Generalized Bulletproofs code audit

Cypher Stack*

January 12, 2025

This report describes the findings of a code audit of a Generalized Bulletproofs implementation intended for use in Monero. It reflects a limited scope and represents a best effort; as with any review or audit, it cannot guarantee that any protocol or implementation is suitably secure for a particular use case, nor that the contents of this report reflect all issues or vulnerabilities that may exist. The author asserts no warranty and disclaims liability for its use. The author further expresses no endorsement of any kind. This report has not undergone any further formal or peer review.

Contents

*`https://cypherstack.com`

1 Summary

Generalized Bulletproofs are a cryptographic construction designed for use in Curve Trees [2], themselves used in a full-chain membership proof construction intended for deployment as a Monero protocol upgrade. The construction of Generalized Bulletproofs (as the name implies) generalizes the arithmetic circuit satisfiability protocol of Bulletproofs [1] to support Pedersen vector commitments. The protocol has previously been formalized and analyzed for security[1]. This report represents an audit of an implementation of Generalized Bulletproofs[2] at commit `7d972e2`.

The following were identified as goals for this audit:

- Assert that implementations of external specifications or protocols are done correctly
- Determine if malicious or unintended edge cases can be exploited
- Identify cases where the code may panic unexpectedly
- Note situations where documentation is insufficient to convey intent or design
- Ensure that secret data is handled as safely as is feasible with respect to memory clearing and constant-time operations
- Identify areas of the implementation where efficiency gains are possible and reasonable
- Determine the extent to which the implementation contains relevant and comprehensive tests

Overall, we find that the implementation is well written and appears suitable for its intended purpose. We identified an issue wherein the implementation uses polynomial indexing (accompanied by witness and weighting matrix restrictions) that does not match the proving system specification. However, we were able to modify the specification in the technical note to account for the discrepancy,

[1] `https://github.com/cypherstack/generalized-bulletproofs/releases/tag/final`

[2] `https://github.com/kayabaNerve/fcmp-plus-plus/tree/develop/crypto/generalized-bulletproofs`

and proved it secure[3]. As a result, we did not identify any outstanding issues of specific immediate concern. Consequently, this report primarily addresses areas of potential improvement. These deal with inconsistencies, documentation, tests, or areas of the codebase where fixes or changes could mitigate future issues.

However, we carefully note that we do not assign severity ratings to the issues contained herein. Such ratings are often subjective, and typically depend on the ease or difficulty of triggering specific behavior, which can be challenging to assess in a consistent way.

2 General

2.1 Random number generation trait bound

The codebase uses the trait bound `CryptoRng + RngCore` for cryptographically-secure random number generation. This is sufficient for the desired purpose; however, it is also possible to use the combined trait bound `CryptoRngCore` (supplied by `rand_core`) instead. This may be desirable to reduce the risk of neglecting the `CryptoRng` trait bound, as well as to more directly signal the intention of the bound.

3 `lib.rs`

3.1 Power-of-two padding

The codebase introduces a utility function `padded_power_of_2` to compute the nearest greater power of two to a given index. The function appears to perform its task; however, it can be replaced by the standard library method `usize::checked_next_power_of_two` for simplicity and robustness.

3.2 Generator index panic

The methods `ProofGenerators::g_bold` and `ProofGenerators::h_bold` can panic if the supplied index is out of bounds. It may be prudent to use a checked variant for these methods to avoid a panic.

3.3 Unnecessary reallocation

The method `PedersenVectorCommitment::commit` unnecessarily reallocates an internal vector due to iterating over input values. Since the number of input values is known at runtime, this can be avoided by allocating this vector to the appropriate size.

[3] `https://github.com/cypherstack/generalized-bulletproofs/releases/tag/implementation`

3.4 Missing unit tests

No unit tests are provided. These may be particularly relevant for commitment-related operations.

3.5 Generators must be independent

It is a requirement of the Generalized Bulletproofs protocol that certain generators used in proofs be independent. That is, they should have no nontrivial efficiently-computable discrete logarithm representation. In particular, all generators represented by `ProofGenerators` must be independent.

