July 25, 2024

Prepared for
Innokentii Sennovskii
Aztec

Prepared by
Malte Leip
Sylvain Pelissier
Mohit Sharma
Zellic

Barretenberg Bigfield
Proof Circuit Security Assessment

Contents

← Back to Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Aztec from June 20th to July 9th, 2024. During this engagement, Zellic reviewed Barretenberg Bigfield's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Are all arithmetic operations and constructors for `bigfield` elements properly constrained?
- Are there completeness issues preventing honest provers from proving operations that should be valid?

1.3. Non-goals and Limitations

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide. In this case, the way that large parts of the code hinge crucially on careful bounds that need to be traced through the codebase, and which are not specified or documented, resulted in a significant amount of time needed per code unit for reviewing the code thoroughly. See section 4.1. ↗ for an expanded discussion of this point.

Due to this, the allocated time did not suffice to thoroughly review the entire in-scope codebase to our usual standard. We list below which parts of the code we reviewed in detail. Given the amount and severity of bugs found, we expect that there may exist additional serious bugs in the parts of code we only were able to review more lightly. We thus strongly recommend a reaudit, optimally after improving the documentation for the code as discussed in sections 4.1. ↗ and 4.2. ↗.

Based on the number of severe findings uncovered during the audit, it is our opinion that the project is not yet ready for production. We strongly advise a comprehensive reassessment before deployment to help identify any potential issues or vulnerabilities introduced by necessary fixes or changes. We also recommend adopting a security-focused development workflow, including (but not limited to) augmenting the repository with comprehensive end-to-end tests that achieve 100% branch coverage using any common, maintainable testing framework, thoroughly documenting all function requirements, and training developers to have a security mindset while writing code.

Parts of the code reviewed thoroughly

We reviewed thoroughly the entire file bigfield.hpp and the following functions in bigfield_impl.hpp:

- `bigfield` constructors
- `operator=`
- `get_value`
- `get_maximum_value`
- `add_to_lower_limb`
- `operator+`
- `operator-`
- `operator*`
- `sum`
- `operator==`
- `reduction_check`
- `self_reduce`
- `unsafe_evaluate_multiply_add` (branch for `HasPlookup<Builder>` only)
- `compute_quotient_remainder_values`
- `compute_maximum_quotient_value`
- `get_quotient_reduction_info`

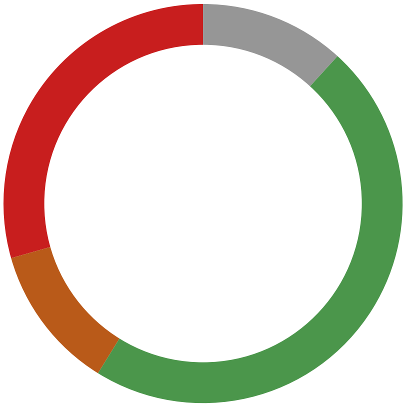
1.4. Results

During our assessment on the scoped Barretenberg Bigfield circuits, we discovered 17 findings. Five critical issues were found. Two were of high impact, eight were of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Aztec's benefit in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	5
🟧 High	2
🟨 Medium	0
🟩 Low	8
◼ Informational	2



2. Introduction

2.1. About Barretenberg Bigfield

Aztec contributed the following description of Barretenberg Bigfield:

> Aztec is a privacy-first L2 on Ethereum.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Underconstrained circuits. The most common type of vulnerability in a ZKP circuit is not adding sufficient constraints to the system. This leads to proofs generated with incorrect witnesses in terms of the specification of the project being accepted by the ZKP verifier. We manually check that the set of constraints satisfies soundness, enough to remove all such possibilities and in some cases provide a proof of the fact.

Overconstrained circuits. While rare, it is possible that a circuit is overconstrained. In this case, appropriately assigning witness will become impossible, leading to a vulnerability. To prevent this, we manually check that the constraint system is set up with completeness so that the proofs generated with the correct set of witnesses indeed pass the ZKP verification.

Missing range checks. This is a popular type of an underconstrained circuit vulnerability. Due to the usage of field arithmetic, overflow checks and range checks serve a huge purpose to build applications that work over the integers. We manually check the code for such missing checks and, in certain cases, provide a proof that the given set of range checks is sufficient to constrain the circuit up to specification.

Cryptography. ZKP technology and their applications are based on various aspects of cryptography. We manually review the cryptography usage of the project and examine the relevant studies and standards for any inconsistencies or vulnerabilities.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices, guidelines, and code quality standards.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding’s impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped circuits itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Barretenberg Bigfield Circuits

Type	C++
Platform	Barretenberg
Target	aztec-packages
Repository	https://github.com/AztecProtocol/aztec-packages ↗
Version	`4ba553ba3170838de3b6c4cf47b609b0198443d0`

Programs	`barretenberg/stdlib/primitives/bigfield/bigfield_impl.hpp`
	`barretenberg/stdlib/primitives/bigfield/bigfield.hpp`
	`barretenberg/stdlib/primitives/bigfield/constants.hpp`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 2.4 person-weeks. The assessment was conducted by three consultants over the course of three calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Malte Leip
Engineer
malte@zellic.io ↗

Sylvain Pelissier
Engineer
sylvain@zellic.io ↗

Mohit Sharma
Engineer
mohit@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

June 20, 2024	Kick-off call
June 20, 2024	Start of primary review period
July 9, 2024	End of primary review period

3. Detailed Findings

3.1. Range check in `unsafe_evaluate_multiply_add` may be ineffective

Target	bigfield_impl.hpp		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

This finding concerns the `unsafe_evaluate_multiply_add` and the `unsafe_evaluate_multiple_multiply_add` functions when `HasPlookup<Builder>` is true. We discuss the case of `unsafe_evaluate_multiply_add`; the situation for `unsafe_evaluate_multiple_multiply_add` is analogous.

The signature of this first function is

```
void bigfield<Builder,
    T>::unsafe_evaluate_multiply_add(const bigfield& input_left,
    const bigfield& input_to_mul, const std::vector<bigfield>& to_add,
    const bigfield& input_quotient,
    const std::vector<bigfield>& input_remainders)
```

and the function is intended to check that `input_left*input_to_mul + (to_add[0] + ... + to_add[-1]) - input_quotient*modulus - (input_remainders[0] + ... + input_remainders[-1]) = 0` modulo the CRT modulus $2^t \cdot r$ where r is the native circuit modulus. To carry out this check, congruence to 0 modulo 2^t and r is checked separately, and for the former check, `evaluate_non_native_field_multiplication` is used:

```
const auto [lo_idx, hi_idx]
    = ctx->evaluate_non_native_field_multiplication(witnesses, false);
```

As described in section 4.4. ↗, that function does not itself constrain the identity to hold, but it is required that the caller perform a range check to verify that the two return values are at most 251264705520246785319773662051664216, a 118-bit number.

The range check performed is as follows:

```
bb::fr neg_prime = -bb::fr(uint256_t(target_basis.modulus));
field_t<Builder>::evaluate_polynomial_identity(left.prime_basis_limb,
    to_mul.prime_basis_limb, quotient.prime_basis_limb * neg_prime,
    -remainder_prime_limb);
```

```
field_t lo = field_t<Builder>::from_witness_index(ctx, lo_idx);
field_t hi = field_t<Builder>::from_witness_index(ctx, hi_idx);
const uint64_t carry_lo_msb = max_lo_bits - (2 * NUM_LIMB_BITS);
const uint64_t carry_hi_msb = max_hi_bits - (2 * NUM_LIMB_BITS);

// if both the hi and lo output limbs have less than 70 bits, we can use our
    custom
// limb accumulation gate (accumulates 2 field elements, each composed of 5
    14-bit limbs, in 3 gates)
if (carry_lo_msb <= 70 && carry_hi_msb <= 70) {
    ctx->range_constrain_two_limbs(
        hi.witness_index, lo.witness_index, size_t(carry_lo_msb),
    size_t(carry_hi_msb));
} else {
    ctx->decompose_into_default_range(hi.normalize().witness_index,
    carry_hi_msb);
    ctx->decompose_into_default_range(lo.normalize().witness_index,
    carry_lo_msb);
}
```

As can be seen here, the values are range checked for `max_lo_bits - (2 * NUM_LIMB_BITS)` and `max_hi_bits - (2 * NUM_LIMB_BITS)` bits. Should those values be 118 bits or larger, then an overflow will be possible, breaking soundness of the check of the identity. Thus, for correct functioning, it is necessary that `max_lo_bits` and `max_hi_bits` are at most 253.

These are defined earlier in the code as follows:

```
const uint512_t max_lo = max_r0 + (max_r1 << NUM_LIMB_BITS) + max_a0;
const uint512_t max_hi = max_r2 + (max_r3 << NUM_LIMB_BITS) + max_a1;

uint64_t max_lo_bits = (max_lo.get_msb() + 1);
uint64_t max_hi_bits = max_hi.get_msb() + 1;
if ((max_lo_bits & 1ULL) == 1ULL) {
    ++max_lo_bits;
}
if ((max_hi_bits & 1ULL) == 1ULL) {
    ++max_hi_bits;
}
```

Before the last increment here, the two values may be at most 252, so `max_lo` and `max_hi` may be at most numbers with a bit width of 252. This in turn implies that `max_r1` and `max_r3` may be at most 184 bits wide.

These are defined as follows:

```
   uint512_t max_b0 = (left.binary_basis_limbs[1].maximum_value *
    to_mul.binary_basis_limbs[0].maximum_value);
   max_b0 += (neg_modulus_limbs_u256[1]
    * quotient.binary_basis_limbs[0].maximum_value);
   uint512_t max_b1 = (left.binary_basis_limbs[0].maximum_value *
    to_mul.binary_basis_limbs[1].maximum_value);
   max_b1 += (neg_modulus_limbs_u256[0]
    * quotient.binary_basis_limbs[1].maximum_value);
   uint512_t max_c0 = (left.binary_basis_limbs[1].maximum_value *
    to_mul.binary_basis_limbs[1].maximum_value);

   // ...

to_mul.binary_basis_limbs[0].maximum_value);
   max_d0 += (neg_modulus_limbs_u256[3]
    * quotient.binary_basis_limbs[0].maximum_value);
   uint512_t max_d1 = (left.binary_basis_limbs[2].maximum_value *
    to_mul.binary_basis_limbs[1].maximum_value);
   max_d1 += (neg_modulus_limbs_u256[2]
    * quotient.binary_basis_limbs[1].maximum_value);
   uint512_t max_d2 = (left.binary_basis_limbs[1].maximum_value *
    to_mul.binary_basis_limbs[2].maximum_value);
   max_d2 += (neg_modulus_limbs_u256[1]
    * quotient.binary_basis_limbs[2].maximum_value);
   uint512_t max_d3 = (left.binary_basis_limbs[0].maximum_value *
    to_mul.binary_basis_limbs[3].maximum_value);
   max_d3 += (neg_modulus_limbs_u256[0]
    * quotient.binary_basis_limbs[3].maximum_value);

   // ...

   const uint512_t max_r1 = max_b0 + max_b1;
   const uint512_t max_r2 = max_c0 + max_c1 + max_c2;
   const uint512_t max_r3 = max_d0 + max_d1 + max_d2 + max_d3;
```

Now if `left.binary_basis_limbs[1].maximum_value` and `to_mul.binary_basis_limbs[0].maximum_value` are each 117 bits wide, then `max_b0` and hence `max_r1` can be up to 233 bits wide, which is more than 184. Similarly, if `left.binary_basis_limbs[1].maximum_value` and `to_mul.binary_basis_limbs[2]` are each 117 bits wide, then `max_d1` and hence `max_r3` can be up to 233 bits wide.

It would appear that limbs of such width should be supported by `unsafe_evaluate_multiply_add`, as such limbs pass `reduction_check`. A call of `unsafe_evaluate_multiply_add` with inputs triggering this bug can in particular be reached through `operator*`.

Impact

It is possible to prove incorrect multiply-add identities for some `bigfield` elements. In particular, it is possible to prove incorrect results for the product of some `bigfield` elements that have sufficiently large limbs.

Recommendations

As a first measure, assert that `max_lo_bits` and `max_hi_bits` are at most 253. This will convert this soundness issue into a completeness issue. For completeness, it may be possible to keep the code for `unsafe_evaluate_multiply_add` and merely make stricter assumptions on the inputs. Otherwise, changes to the function's logic would be required.

Remediation

This issue has been acknowledged by Aztec, and a fix was implemented in commit d9ca76a2 ↗, by reducing the maximum bit width of limbs before reduction from 117 to 78, with execution stopping should a limb have maximum value more than 83 bits wide.

3.2. Overflow possible in `evaluate_non_native_field_multiplication`

Target	UltraCircuitBuilder		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

This finding concerns the method `evaluate_non_native_field_multiplication` of the class `UltraCircuitBuilder_`. We refer to section 4.4. ↗ for background on how this function is intended to work.

The issue is that calculations done by `evaluate_non_native_field_multiplication` can overflow modulo the native circuit prime modulus r. (Note that the function uses an array also called `r` for the remainder; to distinguish the remainder from the prime, we will only refer to the limbs of the remainder `r[0]`, `r[1]`, `r[2]`, and `r[3]` but never use `r` for the array itself.) The core computation of the function is summarized by the following snippet containing the out-of-circuit calculation of the witnesses, which are then constrained elsewhere in the function:

```
FF lo_0 = a[0] * b[0] - r[0] + (a[1] * b[0] + a[0] * b[1]) * LIMB_SHIFT;
FF lo_1 = (lo_0 + q[0] * input.neg_modulus[0] +
           (q[1] * input.neg_modulus[0] + q[0] * input.neg_modulus[1] - r[1])
    * LIMB_SHIFT) *
          LIMB_RSHIFT_2;

FF hi_0 = a[2] * b[0] + a[0] * b[2] + (a[0] * b[3] + a[3] * b[0] - r[3])
    * LIMB_SHIFT;
FF hi_1 = hi_0 + a[1] * b[1] - r[2] + (a[1] * b[2] + a[2] * b[1]) * LIMB_SHIFT;
FF hi_2 = (hi_1 + lo_1 + q[2] * input.neg_modulus[0] +
           (q[3] * input.neg_modulus[0] + q[2] * input.neg_modulus[1])
    * LIMB_SHIFT);
FF hi_3 = (hi_2 + (q[0] * input.neg_modulus[3] + q[1] * input.neg_modulus[2])
    * LIMB_SHIFT +
           (q[0] * input.neg_modulus[2] + q[1] * input.neg_modulus[1])) *
          LIMB_RSHIFT_2;
```

These calculations can overflow modulo r. If an overflow happens, `evaluate_non_native_field_multiplication` can be used to prove incorrect multiplications, so it is crucial that callers ensure that all limbs of the arguments are sufficiently constrained to ensure no overflow can happen.

The method `evaluate_non_native_field_multiplication` under discussion is in particular called by `bigfield::unsafe_evaluate_multiple_multiply_add`, which is in turn used by `bigfield::operator*`. We will now describe how a call to `bigfield::operator*` can ultimately result in an overflow in `evaluate_non_native_field_multiplication` in such a way that an incorrect product can be proven. There are likely also other paths to use this bug; this is just one concrete example.

