Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Aztec

Barretenberg Bigfield

Veridise Inc.

Sep. 18, 2025

► **Prepared For:**

Aztec Labs
https://aztec.network/

► **Prepared By:**

Alp Bassa
Benjamin Sepanski
Tyler Diamond

► **Contact Us:**

contact@veridise.com

► **Version History:**

Sep. 18, 2025	V2 - Incorporating issue fixes
Aug. 07, 2025	V1
Aug. 05, 2025	Initial Draft

Contents

1 Executive Summary

From Jul. 14, 2025 to Aug. 1, 2025, Aztec Labs engaged Veridise to conduct a security assessment of their Barretenberg Bigfield library. The security assessment covered the Bigfield component of the Barretenberg library, which enables zero-knowledge circuits to efficiently operate on field elements that are larger than the native finite field. Veridise conducted the assessment over 9 person-weeks, with 3 security analysts reviewing the project over 3 weeks on commit `480f49d`. The review strategy involved a tool-assisted analysis of the program source code performed by Veridise security analysts as well as thorough code review.

Project Summary. The security assessment covered the implementation of arithmetic operations computed over a Bigfield, i.e. a finite field of cryptographic size. Since Bigfield elements are either close to the size of the native finite field (in this case, BN254) or larger, this requires careful consideration to ensure computations do not overflow the native modulus leading to spurious results. The classical approach involves representing an element of the Bigfield by decomposing it into sufficiently small *limbs*. By ensuring the limbs are small enough, computations on limbs may be performed directly in the native field without fear of overflow. However, this approach can require a large number of range checks and doubling the number of limbs to compute intermediate results during Bigfield multiplication.

To avoid the difficulties described above, Aztec Labs relies on the Chinese Remainder Theorem (CRT). In what follows, we use the following notation: $L = 68$ is the binary limb size, $T = 4 \cdot L$, $2^{253} < n < 2^{254}$ is the native field modulus, $2^{250} < p < 2^{256}$ is the Bigfield modulus, and $M = 2^T n$ is the "CRT modulus". To represent a Bigfield element in $\mathbb{F}_p$, the Barretenberg Bigfield tracks a (possibly non-canonical) member of its equivalence class $a \in [0, M)$. a is represented implicitly by four binary limbs $a_i \in [0, n)$ and a fifth limb $a_{prime} \in [0, n)$ such that

$$a \equiv \sum_{i=0}^{3} a_i 2^{iL} \bmod 2^T, \qquad a \equiv a_{prime} \bmod n. \tag{1.1}$$

Applying the CRT shows that these equations define a unique value in $[0, M)$. This representation allows Aztec Labs to avoid doubling the number of limbs required during multiplications, a common operation required to reduce elements modulo p. However, additional care must be taken. If the limbs become too large, computations may overflow the CRT modulus, i.e. compute a value which is greater than or equal to M. If this occurs, the resultant Bigfield might be incorrect by a factor of M.

To ensure the implementation is safe, Aztec Labs enforces two important invariants on the limbs. First, they constrain that

$$a_{prime} \equiv \sum_{i=0}^{3} a_i 2^{iL} \bmod n. \tag{1.2}$$

Second, they constrain each binary limb a_i to be bounded above. This upper bound is parameterized by a value $Q \geq L$ which must be sufficiently small (see the Barretenberg Bigfield README

for more information). The Bigfield implementation requires that

$$a_i < 2^Q, \qquad 0 \leq i < 4. \tag{1.3}$$

These invariants protect the protocol from both of the risks described above. The bounds from (1.3) ensure that sums of products of limbs can be safely computed without overflowing the native field n. The constraints (1.2) and (1.3) together imply that, when computed over the integers,

$$a = \sum_{i=0}^{3} a_i 2^{iL}.$$

This equality allows Aztec Labs to use upper bounds known on the binary limbs a_i to conclude upper bounds on the implicitly represented value a. This trick leads to the final piece of core architecture in the Barretenberg Bigfield implementation.

To ensure no computations overflow the CRT-modulus M, Aztec Labs tracks upper bounds known on each binary basis limb. These static upper bounds are set whenever Bigfields are constructed (and a corresponding range-check is performed). Upper bounds on outputs from any limb-wise computations are set based on the upper bounds known on inputs to the computation. Whenever a computation may cause the implicitly represented output value a to be greater than or equal to M (as determined by the statically known upper-bounds on the binary basis limbs), the inputs are first reduced.

With this architecture in place, the Barretenberg Bigfield implements all standard field operations, including addition, subtraction, multiplication, and division. It also implements routines to reduce the representative $a \in [0, M)$ to within a factor of 2 of p (more precisely, to be at most $2^{\texttt{numBits}(p)} - 1$), or to fully reduce a to be the canonical representative of its equivalence class modulo p. Finally, several optimized methods which batch together additions with multiplication are provided to take full advantage of the gates used.