The `ProofGenerators` constructor `new` takes proposed generators as input, and therefore cannot assert their independence. It would be helpful for its documentation to make this requirement clear, as protocol soundness depends critically on it.

4 `point_vector.rs`

4.1 Commented-out methods

The following `PointVector` methods are commented out with no explanation:

- `add`
- `sub`
- `mul`
- `add_vec`
- `sub_vec`
- `multiexp_vartime`
- `sum`

4.2 Missing documentation

The following `PointVector` methods lack documentation:

- `mul_vec`
- `multiexp`
- `len`
- `split`

4.3 Assertions

The `PointVector` methods `mul_vec` and `multiexp` contain debug assertions relating to input values. These relate to assumptions on the corresponding values that are not documented and will not be caught in production.

The `PointVector` method `split` contains an assertion on the underlying vector length that is not documented.

4.4 Missing unit tests

No unit tests are provided. These may be particularly relevant for arithmetic operations.

5 `scalar_vector.rs`

5.1 Commented-out method

The `PointVector` method `sum` is commented out with no explanation.

5.2 Missing documentation

The following `ScalarVector` methods lack documentation:

- `powers`
- `inner_product`
- `split`

5.3 Assertions

The `ScalarVector` methods `inner_product` and `split` contains debug assertions relating to input values that are not documented and will not be caught in production.

The `ScalarVector` methods `add`, `sub`, `mul`, `powers`, and `split` contain assertions that are not documented.

5.4 Missing unit tests

No unit tests are provided. These may be particularly relevant for arithmetic operations.

6 transcript.rs

6.1 Vector length panics

When encoding vector lengths for the `Transcript` method `write_commitments` and `VerifierTranscript` method `read_commitments`, it is possible for the methods to panic if any vector length exceeds `u32` bounds.

6.2 Proof serialization is malleable and not canonical

When conducting verification of a serialized proof associated to an `IpStatement` or `ArithmeticCircuitStatement`, the implementation builds and processes a transcript using the `VerifierTranscript` construction as it reads proof elements from a supplied byte slice.

Presumably to better support composition, the verifier does not require that the supplied slice be empty after all required proof elements are read. This means that depending on higher-level handling, byte-serialized proofs can be malleable and are not canonical. It should be carefully checked that all such handling accounts for this.

6.3 Missing documentation

The following `Transcript` methods lack documentation:

- `push_scalar`
- `push_point`

The following `VerifierTranscript` methods lack documentation:

- `read_scalar`
- `read_point`

6.4 Missing unit tests

No unit tests are provided. These may be particularly relevant for testing the relationship between `Transcript` and `VerifierTranscript`.

7 lincomb.rs

7.1 Unused zeroization

The implementation of `Zeroize` for `Variable` appears to be unused within the codebase, and does not perform any action. This may have been intended to handle the case where a higher-level construction uses `Variable` and itself derives `Zeroize`. In this case, the user might assume or require zeroization from `Variable` that is not occurring, which violates the intent of the trait.

Fortunately, it is possible to use the `#[zeroize(skip)]` attribute for higher-level constructions requiring `Zeroize` derived support and which use `Variable`. Using this obviates the need for the implementation in the codebase. This approach is recommended in order to ensure user expectations are understood and met.

7.2 Missing documentation

The following `LinComb` methods lack documentation:

- `add`
- `sub`
- `mul`

The function `accumulate_vector` also lacks documentation.

We note that the aforementioned arithmetic operations on `LinComb` are implementations of existing traits. However, their design is subtle and warrants careful documentation.

7.3 Vector accumulation can panic

The `accumulate_vector` function will panic if the input `values` vector references an `accumulator` index that is out of bounds.

8 `inner_product.rs`

8.1 Unnecessary padding algorithm

The `IpWitness` method `new` pads the input vectors to the nearest power of two using zero scalars. The implementation is correct, but is unnecessary. It is possible simply to use `Vec::resize` instead, which resizes a vector with specified padding.

8.2 Nitpicky error naming

The `IpError` enumeration variant `IncorrectAmountOfGenerators` should more correctly be named `IncorrectNumberOfGenerators`, but this is very nitpicky.