Let `a`, `b`, and `c` be such that `(a<<68) * b = r + c`, with `c<r` and `a,b < 2^117`. This is possible [1] because `2 * 117 + 68 = 302` is significantly bigger than the bit width of r, which is 254. Now suppose we instantiate two `bigfield` elements `A` and `B` with binary limbs (in little endian) `(0, a, 0, 0)` and `(b, 0, 0, 0)`. Applying `operator*` to calculate `A*B`, no reductions would be needed as the limbs of both elements are less than 117 bits and the unsigned integer represented is also small enough, since we are only using the two least significant limbs. The two elements would thus be passed to `unsafe_evaluate_multiple_multiply_add` unchanged.

In the `HasPlookup<Builder>` case, we would ultimately end up in `evaluate_non_native_field_multiplication`, with factors still unchanged `A` and `B`. The calculation of `lo_0` would then overflow:

```
FF lo_0 = a[0] * b[0] - r[0] + (a[1] * b[0] + a[0] * b[1]) * LIMB_SHIFT;
```

This expression calculating `lo_0` simplifies to `a*b*(1<<68) - r[0]`, which is by our assumption equal to `r+c - r[0]`. As the calculation is done modulo r, we obtain an overflow and will have `lo_0=c - r[0]`. We will then have that all further computations, ending up with `lo_1` and `hi_3`, will be zero, if we set the remainder to be proven to `(c, 0, 0, 0)` and the quotient to `(0, 0, 0, 0)`.

Thus, the checks in `evaluate_non_native_field_multiplication` and `bigfield::unsafe_evaluate_multiple_multiply_add` will amount to the incorrect acceptance of `(a<<68) * b` to be equal to `c` modulo `2^t`, which is, however, false. The check of the identity for the prime limbs will also pass, as `(a<<68) * b` actually is equal to `c` modulo r.

Thus, a malicious prover can ultimately prove that $(a << 68) \cdot b = c$ modulo the emulated prime modulus p, even though this is wrong.

Impact

It is possible to prove incorrect product identities modulo 2^t, which can be leveraged from the `bigfield` component in order to (in particular) prove incorrect products of `bigfield` elements.

[1] We can also make an example with concrete values. As $r = 21888242871839275222246405745257275088548364400416034343698204186575808495617$, we can choose $a = b = 8611634657822393288703792040$, a 93-bit number, and $c = 279393233962661376387625408336507998066373033983$, a 158-bit number.

Recommendations

Ensure that no overflows modulo r can happen in evaluate_non_native_field_multiplication. As this function is not passed maximums for its arguments, this likely needs to be done in two steps.

First, document bounds for the arguments to evaluate_non_native_field_multiplication that callers must enforce, and ensure that those bounds are strict enough that no overflows modulo r can happen.

Second, make sure callers such as unsafe_evaluate_multiply_add (perhaps by documenting requirements for its callers) actually enforce those bounds. How to do that might depend from case to case. Carrying out a reduction check is not enough, as the example above demonstrates (operator* does have a reduction check, but as the limbs are all small enough, the elements pass it without reduction). So it might be necessary to check whether the maximums satisfy the bounds, and if not, then reduce both elements and check again.

The situation is complicated by the fact that unsafe_evaluate_multiply_add (which calls evaluate_non_native_field_multiplication) is used by self_reduce, and for that usage, unsafe_evaluate_multiply_add must be able to deal with values for input_left that have limbs that are even bigger than 117 bits wide. Reducing to deal with this is not an option, as this is exactly the call intended to perform the reduction. While input_left might have very large limbs on calls by self_reduce, the second factor input_to_mul will just be one, however. So, it may be feasible to make two sets of assumptions for arguments to unsafe_evaluate_multiply_add, and satisfying either of those two sets of assumptions will ensure the conditions necessary for evaluate_non_native_field_multiplication.

Remediation

This issue has been acknowledged by Aztec. Commit d9ca76a2 ↗ reduces the maximum bit width of limbs before reduction from 117 to 78, with execution stopping should a limb have maximum value more than 83 bits wide, which prevents the example described above leading to an overflow in evaluate_non_native_field_multiplication. We have not checked whether there are other ways to reach an overflow in evaluate_non_native_field_multiplication that are still possible.

3.3. Underflow possible in `evaluate_non_native_field_multiplication`

Target	UltraCircuitBuilder		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

This finding primarily concerns the method `evaluate_non_native_field_multiplication` of the class `UltraCircuitBuilder_`. We refer to section 4.4. ↗ for background on how this function is intended to work. In Finding 3.2. ↗, we discuss how the calculations done in that function can overflow modulo r, the native circuit prime modulus, due to sums of products becoming too large. In this finding, we discuss how subtractions happening in the calculations can cause an underflow modulo r, similarly causing incorrect results, though in this case resulting in a completeness rather than a soundness issue (under mild assumptions on caller behavior).

The core computation of the function is summarized by the following snippet containing the out-of-circuit calculation of the witnesses, which are then constrained elsewhere in the function:

```
FF lo_0 = a[0] * b[0] - r[0] + (a[1] * b[0] + a[0] * b[1]) * LIMB_SHIFT;
FF lo_1 = (lo_0 + q[0] * input.neg_modulus[0] +
           (q[1] * input.neg_modulus[0] + q[0] * input.neg_modulus[1] - r[1])
    * LIMB_SHIFT) *
          LIMB_RSHIFT_2;

FF hi_0 = a[2] * b[0] + a[0] * b[2] + (a[0] * b[3] + a[3] * b[0] - r[3])
    * LIMB_SHIFT;
FF hi_1 = hi_0 + a[1] * b[1] - r[2] + (a[1] * b[2] + a[2] * b[1]) * LIMB_SHIFT;
FF hi_2 = (hi_1 + lo_1 + q[2] * input.neg_modulus[0] +
           (q[3] * input.neg_modulus[0] + q[2] * input.neg_modulus[1])
    * LIMB_SHIFT);
FF hi_3 = (hi_2 + (q[0] * input.neg_modulus[3] + q[1] * input.neg_modulus[2])
    * LIMB_SHIFT +
           (q[0] * input.neg_modulus[2] + q[1] * input.neg_modulus[1])) *
          LIMB_RSHIFT_2;
```

Note that the four remainder limbs are subtracted here. Thus, if the other terms of the expressions for the crucial end results `lo_1/LIMB_RSHIFT_2` and `hi_3/LIMB_RSHIFT_2` are smaller than the terms that get subtracted, which are `r[0] + r[1]*LIMB_SHIFT` and `r[2] + r[3]*LIMB_SHIFT`, respectively, then an underflow and wraparound modulo r will happen. Calls to `evaluate_non_native_field_multiplication` usually have remainders that passed a reduction check before, so `r[0]`,`r[1]`,`r[2]`,`r[3]` should all be at most about 117 bits wide, and so the subtracted terms must be significantly smaller than r. Thus, on wraparound due to underflow, we must obtain a value that is very large (close to r). Putting aside the issue described in Finding 3.1. ↗, the range check on lo_1 or hi_3 would then fail. Thus, the possibility of an underflow in these calculations amounts to a completeness issue.

We now discuss a realistic example that is impacted by this completeness issue. Suppose we construct two bigfield elements with binary limbs (using little-endian notation) (0, 1<<116, 0, 0) and (1, 0, 0, 0) and divide the former by the latter. The former represents the element 1<<(116+68) = 1<<184, and the second the element 1, so the division should result in 1<<184 again. Note that both elements will pass the reduction check.

The call to operator/ will cause a call to internal_div with numerators = [(0, 1<<116, 0, 0)] and denominator = (1, 0, 0, 0). This function then calculates some additional values as follows:

```
const uint1024_t left = uint1024_t(numerator_values);
const uint1024_t right = uint1024_t(denominator.get_value());
const uint1024_t modulus(target_basis.modulus);
uint512_t inverse_value = right.lo.invmod(target_basis.modulus).lo;
uint1024_t inverse_1024(inverse_value);
inverse_value = ((left * inverse_1024) % modulus).lo;

const uint1024_t quotient_1024 =
    (uint1024_t(inverse_value) * right + unreduced_zero().get_value() - left)
    / modulus;
const uint512_t quotient_value = quotient_1024.lo;

// ...
quotient = create_from_u512_as_witness(ctx, quotient_value, false,
    num_quotient_bits);
inverse = create_from_u512_as_witness(ctx, inverse_value);
```

Here we will then have inverse_value = 1<<(116+68). Thus, inverse will be (0, 0, 1<<48, 0). As unreduced_zero() is 263 bits wide if p is 256 bits, we will get that the quotient is at most a 7-bit value of the form (q, 0, 0, 0). In the end, internal_div calls unsafe_evaluate_multiply_add as follows:

```
unsafe_evaluate_multiply_add(denominator, inverse, { unreduced_zero() },
    quotient, numerators);
```

Here is the signature of that function:

```
template <typename Builder, typename T>
void bigfield<Builder,
    T>::unsafe_evaluate_multiply_add(const bigfield& input_left,
    const bigfield& input_to_mul, const std::vector<bigfield>& to_add,
    const bigfield& input_quotient,
    const std::vector<bigfield>& input_remainders)
```

In our case, we will have the following values:

```
input_left = (1, 0, 0, 0)
input_to_mul = (0, 0, 1<<48, 0)
to_add = [(u0, u1, u2, u3)]
input_quotient = (q, 0, 0, 0)
input_remainders = [(0, 1<<116, 0, 0)]
```

Where each `u` is at most 68 bits wide, and `q` is at most 7 bits wide. We assume we are in the `HasPlookup<Builder>` case and the `needs_normalize=true`. The function will add various contributions from `input_remainders` and `to_add` together. At the end we will get this:

```
remainder_limbs = (-u0 + (1<<(68+116)) - (u1<<68), 0, -u2 - u3<<68 , 0)
```

Then, the function `UltraCircuitBuilder_::evaluate_non_native_field_multiplication` is called with the following parameters:

```
a = (1, 0, 0, 0)
b = (0, 0, 1<<48, 0)
q = (q, 0, 0, 0)
r = (-u0 + (1<<(68+116)) - (u1<<68), 0, -u2 - u3<<68 , 0)
```

The function `evaluate_non_native_field_multiplication` will calculate and constrain the following values, with `lo_1` being returned to be range checked:

```
FF lo_0 = a[0] * b[0] - r[0] + (a[1] * b[0] + a[0] * b[1]) * LIMB_SHIFT;
FF lo_1 = (lo_0 + q[0] * input.neg_modulus[0] +
           (q[1] * input.neg_modulus[0] + q[0] * input.neg_modulus[1] - r[1])
    * LIMB_SHIFT) *
          LIMB_RSHIFT_2;
```

In the case under consideration, this will amount to the following:

```
lo_0 = u0 + (u1<<68) - (1<<(68+116)
lo_1 = (u0 + (u1<<68) - (1<<(68+116)) + q*neg_modulus[0]
    + (q*neg_modulus[1])<<68) * LIMB_RSHIFT_2
     = (u0 + (u1<<68) + q*neg_modulus[0]
    + (q*neg_modulus[1])<<68  - (1<<(68+116))) * LIMB_RSHIFT_2
```

As stated before, `u0` and `u1` are at most 68 bits wide, so the first two summands are at most 68 and `2*68=136` bits wide, respectively. As `q` was estimated to be at most 7 bits wide and `neg_modulus[0]` and neg_modulus[1] are at most 68 bits, we furthermore conclude that the third and fourth summands are at most 7+68=75 and 7+68+68=143 bits wide. Thus, the sum of the four positive summands can be at most 144 bits wide. However, 1<<(68+116) is larger than even any 68+116=184-bit–wide value. Thus, we will get an underflow. Note that while our estimates might be off by a small number of bits on intermediate steps (for example if p has less than 256 bits), the gap of 40 bits is so large that such estimation errors do not change the conclusion.

The following test case demonstrates the problem. To demonstrate that the values needed to trigger the error can occur naturally, we instantiate a bigfield element with limbs at most 68 bits wide (the second limb being 1<<67, the others zero), and then add this element to itself nine times, to obtain an element with 1<<76 as the second limb. By testing with this test case, we empirically see that the issue is triggered by this limb 1<<76, while it is not triggered by 1<<75. This is as expected, as 76 is exactly 40 less than 116 — this being precisely the gap of 40 bits we arrived at above.

```
static void test_zellic_division_underflow()
{
    std::cout << "\nIn Zellic division underflow test!" << std::endl;
    auto builder = Builder();
    fq_ct numerator = fq_ct::create_from_u512_as_witness(&builder,
    uint256_t(1) << (68 + 67));
    ASSERT(numerator.binary_basis_limbs[0].element.get_value() == 0);
    ASSERT(numerator.binary_basis_limbs[1].element.get_value()
    == uint256_t(1)<<67);
    ASSERT(numerator.binary_basis_limbs[2].element.get_value() == 0);
    ASSERT(numerator.binary_basis_limbs[3].element.get_value() == 0);
    numerator.binary_basis_limbs[0].maximum_value = 0;
    numerator.binary_basis_limbs[1].maximum_value = uint256_t(1)<<67;
    numerator.binary_basis_limbs[2].maximum_value = 0;
    numerator.binary_basis_limbs[3].maximum_value = 0;

    for (size_t i = 0; i < 9 ; i++) {
        numerator = numerator + numerator;
        info("Doubled ", i, " times, second limb:  ",
    numerator.binary_basis_limbs[1].element);
    }
    info("\nnumerator:");
    for (size_t limb_index = 0; limb_index < 4 ; limb_index++) {
        info("Limb ", limb_index, ":  ",
    numerator.binary_basis_limbs[limb_index]);
    }
    fq_ct denominator = fq_ct::create_from_u512_as_witness(&builder,
    uint256_t(1));
    info("\ndenominator:");
    for (size_t limb_index = 0; limb_index < 4 ; limb_index++) {
        info("Limb ", limb_index, ":  ",
    denominator.binary_basis_limbs[limb_index]);
    }
    fq_ct result = numerator / denominator;
    info("\nback in test case:");
    info("\nresult:");
    for (size_t limb_index = 0; limb_index < 4 ; limb_index++) {
        info("Limb ", limb_index, ": ",
    result.binary_basis_limbs[limb_index]);
    }
    bool checked = CircuitChecker::check(builder);
    EXPECT_TRUE(checked);
}
```

This test case fails with the following output:

```
Failed Arithmetic relation at row idx = 1835
Failed at block idx = 1
/home/user/aztec-
    packages/barretenberg/cpp/src/barretenberg/stdlib/primitives/bigfield/
bigfield.test.cpp:372: Failure
Value of: checked
  Actual: false
Expected: true
[ FAILED ] stdlib_bigfield/1.zellicdivisionunderflow, where TypeParam =
    bb::UltraCircuitBuilder_<bb::UltraArith<bb::field<bb::Bn254FrParams> > >
    (26 ms)
[----------] 1 test from stdlib_bigfield/1 (26 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (26 ms total)
[ PASSED ] 0 tests.
[ FAILED ] 1 test, listed below:
[ FAILED ] stdlib_bigfield/1.zellicdivisionunderflow, where TypeParam =
    bb::UltraCircuitBuilder_<bb::UltraArith<bb::field<bb::Bn254FrParams> > >

 1 FAILED TEST
```

The test case fails both for the UltraCircuitBuilder as expected based on the discussion above. With the StandardCircuitBuilder, the test case fails as well, likely for analogous reasons. However, we have not analyzed the precise flow for this case.

Impact

Some divisions and possibly other arithmetic operations for `bigfield` elements will fail to be proven in the current implementation, amounting to a completeness issue.