Code Assessment. The Aztec Labs developers provided the source code of the Barretenberg Bigfield for the code review, as well as all prior audit reports. The source code appears to be mostly original code written by the Barretenberg Bigfield developers. It contains some documentation in the form of READMEs and documentation comments on functions. The `README.md` provides mathematical arguments on the safety of their design decisions, and justification of certain constants such as the size of each limb. The code has been audited three times previously, and appears to have taken all recommendations for severe issues*.

The developers take great care to enforce the invariants described in the project summary. They add constraints whenever a Bigfield is constructed. Additionally, they carefully track and update the bounds on each binary basis limb. They have applied detailed reasoning to ensure the imposed range checks enforce soundness, and used conservative upper bounds to ensure their implementation stays well within their analysis.

The source code contained a test suite. This test suite tests each of the arithmetic operators for completeness, checking that the values match their expected results and that the resul-

* https://github.com/AztecProtocol/audit-reports/tree/7ee882ac/Barretenberg/Test%20Bigfield%20Audit

tant witnesses satisfy the constraints. Additionally, the test suite tests the tagging system, which is used for debugging and automated pattern identification by Aztec Labs. Optimized arithmetic functions like `sqr()`, `madd()`, `mult_madd()`, `dual_madd()`, `msub_div()`, `invert()`, and `pow()`, utility functions like `conditional_negate()`, `self_reduce()`, `assert_is_in_field()`, `assert_less_than()`, `assert_equal()` and `assert_not_equal()`, some constructors, as well as multiplication, division, and some combinations of the standard arithmetic operators are tested with random inputs, and using both witness variables and constants. Several regressions are included in the test suite, as well as edge cases such as arithmetic operations performed over a Bigfield of maximal size to ensure routines are complete (in addition to the test performing a fixed sequence of operations on a large Bigfield input).

Additionally, Aztec Labs has fuzzed the Barretenberg Bigfield. Their fuzzer[†] performs differential fuzzing with a direct implementation of arithmetic over the Bigfield, using a stack to store results from prior operations so that long sequences of operations are checked.

Summary of Issues Detected. The security assessment uncovered 9 issues, none of which were assessed to be of high or critical severity. 3 isssues were assessed to be of low severity by the Veridise analysts. Specifically V-AZT-VUL-001, which details how scenarios may arise that perform a divide by zero leading to unexpected results, V-AZT-VUL-002 which provides reasoning why exposing the `assert_not_equal` function in its current form can lead to negative consequences, and V-AZT-VUL-003 describe an edge-case in which completeness may be limited by rounding a computation performed on an upper bound in the incorrect direction. Additionally, 3 warnings, and 3 informational findings were found.

Aztec Labs has fixed all of the issues except V-AZT-VUL-002 and V-AZT-VUL-009. V-AZT-VUL-002 describes an incompleteness in certain circuits which may lead to unsatisfiable constraints and denial of service. While the over-constrained circuits are still present for performance reasons, all locations which exposed these circuits to user code (e.g. through the Noir language) have been removed. The only remaining use cases are by the Aztec Labs developers themselves, and rely on cryptographic guarantees to ensure that, except with negligible probability, the over-constrained cases are not encountered. V-AZT-VUL-009 is an informational issue with some proposed optimizations, and poses no security risk. The Veridise team validated that the provided fixes to the remaining issues resolve the raised concerns.

Recommendations. After conducting the assessment of the protocol, the security analysts had a few suggestions to improve the Barretenberg Bigfield implementation.

Formalize Soundness Argument. The provided documentation is very useful for understanding the intended design. However, much of the formal argument focuses on arguing that the bounds are not too tight, i.e. that the applied range constraints do not make the system incomplete. While the bounds are in fact tight enough to prevent overflow, including a more formal writeup of this argument will help future developers to clearly understand the requirements necessary to safely make changes in the code. As a first step towards this, the Veridise analysts have manually encoded one of the core functions (`evaluate_non_native_field_multiplication()`) into the Z3 SMT prover and verified that its constraints, along with proper range checks, imply

[†] https://github.com/AztecProtocol/aztec-packages/blob/2672be7f/barretenberg/cpp/src/barretenberg/stdlib/primitives/bigfield/bigfield.fuzzer.hpp

the desired expression is zero modulo 2^T[‡]. Further efforts, even as traditional write-ups rather than SMT-encoded, could help to strengthen reviewers' understanding of the system.

