H_scalar.bytes);
```

Listing 15: Scalar multiplicand of H

Step 3.5: Compute the scalar vectors $\mathbf{s}_j, \mathbf{s}'_j \in \mathbb{Z}_p^{mn}$ as defined in Step 3 of the verifier's algorithm.

```
        // Compute the number of rounds for the inner-product argument
        const size_t rounds = pd.logM+logN;
        CHECK_AND_ASSERT_MES(rounds > 0, false, "Zero rounds");

        const rct::key *challenges_inv = &inverses[pd.inv_offset];
        const rct::key yinv = inverses[pd.inv_offset + rounds];

        // Compute challenge products
        challenges_cache.resize(1<<rounds);
        challenges_cache[0] = challenges_inv[0];
        challenges_cache[1] = pd.challenges[0];
        for (size_t j = 1; j < rounds; ++j)
        {
            const size_t slots = 1<<(j+1);
            for (size_t s = slots; s-- > 0; --s)
            {
                sc_mul(challenges_cache[s].bytes, challenges_cache[s/2].bytes, pd.challenges[j].bytes);
                sc_mul(challenges_cache[s-1].bytes, challenges_cache[s/2].bytes, challenges_inv[j].bytes);
            }
        }
```

Listing 16: Scalar vectors $\mathbf{s}_j$, $\mathbf{s}'_j \in \mathbb{Z}_p^{mn}$

Step 3.5: Cumulatively compute the scalar multiplicand vectors of $\mathbf{G}, \mathbf{H}$:

$$\mathbf{G}_{\text{scalar}} \mathrel{+}= w_j \cdot \left(e_j \cdot r'_j \cdot \mathbf{s}_j + z_j e_j^2 \mathbf{1}^{m_j n}\right)$$
$$\mathbf{H}_{\text{scalar}} \mathrel{+}= w_j \cdot \left(e_j \cdot s'_j \cdot \mathbf{s}'_j - z_j e_j^2 \cdot \mathbf{1}^{m_j n} - e_j^2 \cdot \mathbf{d}_j \circ \overleftarrow{y}_j^{mn}\right)$$

```
        // Gi and Hi
        rct::key e_r1_w_y;
        sc_mul(e_r1_w_y.bytes, pd.e.bytes, proof.r1.bytes);
        sc_mul(e_r1_w_y.bytes, e_r1_w_y.bytes, weight.bytes);
        rct::key e_s1_w;
        sc_mul(e_s1_w.bytes, pd.e.bytes, proof.s1.bytes);
        sc_mul(e_s1_w.bytes, e_s1_w.bytes, weight.bytes);
        rct::key e_squared_z_w;
        sc_mul(e_squared_z_w.bytes, e_squared.bytes, pd.z.bytes);
        sc_mul(e_squared_z_w.bytes, e_squared_z_w.bytes, weight.bytes);
        rct::key minus_e_squared_z_w;
        sc_sub(minus_e_squared_z_w.bytes, ZERO.bytes, e_squared_z_w.bytes);
        rct::key minus_e_squared_w_y;
        sc_sub(minus_e_squared_w_y.bytes, ZERO.bytes, e_squared.bytes);
        sc_mul(minus_e_squared_w_y.bytes, minus_e_squared_w_y.bytes, weight.bytes);
        sc_mul(minus_e_squared_w_y.bytes, minus_e_squared_w_y.bytes, y_MN.bytes);
        for (size_t i = 0; i < MN; ++i)
        {
            rct::key g_scalar = copy(e_r1_w_y);
            rct::key h_scalar;

            // Use the binary decomposition of the index
            sc_muladd(g_scalar.bytes, g_scalar.bytes, challenges_cache[i].bytes, e_squared_z_w.bytes);
            sc_muladd(h_scalar.bytes, e_s1_w.bytes, challenges_cache[(~i) & (MN-1)].bytes, minus_e_squared_z_w.bytes);

            // Complete the scalar derivation
            sc_add(Gi_scalars[i].bytes, Gi_scalars[i].bytes, g_scalar.bytes);
            sc_muladd(h_scalar.bytes, minus_e_squared_w_y.bytes, d[i].bytes, h_scalar.bytes);
            sc_add(Hi_scalars[i].bytes, Hi_scalars[i].bytes, h_scalar.bytes);

```

```
            // Update iterated values
            sc_mul(e_r1_w_y.bytes, e_r1_w_y.bytes, yinv.bytes);
            sc_mul(minus_e_squared_w_y.bytes, minus_e_squared_w_y.bytes, yinv.bytes);
        }
```

Listing 17: Scalar multiplicand vectors of $\mathbf{G}$, $\mathbf{H}$

Step 3.5: Compute the scalar multiplicand vectors of $\mathbf{L}_j, \mathbf{R}_j \in \mathbb{G}_l^{\log_2(nm_j)}$:

$$\begin{aligned}\mathbf{L}_j: &\quad w_j \cdot e_j^2 \cdot (-x_{1,j}^2, -x_{2,j}^2, \ldots, -x_{\log_2(nm_j)}^2),\\ \mathbf{R}_j: &\quad w_j \cdot e_j^2 \cdot (-x_{1,j}^{-2}, -x_{2,j}^{-2}, \ldots, -x_{\log_2(nm_j)}^{-2}).\end{aligned}$$