Recommendations

As limbs come with maximum bounds but not minimum bounds, this problem cannot be easily fixed merely by checking whether certain bounds are satisfied and if not to reduce. Note that, considering `evaluate_non_native_field_multiplication` itself, there is no reasonable upper bound for the remainder that can reliably prevent an underflow in `lo_1`, as the lower two limbs of `a` and `b` could be zero, with `q` being `1`, so that it would be required to impose the requirement that `r[1] <= input.neg_modulus[1]`. However, as `input.neg_modulus[1]` is just a 68-bit value, this would mean that `r[1]` is restricted from taking some larger 68-bit values, which would make it impossible to represent some remainders for passing to the `evaluate_non_native_field_multiplication` function.

A solution might thus involve restricting the maximum value of the remainder limbs and then adding an appropriate multiple of `1<<(2*68)` to `lo_1` and `hi_3` before multiplication by `LIMB_RSHIFT_2` that is ensured to be larger than the negative contributions from the remainder. This would then prevent an underflow from happening. Care must be taken to prevent an overflow; see Finding 3.2. ↗. Estimates for how wide of a range check `lo_1` and `hi_3` need to be checked with will need to be updated as well. Note that `hi_2` needs to be corrected to account for the modification of `lo_2`.

Remediation

This issue has been acknowledged by Aztec. Commit f567559c ↗ estimates the maximum absolute value of the subtracted summands, and adds a corresponding correction to the low part (`lo_1`) before performing the range check in `unsafe_evaluate_multiply_add` and `unsafe_evaluate_multiple_multiply_add`. This prevents the issue for the low part.

For the high part (`hi_3`), no modification was done. The argument for why this is not needed is roughly as follows[2]: If $a \cdot b - r - p \cdot q \equiv 0 \pmod{s \cdot 2^{4 \cdot 68}}$, then `hi_3` will have the value $(a \cdot b - r - p \cdot q)/2^{2 \cdot 68}$, calculated in the field $\mathbb{F}_s$. By the assumption, this will be a multiple of $s \cdot 2^{2 \cdot 68}$, and hence zero modulo s. Thus the range check on `hi_3` will succeed. Essentially, any hypothetical underflow in the calculation of `hi_3` for valid witnesses must underflow by a multiple of s and hence not actually change the value modulo s.

[2] As r will be used for the remainder, we will use s for the native circuit prime modulus here.

3.4. Missing `field_t` normalization in constructor

Target	bigfield_impl.hpp		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

This finding concerns the following constructor for `bigfield`:

```
template <typename Builder, typename T>
bigfield<Builder, T>::bigfield(const field_t<Builder>& low_bits_in,
                               const field_t<Builder>& high_bits_in,
                               const bool can_overflow,
                               const size_t maximum_bitlength)
```

After the `if` check on `HasPlookup<Builder>`, in the `else` branches, the method `decompose_into_base4_accumulators` of the class `StandardCircuitBuilder_<FF>` is called. This function takes raw witness indexes as input, but the arguments are not normalized in the `bigfield` constructor beforehand, as can be seen here in the case of the lower bits:

```
low_accumulator = context->decompose_into_base4_accumulators(
    low_bits_in.witness_index, static_cast<size_t>(NUM_LIMB_BITS * 2),
    "bigfield: low_bits_in too large.");
```

This applies also for the higher bits.

Impact

If the additive or multiplicative constant of the `field_t` values `low_bits_in` or `high_bits_in` are different from `1` and `0`, respectively, the constructor will construct a `bigfield` element with incorrect values.

Recommendations

Normalize the witness before passing the witness index as an argument to `decompose_into_base4_accumulators`, for example in the lower-bits case,

```
low_accumulator = context->decompose_into_base4_accumulators(
    low_bits_in.witness_index, static_cast<size_t>(NUM_LIMB_BITS * 2), "
        bigfield: low_bits_in too large.");
    low_bits_in.normalize().witness_index, static_cast<size_t>(NUM_LIMB_BITS *
         2), "bigfield: low_bits_in too large.");
```

and similarly for the higher-bits case.

Remediation

This issue has been acknowledged by Aztec, and a fix was implemented in commit 78f47b6d ↗.

3.5. Missing range check in constructors

Target	bigfield.hpp		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

Some `bigfield` constructors do not range check their arguments appropriately. For example, consider the following `bigfield` constructor:

```
// we assume the limbs have already been normalized!
bigfield(const field_t<Builder>& a,
        const field_t<Builder>& b,
        const field_t<Builder>& c,
        const field_t<Builder>& d,
        const bool can_overflow = false)
{
    context = a.context;
    binary_basis_limbs[0] = Limb(field_t(a));
    binary_basis_limbs[1] = Limb(field_t(b));
    binary_basis_limbs[2] = Limb(field_t(c));
    binary_basis_limbs[3] =
        Limb(field_t(d), can_overflow ? DEFAULT_MAXIMUM_LIMB
    : DEFAULT_MAXIMUM_MOST_SIGNIFICANT_LIMB);
    prime_basis_limb =
        (binary_basis_limbs[3].element * shift_3)
            .add_two(binary_basis_limbs[2].element * shift_2,
    binary_basis_limbs[1].element * shift_1);
    prime_basis_limb += (binary_basis_limbs[0].element);
};
```

It takes as input four values in the native field $\mathbb{F}_r$ and creates four `Limb` objects from those values. However, the relevant constructor for the `Limb` object is implemented as follows:

```
Limb(const field_t<Builder>& input, const uint256_t max = uint256_t(0))
            : element(input)
{
    if (input.witness_index == IS_CONSTANT) {
        maximum_value = uint256_t(input.additive_constant) + 1;
    } else if (max != uint256_t(0)) {
        maximum_value = max;
    } else {
        maximum_value = DEFAULT_MAXIMUM_LIMB;
    }
}
```

It assigns the value to the `element` field but does not check for the value being in the correct range that is indicated by `maximum_value`.

The same issue occurs with the `bigfield` constructor taking in an additional argument `prime_limb`.

Impact

All further computations are based on the crucial assumption that the limb elements are less than or equal to the `maximum_value` stored alongside them. Violation of that assumption can, for example, cause overflows on later use, impacting soundness by allowing to prove for example incorrect products of `bigfield` elements.

Recommendations

Implement the appropriate range check in the constructor of `Limb` or in the `bigfield` constructors. Alternatively, specify clearly that the constructors assume that the arguments have already been range checked to lie in specific ranges.

Remediation

This issue has been acknowledged by Aztec, and a fix was implemented in commit ab2308de ↗.

3.6. Missing consistency check for prime limb in constructor

Target	bigfield.hpp		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	Low	**Impact**	High

Description

The functioning of `bigfield` functions rely crucially on the assumption that the prime limb is consistent with the unsigned integer stored in the binary limbs. See the discussion in section 4.3. ↗. Breaking this assumption would allow for proving incorrect relations in, for example, `unsafe_evaluate_multiply_add`, where equality is checked separately modulo 2^t (using the binary limbs) and modulo r (using the prime limbs), which can ultimately mean proving, for example, incorrect products.

However, the following constructor does not ensure that the prime limb is consistent with the binary limbs:

```
// we assume the limbs have already been normalized!
bigfield(const field_t<Builder>& a,
         const field_t<Builder>& b,
         const field_t<Builder>& c,
         const field_t<Builder>& d,
         const field_t<Builder>& prime_limb,
         const bool can_overflow = false)
{
    context = a.context;
    binary_basis_limbs[0] = Limb(field_t(a));
    binary_basis_limbs[1] = Limb(field_t(b));
    binary_basis_limbs[2] = Limb(field_t(c));
    binary_basis_limbs[3] =
        Limb(field_t(d), can_overflow ? DEFAULT_MAXIMUM_LIMB
    : DEFAULT_MAXIMUM_MOST_SIGNIFICANT_LIMB);
    prime_basis_limb = prime_limb;
};
```

Impact

Further computations are based on the crucial assumption that the prime limb is consistent with the binary limbs. Violation of that assumption can, for example, allow proving an incorrect product of such a `bigfield` element with another.

Recommendations

Constrain prime_limb to be consistent with the binary limbs. Alternatively, document clearly that the caller is responsible for ensuring this.

Additionally, for defense in depth, we recommend to assert consistency of the binary limbs with the prime limb in bigfield<Builder, T>::get_value() and for the inputs in the functions that check things separately for binary and prime modulus, such as unsafe_evaluate_multiply_add.

Remediation

This issue has been acknowledged by Aztec, and a fix was implemented in commit ab2308de ↗.

3.7. Proving that multiples of p are unequal to 0 modulo p possible

Target	bigfield_impl.hpp		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	Low	**Impact**	High

Description

The `assert_is_not_equal` function for `bigfield` elements is implemented as follows:

```
template <typename Builder, typename T> void bigfield<Builder,
    T>::assert_is_not_equal(const bigfield& other) const
{
    // Why would we use this for 2 constants? Turns out, in biggroup
    const auto get_overload_count
    = [target_modulus = modulus_u512](const uint512_t& maximum_value) {
        uint512_t target = target_modulus;
        size_t overload_count = 0;
        while (target < maximum_value) {
            ++overload_count;
            target += target_modulus;
        }
        return overload_count;
    };
    const size_t lhs_overload_count = get_overload_count(get_maximum_value());
    const size_t rhs_overload_count
    = get_overload_count(other.get_maximum_value());

    // if (a == b) then (a == b mod n)
    // to save gates, we only check that (a == b mod n)

    // if numeric val of a = a' + p.q
    // we want to check (a' + p.q == b mod n)
    const field_t<Builder> base_diff = prime_basis_limb
    - other.prime_basis_limb;
    auto diff = base_diff;
    field_t<Builder> prime_basis(get_context(), modulus);
    field_t<Builder> prime_basis_accumulator = prime_basis;
    // Each loop iteration adds 1 gate
    // (prime_basis and prime_basis accumulator are constant so only the *
    operator adds a gate)
    for (size_t i = 0; i < lhs_overload_count; ++i) {
        diff = diff * (base_diff - prime_basis_accumulator);
        prime_basis_accumulator += prime_basis;
    }
    prime_basis_accumulator = prime_basis;
    for (size_t i = 0; i < rhs_overload_count; ++i) {
        diff = diff * (base_diff + prime_basis_accumulator);
        prime_basis_accumulator += prime_basis;
    }
    diff.assert_is_not_zero();
}
```

The way this function is intended to work can be summarized as follows. We have two unsigned integers, say a and b, which we want to prove are unequal modulo p. Assume for the moment $a > b$. If $k \cdot p$ is a bound for the maximum value that $a - b$ might be, then this follows if we can show, in integers, that $a - b - i \cdot p \neq 0$ for $i = 0, 1, ..., k$. The case for $b \geq a$ is symmetrical.

The second half of the function's implementation carries out these checks (modulo r, the native circuit prime modulus, but this is not relevant for this finding). To obtain the parameter k so that $a - b \leq k \cdot p$, the code assumes that b has the minimum possible value it could have, namely zero, and a has the maximum value it possibly could have (which can be calculated from the maximum values of its limbs).

The function `get_overload_count` is then intended on argument m to calculate m/p. However, the function is incorrect if m is a multiple of p. In that case, the loop stops one iteration too early, causing the return value to be too small by one. For example, consider the case where the argument is p itself. In that case, `target < maximum_value` will be $p < p$, which is false, so the loop body will never run and the function will return 0. Thus, if $a = p$, and this is the maximum value that a can have, then for the case of b being smaller than a, only $a - b \neq 0$ is checked, but not $a - b - p \neq 0$, as it should be. Thus, if $a = p$ with $b = 0$, then `assert_is_not_equal` will not detect that those values are equal modulo p, so a malicious prover can succeed in incorrectly proving that those values are unequal.

The following test case demonstrates this issue:

```
static void test_zellic_assert_not_equal()
{
    std::cout << "\nIn Zellic assert_is_not_equal!" << std::endl;
    auto builder = Builder();
    fq_ct zero = fq_ct::create_from_u512_as_witness(&builder, uint256_t(0));
    fq_ct alsozero = fq_ct::create_from_u512_as_witness(&builder,
    fq_ct::modulus_u512);
    for (size_t i = 0; i < 4; i++) {
        zero.binary_basis_limbs[i].maximum_value =
    zero.binary_basis_limbs[i].element.get_value();
        alsozero.binary_basis_limbs[i].maximum_value =
    alsozero.binary_basis_limbs[i].element.get_value();
    }
    info("\nzero:");
    for (size_t limb_index = 0; limb_index < 4 ; limb_index++) {
        info("Limb ", limb_index, ": ", zero.binary_basis_limbs[limb_index]);
    }
    info("Prime limb: ", zero.prime_basis_limb);
    info("\nalsozero:");
    for (size_t limb_index = 0; limb_index < 4 ; limb_index++) {
        info("Limb ", limb_index, ": ",
    alsozero.binary_basis_limbs[limb_index]);
    }
    info("Prime limb: ", alsozero.prime_basis_limb);
    zero.assert_is_not_equal(alsozero);
    bool result = CircuitChecker::check(builder);
    EXPECT_EQ(result, true);
}
```

In this test case, `assert_is_not_equal` is used to prove that 0 and p are unequal modulo p. The test case succeeds, even though it should not:

```
% ./build/bin/stdlib_primitives_tests
    '--gtest_filter=stdlib_bigfield/1.zellicassertnotequal'
Running main() from
    /home/user/aztec-packages/barretenberg/cpp/build/_deps/gtest-
    src/googletest/src/gtest_main.cc
Note: Google Test filter = stdlib_bigfield/1.zellicassertnotequal
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from stdlib_bigfield/1, where TypeParam
    = bb::UltraCircuitBuilder_<bb::UltraArith<bb::field<bb::Bn254FrParams> > >
[ RUN ] stdlib_bigfield/1.zellicassertnotequal

In Zellic assert_is_not_equal!

zero:
Limb 0: { 0x0000000000000000000000000000000000000000000000000000000000000000 <
    0x0000000000000000000000000000000000000000000000000000000000000000 }
Limb 1: { 0x0000000000000000000000000000000000000000000000000000000000000000 <
    0x0000000000000000000000000000000000000000000000000000000000000000 }
Limb 2: { 0x0000000000000000000000000000000000000000000000000000000000000000 <
    0x0000000000000000000000000000000000000000000000000000000000000000 }
Limb 3: { 0x0000000000000000000000000000000000000000000000000000000000000000 <
    0x0000000000000000000000000000000000000000000000000000000000000000 }
Prime limb: 0x0000000000000000000000000000000000000000000000000000000000000000

alsozero:
Limb 0: { 0x00000000000000000000000000000000000000000000000d3c208c16d87cfd47 <
    0x00000000000000000000000000000000000000000000000d3c208c16d87cfd47 }
```

```
Limb 1: { 0x000000000000000000000000000000000000000000000005d97816a916871ca8 <
    0x000000000000000000000000000000000000000000000005d97816a916871ca8 }
Limb 2: { 0x00000000000000000000000000000000000000000000000029b85045b6818158 <
    0x00000000000000000000000000000000000000000000000029b85045b6818158 }
Limb 3: { 0x000000000000000000000000000000000000000000000000000030644e72e131a <
    0x000000000000000000000000000000000000000000000000000030644e72e131a }
Prime limb: 0x000000000000000000000000000000006f4d8248eeb859fbf83e9682e87cfd46
[ OK ] stdlib_bigfield/1.zellicassertnotequal (17 ms)
[----------] 1 test from stdlib_bigfield/1 (17 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (17 ms total)
[ PASSED ] 1 test.
```

Impact

Malicious provers can prove that `bigfield` elements are unequal even though they are not, under certain conditions. The condition that is required for this to be the case is that the maximum value for one of the elements has to be a multiple of the emulated prime modulus. As the maximum values are determined on circuit generation and cannot be changed by the malicious prover, this case is unlikely to occur in practice.