Improve Testing. The Veridise analysts assessed that certain portions of the code could benefit from additional testing and recommend the following:

- All public APIs should be tested. A few of the public functions are not directly tested, including various constructors of the Bigfield, one variant of `unsafe_construct_from_limbs()`, `construct_from_limbs()`, `to_byte_array()`, `add_two()`, assignment arithmetic operations such as `operator*=()` or `operator/=()`, `sum()`, and `operator==()`.
- Tests should consider a few additional edge cases. For example, when generating random witnesses, include non-normalized elements which contain limbs larger than `NUM_LIMB_BITS`. Include operations over potentially problematic values such as multiples of n, p, or 2^T. Additionally, consider testing multiple different Bigfields.
- Tests should also include more negative cases. For example, the suite should check that division and inversion with zero checks fail when zero is in the denominator. It should also check that, when provided with input limbs whose maximum values are too large, the expected sanity checks fail.
- Consider also testing important invariants on intermediate results during tests. For example, the limb maximum values should be greater than or equal to the witness values, the prime limb should be consistent with the binary limbs, and the maximum values should have fewer bits than `PROHIBITED_LIMB_BITS`.
- Directly test the correctness of core internal functions such as `unsafe_evaluate_multiply_add()` and `unsafe_evaluate_multiple_multiply_add()`. These functions implement the critical constraints applied during reduction, multiplication, and division. Although they are indirectly tested through the testing of the higher level functions, it would be best to test these directly. Some other public functions are tested but only indirectly, such as `internal_div()`, `div_without_denominator_check()`, `div_check_denominator_nonzero()`, `conditional_select()`, `conditional_assign()`. If time permits, testing these as well could help to ensure future changes do not cause regressions.

The Aztec Labs team has implemented these recommendations and expanded their test suite[§].

Deduplicate Code. Many operations in the codebase involve duplicate code. As a simple example, many unrolled loops apply the same operation to the four binary basis limbs, rather than writing the single operation once inside a loop. As another example, many of the core arithmetic operations include almost identical code snippets: The two Bigfields are both recursively reduced until no further reduction is required. The witnessed result of the operation is then computed outside the circuit. This result is then divided by the target modulus in order to determine the result in the target field (remainder) and the number of times the modulus was "wrapped around" (quotient). Fresh quotient/remainder variables are created using the computed witness values. Abstracting away these operations into functions could help reduce the size and increase the readability of the implementation.

The logic involving the out-of-circuit computations is replicated across multiple functions, in addition to the compile-time bounds checking on inputs and assignment of `maximum_value` bounds. This logic could instead be implemented in a singular function, called by all of the

[‡] https://github.com/VeridiseAuditing/field-emulation-proofs/tree/58c54b0d/field_emulation_proofs/aztec

[§] https://github.com/AztecProtocol/aztec-packages/pull/16802/

arithmetic implementations. This would make the soundness of the implementation easier to review.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Barretenberg Bigfield	`480f49d`	C++	Barretenberg

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Jul. 14–Aug. 1, 2025	Manual & Tools	3	9 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	0	0	0
High-Severity Issues	0	0	0
Medium-Severity Issues	0	0	0
Low-Severity Issues	3	3	2
Warning-Severity Issues	3	3	3
Informational-Severity Issues	3	3	2
TOTAL	9	9	7

Table 2.4: Category Breakdown.

Name	Number
Logic Error	3
Maintainability	3
Over-constrained Circuit	1
Usability Issue	1
Data Validation	1

3 Security Assessment Goals and Scope

3.1 Security Assessment Goals

The engagement was scoped to provide a security assessment of Aztec Labs's Barretenberg Bigfield implementation. During the assessment, the security analysts aimed to answer questions such as:

- Are all witnesses appropriately range-checked?
- Are inputs that are too large reduced before performing arithmetic operations?
- Are zero values, such as in division, properly handled?
- Can congruent representations of values cause unintended behavior?
- Are all valid values of the target field able to be represented and acted upon?
- Are Bigfield elements always properly constrained on construction?
- Are input bounds documented clearly?
- Can any sequence using publicly available APIs violate important Bigfield invariants?
- Can overflows in any field other than the Bigfield occur and change the result?
- Are any circuits under- or over-constrained?
- Are static upper bounds tracked correctly based on the operations performed on individual limbs?
- Can the prime basis limb become out of sync with the binary basis limbs?
- Is the library thoroughly tested?

3.2 Security Assessment Methodology & Scope

Security Assessment Methodology. To address the questions above, the security assessment involved a combination of human experts and automated program analysis & testing tools. In particular, the security assessment was conducted with the aid of the following techniques:

- *Formal Verification.* Security analysts encoded some functions into SMT-formulas using the Z3 prover. They then used the prover to verify the soundness of the constraints.
- *Semgrep.* Security analysts wrote Semgrep rules to statically analyze if certain invariants did not hold, such as the `binary_basis_limbs` and `prime_basis_limb` becoming out of sync.

Scope. The scope of this security assessment is limited to the following files in the `barrentenberg/cpp/src/barrentenberg/stdlib/primitives/bigfield/` folder:

- `bigfield.hpp`
- `bigfield_impl.hpp`
- `bigfield.test.cpp`

Methodology. Veridise security analysts reviewed the reports of previous audits for Barretenberg Bigfield, inspected the provided tests, and read the Barretenberg Bigfield documentation. They then began a review of the code.

During the security assessment, the Veridise security analysts regularly met with the Barretenberg Bigfield developers to ask questions about the code.