```
        // L_j: -weight*e*e*challenges[j]**2
        // R_j: -weight*e*e*challenges[j]**(-2)
        for (size_t j = 0; j < rounds; ++j)
        {
            sc_mul(temp.bytes, pd.challenges[j].bytes, pd.challenges[j].bytes);
            sc_mul(temp.bytes, temp.bytes, minus_weight_e_squared.bytes);
            multiexp_data.emplace_back(temp, proof8_L[j]);

            sc_mul(temp.bytes, challenges_inv[j].bytes, challenges_inv[j].bytes);
            sc_mul(temp.bytes, temp.bytes, minus_weight_e_squared.bytes);
            multiexp_data.emplace_back(temp, proof8_R[j]);
        }
```

Listing 18: Scalar multiplicand vectors of $\mathbf{L}_j$, $\mathbf{R}_j$

Step 3.6: Final verification check:

```
    // Verify all proofs in the weighted batch
    multiexp_data.emplace_back(G_scalar, rct::G);
    multiexp_data.emplace_back(H_scalar, rct::H);
    for (size_t i = 0; i < maxMN; ++i)
    {
        multiexp_data[i * 2] = {Gi_scalars[i], Gi_p3[i]};
        multiexp_data[i * 2 + 1] = {Hi_scalars[i], Hi_p3[i]};
    }
    if (!(multiexp(multiexp_data, 2 * maxMN) == rct::identity()))
    {
        MERROR("Verification failure");
        return false;
    }

    return true;
```

Listing 19: Batch Verification using a single multi-scalar multiplication

5 Vulnerabilities

5.1 Missing 'point at infinity' check on group elements

In the verification of a `BulletproofPlus` proof, we multiply the group elements $A, A_1, B, \mathbf{L}, \mathbf{R}, \mathbf{V}$ of the proof with 8 so as to ensure that if these group elements contained any torsion elements [5], they could be nullified off. Concretely, if a proof element has a torsion component $A' = A + K$ where $K \in \mathbb{G}_8$, then $8A' = 8A + 8K = 8A$ because $8K = \mathcal{O}$. However, we do not check if the resulting element A' is a point at infinity. If a malicious prover passes all the group elements of the proof as torsion elements except A and sets A to be of a certain value, the malicious prover could succeed in verifying a false proof as correct. We next describe the attack in detail.

Before we proceed, here is how the verification of a single `BulletproofPlus` proof using a single multi-scalar multiplication looks like.

$$\begin{aligned}
\left(e \cdot r_1 \cdot \mathbf{s} + ze^2\mathbf{1}^m\right) * \mathbf{G} \;+& \\
\left(e \cdot s_1 \cdot \mathbf{s}' - ze^2 \cdot \mathbf{1}^{mn} - e^2 \cdot \mathbf{d} \circ \overleftarrow{y}^{mn}\right) * \mathbf{H} \;+& \\
\left(r \odot s_1 - e^2\zeta(y,z)\right) * H \;+& \\
(d_1) * G \;+& \quad \stackrel{?}{=} \quad \mathcal{O} \\
\left(-e^2\right) * A \;+& \\
\left(-e^2 y^{mn+1} \cdot z^2 \cdot \mathbf{z}^m\right) * \mathbf{V} \;+& \\
(\mathbf{x}_L) * \mathbf{L} \;+ (\mathbf{x}_R) * \mathbf{R} \;+& \\
(-e) * A_1 - B&
\end{aligned}$$

where we have

$$\begin{aligned}
\zeta(y,z) &= (z - z^2)y \cdot \langle \mathbf{1}^{mn}, \mathbf{y}^{nm} \rangle - zy^{mn+1} \cdot \langle \mathbf{1}^{mn}, \mathbf{d} \rangle, \\
\mathbf{x}_L &= e^2 \cdot \left(-x_1^2, -x_2^2, \ldots, -x_{\log_2(nm)}^2\right), \\
\mathbf{x}_R &= e^2 \cdot \left(-x_1^{-2}, -x_2^{-2}, \ldots, -x_{\log_2(nm)}^{-2}\right),
\end{aligned}$$

and $\mathbf{s} = (s_1, s_2, \ldots, s_{mn}), \mathbf{s}' = (s_1^{-1}, s_2^{-1}, \ldots, s_{mn}^{-1}) \in \mathbb{Z}_p^{mn}$ such that for all $i \in [nm]$

$$s_i = \prod_{k=1}^{\log_2(nm)} x_k^{b(i,k)} \quad \text{where} \quad b(i,k) = \begin{cases} 1 & \text{if } k\text{-th bit of } (i-1) \text{ is } 1 \\ -1 & \text{otherwise.} \end{cases}$$

Attack: A malicious prover creates the following false proof:

$$\texttt{BulletproofPlus}(A, A_1, B, \mathbf{L}, \mathbf{R}, r_1, s_1, d_1)$$

such that $A_1, B \in \mathbb{G}_8, \mathbf{L}, \mathbf{R} \in \mathbb{G}_8^{\log_2(mn)}$, $\mathbf{V} \in \mathbb{G}_8^m$ and $r_1 = s_1 = d_1 = 0$. In this case, the terms corresponding to $G, A, \mathbf{V}, \mathbf{L}, \mathbf{R}, A_1, B$ would be $\mathcal{O}$. We can now set the value of A to be:

$$A = (z\mathbf{1}^m) * \mathbf{G} - (z\mathbf{1}^m + \mathbf{d} \circ \overleftarrow{y}^{mn}) * \mathbf{H} - \zeta(y,z) * H.$$

This means that a false proof would pass the verification check. However, this attack is *very unlikely* to be exploited because the expression of A depends on the challenges y, z but the challenges y, z themselves are drawn up by hashing the transcript with A. There is a circular dependency in the computation of A and the challenges y, z. This reduces the security of the protocol to pre-image resistance of the hashing function.

[5] The elements present in the small subgroup $\mathbb{G}_8$ of size 8 (of the curve Ed25519) are known as torsion elements. They are listed on line 115 in ../tests/unit_tests/bulletproofs_plus.cpp.

Recommendation: Although this attack is very difficult and impractical to exploit, it is better to insert the check if the group elements are points at infinity after we multiply them by 8. To be precise, we should have:

```
    // Rescale previously offset proof elements
    //
    // This ensures that all such group elements are in the prime-order subgroup.
    proof8_V.resize(proof.V.size());
    for (size_t i = 0; i < proof.V.size(); ++i) {
        rct::scalarmult8(proof8_V[i], proof.V[i]);
        // Point at infinity check
++      CHECK_AND_ASSERT_THROW_MES(proof8_V[i] != rct::identity(), "Commitment V cannot
    contain a point at infinity!");
    }
    proof8_L.resize(proof.L.size());
    for (size_t i = 0; i < proof.L.size(); ++i) {
        rct::scalarmult8(proof8_L[i], proof.L[i]);
        // Point at infinity check
++      CHECK_AND_ASSERT_THROW_MES(proof8_L[i] != rct::identity(), "Proof element L cannot
    contain a point at infinity!");
    }
    proof8_R.resize(proof.R.size());
    for (size_t i = 0; i < proof.R.size(); ++i) {
        rct::scalarmult8(proof8_R[i], proof.R[i]);
        // Point at infinity check
++      CHECK_AND_ASSERT_THROW_MES(proof8_R[i] != rct::identity(), "Proof element R cannot
    contain a point at infinity!");
    }
    ge_p3 proof8_A1;
    ge_p3 proof8_B;
    ge_p3 proof8_A;
    rct::scalarmult8(proof8_A1, proof.A1);
    rct::scalarmult8(proof8_B, proof.B);
    rct::scalarmult8(proof8_A, proof.A);

    // Point at infinity checks
++  CHECK_AND_ASSERT_THROW_MES(proof8_A1 != rct::identity(), "Proof element A1 cannot be a
    point at infinity!");
++  CHECK_AND_ASSERT_THROW_MES(proof8_A != rct::identity(), "Proof element A cannot be a
    point at infinity!");
++  CHECK_AND_ASSERT_THROW_MES(proof8_B != rct::identity(), "Proof element B cannot be a
    point at infinity!");
```

Listing 20: Point at infinity check after torsion check

Response from MRL: Even if the implementation were to use a prime-order curve group (and therefore not need to use any multiplication or division by a cofactor), a prover could still provide identity elements or zero scalars as proof elements; this is not inherently invalid. Further, the infeasibility of such a malicious prover to construct a valid proof by solving the identified relation seems to be precisely the result of proper use of the Fiat-Shamir construction! This appears to have nothing to do with the use of torsion elimination, and as a result, it does not appear that any security is gained or risk eliminated by such a check against identity elements.

5.2 Missing check if challenge 'e' is 0

Challenge e is computed by hashing the transcript and relevant proof elements (A_1, B) during the last round of the weighted inner-product protocol by the prover. However, unlike the other challenges, e is not checked if it equals to $0 \in \mathbb{Z}_p$.