Recommendations

Change the condition in the loop from < to <=.

```
while (target < maximum_value) {
while (target <= maximum_value) {
    ++overload_count;
    target += target_modulus;
}
```

Remediation

This issue has been acknowledged by Aztec, and a fix was implemented in commit 763d5eb3 ↗.

3.8. Equation checks not enforced

Target	field.cpp		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Low

Description

The function `evaluate_linear_identity` is used to check the linear identity $a + b + c + d = 0$. In the `bigfield` constructor, this function is used to constrain the construction of the low bits of the argument to be equal to `limb_0 + limb_1 << 68`. The function is used similarly for the highest bits of the argument. However, in case all the arguments of `evaluate_linear_identity` are marked as constant, the check is not enforced:

```
void field_t<Builder>::evaluate_linear_identity(const field_t& a,
    const field_t& b, const field_t& c, const field_t& d)
{
    Builder* ctx = a.context == nullptr
                       ? (b.context == nullptr ? (c.context == nullptr ?
    d.context : c.context)
    : b.context)
                       : a.context;

    if (a.witness_index == IS_CONSTANT && b.witness_index == IS_CONSTANT
    && c.witness_index == IS_CONSTANT &&
        d.witness_index == IS_CONSTANT) {
        return;
    }
```

In fact, nothing further is done, and the function returns.

The problem is also present in the function `evaluate_polynomial_identity`, but in this case, the relation to be checked is $a \cdot b + c + d = 0$. Again, if the arguments are constants, then the function simply returns.

Impact

The previous identity could be wrong for some constants and still pass the check. In the `bigfield` component, the two functions are likely only called in branches in which at least one of the arguments will not be constant, so this case is not hit. Impact for other callers depends on usage.

Recommendations

In case the arguments are constant, assert the equality of the identity for the values.

Remediation

This issue has been acknowledged by Aztec, and a fix was implemented in commit `a9ffba18` ↗.

3.9. Large limbs for constant inputs to `conditional_negate`

Target	bigfield_impl.hpp		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Low

Description

The function `bigfield::conditional_negate` is implemented as follows for constant arguments:

```
template <typename Builder, typename T>
bigfield<Builder, T> bigfield<Builder,
    T>::conditional_negate(const bool_t<Builder>& predicate) const
{
    Builder* ctx = context ? context : predicate.context;

    if (is_constant() && predicate.is_constant()) {
        if (predicate.get_value()) {
            uint512_t out_val = (modulus_u512 - get_value()) % modulus_u512;
            return bigfield(ctx, out_val.lo);
        }
        return *this;
    }

    // ...
}
```

The calculation of the return value will involve an underflow and wraparound modulo the native circuit prime modulus r if `get_value()` returns an unsigned integer that is bigger than the emulated prime modulus p. This would make the result incorrect.

Impact

Should `conditional_negate` be used for constant arguments, with the value to be negated being bigger than p, then the return value will be incorrect. Most operations for constant `bigfield` elements will reduce modulo p, so their results will not have vulnerable form. It is possible to construct such constants with the usual `bigfield` constructors, however.

Recommendations

Reduce `get_value()` modulo `modulus_u512` before subtracting it.

Remediation

This issue has been acknowledged by Aztec. Commit `718d5b7c` ↗ fixes the issue by asserting that `get_value() < modulus_u512`.

3.10. Incomplete constant check

Target	bigfield.hpp		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The function `is_constant` is implemented as follows:

```
bool is_constant() const {
    return prime_basis_limb.witness_index == IS_CONSTANT; }
```

It only checks whether the prime limb is constant, but no check is done on the binary limbs. This causes issues in some of the functions using this if one were to call them with a `bigfield` element that has a nonconstant binary limb but constant prime limb. There is a constructor that can be used to construct such `bigfield`, so this situation can occur with the current implementation of the `bigfield` component.

For example, in `reduction_check`, there would be a problem if only the prime limb is constant, as it will replace the current `bigfield` element with one that is constant, with a value taken from the binary limbs, even if they are not constant.

In the `self_reduce` function, there is already a comment about handling the situation where some limbs are constant and others are not, which may be in reference to the issue under discussion:

```
template <typename Builder, typename T> void bigfield<Builder,
    T>::self_reduce() const
{
    // Warning: this assumes we have run circuit construction at least once in
    debug mode where large non reduced
    // constants are disallowed via ASSERT
    if (is_constant()) {
        return;
    }
    // TODO: handle situation where some limbs are constant and others are not
    constant
    const auto [quotient_value, remainder_value]
    = get_value().divmod(target_basis.modulus);
```

Impact

This problem leads to a completeness issue. One concrete example where a user might construct a `bigfield` with a constant prime limb and not-constant binary limbs might be if the user wants to have a value modulo p that is either 0 or r. Then, the prime limb would always be 0, so it could be constant. In that case, running the circuit to generate the proving key would bake in one particular value as a constant, so proving the circuit with the other choice would later fail.

Recommendations

There are two options. If `bigfield` elements in which some, but not all, limbs are constant should be supported, then the `is_constant` function should only return `true` if all five limbs are constant. If this is not to be supported, then the constructors should ensure that if the prime limb is constant, then the binary limbs are constant as well.

Remediation

This issue has been acknowledged by Aztec, and a fix was implemented in commit `3fdd8022` ↗. The remediation includes a check in the constructors to ensure that if the prime limb is constant or any binary limbs is constant then all other limbs must be constant.

3.11. Null-pointer dereference

Target	bigfield_impl.hpp		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

In many functions, for example in `unsafe_evaluate_multiply_add`, the context is assigned from one of the arguments similarly to the following snippet:

```
Builder* ctx = left.context ? left.context : to_mul.context;
```

The pointer `ctx` is then used later, sometimes only in some branches. When both `left` and `to_mul` are constants, it can happen that both their contexts are the null pointer, leading to issues should the pointer be dereferenced later. In many cases, this should not occur normally, as only constant `bigfield` elements should have a null-pointer context, and, for example, the method `operator*` would not call `unsafe_evaluate_multiply_add` if both factors are constant.

However, there are also cases in which a null-pointer dereference can occur. One example is the function `msub_div`, an extract of whose implementation we show below:

```
template <typename Builder, typename T>
bigfield<Builder, T> bigfield<Builder,
    T>::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)
{
    Builder* ctx = divisor.context;

    // ...

    bool products_constant = true;

    // ...

    // Compute the sum of products
    for (size_t i = 0; i < num_multiplications; ++i) {
        const native mul_left_native(uint512_t(mul_left[i].get_value()
    % modulus_u512).lo);
        const native mul_right_native(uint512_t(mul_right[i].get_value()
    % modulus_u512).lo);
        product_native += (mul_left_native * -mul_right_native);
        products_constant = products_constant && mul_left[i].is_constant()
    && mul_right[i].is_constant();
    }

    // Compute the sum of to_sub
    native sub_native(0);
    bool sub_constant = true;
    for (const auto& sub : to_sub) {
        sub_native += (uint512_t(sub.get_value() % modulus_u512).lo);
        sub_constant = sub_constant && sub.is_constant();
    }

    // ...

    // If everything is constant, then we just return the constant
    if (sub_constant && products_constant && divisor.is_constant()) {
        return bigfield(ctx, uint256_t(result_value.lo.lo));
    }

    // Create the result witness
    bigfield result = create_from_u512_as_witness(ctx, result_value.lo);

    // ...
}
```

As can be seen here, the `ctx` pointer is taken directly from `divisor.context`. Thus, if `divisor` is a constant, this could be the null pointer. There then is a separate branch that returns early, for the case of constant arguments. But it is only taken if all the arguments are constants, so this will not be used if `divisor` is a constant but one of the summands or factors is not. In that case, `create_from_u512_as_witness` will be called with a null pointer as `ctx`. However, in the `HasPlookup<Builder>` case, that function will dereference `ctx`:

```
template <typename Builder, typename T>
bigfield<Builder, T> bigfield<Builder,
    T>::create_from_u512_as_witness(Builder* ctx,
                                                                       const
    uint512_t& value,
                                                                       const
    bool can_overflow,
                                                                       const
    size_t maximum_bitlength)
{
    // ...
    if constexpr (HasPlookup<Builder>) {
        // ...
        ctx->create_big_add_gate({ limb_1.witness_index,
                                   limb_2.witness_index,
                                   limb_3.witness_index,
                                   prime_limb.witness_index,
                                   shift_1,
                                   shift_2,
                                   shift_3,
                                   -1,
                                   0 },
                                 true);

         // ...

    } else {
        return bigfield(witness_t(ctx, fr(limbs[0] + limbs[1] * shift_1)),
                        witness_t(ctx, fr(limbs[2] + limbs[3] * shift_1)),
                        can_overflow,
                        maximum_bitlength);
    }
}
```

Thus, calling `msub_div` with a constant divisor and at least one other argument nonconstant will cause a null-pointer dereference in `create_from_u512_as_witness` (at least in the `HasPlookup<Builder>` case), leading to undefined behavior and most likely a crash.

The following test case demonstrates the crash:

```
static void test_zellic_ctx_crash()
{
    auto builder = Builder();
    fq_ct witness_one = fq_ct::create_from_u512_as_witness(&builder,
    uint256_t(1));
    fq_ct constant_one(1);


    info("\nwitness_one:");
    for (size_t limb_index = 0; limb_index < 4 ; limb_index++) {
        info("Limb ", limb_index, ": ",
    witness_one.binary_basis_limbs[limb_index]);
    }
    info("Prime limb: ", witness_one.prime_basis_limb);
    info("Context: ", witness_one.context);
    info("is_constant: ", witness_one.is_constant());
    info("\nconstant_one:");
    for (size_t limb_index = 0; limb_index < 4 ; limb_index++) {
        info("Limb ", limb_index, ": ",
    constant_one.binary_basis_limbs[limb_index]);
    }
    info("Prime limb: ", constant_one.prime_basis_limb);
    info("Context: ", constant_one.context);
    info("is_constant: ", constant_one.is_constant());

    fq_ct::msub_div({witness_one}, {witness_one}, constant_one, {witness_one},
    true);

    bool result = CircuitChecker::check(builder);
    EXPECT_EQ(result, true);
}
```

When the call to `msub_div` is replaced by

```
fq_ct::msub_div({witness_one}, {witness_one}, witness_one, {witness_one},
    true);
```

or

```
fq_ct::msub_div({constant_one}, {constant_one}, constant_one, {constant_one},
    true);
```

the test case succeeds as expected. However, the version above, where the divisor is constant but the summand and factors are not, crashes:

```
% ./build/bin/stdlib_primitives_tests
    '--gtest_filter=stdlib_bigfield/1.zellicctxcrash'
Running main() from
    /home/user/aztec-packages/barretenberg/cpp/build/_deps/gtest-
    src/googletest/src/gtest_main.cc
Note: Google Test filter = stdlib_bigfield/1.zellicctxcrash
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from stdlib_bigfield/1, where TypeParam
    = bb::UltraCircuitBuilder_<bb::UltraArith<bb::field<bb::Bn254FrParams> > >
[ RUN ] stdlib_bigfield/1.zellicctxcrash

In create_from_u512_as_witness, ctx=0x7ffe9694d7b0

witness_one:
Limb 0:  { 0x0000000000000000000000000000000000000000000000000000000000000001 <
```

```
    0x00000000000000000000000000000000000000000000000fffffffffffffffff }
Limb 1: { 0x0000000000000000000000000000000000000000000000000000000000000000 <
    0x00000000000000000000000000000000000000000000000fffffffffffffffff }
Limb 2: { 0x0000000000000000000000000000000000000000000000000000000000000000 <
    0x00000000000000000000000000000000000000000000000fffffffffffffffff }
Limb 3: { 0x0000000000000000000000000000000000000000000000000000000000000000 <
    0x0000000000000000000000000000000000000000000000000003ffffffffffff }
Prime limb: 0x0000000000000000000000000000000000000000000000000000000000000001
Context: 0x7ffe9694d7b0
is_constant: 0

constant_one:
Limb 0: { 0x0000000000000000000000000000000000000000000000000000000000000001 <
    0x0000000000000000000000000000000000000000000000000000000000000002 }
Limb 1: { 0x0000000000000000000000000000000000000000000000000000000000000000 <
    0x0000000000000000000000000000000000000000000000000000000000000001 }
Limb 2: { 0x0000000000000000000000000000000000000000000000000000000000000000 <
    0x0000000000000000000000000000000000000000000000000000000000000001 }
Limb 3: { 0x0000000000000000000000000000000000000000000000000000000000000000 <
    0x0000000000000000000000000000000000000000000000000000000000000001 }
Prime limb: 0x0000000000000000000000000000000000000000000000000000000000000001
Context: 0
is_constant: 1
In create_from_u512_as_witness, ctx=0
zsh: segmentation fault ./build/bin/stdlib_primitives_tests
```

Impact

Dereferencing a null pointer is undefined behavior, so it can result in incorrect computation or most likely a crash.

Recommendations

In the case of `msub_div` and other functions with the analogous problem, we recommend to take the context passed as an argument to other functions from an argument that has a non-null-pointer context, similarly to how it is done in `unsafe_evaluate_multiply_add` or `internal_div`. We have not done an exhaustive search for functions with this problem, so we recommend to review the code for cases analogous to `msub_div`.

In general, we recommend to assert that the context pointer is not the null pointer before using it, to prevent undefined behavior and obtain an error that is easier to debug.

Remediation

This issue has been acknowledged by Aztec and the commit 573d2982 ↗ fixes the problem in the `msub_div` function.

3.12. Handling of `max` argument to `Limb` constructor

Target	bigfield.hpp		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `Limb` struct is a wrapper around a circuit variable that also tracks that variable's maximum possible value with an out-of-circuit `uint256_t`. The constructor is implemented as follows:

```
Limb(const field_t<Builder>& input, const uint256_t max = uint256_t(0))
    : element(input)
{
    if (input.witness_index == IS_CONSTANT) {
        maximum_value = uint256_t(input.additive_constant) + 1;
    } else if (max != uint256_t(0)) {
        maximum_value = max;
    } else {
        maximum_value = DEFAULT_MAXIMUM_LIMB;
    }
}
```

The default value for the argument `max` is zero, and if `max` is zero, then `maximum_value` is set to `DEFAULT_MAXIMUM_LIMB`, which is a different value.

Thus, a caller that passes zero for `max` as the actual intended maximum value will have this choice overriden, with `maximum_value` set to the much larger `DEFAULT_MAXIMUM_LIMB` instead. This kind of call actually appears in the codebase, for example in the `bigfield<Builder, T>::self_reduce()` function in `cpp/src/barretenberg/stdlib/primitives/bigfield/bigfield_impl.hpp`:

```
// TODO: implicit assumption here - NUM_LIMB_BITS large enough for all the
    quotient
quotient.binary_basis_limbs[0] = Limb(quotient_limb, uint256_t(1)
    << maximum_quotient_bits);
quotient.binary_basis_limbs[1]
    = Limb(field_t<Builder>::from_witness_index(context, context->zero_idx),
    0);
quotient.binary_basis_limbs[2]
    = Limb(field_t<Builder>::from_witness_index(context, context->zero_idx),
    0);
```

```
quotient.binary_basis_limbs[3]
    = Limb(field_t<Builder>::from_witness_index(context, context->zero_idx),
    0);
```

Additionally, the `max` argument is ignored if `input` is a constant. In that case, it would be expected behavior that the function asserts that the constant satisfies the passed maximum and either uses the actual constant value as `maximum_value` or the passed `max`. Actual behavior of the constructor is not documented.