3.3 Classification of Vulnerabilities

When Veridise security analysts discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise.

The severity of a vulnerability is evaluated according to the Table 3.1.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

The likelihood of a vulnerability is evaluated according to the Table 3.2.

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

The impact of a vulnerability is evaluated according to the Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

This section presents the vulnerabilities found during the security assessment. For each issue found, the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.) is specified. Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-AZT-VUL-001	Missing zero-check for division by constant	Low	Fixed
V-AZT-VUL-002	Incomplete ‘assert_is_not_equal‘ assumes . . .	Low	Acknowledged
V-AZT-VUL-003	borrow_lo rounds down	Low	Fixed
V-AZT-VUL-004	Unsafe default for ‘msub_div()‘	Warning	Fixed
V-AZT-VUL-005	Limbs with a maximum value of 0 trigger . . .	Warning	Fixed
V-AZT-VUL-006	Off-by-one comparison of . . .	Warning	Fixed
V-AZT-VUL-007	Maintainability Improvements	Info	Fixed
V-AZT-VUL-008	Incorrect term used during calculation on . . .	Info	Fixed
V-AZT-VUL-009	Optimization Improvements	Info	Acknowledged

4.1 Detailed Description of Issues

4.1.1 V-AZT-VUL-001: Missing zero-check for division by constant

Severity	Low	**Commit**	480f49d
Type	Logic Error	**Status**	Fixed
Location(s)	barretenberg/cpp/src/[...]/bigfield_impl.hpp:793-838		
Confirmed Fix At	0b17711, a51b732, a51b732, d8f4dd0		

The `internal_div()` function performs a division of multiple numerators by a single denominator. The `check_for_zero` flag is intended to signal whether the denominator should be validated to avoid division by zero. However, in the constant-constant case (i.e., both numerator and denominator are constants), the check_for_zero flag is ignored. Instead, the function returns `0` if the denominator value is `0`, without enforcing a runtime check or constraint.

```
if (numerators.empty()) {
    return bigfield<Builder, T>(denominator.get_context(), uint256_t(0));
}

// [VERIDISE] elided...

// a / b = c
// => c * b = a mod p
const uint1024_t left = uint1024_t(numerator_values);
const uint1024_t right = uint1024_t(denominator.get_value());
const uint1024_t modulus(target_basis.modulus);
// We don't want to trigger the uint assert
uint512_t inverse_value(0);
if (right.lo != uint512_t(0)) {
    inverse_value = right.lo.invmod(target_basis.modulus).lo;
}
uint1024_t inverse_1024(inverse_value);
inverse_value = ((left * inverse_1024) % modulus).lo;

// [VERIDISE] elided...

if (numerator_constant && denominator.is_constant()) {
    inverse = bigfield(ctx, uint256_t(inverse_value));
    inverse.set_origin_tag(tag);
    return inverse;
}
```

Snippet 4.1: Snippet from `internal_div()`

In this code, even if `check_for_zero` is true, there is no validation that the denominator is non-zero. If the denominator is `0`, `right.lo.invmod(...)` will not be called and `inverse_value` remains `0`, silently returning `0` as the result.

Impact If constant values are used or the `numerators` array is empty, the function may silently perform division by zero and return 0, bypassing any intended safeguard. This undermines

assumptions about field arithmetic safety and may lead to compilation of incorrect or malicious circuits.

Recommendation Insert an explicit assertion or runtime check to fail when a constant denominator is zero and `check_for_zero` is set. This guards against unintended zero-division behavior and preserves semantic consistency across dynamic and constant cases.

Developer Response The developers implemented the recommendation.

4.1.2 V-AZT-VUL-002: Incomplete 'assert_is_not_equal' assumes random distribution

Severity	Low	**Commit**	480f49d
Type	Over-constrained Circuit	**Status**	Acknowledged
Location(s)	`barretenberg/cpp/src/[...]/bigfield_impl.hpp:1948`		
Confirmed Fix At	N/A		

The `assert_is_not_equal()` is designed to be partially incomplete.

```
// construct a proof that points are different mod p, when they are different mod r
// WARNING: This method doesn't have perfect completeness - for points equal mod r (
    or with certain difference kp
// mod r) but different mod p, you can't construct a proof. [VERIDISE: elided....]
template <typename Builder, typename T> void bigfield<Builder, T>::
    assert_is_not_equal(const bigfield& other) const
```

Snippet 4.2: Snippet from `assert_is_not_equal()`

In more detail, the goal of this function is to assert that two emulated field elements `a` and `b` do *not* satisfy `(a - b) mod p == 0` (where `p` is the modulus of the emulated bigfield). To do this, `assert_is_not_equal()` asserts that `(a-b) != k * p` for any possible `k`. To make this constraint concrete, the `bigfield` module tracks upper bounds `A` and `B` such that `a <= A` and `b <= B`, and only checks values of `k` with `-B/p <= k <= A/p`. However, for efficiency, this is only asserted over the native field. To summarize, `assert_is_not_equal()` asserts that `(a-b) mod n != k * p mod n` for all `k` in the range `[-B/p, A/p]`.