```
    rct::key e = transcript_update(transcript, A1, B);

++  if (e == rct::zero())
++  {
++      MINFO("challenge e is 0, trying again");
++      goto try_again;
++  }
```

```
rct::key e_squared;
sc_mul(e_squared.bytes, e.bytes, e.bytes);
```

Listing 21: Missing zero value check of challenge e.

Recommendation: This can be fixed by a simple check and re-doing the computation if the check fails, just how the check is already done for other challenges like y, z.

6 Weaknesses

6.1 Inconsistency in verifier transcript computation

Verifier transcript is generated before multiplication by 8: as a result the proof elements that include small order subgroup elements will change the transcript. Specifically, the challenges generated by the verifier would include the torsion elements from the proof elements while hashing. For example, a malicious prover can change the amount commitments to include a torsion component: $V'_j = V_j + kT,\ j \in [m]$ where $k \in [8], \langle T \rangle = \mathbb{G}_8$. When such a proof with commitments $\{V'_j\}_{j\in[m]}$ is verified, the transcript challenges would be computed by the verifier before we filter out the torsion component in the commitments. Note that the verification would not pass because the verifier's challenge computed would not match the one used while generating the proof. We suggest that the implementation should comply with the transcript computation in the paper and the challenges should be computed after group elements are multiplied by 8.

Note that only re-ordering of the verifier transcript computation would not be sufficient because that would allow a malicious prover to create false proofs passing the verification. In that case, the strategy of multiplication of group elements by 8 to filter out torsion components might not work any more. To ensure that the group elements are in the prime order subgroup, it might be better to switch to the more expensive route: multiplying the group elements by the prime-order subgroup's order p.

Response from MRL: To avoid malleability, it is important that the prover not be able to swap out proof elements in a way that yields identical challenges and a valid proof. The current implementation does precisely this: any attempt to change proof elements (whether by including nonzero torsion, or by any other means) will change the proof transcripts and correctly result in failed verification.

6.2 Incomplete condition for checking 'power of two'

In the function `vector_of_scalar_powers()`, the check for n to be a power of two on line 246 in the file `../src/ringct/bulletproofs_plus.cc` does not comply with the case $n = 0$. The check should not pass but it passes for the case $n = 0$.

```
241 // Given a scalar, construct the sum of its powers from 2 to n (where n is a power of 2):
242 //
243 // Output x**2 + x**4 + x**6 + ... + x**n
244 static rct::key sum_of_even_powers(const rct::key &x, size_t n)
245 {
246     CHECK_AND_ASSERT_THROW_MES((n & (n - 1)) == 0, "Need n to be a power of 2");
247 
248     rct::key x1 = copy(x);
249     sc_mul(x1.bytes, x1.bytes, x1.bytes);
250 
251     rct::key res = copy(x1);
252     while (n > 2)
253     {
254         sc_muladd(res.bytes, x1.bytes, res.bytes, res.bytes);
255         sc_mul(x1.bytes, x1.bytes, x1.bytes);
256         n /= 2;
257     }
258 
259     return res;
260 }
```

Recommendation: Change the check to:

```
    CHECK_AND_ASSERT_THROW_MES (!n && (n & (n - 1)) == 0, "Need n to be a power of 2");
```

Similarly, in the function `sum_of_scalar_powers()` on line 274 in the same file, the check does not perform correctly if $n = 0$. The function wrongly returns $x \in \mathbb{Z}_p$ when $n = 0$.

```
262 // Given a scalar, return the sum of its powers from 1 to n
263 //
264 // Output x**1 + x**2 + x**3 + ... + x**n
265 static rct::key sum_of_scalar_powers(const rct::key &x, size_t n)
266 {
267     rct::key res = ONE;
268     if (n == 1)
269         return res;
270 
271     n += 1;
272     rct::key x1 = copy(x);
273 
274     const bool is_power_of_2 = (n & (n - 1)) == 0;
275     if (is_power_of_2)
276     {
277         sc_add(res.bytes, res.bytes, x1.bytes);
278         while (n > 2)
279         {
280             sc_mul(x1.bytes, x1.bytes, x1.bytes);
281             sc_muladd(res.bytes, x1.bytes, res.bytes, res.bytes);
282             n /= 2;
283         }
284     }
285     else
286     {
287         rct::key prev = x1;
288         for (size_t i = 1; i < n; ++i)
289         {
290             if (i > 1)
291                 sc_mul(prev.bytes, prev.bytes, x1.bytes);
292             sc_add(res.bytes, res.bytes, prev.bytes);
293         }
294     }
295     sc_sub(res.bytes, res.bytes, ONE.bytes);
296 
297     return res;
298 }
```

Recommendation: Change the condition to:

```
    const bool is_power_of_two = (!n) && (n & (n - 1)) == 0;
```

6.3 Input parameter edge case consideration in `vector_of_scalar_powers()`

In the function `vector_of_scalar_powers(const rct::key &x, size_t n)`, when $n = 0$, we return an empty vector. This will cause a segmentation fault if a user tries to access an empty vector. We recommend throwing an error message when the function is called with $n = 0$ so it allows the user to know that the input parameter passed is not right.