8.3 Confusing notation

In `IpStatement`, the member `u` is specified to be a discrete logarithm against a curve group generator `G`. Later, in the `prove` method, this is redefined to be a group element formed by multiplying `u` by `G`. Because this is presumably intended to (roughly) match the protocol specified by the Generalized Bulletproofs technical note, this notation should be modified for clarity.

8.4 Unnecessary vector length computations

In the `IpStatement` method `prove`, the value `n_hat` is computed as a vector length. When subsequently allocating the `L_terms`, `R_terms`, `g_bold`, and `h_bold` vectors, this length is recomputed unnecessarily.

8.5 Verification assumes generator consistency

When queueing the proof associated to an `IpStatement` for batch verification, the `verify` method does not check that the generators are consistent between proofs, and any such requirement is not documented.

8.6 Verification does not check for zero weight

When queueing the proof associated to an `IpStatement` for batch verification, the `verify` method applies a supplied weight to corresponding terms. It is required by the protocol that this weight be nonzero and sampled uniformly at random by the verifier. If the weight is zero, verification of the associated proof is bypassed. While random sampling cannot be asserted internally, the method does not check that the weight is nonzero. It would be easy and prudent to check this.

8.7 Unclear algorithm

In the Bulletproofs preprint[4], the given inner-product argument is for the following relation:

$$\left\{\vec{G}, \vec{H}, U, P; (\vec{a}, \vec{b}) : P = \vec{a}\vec{G} + \vec{b}\vec{H} + \langle \vec{a}, \vec{b} \rangle U\right\}$$

Here we use additive notation for clarity. Capital letters denote elements or vectors in the group $\mathbb{G}$, and lowercase letters denote elements or vectors in the corresponding scalar field $\mathbb{F}$.

The implementation effectively performs a slight modification intended to accommodate the specific requirements of the arithmetic circuit satisfiability proving system used in Generalized Bulletproofs, which is for the following protocol:

$$\left\{\vec{G}, \vec{H}, P, c; (\vec{a}, \vec{b}) : P = \vec{a}\vec{G} + \vec{b}\vec{H}, c = \langle \vec{a}, \vec{b} \rangle\right\}$$

Instantiating an argument for the former protocol using the latter requires sampling U and a separate scalar x in specific ways.

Of note is that the batch verification queueing functionality in `verify` does not process the value P in the protocol, leaving this to the caller to handle. Using informal notation from the preprint, the queueing functionality adds a weighting of the linear combination

$$\sum_j \left(x_j^2 L_j + x_j^{-2} R_j\right) - a\vec{s}\vec{G} - b\vec{s}^{-1}\vec{H} - abxU$$

[4] `https://eprint.iacr.org/2017/1066`

to the batch verifier, which (in a correct proof) precisely yields the corresponding weighting of $-P$.

This is handled correctly in the Generalized Bulletproofs arithmetic circuit satisfiability implementation, but is poorly documented here. Specifically, it means that the implementation here is not itself a self-contained verifier for the protocol, which might not be immediately clear to other implementers.

8.8 Denial of service

When queueing the proof associated to an `IpStatement` for batch verification, the `verify` method performs computations on round challenges prior to reading the values a and b from the transcript.

This means that a malicious prover could supply invalid values that should immediately fail verification, but which force the verifier to perform unnecessary computations.

Such a minor denial-of-service can be avoided by reading the entire transcript prior to performing round challenge computations.

8.9 Possibly confusing verifier naming

The verification functionality performed by the `IpStatement` method `verify`, as documented, does not itself perform proof verification; instead, it adds a semantically-valid proof to a batch verifier for later verification.

Despite its documentation, the naming of this method may imply to an implementer that a non-error return indicates a valid proof, the result of which could be catastrophic. A more clear name like `queue` may provide an extra layer of protection against this.

9 `arithmetic_circuit_proof.rs`

9.1 Manual zeroization

The `Zeroize` trait is manually implemented for `ArithmeticCircuitStatement` against all members except for the generators, which are presumably public and global. This is implemented correctly, but is brittle against future member changes.

One option is to derive `Zeroize` and use the `#[zeroize(skip)]` attribute against the generators. Another is to use destructuring prior to member zeroization.