Since `p != n`, if `a-b mod n = 0`, then the circuit will become unsatisfiable *even if* `a - b mod p != 0`. This is expected behavior to the developers, and documented on the function. This incompleteness is treated as acceptable because, if `*this` and `other` are drawn randomly from the set of possible bigfield representatives, the probability that `a - b mod n = 0` is negligible. In most cryptographic applications using these methods, this assumption is satisfied.

However, not all applications may be cryptographic in nature. More concerning, a malicious party may *intentionally* construct an example which exploits this incompleteness. For example, consider a ZK-Rollup which emulates the Ethereum Virtual Machine (EVM). To support EVM precompiles, it might use the `bigfield` library to implement `ecrecover`, which performs ECDSA signature verification. A smart contract deployed on this network might use the `ecrecover` function for a variety of reasons such as executing meta-transactions or cross-chain validation. Below, we demonstrate how an attacker could create a valid ECDSA signature whose verification is unsatisfiable by the above constraints. This signature could then be submitted to the smart contract as part of a valid transaction, halting the network.

Before proceeding with the example, we make a few important notes:

1. This method is used by other parts of the codebase including `internal_div`, `operator/`, and `inverse`, making this incompleteness a critical consideration for any emulated field arithmetic.
2. The below example requires the attacker to choose a public key *after* the hash of the message is known. For applications like transaction-signing, this will be infeasible under

standard cryptographic assumptions. However, a valid EVM must be able to simulate transactions in which `ecrecover` returns `0` (i.e. the signature verification fails on-chain). More advanced constructions/attacks may be able to circumvent this limitation.

Example (ECDSA Denial of Service):

ECDSA (Elliptic Curve Digital Signature Algorithm) is described here. Given a message hash `z = H(m)`, private key `d`, and ephemeral key `k`, the honest signer computes:

- `r = (k*G).x mod p`
- `s = inverse(k) * (z + r*d) mod p`

The verifier checks the signature by computing:

- `s_inv = inverse(s) mod p`
- `u1 = z*s_inv mod p`, `u_2 = r * s_inv mod p`
- Verifies: `u_1 G + u_2 Q == R`

Critically, the verifier must compute the inverse of `s` modulo `p`.

If the prover uses `bigfield::inverse` to implement this division inside a zk-SNARK, then calling `assert_is_not_equal(s, 0)` becomes necessary to avoid division by zero. A malicious prover can

1. Choose a message `m`, and compute `z = H(m)`
2. Choose `k = inverse(n) mod p` (i.e. `k` such that `k * n mod p == 1`).
3. Compute `r = (k * G).x mod p`.
4. Choose private key `d = inv(r) * (1 - z) mod p`.

The attacker can easily produce an ECDSA signature for the public key `d * G` using the above parameters. Note that

```
s = inverse(k) * (z + r * d) mod p
  = inverse(inverse(n)) * (z + r * inv(r) * (1 - z)) mod p
  = n * (z + (1-z)) mod p
  = n * 1 mod p
  = n mod p
```

As a consequence, the signature will verify *outside* of the circuit, but attempted verification of the signature within the zk context will produce an unsatisfiable circuit.

Impact Applications using `assert_is_not_equal()` or any method which divides by zero cannot be fully DoS-resistant. This is especially impactful in any scenario in which either

1. An action performed out-of-circuit must be provable in-circuit. For example, restaking services need to be able to prove that participants' signatures can be checked in order to perform slashing. If this is implemented inside of a ZK context, malicious actors may become unslashable.
2. An action must be completed in-circuit to continue further processing. For example, if an application takes in a queue of signatures for validation, and then proves them in a batch, this could DoS the batch.

Recommendation A complete alternative should be used. For example, after reducing `*this` and `other`, the only possible differences (when they are equal) are `{-2p, -p, 0, p, 2p}`. Checking the binary limbs for these cases could provide a complete and sound approach.

If the method is left incomplete, all public functions which invoke `assert_is_not_equal()` should add a clear warning to their documentation indicating their usage may introduce a DoS vector into an application.

Developer Response This is a known limitation, and an acceptable risk for its usage within the core libraries.

To ensure it is not publicly exposed, this commit has removed the `bigint_constraint.hpp, bigint_constraint.cpp` which means noir no longer exposes any of the `bigfield` operations to devs.