Impact

The unnecessarily large `maximum_value` for limbs that the caller has ensured are zero can cause unnecessary reductions on later usage.

Should a caller pass a calculated constant to this constructor along with an incorrectly estimated maximum, expecting the constructor to check this bound, unintended behavior can arise, depending on the caller code.

Recommendations

Change the default value of `max` to `DEFAULT_MAXIMUM_LIMB`, and then do not treat any special value of `max` different than the others. This will ensure that a caller passing a value for `max` will obtain the intended behavior.

Add an assert to check that `input.additive_constant <= max` in the case of a constant `input`.

Remediation

This issue has been acknowledged by Aztec, and a fix was implemented in commit `0ecedcc2` ↗.

3.13. Mistakes in calculation of `MAX_UNREDUCED_LIMB_SIZE`

Target	bigfield.hpp		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The maximum unreduced limb size is computed as follows:

```
// a (currently generous) upper bound on the log of number of fr additions in
    any of the class operations
static constexpr uint64_t MAX_ADDITION_LOG = 10;
// the rationale of the expression is we should not overflow Fr when applying
    any bigfield operation (e.g. *) and
// starting with this max limb size
static constexpr uint64_t MAX_UNREDUCED_LIMB_SIZE = (bb::fr::modulus.get_msb()
    + 1) / 2 - MAX_ADDITION_LOG;

static constexpr uint256_t get_maximum_unreduced_limb_value() {
    return uint256_t(1) << MAX_UNREDUCED_LIMB_SIZE; }
```

Let r be the native prime modulus of the circuit, p the emulated modulus (so `bb::fr::modulus`) and `b = bb::fr::modulus.get_msb()`.

The intention here seems to be as follows. What is needed is a bound m so that $\sum_{i=1}^{k} a_i \cdot b_i < p$ as long as $k \leq 2^{10}$ and $a_i, b_i \leq m$.

Let us first consider the calculation of m (as `get_maximum_unreduced_limb_value()`) in the code in the case where the bound for k were $k \leq 1$, so we only consider a single summand. This corresponds to `MAX_ADDITION_LOG = 0`. Then we would have $m = 2^{\lfloor \frac{b+1}{2} \rfloor}$. From the definition of b, it is the most significant bit of p that is set, so we have $2^b < p < 2^{b+1}$. (The reason we have $2^b < p$ and not just $2^b \leq p$ is that p is an odd prime.) If we assume that b is odd, then we would have $m^2 = 2^{b+1} > p$. Thus, the intended property of m would not be satisfied. The issue is the addition of one; if we took $m = 2^{\lfloor \frac{b}{2} \rfloor}$ instead, then we would have $m^2 \leq 2^b < p$.

However, there appears to be another mistake, this time making the bound less sharp, which overcompensates for the just discussed one. Concretely, suppose m_1 is a correct bound as above for the one-summand case, so that $m_1^2 < p$. Then for the case of up to 2^{10} summands, the code uses $m = m_1 \cdot 2^{-10}$. Then we get that $\sum_{i=1}^{k} a_i$ cot $b_i \leq 2^{10} \cdot m_1^2 \cdot 2^{-10 \cdot 2} < 2^{-10} \cdot p$, so m is smaller than necessary. Instead of reducing the bit width of both factors by 10 bits, only the products need to be reduced in bit width by 10 bits, for which it suffices to reduce the bit width of each factor by five bits, so in MAX_UNREDUCED_LIMB_SIZE, the contribution of MAX_ADDITION_LOG could be divided by two as well.

Impact

The mistake reducing sharpness overcorrects for the other mistake, so there is no impact currently. However, there is a risk that making the MAX_ADDITION_LOG contribution sharp at a later time without correcting the other mistake will cause a bound that violates the required assumption.

Recommendations

We recommend to replace

```
static constexpr uint64_t MAX_UNREDUCED_LIMB_SIZE = (bb::fr::modulus.get_msb()
    + 1) / 2 - MAX_ADDITION_LOG;
```

by

```
static constexpr uint64_t MAX_UNREDUCED_LIMB_SIZE = (bb::fr::modulus.get_msb()
    - MAX_ADDITION_LOG) / 2;
```

Remediation

This issue has been acknowledged by Aztec.

3.14. Behavior of `assert_equal` for constant operands

Target	bigfield_impl.hpp		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The function `bigfield::assert_equal` is intended to constrain `this` to be equal to the argument `other`. This function exhibits unexpected behavior if one or both of the operands are constant.

If both are constant, the implementation is as follows:

```
if (is_constant() && other.is_constant()) {
    std::cerr << "bigfield: calling assert equal on 2 CONSTANT bigfield
    elements...is this intended?"
              << std::endl;
    return;
}
```

Thus, in this case, an error message will be printed. However, execution will otherwise proceed. In particular, it is not checked that the two constants are actually equal. Expected behavior would be to assert that the two values are equal and thus stop execution should this assertion fail.

If exactly one of the two operands is constant, the following snippet is the relevant implementation:

```
else if (other.is_constant()) {
    // TODO(https://github.com/AztecProtocol/barretenberg/issues/998):
    Something is fishy here
    // evaluate a strict equality - make sure *this is reduced first, or an
    honest prover
    // might not be able to satisfy these constraints.
    field_t<Builder> t0
    = (binary_basis_limbs[0].element - other.binary_basis_limbs[0].element);
    field_t<Builder> t1
    = (binary_basis_limbs[1].element - other.binary_basis_limbs[1].element);
    field_t<Builder> t2
    = (binary_basis_limbs[2].element - other.binary_basis_limbs[2].element);
    field_t<Builder> t3
    = (binary_basis_limbs[3].element - other.binary_basis_limbs[3].element);
    field_t<Builder> t4 = (prime_basis_limb - other.prime_basis_limb);
```

```
        t0.assert_is_zero();
        t1.assert_is_zero();
        t2.assert_is_zero();
        t3.assert_is_zero();
        t4.assert_is_zero();
        return;
    } else if (is_constant()) {
        other.assert_equal(*this);
        return;
    }
```

The code constrains that the constant and nonconstant operands agree in all binary limbs and the prime limb. As the TODO comment indicates, there is a completeness issue here, as there may be multiple representations possible for `bigfield` elements for the same element modulo the emulated modulus p.

Impact

Against expectation, the function does not actually assert equality if both operands are constants, which is a soundness issue. The error message printed no matter whether the equality holds or not, however, suggests that usage of `assert_equal` is not intended in case both operands are constant, so such usage is to be considered user error anyway.

If exactly one of the operands is constant, then there is a completeness issue for unreduced operands. This behavior is documented in the code already, however.

Recommendations

Assert equality of the values of the operands in the both-constant case. In the case of exactly one operand being constant, document that they must be reduced when using the function, or carry out a reduction in the function prior to the checks.

Remediation

This issue has been acknowledged by Aztec, and a fix was implemented in commit `b3e01f44` ↗.

3.15. Assert for `add_to_lower_limb` could be ineffective due to overflow

Target	bigfield_impl.hpp		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The function `add_to_lower_limb` starts like this:

```
/**
 * @brief Add a field element to the lower limb. CAUTION (the element has to be
   constrained before using this function)
 *
 * @details Sometimes we need to add a small constrained value to a bigfield
   element (for example, a boolean value), but
 * we don't want to construct a full bigfield element for that as it would take
   too many gates. If the maximum value of
 * the field element being added is small enough, we can simply add it to the
   lowest limb and increase its maximum
 * value. That will create 2 additional constraints instead of 5/3 needed to
   add 2 bigfield elements and several needed
 * to construct a bigfield element.
 *
 * @tparam Builder Builder
 * @tparam T Field Parameters
 * @param other Field element that will be added to the lower
 * @param other_maximum_value The maximum value of other
 * @return bigfield<Builder, T> Result
 */
template <typename Builder, typename T>
bigfield<Builder, T> bigfield<Builder,
    T>::add_to_lower_limb(const field_t<Builder>& other,
                                                                  uint256_t
    other_maximum_value)
    const
{
    reduction_check();
    ASSERT((other_maximum_value + binary_basis_limbs[0].maximum_value)
    <= get_maximum_unreduced_limb_value());
```

What is needed for the function to work properly is that other_maximum_value + binary_basis_limbs[0].maximum_value must be smaller or equal to get_maximum_unreduced_limb_value(). The ASSERT in the snippet above checks this under the assumption that the sum on the left-hand side does not wrap around (so if the sum is not 1<<256 or larger). While the documenting comments for add_to_lower_limb suggest that the function is needed for situations in which only a small value such as a boolean is added, the function does not prevent usage with values for other_maximum_value that are close to the maximum value for 256-bit unsigned integers. In that case, the overflow could actually happen.

To ensure an overflow does not happen, it could additionally be asserted that both other_maximum_value and binary_basis_limbs[0].maximum_value are less than or equal to get_maximum_unreduced_limb_value(). The latter of these two checks is already enforced by reduction_check(), so it does not need to be repeated.

However, note that the reduction check does not actually ensure that the crucial assert will hold. While it will ensure that binary_basis_limbs[0].maximum_value <= get_maximum_unreduced_limb_value(), it could happen that binary_basis_limbs[0].maximum_value = get_maximum_unreduced_limb_value(), in which case even a low value as other_maximum_value=1 will fail the assertion after the reduction check in add_to_lower_limb.

Impact

Under the unlikely case that a caller uses add_to_lower_limb with a very large value for other_maximum_value so that other_maximum_value + binary_basis_limbs[0].maximum_value >= 1<<256, the function would produce the wrong result.

Furthermore, it is possible that the assert fails even with benign inputs.

Recommendations

To prevent the edge case of a too large other_maximum_value, we recommend to add an assert such as

```
template <typename Builder, typename T>
bigfield<Builder, T> bigfield<Builder,
    T>::add_to_lower_limb(const field_t<Builder>& other,
                                                              uint256_t
    other_maximum_value)
    const
{
    reduction_check();
    ASSERT((other_maximum_value + binary_basis_limbs[0].maximum_value)
    <= get_maximum_unreduced_limb_value());
    ASSERT(other_maximum_value <= get_maximum_unreduced_limb_value());
```

Alternatively, perform the existing assert with 512-bit unsigned integers, to ensure a wraparound cannot happen.

To ensure that the main assert will not fail with benign inputs, one might consider doing something like in the following pseudocode:

```
// First check that the sum won't overflow in Fr
ASSERT((uint512_t)other_maximum_value
  + (uint512_t)binary_basis_limbs[0].maximum_value <
  (uint512_t)bb::fr::modulus);
// calculate result
// ...
// reduce result if needed
result.reduction_check();
```

In this case, the function will only fail if one of the inputs is very unusually large so that the first assert fails. The reduction check on the result at the end will ensure that the lowest-resulting limb will be at most `get_maximum_unreduced_limb_value()`.

Remediation

This issue has been acknowledged by Aztec, and a fix was implemented in commit `41a66f0a` ↗.

3.16. Missing limb maximum-value check

Target	bigfield_impl.hpp		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `self_reduce` function reduces a `bigfield` element to a representation of at most the bit width as the emulated modulus p. An assumption is made that a value `quotient_limb` that is calculated has at most `NUM_LIMB_BITS` bits, as is also mentioned in the following comment:

```
// TODO: implicit assumption here - NUM_LIMB_BITS large enough for all the
    quotient
```

However, it is not checked whether this assumption holds.

Impact

There does not seem to be direct impact, as the relevant limb is constructed with maximum value taken from the actual maximum value for the quotient, and it is separately checked that that size is small enough to not cause problems with an overflow of the CRT modulus in `unsafe_evaluate_multiply_add`.

Recommendations

Instead of making an implicit assumption, we recommend asserting it:

```
template <typename Builder, typename T> void bigfield<Builder,
    T>::self_reduce() const
{
    // Warning: this assumes we have run circuit construction at least once in
    debug mode where large non reduced
    // constants are disallowed via ASSERT
    if (is_constant()) {
        return;
    }
    // TODO: handle situation where some limbs are constant and others are not
    constant
    const auto [quotient_value, remainder_value]
    = get_value().divmod(target_basis.modulus);

    bigfield quotient(context);

    uint512_t maximum_quotient_size = get_maximum_value()
    / target_basis.modulus;
    uint64_t maximum_quotient_bits = maximum_quotient_size.get_msb() + 1;
    if ((maximum_quotient_bits & 1ULL) == 1ULL) {
        ++maximum_quotient_bits;
    }
   // TODO: implicit assumption here - NUM_LIMB_BITS large enough for all
       the quotient
   ASSERT(maximum_quotient_bits <= NUM_LIMB_BITS)
    uint32_t quotient_limb_index
    = context->add_variable(bb::fr(quotient_value.lo));
```

Remediation

This issue has been acknowledged by Aztec, and a fix was implemented in commit 6d3556c5 ↗.

3.17. Equality comparison for null-pointer context

Target	bigfield_impl.hpp		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `bigfield::operator==` function should constrain and return an in-circuit boolean indicating whether the two arguments are equal or not. The implementation of the function begins as follows:

```
template <typename Builder, typename T> bool_t<Builder> bigfield<Builder,
    T>::operator==(const bigfield& other) const
{
    Builder* ctx = context ? context : other.get_context();
    auto lhs = get_value() % modulus_u512;
    auto rhs = other.get_value() % modulus_u512;
    bool is_equal_raw = (lhs == rhs);
    if (!ctx) {
        // TODO(https://github.com/AztecProtocol/barretenberg/issues/660):
    null context _should_ mean that both are
        // constant, but we check with an assertion to be sure.
        ASSERT(is_constant() == other.is_constant());
        return is_equal_raw;
    }
```

Should both operands have null pointers as context, their values are compared directly and a constant with that boolean value is returned. This is only correct should both operands be constants. The comment points this out and mentions that while null contexts should only happen for constants, to be sure, it should be asserted that both operands are constant. However, the actual assert does not check this, but instead only checks that one operand is constant if and only if the other is. Thus, the case that both operands are not constants but still have a null pointer as context is not ruled out.

Impact

The assert intended to prevent this function being called with operands that have null-pointer contexts without being constants does not work as intended, so this situation is not prevented. However, this assert is intended as a defense-in-depth measure, so direct impact is unlikely.

Recommendations

Replace the assert with one that checks that both operands are constant:

```
ASSERT(is_constant() == other.is_constant());
ASSERT(is_constant() && other.is_constant());
```

Remediation

This issue has been acknowledged by Aztec, and a fix was implemented in commit c60705dd ↗.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Maturity of the codebase and lack of specifications

The reviewed code of the `bigfield` component is a low-level component handling arithmetic for non-native fields. The manner in which non-native fields are emulated by `bigfield` hinge crucially on careful bounds on the data operated on to prevent issues due to overflows and underflows in the bn254 scalar field or other arithmetic types. Ensuring that no overflows or underflows happen requires a careful analysis tracing such bounds through the code.