9.2 Nitpicky error naming

The `AcError` enumeration variant `InconsistentAmountOfConstraints` should more correctly be named `InconsistentNumberOfConstraints`, but this is very nitpicky.

9.3 Missing documentation

The `ArithmeticCircuitStatement` method `yz_challenges` lacks documentation

We note that this method is private and is only called with inputs that are produced as nonzero transcript challenges. It may be prudent to note this, since the method can otherwise panic due to attempted zero inversion.

9.4 Verifier weights can be zero

When queueing the proof associated to an `ArithmeticCircuitStatement` for batch verification, the `verify` method applies weights to corresponding terms. It is required by the protocol that these weights be nonzero and sampled uniformly at random by the verifier. If any weight is zero, verification of the associated proof is bypassed.

The weights are sampled uniformly at random, but may be zero either negligibly or due to an internal failure. It is easy to check that this is not the case, which would also make the intent clear in code.

9.5 Unnecessary padding algorithm

The `ArithmeticCircuitStatement` method `prove` pads witness vectors to the nearest power of two using zero scalars. The implementation is correct, but is unnecessary. It is possible simply to use `Vec::resize` instead, which resizes a vector with specified padding.

9.6 Witness consistency is not checked in production

The `ArithmeticCircuitStatement` method `prove` does not check witness consistency in production. This does not affect security, but may be useful (if not sufficiently inefficient) to detect inconsistencies that could arise in production.

9.7 Reused challenge notation

The implementation's prover derives challenges y and z as specified by the protocol, and uses them to construct exponent vectors $\vec{y}^n$, $\vec{y}^{-n}$, and $\vec{z}^{Q+1}_{[1:]}$. However, it reuses the challenge notation, subsequently setting $y := \vec{y}^n$ and $z := \vec{z}^{Q+1}_{[1:]}$.

The verifier uses the notation x to refer both to a vector of challenge exponents and a verifier weighting of the same vector.

This reuse reduces clarity, and should be avoided.

9.8 Polynomial indexing does not match

When defining indexes for the $\vec{l}$ and $\vec{r}$ polynomials in the implementation, the prover and verifier set (using the notation from the Generalized Bulletproofs

technical note) $n' = 2 + 2\lfloor \frac{n_c}{2} \rfloor$. However, the technical note specifies that $n' = 2(n_c + 1)$ instead. Accompanying this change, the implementation only addresses the case where all $\vec{c}_{k,R}$ witness elements and corresponding weighting matrices $W_{k,R}$ vanish for $k \in [1, n_c]$.

This means the implementation is not strictly that of the Generalized Bulletproofs construction, but of a modified version that resembles an earlier version of the technical note that did not admit a correct security proof.

The technical note has since been updated, such that the protocol it presents and proves secure now matches that used in the implementation. We caution that the updated security proof has subtle requirements relating to completeness and witness-extended emulation that must be met by any higher-level protocol that uses Generalized Bulletproofs as a subprotocol.

9.9 Reused polynomial notation

The implementation's prover uses the same notation l and r to refer to both corresponding polynomials $\vec{l}(X)$ and $\vec{r}(X)$, as well as to their challenge evaluations $\vec{l}(x)$ and $\vec{r}(x)$, which can hinder clarity. It may be helpful to differentiate the notation.

9.10 Polynomial evaluation is duplicated

The implementation's prover provides functionality for polynomial evaluation. This is used when computing $\vec{l}(x)$ and $\vec{r}(x)$, as well as when computing τ_x. Because of the reuse, it may be helpful to refactor this functionality out, which can additionally admit unit testing.

9.11 Denial of service

The `ArithmeticCircuitStatement` method `verify` computes powers of the challenge x immediately after computing this challenge from the transcript. Because the transcript is not yet complete, it could be the case that a malicious prover has not supplied subsequent valid transcript elements, which would render the proof invalid.

To avoid a minor denial-of-service, it may be prudent to avoid such computations until the transcript is completed to the extent possible.

9.12 Reused weighting notation

The implementation's verifier samples two verifier weights, each of which is used for a different verification equation. It is important that these be applied correctly. However, it uses the same `verifier_weight` notation for both, which reduces clarity and could increase future engineering risk.