```
// Given a scalar, construct a vector of its powers:
//
// Output (1,x,x**2,...,x**{n-1})
static rct::keyV vector_of_scalar_powers(const rct::key &x, size_t n)
{
    rct::keyV res(n);
--   if (n == 0)
--       return res;
++   CHECK_AND_ASSERT_THROW_MES(n != 0, "Need n to be non-zero")
    res[0] = rct::identity();
    if (n == 1)
        return res;
    res[1] = x;
    for (size_t i = 2; i < n; ++i)
    {
        sc_mul(res[i].bytes, res[i-1].bytes, x.bytes);
    }
    return res;
```

```
}
```

Note: The issues listed in § 6.3, § 6.2 are not triggered in the current Bulletproofs+ implementation but we suggest adding the necessary checks for the sake of completeness, since these functions could be used in future outside the Bulletproofs+ module.

6.4 Redundant Assertions

We noticed some assertions in the code which are always either true or false and lead to redundant checks. We suggest to either correct these assertions or remove them if unnecessary.

1. In the function `multiexp()`, `STRAUS_SIZE_LIMIT` is defined as 232 at the start of the program. Therefore the assertion in the function (line 94) to check if `STRAUS_SIZE_LIMIT` ≤ 232 would always be true.

```
90  static inline rct::key multiexp(const std::vector<MultiexpData> &data, size_t HiGi_size)
91  {
92      if (HiGi_size > 0)
93      {
94          static_assert(232 <= STRAUS_SIZE_LIMIT, "Straus in precalc mode can only be
    calculated till STRAUS_SIZE_LIMIT");
95          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()));
96      }
97      else
98      {
99          return data.size() <= 95 ? straus(data, NULL, 0) : pippenger(data, NULL, 0,
    get_pippenger_c(data.size()));
100     }
101 }
```

Listing 22: Redundant assertion on line 94

2. In the function `bulletproof_plus_PROVE()`, we declare `logN` $= 6$ at the very beginning on line 816. This is because `N` $= 64$ since each proof is a 64-bit range proof. Clearly, the inner product argument of an aggregated Bulletproofs+ will consist of $\log_2(mn) = \log_2(m) + \log_2(n) > 6$ rounds. Therefore, an assertion of the form `rounds` > 0 on line 868 becomes redundant.

```
811     // Given a batch of range proofs, determine if they are all valid
812     bool bulletproof_plus_VERIFY(const std::vector<const BulletproofPlus*> &proofs)
813     {
814         init_exponents();
815
816         const size_t logN = 6;
817         const size_t N = 1 << logN;
818         ...
819         ...
```

```
859             ...
860             ...
861             // Determine the number of inner-product rounds based on proof size
862             size_t M;
863             for (pd.logM = 0; (M = 1<<pd.logM) <= maxM && M < proof.V.size(); ++pd.logM);
864             CHECK_AND_ASSERT_MES(proof.L.size() == 6+pd.logM, false, "Proof is not the
    expected size");
865             max_logM = std::max(pd.logM, max_logM);
866
867             const size_t rounds = pd.logM+logN;
868             CHECK_AND_ASSERT_MES(rounds > 0, false, "Zero rounds");
869             ...
870             ...
```

Listing 23: Redundant assertion on line 868

3. In the function `pippenger()` in the file `src/ringct/multiexp.cc`, the last argument c is always equal to `get_pippenger_c(data.size())` in the context of the `ringct` module. Further, the

function `get_pippenger_c(data.size())` always returns the integer 9. Therefore the assertion in the function `pippenger()`, line 615, to check if $c \leq 9$ would always be true.