Given this, we strongly recommend to specify assumptions made for the arguments to each function, together with what that function should then enforce. Additionally, in places where it is not obvious that overflows can not occur or a call to another function satisfies its requirements, we recommend to add comments outlining the reasoning why the assumptions imply that these issues can not occur.

This will help prevent bugs already during development and also make reviewing the code for correctness significantly easier. Fully documenting the interface of the implemented functions that are to be used by callers outside of the `bigfield` component itself is also important to ensure correct usage by external callers.

The current codebase largely lacks such specifications, and we identified several bugs that might have been prevented by consistently specifying intended behavior as suggested, such as the critical findings in sections 3.1. ↗, 3.3. ↗, and 3.2. ↗.

We strongly recommend to add specifications and comments as described above to the `bigfield` codebase before a possible second audit.

4.2. Concrete examples for assumptions for bounds and how to document them

There are many implicit assumptions throughout the codebase regarding bounds that are undocumented but required for correct functioning. We have generally only mentioned those instances in this report, which plausibly could cause a problem with likely usage, without a more serious issue occurring as well. In this section, in expansion of what was discussed in 4.1. ↗, we discuss two concrete representative examples of the kind of implicit assumptions scattered throughout the codebase. For the second example, we discuss how one might specify and document the code with the assumptions that the caller should ensure.

Example for common implicit assumptions on bounds

The `unsafe_evaluate_multiple_multiply_add` contains the following calculations, done with types that are 512-bits wide:

```
max_d0 += (neg_modulus_limbs_u256[3]
    * quotient.binary_basis_limbs[0].maximum_value);
uint512_t max_d1 = (left.binary_basis_limbs[2].maximum_value *
    to_mul.binary_basis_limbs[1].maximum_value);
max_d1 += (neg_modulus_limbs_u256[2]
    * quotient.binary_basis_limbs[1].maximum_value);
uint512_t max_d2 = (left.binary_basis_limbs[1].maximum_value *
    to_mul.binary_basis_limbs[2].maximum_value);
max_d2 += (neg_modulus_limbs_u256[1]
    * quotient.binary_basis_limbs[2].maximum_value);
uint512_t max_d3 = (left.binary_basis_limbs[0].maximum_value *
    to_mul.binary_basis_limbs[3].maximum_value);
max_d3 += (neg_modulus_limbs_u256[0]
    * quotient.binary_basis_limbs[3].maximum_value);

uint512_t max_r0 = left.binary_basis_limbs[0].maximum_value *
    to_mul.binary_basis_limbs[0].maximum_value;
max_r0 += (neg_modulus_limbs_u256[0]
    * quotient.binary_basis_limbs[0].maximum_value);

// @AUDIT: these are the maximums of the convolution product limbs of
    left*to_mul + (-p)*quotient
const uint512_t max_r1 = max_b0 + max_b1;
const uint512_t max_r2 = max_c0 + max_c1 + max_c2;
const uint512_t max_r3 = max_d0 + max_d1 + max_d2 + max_d3;
```

Each of `max_d0`, `max_d1`, `max_d2`, and `max_d3` contain a summand that is the product of maximum values of limbs of `left` and `to_mul`. Those maximum values are stored as 256-bit unsigned integers. Their products could thus reach up to 512 bits in width, so four summands of that size could overflow when performing the addition in 512 bits. The function has no documentation detailing which bounds the caller needs to ensure.

The just discussed overflow could only happen in much more extreme and unlikely situations than those in Finding 3.2. ↗. There are many similar pieces of code through the `bigfield` codebase of the same type, however.

Example for documenting bounds

We describe how one could specify assumptions for functions.

Consider the following example:

```
template <typename Builder, typename T>
uint512_t bigfield<Builder,
    T>::compute_maximum_quotient_value(const std::vector<uint512_t>& as,
    const std::vector<uint512_t>& bs, const std::vector<uint512_t>& to_add)
{
    ASSERT(as.size() == bs.size());
    uint512_t add_values(0);
    for (const auto& add_element : to_add) {
        add_values += add_element;
    }
    uint1024_t product_sum(0);
    for (size_t i = 0; i < as.size(); i++) {
        product_sum += uint1024_t(as[i]) * uint1024_t(bs[i]);
    }
    const uint1024_t add_right(add_values);
    const uint1024_t modulus(target_basis.modulus);

    const auto [quotient_1024, remainder_1024] = (product_sum
    + add_right).divmod(modulus);

    return quotient_1024.lo;
}
```

This function takes as arguments various unsigned 512-bit integers. It then computes component-wise products of `as` and `bs` and adds all components up, together with the components of `to_add`. Computation of the sum of components of `to_add` is done in 512 bits, and the rest of the computation is done with a maximum bit width of 1,024 bits.

Already the addition of two components of `to_add` could overflow in 512 bits, if at least one of them is 512 bits wide. So, to properly work, this function requires bounds on the components of `to_add`. The most permissive bounds ruling out overflows would be the following:

- `sum(to_add) < 2^512`
- `dot_product(as, bs) + sum(to_add) < 2^1024`

However, requiring these kind of conditions on the caller might not be very user-friendly. It would be easier for the caller to think about bounds for individual arguments, and it seems likely that those can indeed be assumed. For example, all values passed as arguments will likely be at most the maximum value that could possibly be represented by a `bigfield` element, obtained via `bigfield::get_maximum_value()`. This would be the value we obtain if `maximum_value` for all four binary limbs is the maximum value it can be, which is `2^256 - 1`.

Thus, we can estimate the maximum value that can be returned by `bigfield::get_maximum_value()` to be less than `2^256 * (1 + 2^68 + 2^(2*68) + 2^(3*68)) < 2^(256+3*68+1) = 2^461`. A sum of values that are at most 461 bits wide can only overflow with 512 bits available if there are at least `2^51` summands. As passing a vector with that many summands would require petabytes of memory, we can rule this out as happening in practice. So instead of asking the caller to ensure `sum(to_add) < 2^512` (which means having to make the kind of reasoning we just carried out each time one adds a new call to `compute_maximum_quotient_value` elsewhere in the code), it is likely less cumbersome to request that all components of `to_add` are at most 461 bits wide.

Similar reasoning can be carried out for the other relevant bounds for this function. Ultimately, we might end with the function documented (with respect to bounds, the function could also benefit from a documenting comment regarding what it is intended to do) as follows:

```
// Assumptions:
// 1. All components of `as`, `bs`, and `to_add` must be smaller than `1<<461`.
// Note that the above assumption will be satisfied for return values of
    `bigfield::get_maximum_value()`.
// Reason for that: The maximum value that can be returned is `(2^256 - 1) * (1
    + 2^68 + 2^(2*68) + 2^(3*68))`, which is less than `1<<461`.
// Promises:
// - The return value will be less than `1<<820`.
template <typename Builder, typename T>
uint512_t bigfield<Builder,
    T>::compute_maximum_quotient_value(const std::vector<uint512_t>& as,
    const std::vector<uint512_t>& bs, const std::vector<uint512_t>& to_add)
{
    ASSERT(as.size() == bs.size());
    // The below loop calculates the sum of the components of `to_add` in the
    `uint512_t` type.
    // An overflow would require at least `2^(512-461) = 2^51` summands, which
    we can reject as unrealistic.
    uint512_t add_values(0);
    for (const auto& add_element : to_add) {
        add_values += add_element;
    }
    // Each summand of `product_sum` will be less than `2^(2*461) = 2^922`.
    // Again, `2^100` or more summands are unrealistic, so this sum will not
    overflow, and will also stay less than `2^1023`.
    uint1024_t product_sum(0);
    for (size_t i = 0; i < as.size(); i++) {
        product_sum += uint1024_t(as[i]) * uint1024_t(bs[i]);
    }
    const uint1024_t add_right(add_values);
    const uint1024_t modulus(target_basis.modulus);

    // The sum `product_sum + add_right` does not overflow, as `add_right` can
    be at most 512 bits wide, and we argued that `product_sum < 2^1023`.
    const auto [quotient_1024, remainder_1024] = (product_sum
    + add_right).divmod(modulus);
    // `modulus` must be bigger than or equal to `2^(3*68) = 2^204` by general
    assumption for this class.
    // Thus, with the sum being divided being less than `2^1024`, we conclude
    that the quotient must be smaller than `2^(1024-204)=2^820`.
    return quotient_1024.lo;
}
```

The above is merely an example for how one might thoroughly document reasoning about bounds. Different bounds may be desired for this function.

4.3. Undocumented assumption regarding meaning of `bigfield` fields

Every `bigfield` consists of four binary limbs and a prime limb. The prime limb directly determines a value modulo the native circuit field modulus r. The binary limbs are interpreted as an integer similarly to four digits of an integer in 2^{68}-ary representation. Thus, if the limbs are l_0, l_1, l_2, l_3, then the value will be $\sum_{i=0}^{3} l_i \cdot 2^{i \cdot 68}$. However, unlike true 2^{68}-ary representation, the limbs can also be bigger than or equal to 2^{68}; the represented integer is still given by the above sum.

The documentation[3] contains passages like the following:

> So technically, if we could perform operations modulo some number m with $m > p^2$, then it would be enough to compute values modulo p. How do we get a modulus $m > p^2$, when we have just operations modulo p. Well, let's imagine for a second that we can implement operations modulo some number $z > p$, such that $z * n > p^2$ and z and n are coprime. Then we can use Chinese Remainder Theorem to implement operations modulo $z * n$. If you don't know how CRT works, you can read about it under spoiler.

In this quote, n refers to the native modulus of the circuit (which we refer to by r), and p refers to the emulated modulus. Given this documentation, the reader might think that it is possible to use `bigfield` instances in which the value is given by the CRT of the values modulo the prime and binary modulus $2^t = 2^{4 \cdot 68}$. For example, if one were to want to construct the value 2^t modulo p, it may be reasonable to think that one can do so by setting the binary limbs to zero and the prime limb to $2^t \mod r$. The CRT of those values would then be the intended 2^t. The quote also suggests that the integer represented by the binary limbs will be interpreted as representing a value modulo 2^t.

However, the code differs from this interpretation in two ways.

Firstly, the values stored in the limbs are not interpreted using a CRT. Instead, only the value obtained from the binary limbs is used as the value of the `bigfield` element. The prime limb is then assumed to be the reduction of that value modulo the native modulus. This can be seen, for example, in the often-used function `get_value`, in which the return value depends only on the binary limbs, with the prime limb not being used:

[3] An unpublished PDF document called *Biggroup and bigfield in Barretenberg* was provided prior to the audit.

```
template <typename Builder, typename T> uint512_t bigfield<Builder,
    T>::get_value() const
{
    uint512_t t0 = uint256_t(binary_basis_limbs[0].element.get_value());
    uint512_t t1 = uint256_t(binary_basis_limbs[1].element.get_value());
    uint512_t t2 = uint256_t(binary_basis_limbs[2].element.get_value());
    uint512_t t3 = uint256_t(binary_basis_limbs[3].element.get_value());
    return t0 + (t1 << (NUM_LIMB_BITS)) + (t2 << (2 * NUM_LIMB_BITS)) + (t3
    << (3 * NUM_LIMB_BITS));
}
```

The prime limb is thus treated as redundant information. In principle, the prime limb could thus be removed from the `bigfield` class and recomputed wherever it is needed internally in computations, which happens when checking multiplicative relations of `bigfield` elements. However, recomputation of the prime limb costs about two gates. If many multiplications are using the same factors, computing the prime limb only once instead of recomputing it each time can thus save gates. Furthermore, on some operations, such as addition, it only costs a single gate to compute the prime limb of the result from the prime limbs of the inputs. The purpose of the prime limb is thus not to store data but to essentially cache the value represented by the binary limbs modulo r as a performance improvement.

Secondly, the integer represented by the binary limbs is used as is and not interpreted as a congruence class modulo 2^t. If it were treated as a congruence class modulo 2^t, we would expect the smallest nonnegative representative to be used to convert to a number modulo p. In other words, if l_0, l_1, l_2, l_3 are the binary limbs, then the value modulo p represented by this would be expected to be $\left(\left(\sum_{i=0}^{3} l_i \cdot 2^{i \cdot 68}\right) \mod 2^t\right) \mod p$. However, the code actually interprets those limbs as representing $\left(\sum_{i=0}^{3} l_i \cdot 2^{i \cdot 68}\right) \mod p$.

As a concrete example, if we were to store the integer 2^t in the binary limbs, then the code will treat this as $2^t \mod p$ and not as $(2^t \mod 2^t) \mod p = 0 \mod p = 0$. This can also be seen with the following test case:

```
static void test_zellic_binary_modulus()
{
    size_t shift = 68;
    std::cout << "\nIn Zellic binary modulus test!" << std::endl;
    auto builder = Builder();
    fq_ct num(
        witness_ct(&builder, fr(uint256_t(0))),
        witness_ct(&builder, fr(uint256_t(0))),
        witness_ct(&builder, fr(uint256_t(0))),
        witness_ct(&builder, fr(uint256_t(1) << shift)),
        witness_ct(&builder, fr(uint256_t(0))),
        true
    );
    ASSERT(num.binary_basis_limbs[0].element.get_value() == 0);
    ASSERT(num.binary_basis_limbs[1].element.get_value() == 0);
    ASSERT(num.binary_basis_limbs[2].element.get_value() == 0);
    ASSERT(num.binary_basis_limbs[3].element.get_value()
    == uint256_t(1)<<shift);
    num.binary_basis_limbs[0].maximum_value = 0;
    num.binary_basis_limbs[1].maximum_value = 0;
    num.binary_basis_limbs[2].maximum_value = 0;
    num.binary_basis_limbs[3].maximum_value = uint256_t(1)<<shift;
    info("\nbefore reduction:");
    for (size_t limb_index = 0; limb_index < 4 ; limb_index++) {
        info("Limb ", limb_index, ": ", num.binary_basis_limbs[limb_index]);
    }
    num.self_reduce();
    info("\nafter reduction:");
    for (size_t limb_index = 0; limb_index < 4 ; limb_index++) {
        info("Limb ", limb_index, ": ", num.binary_basis_limbs[limb_index]);
    }
    bool result = CircuitChecker::check(builder);
    EXPECT_EQ(result, true);
}
```

With an interpretation of binary and prime limbs via CRT, setting them independently should be possible. And if they need to be consistent but the binary limbs are reduced modulo $2^t = 2^{4\cdot 68}$, then making the prime limb zero would be consistent with the binary limbs representing 2^t.