4.1.3 V-AZT-VUL-003: borrow_lo rounds down

Severity	Low	**Commit**	480f49d
Type	Logic Error	**Status**	Fixed
Location(s)	barretenberg/cpp/src/[...]/bigfield_impl.hpp:2086		
Confirmed Fix At	2953d49		

The `unsafe_evaluate_multiply_add()` function validates the correct relationship between the input elements. More specifically, the relationship (omitting the summation of `to_add`) validates `a*b - p*q - r = 0` by checking it `mod 2^T` and `mod n`, separately. For the `mod 2^T` computation `-p` is replaced by its canonical representative `mod 2^T`, leaving `r` as the only negative contribution. To fit in one native field element operations are split into two parts consisting of the lower two and upper two limbs. To avoid underflow for the computation in the lower part, a sufficiently large multiple of `2^(2*L)` is borrowed from the upper two limbs. The necessary borrow is computed as in `Line 2086` as follows:

```
1 uint256_t borrow_lo_value = max_remainders_lo >> (2 * NUM_LIMB_BITS);
```

Here `max_remainders_lo` is the available upper bound for the sum of the lower parts of the remainder terms. This amounts to computing `floor(max_remainders_lo/2^(2*L))`. To ensure that `borrow_lo_value * 2^(2*L) > max_remainders_lo` , the correct expression for the borrow should be `ceil(max_remainders_lo/2^(2*L))`.

Impact Using the `floor` rather than the `ceil` can result in a negative value of the element `lo` on `Line 2228`. The subsequent check to verify it is small will fail. Valid values of the input to this function for which the lower part remains negative despite adding the borrow will not satisfy the constraints, resulting in completeness issues.

Recommendation Take the ceiling of `max_remainders_lo`, instead of the floor, by using

```
1 (max_remainders_lo +(1 << (2 * NUM_LIMB_BITS)-1) ) >> (2 * NUM_LIMB_BITS)
```

Developer Response The developers implemented the recommendation in both `unsafe_evaluate_multiply_add()` and `unsafe_evaluate_multiple_multiply_add()`.

4.1.4 V-AZT-VUL-004: Unsafe default for 'msub_div()'

Severity	Warning	**Commit**	480f49d
Type	Usability Issue	**Status**	Fixed
Location(s)	`barretenberg/cpp/src/barretenberg/[...]/bigfield.hpp:567`		
Confirmed Fix At	fc80c59		

The function `msub_div()` performs a sequence of multi-element big integer operations:

```
static bigfield msub_div(const std::vector<bigfield>& mul_left,
                         const std::vector<bigfield>& mul_right,
                         const bigfield& divisor,
                         const std::vector<bigfield>& to_sub,
                         bool enable_divisor_nz_check = false);
```

By default, the `enable_divisor_nz_check` parameter is set to `false`. When `false`, no check is performed to ensure that `divisor != 0` prior to performing a division. As a result, invoking this function with a zero divisor can lead to an unconstrained result.

Since the default behavior omits the zero-check, developers may overlook the need to explicitly enable it, resulting in fragile or unsafe usage patterns. For example, scanning a codebase for misuses would be nontrivial if this parameter is frequently omitted.

Impact Silent reliance on the caller to enable `enable_divisor_nz_check` opens the door for misuse. In the worst case, unguarded division by zero could lead to unconstrained results, allowing attackers to choose an arbitrary result of the computation.

Recommendation Change the default value of `enable_divisor_nz_check` to `true` to ensure that the divisor is validated by default. This shifts the burden of disabling safety to deliberate opt-out cases rather than requiring developers to opt-in for safety. Additionally, audit the codebase for existing calls to `msub_div()` where the divisor is not guaranteed to be non-zero and the check is disabled.

Developer Response The developers implemented the recommendation.

4.1.5 V-AZT-VUL-005: Limbs with a maximum value of 0 trigger compilation failure

Severity	Warning	**Commit**	480f49d
Type	Data Validation	**Status**	Fixed
Location(s)	`barretenberg/cpp/src/[...]/bigfield_impl.hpp:2233-2241`		
Confirmed Fix At	`https://github.com/AztecProtocol/aztec-packages/pull/16842`, bb3a089		

The `carry_lo_msb` and `carry_hi_msb` define the highest significant bit that the low and high carry outs from `evaluate_non_native_field_multiplication()` may possess. The carry outs (`lo_idx` and `hi_idx`) are range constrained to their corresponding `msb`:

```
if (carry_lo_msb <= 70 && carry_hi_msb <= 70) {
        ctx->range_constrain_two_limbs(hi.get_normalized_witness_index(),
                                       lo.get_normalized_witness_index(),
                                       static_cast<size_t>(carry_hi_msb),
                                       static_cast<size_t>(carry_lo_msb));
    } else {
        ctx->decompose_into_default_range(hi.get_normalized_witness_index(),
    carry_hi_msb);
        ctx->decompose_into_default_range(lo.get_normalized_witness_index(),
    carry_lo_msb);
    }
```

The `range_constrain_two_limbs()` is used when possible for efficiency reasons. Notably, the `decompose_into_default_range()` does not allow a target `num_bits` of 0. When a scenario occurs, such as `carry_hi_msb > 70` and `carry_lo_msb = 0`, then the circuit cannot be compiled.

Impact Certain use-cases will not be able to compile their circuits due to this limitation.

Recommendation When a `0` value is observed for either variable, utilize a function that can handle this case.