```
608 rct::key pippenger(const std::vector<MultiexpData> &data, const std::shared_ptr<
        pippenger_cached_data> &cache, size_t cache_size, size_t c)
609 {
610     if (cache != NULL && cache_size == 0)
611         cache_size = cache->size;
612     CHECK_AND_ASSERT_THROW_MES(cache == NULL || cache_size <= cache->size, "Cache is too
        small");
613     if (c == 0)
614         c = get_pippenger_c(data.size());
615     CHECK_AND_ASSERT_THROW_MES(c <= 9, "c is too large");
616     ...
617     ...
618     ...
```

Listing 24: Redundant assertion on line `src/ringct/multiexp.cc:614`

6.5 Caution on usage of amount commitments outside Bulletproofs+

The commitments V_j to the amounts $a_j \in \mathbb{Z}_p$ in Bulletproofs+ and Bulletproofs are computed as:

$$V_j = \left(\frac{\gamma_j}{8}\right) * G + \left(\frac{a_j}{8}\right) * H.$$

This is done as an optimisation for the verification computation. In the verification, the group elements of the proof are multiplied by 8 to ensure that the torsion components (if any) in them are filtered out. Here, when we multiply V_j by 8, we get $8V_j = \gamma_j G + a_j H$, which is the desired Pedersen commitment. If we had not multiplied the amount and the blinding factor by $\frac{1}{8}$, we would have to multiply V_j first by 8 and then by $\frac{1}{8}$ to ensure the torsion components are filtered out without changing the original group elements.

Pedersen commitments in Monero are defined as $V = \gamma G + aH$ to amount $a \in \mathbb{Z}_p$ and $\gamma \in \mathbb{Z}_p$. Note that the same commitments V_j to the amounts are also used in the RingCT transaction structure in the CryptoNote protocol [14]. Therefore, we caution about using the amount commitments from the `BulletproofPlus` and `Bulletproof` proofs outside of the range proving systems.

7 Code Improvements

7.1 Reduction in multi-scalar multiplications relevant to amount commitments

For filtering out the torsion component out of a group element A, we need to first multiply it by 8 to get $A_8 = 8A$, which brings down the torsion component to zero. To get back the original group element, we need to compute $8^{-1}A_8 = A_{\text{filtered}}$. As explained in § 4.2 in the code overview, the group elements in a `BulletproofPlus` proof like $\mathbf{V}, A, A_1, B$ are computed from scalars pre-multiplied by $8^{-1} \in \mathbb{Z}_p$. This is done to avoid multi-scalar multiplication by 8^{-1} of the group elements in the verification after they are multiplied by 8 so as to filter out the torsion component. This optimisation works neatly for all the group elements except the amount commitments $\mathbf{V}$.

The amount commitments are denoted by the `outPK` vector in the transaction structure. In the Bulletproofs+ and Bulletproofs modules, we represent the amount commitments by the vector $\mathbf{V}$. Before a Bulletproofs range proof is verified, during the parsing of the transaction, the verifier has to fill in the vector $\mathbf{V}$ by computing $\mathbf{V}[j] = 8^{-1} * \texttt{outPK}[j]$.

```
202         CHECK_AND_ASSERT_MES(n_amounts == rv.outPk.size(), false, "Internal error filling
    out V");
203         rv.p.bulletproofs[0].V.resize(n_amounts);
204         for (size_t i = 0; i < n_amounts; ++i)
205         rv.p.bulletproofs[0].V[i] = rct::scalarmultKey(rv.outPk[i].mask, rct::INV_EIGHT);
```

Listing 25: Filling $\mathbf{V}$ in `src/cryptonote_basic/cryptonote_format_utils.cpp:L205`

This is done so as to ensure that the amount commitments of the verifier match that of the prover's (recall, the prover computes $\mathbf{V}[j] = (8^{-1} * a_j) * H + (8^{-1} * \gamma_j) * G$).

We claim that the multi-scalar multiplication while parsing the transaction can be avoided. Instead, we could shift the multiplication by 8^{-1} of the amount commitments in the verification algorithm with reduced computational overhead. When we compute the multiplicand of the vector $\mathbf{V}$ for the final verification check in the verifier, we can multiply those multiplicands by 8^{-1} as follows.

```

    // V_j: -e**2 * z**(2*j+1) * y**(MN+1) * weight
    rct::key e_squared;
    sc_mul(e_squared.bytes, pd.e.bytes, pd.e.bytes);

    rct::key z_squared;
    sc_mul(z_squared.bytes, pd.z.bytes, pd.z.bytes);

    sc_sub(temp.bytes, ZERO.bytes, e_squared.bytes);
    sc_mul(temp.bytes, temp.bytes, y_MN_1.bytes);
    sc_mul(temp.bytes, temp.bytes, weight.bytes);

++  // Multiply by additional factor of (1 / 8)
++  sc_mul(temp.bytes, temp.bytes, INV_EIGHT.bytes);

    for (size_t j = 0; j < proof8_V.size(); j++)
    {
        sc_mul(temp.bytes, temp.bytes, z_squared.bytes);
        multiexp_data.emplace_back(temp, proof8_V[j]);
    }
```

Listing 26: Modifying multiplicands of $\mathbf{V}$ in `src/ringct/bulletproofs_plus.cpp`

Note that we would have to also change the way prover computes the amount commitments $\mathbf{V}$. Since the multiplication and division by 8 both are being taken care of in the verification function for the vector $\mathbf{V}$, the prover must simply compute the amount commitments to be $\mathbf{V}[j] = a_j * H + \gamma_j * G$. This would save the verifier m scalar multiplications per proof in the transaction parsing step (note that m can vary across different `BulletproofPlus` proofs). Since scalar multiplication (group operations) are the costliest computation in elliptic curve cryptography, this optimisation is expected to save a marginal but notable computation for the verifier.

On a more general note, we can do the same thing for all other group elements: multiplication and division by 8 both in the verification algorithm. This would negligibly increase the verifier computation (due to additional field multiplications) but would make the code a bit more intuitive to understand.

7.2 Reduction in field multiplications in `compute_LR()`

In the function `compute_LR()`, we compute the group elements $L, R \in \mathbb{G}_l$ for a given inner product round. Function inputs: $(n, y, \mathbf{G}, G_0, \mathbf{H}, H_0, \mathbf{a}, a_0, \mathbf{b}, b_0, c, d)$

$$\begin{aligned} L = \left(\frac{y}{8} \cdot \mathbf{a}[a_0 : a_0 + n]\right) * \mathbf{G}[g_0 : g_0 + n] \; + \\ \left(\frac{1}{8} \cdot \mathbf{b}[b_0 : b_0 + n]\right) * \mathbf{H}[h_0 : h_0 + n] \; + \\ \frac{c}{8} * H + \frac{d}{8} * G. \end{aligned}$$