However, neither of those are the case, as can be seen by the fact that this test case will fail:

```
[ RUN ] stdlib_bigfield/1.zellicbinarymodulus

In Zellic binary modulus test!

before reduction:
Limb 0: { 0x0000000000000000000000000000000000000000000000000000000000000000 <
    0x0000000000000000000000000000000000000000000000000000000000000000 }
Limb 1: { 0x0000000000000000000000000000000000000000000000000000000000000000 <
    0x0000000000000000000000000000000000000000000000000000000000000000 }
Limb 2: { 0x0000000000000000000000000000000000000000000000000000000000000000 <
    0x0000000000000000000000000000000000000000000000000000000000000000 }
Limb 3: { 0x0000000000000000000000000000000000000000000000100000000000000000 <
    0x0000000000000000000000000000000000000000000000100000000000000000 }

after reduction:
Limb 0: { 0x0000000000000000000000000000000000000000000000002e0850a4e1bc3b4f <
    0x00000000000000000000000000000000000000000000000fffffffffffffffff }
Limb 1: { 0x00000000000000000000000000000000000000000000000b87d765f422f21d2d <
```

```
    0x00000000000000000000000000000000000000000000000fffffffffffffffff }
Limb 2: { 0x00000000000000000000000000000000000000000000000b955105616bf7c17a <
    0x00000000000000000000000000000000000000000000000fffffffffffffffff }
Limb 3: { 0x0000000000000000000000000000000000000000000000000000d42a313121fe <
    0x0000000000000000000000000000000000000000000000000003ffffffffffff }
Failed Arithmetic relation at row idx = 1744
Failed at block idx = 1
/home/user/aztec-
    packages/barretenberg/cpp/src/barretenberg/stdlib/primitives/bigfield/
bigfield.test.cpp:368: Failure
Expected equality of these values:
  result
    Which is: false
  true
[ FAILED ] stdlib_bigfield/1.zellicbinarymodulus, where TypeParam
    = bb::UltraCircuitBuilder_<bb::UltraArith<bb::field<bb::Bn254FrParams> > >
    (29 ms)
```

We recommend to clearly document the correct interpretation of the binary and prime limbs as used by the implementation of the `bigfield` class.

4.4. Functioning of `evaluate_non_native_field_multiplication`

We give a rough overview[4] of how the function `evaluate_non_native_field_multiplication` in `cpp/src/barretenberg/stdlib_circuit_builders/ultra_circuit_builder.cpp` is intended to work. Note that this function was out of scope for our audit, so we did not verify in detail whether the actual constraints imposed accurately reflect the intended specification for this function.

The function gets `a`, `b`, `q`, and `neg_modulus` (which we will call `-p` from now on) as arguments, each given with four binary limbs t_0, t_1, t_2, t_3 (there is also a prime limb, but it is not used in this function), with the value represented given by $t_0 + 2^{68} \cdot t_1 + 2^{2 \cdot 68} \cdot t_2 + 2^{3 \cdot 68} \cdot t_3$. The binary limbs can hold values wider than 68 bits, however.

The core computation of the function is summarized by the following snippet containing the out-of-circuit calculation of the witnesses, which are then constrained elsewhere in the function:

```
FF lo_0 = a[0] * b[0] - r[0] + (a[1] * b[0] + a[0] * b[1]) * LIMB_SHIFT;
FF lo_1 = (lo_0 + q[0] * input.neg_modulus[0] + (q[1] * input.neg_modulus[0]
    + q[0] * input.neg_modulus[1] - r[1]) * LIMB_SHIFT) * LIMB_RSHIFT_2;

FF hi_0 = a[2] * b[0] + a[0] * b[2] + (a[0] * b[3] + a[3] * b[0] - r[3])
```

[4] We are only considering the case `range_constrain_quotient_and_remainder=false`.

```
    * LIMB_SHIFT;
FF hi_1 = hi_0 + a[1] * b[1] - r[2] + (a[1] * b[2] + a[2] * b[1]) * LIMB_SHIFT;
FF hi_2 = (hi_1 + lo_1 + q[2] * input.neg_modulus[0] + (q[3]
    * input.neg_modulus[0] + q[2] * input.neg_modulus[1]) * LIMB_SHIFT);
FF hi_3 = (hi_2 + (q[0] * input.neg_modulus[3] + q[1] * input.neg_modulus[2])
    * LIMB_SHIFT + (q[0] * input.neg_modulus[2] + q[1]
    * input.neg_modulus[1])) * LIMB_RSHIFT_2;
```

The constant `LIMB_SHIFT` has value `1 << 68`, and `LIMB_RSHIFT_2` is the inverse of `1 << (2*68)` in the scalar field $\mathbb{F}_r$, where r is the native circuit prime modulus.

Let us assume there are no overflows or underflows in the calculations above so that we may assume calculations happen in integers rather than $\mathbb{F}_r$.

Then `lo_1 * LIMB_SHIFT^2` is congruent to `a*b - p * q - r` modulo `2^(2 * 68)`. Thus, we can ensure that `a*b - p * q - r = 0` modulo `2^(2 * 68)` as follows. We range check `lo_1` to be less than `r / 2^(2 * 68)`. Then, no overflow will happen in the term `lo_1 * LIMB_SHIFT^2` so that we obtain that `a*b - p * q - r` is indeed congruent to a multiple of `2^(2 * 68)` modulo `2^(2 * 68)` and hence congruent to `0`.

If we assume that `a*b - p * q - r` is congruent to `0` modulo `2^(2 * 68)`, then `hi_3 * LIMB_SHIFT^2` will calculate `(a * b - p * q - r) / 2^(2 * 68)` modulo `2^(2 * 68)`. Similarly as before, range checking `hi_3` to be less than `r / 2^(2 * 68)` ensures that `(a * b - p * q - r) / 2^(2* 68)` is congruent to `0` modulo `2^(2 * 68)` and thus that `a * b - p * q - r` is congruent to `0` modulo `2^(4 * 68)`.

Note that `r / 2^(2 * 68)` rounds down to `251264705520246785319773662051664216`, which has 118 bits. So a range check for 117 bits would suffice.

What `evaluate_non_native_field_multiplication` returns are the variable indexes of `lo_1` and `hi_3`. However, the range check is not performed in the function itself but must be handled by the caller. The function is documented as follows:

```
* @brief Queue up non-native field multiplication data.
*
* @details The data queued represents a non-native field multiplication
    identity a * b = q * p + r,
* where a, b, q, r are all emulated non-native field elements that are each
    split across 4 distinct witness variables.
*
* Without this queue some functions, such as
    bb::stdlib::element::multiple_montgomery_ladder, would
* duplicate non-native field operations, which can be quite expensive.
    We queue up these operations, and remove
* duplicates in the circuit finishing stage of the proving key computation.
*
* The non-native field modulus, p, is a circuit constant
*
```

```
* The return value are the witness indices of the two remainder limbs `lo_1,
    hi_2`
*
* N.B.: This method does NOT evaluate the prime field component of non-native
    field multiplications.
```

These documentation comments do not make clear that the identity `a * b = q * p + r` is not enforced by the function but that the caller must perform an appropriate range check on the two returned circuit variables. We recommend to make this clearer.

4.5. Assumptions regarding bit width of the modulus

The implementation of the `bigfield` class assumes that the emulated modulus has a bit width between 204 and 256 bits, as can be seen from these lines:

```
// code assumes modulus is at most 256 bits so good to define it via a uint256_t
static constexpr uint256_t modulus = (uint256_t(T::modulus_0, T::modulus_1,
    T::modulus_2, T::modulus_3));
// ...
static constexpr uint64_t NUM_LAST_LIMB_BITS = modulus_u512.get_msb()
    + 1 - (NUM_LIMB_BITS * 3);
```

Here, `NUM_LIMB_BITS` has value 68, so the bit width of the modulus must be at least `3*68 = 204`. We recommend documenting this requirement. Optimally, a compile-time check that the modulus is in the required range would also be added.

4.6. Behavior in simulator mode

The circuit code can be run with a circuit simulator instead of a real circuit builder for faster testing. In simulator mode, generally code paths should be used that are analogous to the case of constant values. This is described in comments in `cpp/src/barretenberg/stdlib_circuit_builders/circuit_simulator.hpp`:

```
* The general strategy is to use the "constant" code paths that already exist
    in the stdlib, but with a circuit
* simulator as its context (rather than, for instance, a nullptr). In cases
    where this doesn't quite work, we use
* template metaprogramming to provide an alternative implementation.
*
```

```
* This means changing the data model in some ways. Since a simulator does not
    contain a `variables` vector, we cannot
* work with witness indices anymore. In particular, the witness index of every
    witness instantiated with a simulator
* type will be `IS_CONSTANT`, and the witness is just a wrapper for a value. A
    stdlib `field_t` instance will now store
* all of its data in the `additive_constant`, and something similar is true for
    the `uint` classes.
```

Note that the documentation here is incorrect regarding the witness index being `IS_CONSTANT`. The value of `IS_CONSTANT` is `0xffffffff=4294967295`, whereas the circuit simulator returns `1028` as witness indexes — see, for example, in these functions still in `cpp/src/barretenberg/stdlib_circuit_builders/circuit_simulator.hpp`:

```
inline uint32_t add_variable([[maybe_unused]] const bb::fr index) const {
    return 1028; }
inline bb::fr get_variable([[maybe_unused]] const uint32_t index) const {
    return 1028; }

uint32_t put_constant_variable([[maybe_unused]] const bb::fr& variable) {
    return 1028; }
```

In the `bigfield` component, there some places where simulator mode behaves unusually, which may result in incomplete testing.

The function `bigfield::assert_equal` just returns in simulator mode, rather than asserting that the values are equal, which would be the expected behavior (see also Finding 3.14. ↗ for the same issue in the case of both operands being constant). There is already a TODO and issue for this.

Similarly in the following constructor,

```
template <typename Builder, typename T>
bigfield<Builder, T>::bigfield(const field_t<Builder>& low_bits_in,
                               const field_t<Builder>& high_bits_in,
                               const bool can_overflow,
                               const size_t maximum_bitlength)
```

the limbs are treated differently in the simulated case. This constructor is passed low and high halves of the value, so, for example, `low_bits_in` contains the value that will be represented by the two lower binary limbs, which together are `2*68` bits wide. In the nonconstant `!HasPlookup<Builder> && !IsSimulator<Builder>` case, this element will be decomposed into two limbs using `decompose_into_base4_accumulators` as follows:

```
low_accumulator = context->decompose_into_base4_accumulators(
    low_bits_in.witness_index, static_cast<size_t>(NUM_LIMB_BITS * 2),
    "bigfield: low_bits_in too large.");
```

```
mid_index = static_cast<size_t>((NUM_LIMB_BITS / 2) - 1);
// Range constraint returns an array of partial sums, midpoint will happen to
    hold the big limb
// value
if constexpr (!IsSimulator<Builder>) {
    limb_1.witness_index = low_accumulator[mid_index];
}
// We can get the first half bits of low_bits_in from the variables we already
    created
limb_0 = (low_bits_in - (limb_1 * shift_1));
```

Here, `limb_1` is first assigned the correct value from the return value of `decompose_into_base4_accumulators`, and then `limb_0` is derived from this. As `decompose_into_base4_accumulators` will constrain `low_bits_in` to be at most `2*68` bits wide, and it ensures that `limb_1` assigned the way it is will be the high 68 bits, setting `limb_0 = (low_bits_in - (limb_1 * shift_1))` ensures that `limb_0` will be the low 68 bits of `low_bits_in`. In particular, both `limb_0` and `limb_1` will be at most 68 bits wide.

However, in the nonconstant `!HasPlookup<Builder> && IsSimulator<Builder>` case, we skip setting the `limb_1` witness index. This in itself makes sense, as the witness index does not carry information in the simulated case, and the `std::vector<uint32_t> decompose_into_base4_accumulators` function is also not properly implemented for `CircuitSimulatorBN254`. It just returns a single index, rather than a list with the expected number of elements. Note that this can cause crashes due to reading out of bounds if any caller does not special-case the `IsSimulator<Builder>` case for return values of `decompose_into_base4_accumulators`. However, the end result in the simulated case will be that `limb_0` will be up to `2*68` bits wide, and `limb_1` will have value zero. As the first limb still gets attached a default maximum value that is 68 bits wide, the assumption that limb values are always at most their maximum value will be broken:

```
binary_basis_limbs[0] = Limb(limb_0, DEFAULT_MAXIMUM_LIMB);
```

It is unclear whether this would have any impact, as code paths for constant values usually do not use the maximum values. Furthermore, it is unclear whether `low_bits_in.witness_index != IS_CONSTANT && !HasPlookup<Builder> && IsSimulator<Builder>` can ever be true, as the construction of `field_t` out of witnesses special-cases the simulated case and makes the field element a constant (`cpp/src/barretenberg/stdlib/primitives/field/field.cpp`):

```
template <typename Builder>
field_t<Builder>::field_t(const witness_t<Builder>& value)
    : context(value.context)
{
    if constexpr (IsSimulator<Builder>) {
        additive_constant = value.witness;
        multiplicative_constant = 1;
        witness_index = IS_CONSTANT;
    } else {
        additive_constant = 0;
        multiplicative_constant = 1;
        witness_index = value.witness_index;
    }
}
```

4.7. Handling of too large ranges in `uint256_t::slice`

The function `uint256_t::slice` is implemented as follows:

```
/**
 * Viewing `this` uint256_t as a bit string, and counting bits from 0, slices a
    substring.
 * @returns the uint256_t equal to the substring of bits from (and including)
    the `start`-th bit, to (but excluding) the
 * `end`-th bit of `this`.
 */
constexpr uint256_t uint256_t::slice(const uint64_t start, const uint64_t end)
    const
{
    const uint64_t range = end - start;
    const uint256_t mask = (range == 256) ? -uint256_t(1) : (uint256_t(1)
    << range) - 1;
    return ((*this) >> start) & mask;
}
```

Note that this function works correctly even when `end` is bigger than 256 (essentially, the 256-bit value is extended by zeros for more significant bits). However, the implementation will not handle ranges bigger than 256 correctly. To prevent incorrect usage, we recommend to assert that `range <= 256`, or alternatively document that the caller must ensure this.

4.8. Missing maximum-bit number checks in constructor

For the following `bigfield` constructor, the `maximum_bitlength` argument is used to determine how many bits wide the most significant binary limb will be. The case of `maximum_bitlength > 4*NUM_LIMB_BITS` is not intended to be supported by the function; we recommend to assert this in the constructor.

```
template <typename Builder, typename T>
bigfield<Builder, T>::bigfield(const field_t<Builder>& low_bits_in,
                               const field_t<Builder>& high_bits_in,
                               const bool can_overflow,
                               const size_t maximum_bitlength)
{
    ASSERT((can_overflow == true && maximum_bitlength == 0) ||
           (can_overflow == false && (maximum_bitlength
    == 0 || maximum_bitlength > (3 * NUM_LIMB_BITS))));
   ASSERT(maximum_bitlength <= 4*NUM_LIMB_BITS)
```

4.9. Fix for `decompose_into_bits`

The function `field_t<Builder>::decompose_into_bits` in ‘cpp/src/barretenberg/stdlib/primitives/field/field.cpp contains the following piece of code:

```
// TODO: Guido will make a PR that will fix an error here; hard-coded 127 is
    incorrect when 128 < num_bits < 256.
// Extract bit vector and show that it has the same value as `this`.
for (size_t i = 0; i < num_bits; ++i) {
    bool_t<Builder> bit = get_bit(context, num_bits - 1 - i, val_u256);
    result[num_bits - 1 - i] = bit;
    bb::fr scaling_factor_value = fr(2).pow(static_cast<uint64_t>(num_bits
    - 1 - i));
    field_t<Builder> scaling_factor(context, scaling_factor_value);

    sum = sum + (scaling_factor * bit);
    if (i == 127)
        shifted_high_limb = sum;
}
```

As the TODO comment indicates, the function does not function correctly for `num_bits > 128`. It appears that a fix for this would be to replace the condition `i == 127` by `num_bits == 128 + 1 + i`. We also recommend to expand the `decompose_into_bits` test in `cpp/src/barretenberg/stdlib/primitives/field/field.test.cpp` to cover the case of `num_bits` being a value other than 256.

4.10. Overflow checks for unreduced elements

Many `bigfield` functions require the individual binary limbs and full value to satisfy certain bounds for proper functioning. The `reduction_check` function checks whether these bounds are satisfied, and if not, it carries out a reduction to replace the element with another representation that does satisfy those bounds.