Developer Response The developers now use the `create_range_constraint()` function in the `else` branch. This function allows for an arbitrary-sized range constraint, including 0. When the target bit-size is 0, a constraint is created that enforces the value itself is equal to 0.

4.1.6 V-AZT-VUL-006: Off-by-one comparison of maximum_unreduced_limb_value

Severity	Warning	**Commit**	480f49d
Type	Maintainability	**Status**	Fixed
Location(s)	`barretenberg/cpp/src/[...]/bigfield_impl.hpp:1787-1795`		
Confirmed Fix At	0fb812d		

The `reduction_check()` function will determine if the element must be reduced by checking if any of its limbs maximum values are greater than (>) the `get_maximum_unreduced_limb_value()`. However, this function is defined as `1 << MAX_UNREDUCED_LIMB_BITS`. With this definition, `get_maximum_unreduced_limb_value()` should be considered the *exclusive* upper limit on a limb's maximum value. And therefore, the correct comparison would be >=.

Impact A value that is equal to the `get_maximum_unreduced_limb_value()` will not be reduced when it should be.

Recommendation Change the definition of `get_maximum_unreduced_limb_value()` to subtract 1 from the bitshift, as this more accurately reflects its name, which implies this is an *inclusive* limit.

Developer Response The developers implemented the recommendation.

4.1.7 V-AZT-VUL-007: Maintainability Improvements

Severity	Info	**Commit**	480f49d
Type	Maintainability	**Status**	Fixed
Location(s)	`barretenberg/cpp/src/barretenberg/stdlib/primitives/[...]/` ▶ `bigfield.hpp` ▶ `bigfield_impl.hpp`		
Confirmed Fix At	ce539cb		

The maintainability of the code may be improved by changes in the following locations:

1. `bigfield.hpp:L325-L335`: The following constants are defined, but unused
   a) `bigfield.prime_basis_maximum_limb`
   b) `bigfield.shift_right_1`
   c) `bigfield.shift_right_2`
   d) `bigfield.negative_prime_modulus_mod_binary_basis`
2. `bigfield.hpp:L335`: Consider renaming `negative_prime_modulus_mod_binary_basis` to `negative_prime_modulus_mod_native_basis` to more accurately describe the variable.
3. `bigfield_impl.hpp:L332`: Use the `NUM_LIMBS` constant instead of a hardcoded `4`.
4. `bigfield_impl.hpp:L2306-2308`: Use the variable `L` instead of `t` in order to match the rest of the codebase/documentation.
5. The functions `get_maximum_unreduced_limb_value()`, `get_maximum_value()` and `get_maximum_unreduced_value()` are inconsistent in their definition of "maximum". They should be reworked to strictly be the *inclusive* upper limit of their respective values, and the comparisons should reflect that.
6. `bigfield.hpp:L891-922` : The definition of `MAX_UNREDUCED_LIMB_BITS` should use the `MAX_ADDITION_LOG` constant, in case it is changed in the future.

```
1 static constexpr uint64_t MAX_ADDITION_LOG = 10;
2 // ...
3 static constexpr uint64_t MAX_UNREDUCED_LIMB_BITS = NUM_LIMB_BITS + 10;
```

De-duplicating instances of duplicate code also improves the maintainability of the codebase:

1. `bigfield_impl.hpp:L2224`: The `neg_prime` is already a defined constant `negative_prime_modulus_mod_binary_basis`.
2. `bigfield_impl.hpp:L2509`: The `neg_prime` is already a defined constant `negative_prime_modulus_mod_binary_basis`.
3. In `bigfield_impl.hpp`, there are multiple instance of duplicating the functionality of `bigfield::is_constant()`, such as:
   a) `L393-406`
   b) `L638-650`
4. `bigfield_impl.hpp:L2256`: The assertion on this line is a duplicate of assertions on previous lines.

The following typos were found in the codebase:

1. `bigfield.hpp:58`: The print statement should print the value is <= the maximum value, not <.
2. `bigfield.hpp:767`: `q` is the quotient, not `p`.
3. `bigfield_impl.hpp:L1945`: comment should be corrected as "for k in a range [-R,L] for largest L and R such that L*p<=a and R*p<=b".

Impact Issues may arise in the future from inconsistencies in the codebase. Additionally, the recommended changes will improve the ability to understand the code.

Recommendation Implement the recommendations.

Developer Response The developers implemented the recommendations.