```
187 // Helper function used to compute the L and R terms used in the inner-product round function
188 static rct::key compute_LR(size_t size, const rct::key &y, const std::vector<ge_p3> &G, size_t G0, const std::vector<ge_p3> &H, size_t H0, const rct::keyV &a, size_t a0, const rct::keyV &b, size_t b0, const rct::key &c, const rct::key &d)
189 {
190     CHECK_AND_ASSERT_THROW_MES(size + G0 <= G.size(), "Incompatible size for G");
191     CHECK_AND_ASSERT_THROW_MES(size + H0 <= H.size(), "Incompatible size for H");
192     CHECK_AND_ASSERT_THROW_MES(size + a0 <= a.size(), "Incompatible size for a");
193     CHECK_AND_ASSERT_THROW_MES(size + b0 <= b.size(), "Incompatible size for b");
194     CHECK_AND_ASSERT_THROW_MES(size <= maxN*maxM, "size is too large");
195
196     std::vector<MultiexpData> multiexp_data;
197     multiexp_data.resize(size*2 + 2);
198     rct::key temp;
199     for (size_t i = 0; i < size; ++i)
200     {
201         sc_mul(temp.bytes, a[a0+i].bytes, y.bytes);
202         sc_mul(multiexp_data[i*2].scalar.bytes, temp.bytes, INV_EIGHT.bytes);
203         multiexp_data[i*2].point = G[G0+i];
204
205         sc_mul(multiexp_data[i*2+1].scalar.bytes, b[b0+i].bytes, INV_EIGHT.bytes);
206         multiexp_data[i*2+1].point = H[H0+i];
207     }
208
209     sc_mul(multiexp_data[2*size].scalar.bytes, c.bytes, INV_EIGHT.bytes);
210     ge_p3 H_p3;
211     ge_frombytes_vartime(&H_p3, rct::H.bytes);
212     multiexp_data[2*size].point = H_p3;
213
214     sc_mul(multiexp_data[2*size+1].scalar.bytes, d.bytes, INV_EIGHT.bytes);
215     ge_p3 G_p3;
216     ge_frombytes_vartime(&G_p3, rct::G.bytes);
217     multiexp_data[2*size+1].point = G_p3;
218
219     return multiexp(multiexp_data, 0);
220 }
```

Listing 27: Batch Verification using a single multi-scalar multiplication

The scalar multiplicand of $\mathbf{G}$ is computed in two steps:

(i) `temp` $= y \cdot \mathbf{a}[a_0 + i]$ (line 201)

(ii) `G_scalar` $= \frac{1}{8} \cdot$ `temp` (line 202)

So, we effectively use 2 scalar multiplications for each $i \in [n]$. We could instead reduce it a single field multiplication by pre-computing $y' = \frac{y}{8}$ and then `G_scalar` $= y' \cdot \mathbf{a}[a_0 + i]$. Since this function is called for $\log_2(mn)$ times for sizes $\left(\frac{mn}{2}, \frac{mn}{4}, \ldots, 2\right)$, we can save a total of $mn - \log_2(mn)$ field multiplications.

7.3 Redundant loop and copy before inner-product rounds

Before beginning the computation in the recursive inner product protocol, the witness and the generator vectors are copied to temporary vectors in the loop on line 671.

```
661     // These are used in the inner product rounds
662     size_t nprime = MN;
663     std::vector<ge_p3> Gprime(MN);
664     std::vector<ge_p3> Hprime(MN);
665     rct::keyV aprime(MN);
666     rct::keyV bprime(MN);
667
668     const rct::key yinv = invert(y);
669     rct::keyV yinvpow(MN);
670     yinvpow[0] = ONE;
671     for (size_t i = 0; i < MN; ++i)
672     {
673         Gprime[i] = Gi_p3[i];
674         Hprime[i] = Hi_p3[i];
675         if (i > 0)
676         {
677             sc_mul(yinvpow[i].bytes, yinvpow[i-1].bytes, yinv.bytes);
678         }
679         aprime[i] = aL1[i];
680         bprime[i] = aR1[i];
681     }
```

Listing 28: Setting up before the inner-product rounds

We also compute the terms of the vector $\overleftarrow{y}^{mn} = (1, y^{-1}, y^{-2}, \ldots, y^{mn-1})$ in the same loop. We can shift the computation in this loop to the loop on the line 645 as shown below.