Separate notation should be considered for this.

9.13 Mismatched index notation

The implementation's verifier computes scalars associated to the generator vector $\vec{H}'$ using, in part, powers of the (verifier-weighted) challenge x linked to polynomial indexes. However, it uses the index j_{LR} in several places where the technical note specifies i_{LR}. This is trivially correct since $i_{LR} = j_{LR}$ by definition, but could be changed for clarity and consistency.

9.14 Inconsistent verifier accumulation operations

The implementation's verifier must carefully accumulate scalars associated to common group generators across proofs, in order to perform a subsequent multi-scalar multiplication more efficiently. It does so somewhat inconsistently, using `Add` and `AddAssign` (or `Sub` and `SubAssign`, as appropriate) implementations interchangeably.

All such operations appear to be implemented correctly. However, the sensitivity of these operations may warrant more consistent treatment to improve clarity and avoid future errors.

9.15 Possibly confusing verifier naming

The verification functionality performed by the `ArithmeticCircuitStatement` method `verify`, as documented, does not itself perform proof verification; instead, it adds a semantically-valid proof to a batch verifier for later verification.

Despite its documentation, the naming of this method may imply to an implementer that a non-error return indicates a valid proof, the result of which could be catastrophic. A more clear name like `queue` may provide an extra layer of protection against this.

10 `tests/inner_product.rs`

10.1 Tests lack correct documentation

The `test_zero_inner_product` test lacks documentation on the statement and witness corresponding to the proof, which would be useful to improve clarity.

The `test_inner_product` test documentation specifies that it defines $P = \vec{a}\vec{G} + \vec{b}\vec{H}$. However, the test actually defines $P = \vec{a}\vec{G} + \vec{b}\vec{H} + \langle \vec{a}, \vec{b} \rangle G$ as required by the inner-product argument (and as noted at the top of the test file). This documentation should be corrected.

10.2 No test exercising parameters nontrivially

The `test_inner_product` test only generates and verifies proofs against statements where `h_bold_weights`, `u`, and the verifier weight associated to `P` are set to unit scalars. This means nontrivial functionality with respect to these values is not directly tested.

10.3 No test against invalid generator set

The `test_inner_product` test parameterizes its reduced generator set to check all valid powers of two up to the set limit. However, it does not check the failure mode arising from larger sets.

10.4 No tests for malicious transcripts

There are no tests against malicious transcripts that have been manipulated in any way, to check verifier rejection.

10.5 No tests for mismatched statements

There are no tests against mismatched statements (represented by differing transcript context), to check verifier rejection.

10.6 No verifier fuzz testing

There is no fuzz testing of proofs provided to the verifier as byte slices.

Because of the complexity of the verification process, it may be extremely useful to check that fuzzed proofs cannot meaningfully result in panic or other unexpected behavior for a given statement.

11 `tests/arithmetic_circuit.rs`

11.1 No tests for invalid generator set

There are no tests against generator sets that are invalid for an associated proof.

11.2 No tests for malicious transcripts

There are no tests against malicious transcripts that have been manipulated in any way, to check verifier rejection.

11.3 No tests for mismatched statements

There are no tests against mismatched statements (represented by differing transcript context), to check verifier rejection.

11.4 No tests for batch verification

There are no tests against nontrivial batch verification, either for correctness or to show that an invalid proof within a batch will result in verifier rejection.

11.5 No verifier fuzz testing

While the `fuzz_test_arithmetic_circuit` test generates proofs using random constraints as a form of fuzzing, there is no fuzz testing of proofs provided to the verifier as byte slices.

Because of the complexity of the verification process, it may be extremely useful to check that fuzzed proofs cannot meaningfully result in panic or other unexpected behavior for a given statement.

References

[1] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. Cryptology ePrint Archive, Paper 2017/1066, 2017.

[2] Matteo Campanelli, Mathias Hall-Andersen, and Simon Holmgaard Kamp. Curve trees: Practical and transparent zero-knowledge accumulators. Cryptology ePrint Archive, Paper 2022/756, 2022.