4.1.8 V-AZT-VUL-008: Incorrect term used during calculation on bound of upper limb

Severity	Info	**Commit**	480f49d
Type	Logic Error	**Status**	Fixed
Location(s)	`barretenberg/cpp/src/barretenberg/stdlib/primitives/[...]/` ▶ `README.md:417` ▶ `bigfield.hpp:916-917`		
Confirmed Fix At	2d5ea7e		

The calculation of the resultant maximum bound of the upper limb (named `D_{hi}`) of the product of multiplication is defined as follows, with `Q` representing the maximum value of a limb:

```
{D_{hi}=3*(2Q-1)^2+4*(2Q-1)^2*2^L}
```

We can simplify this calculation by making two approximations:

1. `(2^Q-1)^2 \approx 2^{2Q}`: This is the upper bound of the resulting expanded form
2. `{3*(2^Q-1)^2 \approx 4*2^L*(2^Q-1)^2}` : This allows us to combine the two terms easier. The `2^L` term is negligent compared to the `2^{2Q}` term for production values

```
D_{hi} < 4*2^L*2^{2Q} + 4*2^L*2^{2Q}
D_{hi} < 2^3*2^L*2^{2Q}
D_{hi} < 2^{2Q+L+3}
```

The below script provides examples on when the original equation does not hold:

```python
def lhs(q, ell):
    return 3 * (2**q - 1)**2 + 4 * (2**q - 1)**2 * 2**ell

def rhs(q, ell):
    return 2**(2*q + ell + 2)

for q in range(0, 5):
    for ell in range(0, 5):
        if lhs(q, ell) >= rhs(q, ell):
            print(f"q = {q}, ell = {ell}, lhs = {lhs(q, ell)}, rhs = {rhs(q, ell)}")
```

Fortunately, for the given `n`, `L` and `k` values, the same result of `Q=86` is calculated.

Impact For a given `n`, `L` and `k`, a too-low bound may be calculated for the maximum overflow bits value, `Q`.

Recommendation Update the equations to calculate `Q`.

Developer Response The developers implemented the recommendation.

4.1.9 V-AZT-VUL-009: Optimization Improvements

Severity	Info	**Commit**	480f49d
Type	Maintainability	**Status**	Acknowledged
Location(s)	`barretenberg/cpp/src/barretenberg/[...]/bigfield_impl.hpp`		
Confirmed Fix At	https://github.com/AztecProtocol/aztec-packages/pull/16842		

The security analysts noted the following opportunities for performance improvements:

1. `bigfield_impl.hpp:467`: The function will only optimize constants if all three elements are constant, but any two constant elements can also be added together and reduced by the prime.
2. `bigfield_impl.hpp:507-521`: In the case where both elements are constant, the two modular reductions are unnecessary, as `reduction_check()` already handled them. Likewise, in the case where only `other` is constant, it too does not need to be reduced.
3. `bigfield_impl.hpp:get_quotient_reduction_info()`: This function calculates the `maximum_quotient` via `compute_maximum_quotient_value()`, which takes into account the maximum values of all terms involved in a multiplication+addition. Alternatively, `num_quotient_bits` is calculated from `get_quotient_max_bits()`, which only takes into account the maximum values of the remainders. This function should return the lower of the two bounds to reduce the size of the range check..

Impact The witness generator may be more efficient, or the constraints of the generated circuit may be reduced.

Recommendation Implement the recommended optimizations.

Developer Response

1. In the function `add_two`, we could add another `if` condition when two of the three inputs are constant and reduced by the prime. But the current implementation should handle that implicitly on calling `add_two` on each limb. Note that no additional gates would be added since each `add_two` call on limbs would just update additive constants on the witness.
2. Agree with the observation but we think its fine to have redundant reductions in constant case as that would cause negligible impact on prover performance.
3. It is a good suggestion to use a tighter bound for range constraint on the quotient when reduction is not necessary. However, while creating the quotient witness using the constructor:

```
1 template <typename Builder, typename T>
2 bigfield<Builder, T> bigfield<Builder, T>::create_from_u512_as_witness(Builder* ctx,
3                                                                        const
      uint512_t& value,
4                                                                        const bool
      can_overflow,
5                                                                        const size_t
      maximum_bitlength)
```

when `can_overflow` is set to `false`, `maximum_bitlength` must be greater than 3L. This assertion breaks if we use a tighter range constraint on the quotient. This needs to be investigated further.

Glossary

BN254 Also called BN-128 or `alt_bn_128`, this pairing-friendly elliptic curve is commonly used in zero knowledge proof systems. See `https://hackmd.io/@jpw/bn254` to learn more. 1

Chinese Remainder Theorem A famous theorem proving how to relate a number modulo several divisors to that same number modulo their product. See `https://en.wikipedia.org/wiki/Chinese_remainder_theorem` to learn more. 1, 24

CRT Chinese Remainder Theorem. 1

Satisfiability Modulo Theories The problem of determining whether a certain mathematical statement has any solutions. SMT solvers attempt to do this automatically. See `https://en.wikipedia.org/wiki/Satisfiability_modulo_theories` to learn more. 24

Semgrep Semgrep is an open-source, static analysis tool. See `https://semgrep.dev` to learn more. 7

SMT Satisfiability Modulo Theories. 3

zero-knowledge circuit A cryptographic construct that allows a prover to demonstrate to a verifier that a certain statement is true, without revealing any specific information about the statement itself. See `https://en.wikipedia.org/wiki/Zero-knowledge_proof` for more. 1