```
640     // Prepare inner product terms
641     rct::keyV aprime = vector_subtract(aL, z);
642
643     // declare d_y, yinvpow, Gprime, Hprime
644     rct::keyV bprime = vector_add(aR, z);
645     rct::keyV d_y(MN);
646
647     const rct::key yinv = invert(y);
648     rct::keyV yinvpow(MN);
649     yinvpow[0] = ONE;
650
651     std::vector<ge_p3> Gprime(MN);
652     std::vector<ge_p3> Hprime(MN);
653
654     for (size_t i = 0; i < MN; i++)
655     {
656         sc_mul(d_y[i].bytes, d[i].bytes, y_powers[MN-i].bytes);
657
658         Gprime[i] = Gi_p3[i];
659         Hprime[i] = Hi_p3[i];
660         if (i > 0)
661         {
662             sc_mul(yinvpow[i].bytes, yinvpow[i-1].bytes, yinv.bytes);
663         }
664     }
665     bprime = vector_add(bprime, d_y);
```

Listing 29: Modification of the loop at line `src/ringct/bulletproofs_plus.cc::L645`

Note that we have also renamed aL1 to `aprime` and aR1 to `bprime` to avoid copying `aL1, aR1` to new vectors `aprime, bprime`. This would save the redundant copying of two vectors of size mn along with getting rid of a mn-sized loop.

References

[1] Suyash Bagad. *Bulletproofs+ Audit for Monero*. 2020. URL: https://repo.getmonero.org/monero-project/ccs-proposals/-/merge_requests/197.

[2] Suyash Bagad. *Comparing Bulletproofs+ and Bulletproofs*. 2020. URL: https://suyash67.github.io/homepage/project/2020/07/03/bulletproofs_plus_part1.html; https://suyash67.github.io/homepage/project/2020/07/03/bulletproofs_plus_part2.html.

[3] Jonathan Bootle et al. "Efficient Zero-Knowledge Arguments for Arithmetic Circuits in the Discrete Log Setting". In: *Advances in Cryptology – EUROCRYPT 2016*. Ed. by Marc Fischlin and Jean-Sébastien Coron. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 327–357. ISBN: 978-3-662-49896-5.

[4] B. Bünz et al. "Bulletproofs: Short Proofs for Confidential Transactions and More". In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, pp. 315–334. DOI: 10.1109/SP.2018.00020.

[5] Heewon Chung et al. *Bulletproofs+: Shorter Proofs for Privacy-Enhanced Distributed Ledger*. Cryptology ePrint Archive, Report 2020/735. https://eprint.iacr.org/2020/735. 2020.

[6] Sebastian Faust et al. *On the Non-malleability of the Fiat-Shamir Transform*. Cryptology ePrint Archive, Report 2012/704. https://eprint.iacr.org/2012/704. 2012.

[7] Jens Groth and Y. Ishai. "Sub-linear Zero-Knowledge Argument for Correctness of a Shuffle". In: *EUROCRYPT*. 2008.

[8] Kyoohyung (Kay) Han. *Bulletproofs+ Implementation*. 2020. URL: https://github.com/KyoohyungHan/BulletProofsPlus.

[9] Koe, Kurt M. Alonso, and Sarang Noether. *Zero to Monero: Second Edition*. 2020. URL: https://web.getmonero.org/library/Zero-to-Monero-2-0-0.pdf.

[10] Yehuda Lindell. "Parallel Coin-Tossing and Constant-Round Secure Two-Party Computation". In: *Journal of Cryptology* 16 (2003), pp. 143–184.

[11] Dr. Sarang Noether. *Bulletproofs+ in Monero*. URL: https://web.getmonero.org/2020/12/24/Bulletproofs+-in-Monero.html.

[12] Sarang Noether. *Bulletproofs+ in Monero*. 2020. URL: https://gist.github.com/SarangNoether/ee6367fa8b5500120b2a4dbe23b71694.

[13] Quarkslab. *Evaluation of Bulletproof Implementation*. Audit Report. 2018. URL: https://blog.quarkslab.com/resources/2018-10-22-audit-monero-bulletproof/18-06-439-REP-monero-bulletproof-sec-assessment.pdf.

[14] Nicolas van Saberhagen. *CryptoNote v 2.0*. White paper. 2013. URL: https://cryptonote.org/whitepaper.pdf.

[15] Kudelski Security. *Monero Bulletproofs Security Audit*. Audit Report. 2018. URL: https://cybermashup.files.wordpress.com/2018/07/monero-audit2.pdf.

[16] Riccardo "fluffypony" Spagni and luigi1111. *Disclosure of a Major Bug in CryptoNote Based Currencies*. 2017. URL: https://web.getmonero.org/2017/05/17/disclosure-of-a-major-bug-in-cryptonote-based-currencies.html.