Note that in particular this means that `reduction_check` must be able to handle input that does not satisfy the usual bounds. The function, however, also requires certain bounds to be satisfied to function properly. For example, `bigfield` elements where limbs have maximum values bigger than or equal to the native circuit prime modulus do not represent a well-defined unsigned integer anymore, and hence this occurring should be ruled out. While this is very unlikely to happen on normal usage, for defense in depth, we recommend to assert that the maximum value of all limbs is less than the native circuit prime modulus in `reduction_check`.

4.11. Prime limb witness indexes equality check in `operator-` and `operator+`

In `operator-` and `operator+`, the optimized implementation of the last calculation step in the `HasPlookup<Builder>` case is skipped if a boolean `limbconst` is true:

```
if constexpr (HasPlookup<Builder>) {
    if (prime_basis_limb.multiplicative_constant == 1 &&
    other.prime_basis_limb.multiplicative_constant == 1 &&
        !is_constant() && !other.is_constant()) {
        bool limbconst = binary_basis_limbs[0].element.is_constant();
        limbconst = limbconst || binary_basis_limbs[1].element.is_constant();
        limbconst = limbconst || binary_basis_limbs[2].element.is_constant();
        limbconst = limbconst || binary_basis_limbs[3].element.is_constant();
        limbconst = limbconst || prime_basis_limb.is_constant();
        limbconst = limbconst
    || other.binary_basis_limbs[0].element.is_constant();
        limbconst = limbconst
    || other.binary_basis_limbs[1].element.is_constant();
        limbconst = limbconst
    || other.binary_basis_limbs[2].element.is_constant();
        limbconst = limbconst
    || other.binary_basis_limbs[3].element.is_constant();
        limbconst = limbconst || other.prime_basis_limb.is_constant();
        limbconst = limbconst || (prime_basis_limb.witness_index ==
    other.prime_basis_limb.witness_index);
        if (!limbconst) {
            // optimized calculation saving a gate
            // ...
            return result;
        }
    }
}
// normal case calculation
// ...
return result;
```

The last condition in `limbconst`, namely whether the two prime limbs have the same witness index, is not something that is necessary to ensure the code in the `!limbconst` branch functions correctly. Thus, this check could be removed from `limbconst`.

For `operator-`, the reason it is assumed that the witness indexes are not the same is that if `*this` and `other` are equal, then instead of using up gates to constrain the subtraction, one could use the result zero directly. If usage of `operator-` is to be prevented in those cases, it should instead be asserted that the indexes are different.

4.12. Inefficient loops

In some parts of the codebase, unnecessary loops are used — for example, this loop in `bigfield<Builder, T>::operator-`,

```
uint512_t constant_to_add = modulus_u512;
// add a large enough multiple of p to not get negative result in subtraction
while (constant_to_add.slice(NUM_LIMB_BITS * 3, NUM_LIMB_BITS
    * 4).lo <= limb_3_maximum_value) {
    constant_to_add += modulus_u512;
}
```

which could be replaced by (pseudocode):

```
constant_to_add_factor = ((limb_3_maximum_value << (3*NUM_LIMB_BITS))
    / modulus_u512) + 1;
constant_to_add = constant_to_add_factor * modulus_u512;
```

Note, however, that the above loop does not calculate the *minimum* multiple of p so as to avoid a negative result on subtraction. For that, the loop should only advance on < rather than <=. However, even though the documenting comments for `operator-` emphasize that the minimum should be calculated, this is not actually necessary (though using a larger-than-necessary multiple of p has performance downsides):

```
* We must compute the MINIMUM value of `m` that ensures that none of the
    bigfield limbs will underflow!
```

```
*
* i.e. We must compute the MINIMUM value of `m` such that, for each limb `i`,
    the following result is positive:
*
* *this.limb[i] + X.limb[i] - other.limb[i]
```

We recommend to update the documentation accordingly. There is also an inconsistency in the comments regarding usage of "positive" (usually meaning > 0) versus "non-negative" (meaning >= 0). In this case, the goal is preventing an underflow, so >= 0 is the precise condition, and so usage of "non-negative" would be clearer.

To replace the variant of the loop that uses <, one could use the following:

```
constant_to_add_factor = (((limb_3_maximum_value << (3*NUM_LIMB_BITS))
    + modulus_u512 - 1 ) / modulus_u512);
constant_to_add = constant_to_add_factor * modulus_u512
```

Similarly, in `bigfield<Builder, T>::assert_is_not_equal`, the loop

```
uint512_t target = target_modulus;
size_t overload_count = 0;
while (target < maximum_value) {
    ++overload_count;
    target += target_modulus;
}
return overload_count
```

could be replaced by

```
if (maximum_value == 0)
    return 0;
else
    return (maximum_value - 1) / target_modulus
```

See also Finding 3.7. ↗, according to which the first branch and `-1` should be removed as well, corresponding to changing < in the condition of the loop to <=.

4.13. Possible shaper bounds

We list some calculations of bounds that could be made sharper.

Maximum value for constant limbs

The Limb structure has a field maximum_value for the maximum value that the wrapped circuit variable may have. From usage in the remaining codebase, it is clear that this is intended as an upper bound that can be reached. However, the constructor of Limb sets maximum_value to one higher than the actual value in the case of a constant:

```
Limb(const field_t<Builder>& input, const uint256_t max = uint256_t(0))
    : element(input)
{
    if (input.witness_index == IS_CONSTANT) {
        maximum_value = uint256_t(input.additive_constant) + 1;
    } else if (max != uint256_t(0)) {
        maximum_value = max;
    } else {
        maximum_value = DEFAULT_MAXIMUM_LIMB;
    }
}
```

It would be sharper to use maximum_value = uint256_t(input.additive_constant); in the constant case. However, see also Finding 3.12. ↗.

The following member function for Limb produces output that reflects an incorrect meaning of maximum_value; it should use <= or ≤ instead of <:

```
friend std::ostream& operator<<(std::ostream& os, const Limb& a)
{
    os << "{ " << a.element << " < " << a.maximum_value << " }";
    return os;
}
```

Lower bound for square root of maximum_product in get_maximum_unreduced_value

The get_maximum_unreduced_value function is intended to return a lower bound for the square root of maximum_product.

```
static constexpr uint512_t get_maximum_unreduced_value(const size_t
    num_products = 1)
{
    // return (uint512_t(1) << 256);
    uint1024_t maximum_product = uint1024_t(binary_basis.modulus)
    * uint1024_t(prime_basis.modulus)
    / uint1024_t(static_cast<uint64_t>(num_products));
    // TODO: compute square root (the following is a lower bound, so good for
    the CRT use)
    uint64_t maximum_product_bits = maximum_product.get_msb() - 1;
    return (uint512_t(1) << (maximum_product_bits >> 1)) - uint512_t(1);
}
```

The bound can be sharpened, as we now explain. In our exposition, we will assume `num_products=1`, which does not change any of the arguments, though.

The return value must satisfy `return_value <= maximum_product^(1/2)`, if `maximum_product` is the maximum value that a product is allowed to take. However, as `maximum_product`, as calculated in the current code, is the CRT modulus, this is actually already too big. To avoid wraparound modulo the CRT modulus, `maximum_product` should be calculated as follows (pseudocode):

```
maximum_product = (((binary_basis.modulus * prime_basis.modulus) - 1)
    / num_products);
```

Note that due to the remainder of the function not being sharp, this small mistake does not cause any issues in the current code.

Now, with the corrected `maximum_product`, we must ensure that `return_value <= maximum_product^(1/2)`. Let us assume we have a value for `maximum_product_bits` so that `maximum_product >= 2^maximum_product_bits`. Then it follows from this that

```
2^(maximum_product_bits >> 1) <= 2^(maximum_product_bits / 2)
    = (2^maximum_product_bits)^(1/2) <= maximum_product^(1/2)
```

A value for `maximum_product_bits` that satisfies `maximum_product >= 2^maximum_product_bits` is `maximum_product.get_msb()`, as if the `i`-th bit is set in `maximum_product`, then we must have `2^i <= maximum_product`.

Thus, a sharper bound could be obtained with the following pseudocode:

```
static constexpr uint512_t get_maximum_unreduced_value(const size_t
    num_products = 1)
{
    // return (uint512_t(1) << 256);
    uint1024_t maximum_product = ((uint1024_t(binary_basis.modulus)
    * uint1024_t(prime_basis.modulus)) - 1)
    / uint1024_t(static_cast<uint64_t>(num_products));
    // TODO: compute square root (the following is a lower bound, so good for
    the CRT use)
    uint64_t maximum_product_bits = maximum_product.get_msb();
    return (uint512_t(1) << (maximum_product_bits >> 1));
}
```

Maximum value of last limb in `bigfield` constructor

In the constructor `bigfield<Builder, T>::bigfield(const byte_array<Builder>& bytes)`, at the end, the bound is computed with

```
const auto num_last_limb_bits = 256 - (NUM_LIMB_BITS * 3);
res.binary_basis_limbs[3].maximum_value = (uint64_t(1) << num_last_limb_bits);
```

The maximum value here is not as sharp as it could be. One can be subtracted since `maximum_value` is an upper bound that can be attained and `(uint64_t(1) << num_last_limb_bits) - 1` is the maximum that a value that is `num_last_limb_bits`-bits wide can attain.

Maximum quotient bits

The `get_quotient_max_bits` function is implemented as follows:

```
/**
 * @brief Compute the maximum number of bits for quotient range proof to
    protect against CRT underflow
 *
 * @param remainders_max Maximum sizes of resulting remainders
 * @return Desired length of range proof
 */
static size_t get_quotient_max_bits(const std::vector<uint1024_t>&
    remainders_max)
{
    // find q_max * p + ...remainders_max < nT
    uint1024_t base = get_maximum_crt_product();
    for (const auto& r : remainders_max) {
        base -= r;
    }
    base /= modulus_u512;
    return static_cast<size_t>(base.get_msb() - 1);
}
```

As the comments indicate, it is intended to compute the maximum bit width that ensures that values q with that bit width still satisfy `q*p + sum(remainders_max) < CRT modulus`.

In the implementation, the intention is to first find the maximum value the product `q*p` is allowed to take (`base` after the loop), then divide by `p` to obtain the maximum value for `q`, and finally obtain the maximum bits that ensure that numbers with that bit width are less than or equal to the obtained maximum for `q`.

For the first step, there is a small mistake in the implementation in that `base` should start with `uint1024_t base = get_maximum_crt_product() - 1;`, so one smaller than in the current implementation. This is because `q*p + sum(remainders_max)` is to be strictly smaller than the CRT modulus, not smaller than or equal to it.

This mistake is, however, overcompensated for with a less-strict-than-necessary estimate for the last step. If after division `base` is the maximum value that is allowed for `q`, then with `b = base.get_msb()`, we have that the `b`-th bit is the most significant bit set in `q`. Thus, we have `2^b <= q < 2^(b+1)`. Any value with a bit width less than or equal to `b` is at most `(2^b) - 1` (bits `0` through `b-1` set). The upshot is that it is fine to return `base.get_msb()` instead of `base.get_msb() - 1`. This is assuming the change reducing the initial value of `base` by `1` is made as well.

4.14. Unnecessary assert in `mul_product_overflows_crt_modulus`

The two functions `mul_product_overflows_crt_modulus` are used to check if terms of the form $\sum_{i=0}^{k} a_i \cdot b_i + \sum_{i=0}^{l} c_i$ would overflow modulo the CRT modulus $2^t \cdot r$, where r is the native circuit prime modulus. One variant is for the case in which $k = 0$ — otherwise they are identical. The single-product variant is implemented as follows:

```
static bool mul_product_overflows_crt_modulus(const uint1024_t& a_max,
                                              const uint1024_t& b_max,
                                              const
    std::vector<bigfield>& to_add)
{
    uint1024_t product = a_max * b_max;
    uint1024_t add_term;
    for (const auto& add : to_add) {
        add_term += add.get_maximum_value();
    }
    constexpr uint1024_t maximum_default_bigint = uint1024_t(1)
    << (NUM_LIMB_BITS * 6 + NUM_LAST_LIMB_BITS * 2);

    // check that the add terms alone cannot overflow the crt modulus. v.
    unlikely so just forbid circuits that
    // trigger this case
    ASSERT(add_term + maximum_default_bigint < get_maximum_crt_product());
    return ((product + add_term) >= get_maximum_crt_product());
}
```

It is unclear what the reason of the assertion `add_term + maximum_default_bigint < get_maximum_crt_product()` is here. The comment suggests the intention to forbid cases in which the sum ($\sum_{i=0}^{l} c_i$ in the expression at the start) would already overflow. This would be the check `add_term < get_maximum_crt_product()`, so it is unclear why `maximum_default_bigint` is added.

Note that we have the following bound for `get_maximum_crt_product`:

```
get_maximum_crt_product() = r*2^t = r * 2^(4*68) > (r-1) * 2^(272)
```

The maximum value an unreduced `bigfield` could possibly attain is one in which all four binary limbs are `r-1`. This value is

```
(r-1)*(1 + 2^68 + 2^(2*68) + 2^(3*68)) < (r-1) * 2^(3*68+1) = (r-1) * 2^205
```

The number of addition terms needed to overflow the CRT modulus in a sum are thus at least `2^(272-205) = 2^67`. This is an unrealistic number of summands, so such an overflow should never happen in practice.

The value of `maximum_default_bigint` is `2^(2*s)`, where `s` is the number of bits of `p`. Note that `s<=256` as the emulated prime can be at most 256 bits. Thus, the number of addition terms needed for `add_term + maximum_default_bigint >= get_maximum_crt_product()` to be possible is at least

```
  ((r-1) * 2^272 - 2^512) /  ((r-1) * 2^205)
> (2^253 * 2^272 - 2^512) /  (2^254 * 2^205)
= (2^525 - 2^512) / 2^459
> 2^525 / 2^459
= 2^66
```

This is still an unrealistic number of summands, so the current assert should also never trigger in practice.

The maximum value `get_maximum_value()` can return for a `bigfield` element is less than `2^(256+3*68)=2^460`, so a product of two such values will be less than `2^920`. Thus, an overflow due to adding `product` is also not possible in practice, as long as `a_max` and `b_max` are obtained from such a call to `get_maximum_value()`.

4.15. Unnecessary reductons in `operator+`

The implementation of `operator+` essentially obtains the limbs of the result by adding the relevant limbs of the two summands. A problem can occur should addition of any of the prime limbs cause a wraparound modulo the native circuit prime modulus r (which is a 254-bit prime). This is prevented by carrying out a reduction check on both summands at the start:

```
template <typename Builder, typename T>
bigfield<Builder, T> bigfield<Builder, T>::operator+(const bigfield& other)
    const
{
    reduction_check();
    other.reduction_check();
```

This will, if necessary, use a reduction to ensure that all binary limbs are at most about 117 bits.[5] Note that as the sum of two 117-bit values can at most be a 118-bit value, there is a very large buffer between this and the 254-bit number r. It is also unlikely that normal usage of `bigfield` binary limbs with maximum values close to r would occur. It thus might make sense to remove these two reduction checks and instead only assert that the sum of binary limbs is less than r, or perhaps reduce the two summands conditionally on this occurring. This could save on unnecessary reductions.

4.16. Unreachable code

Some parts of the code are not reachable and should be removed.

Unreachable branch in `assert_equal` function

The `assert_equal` function has several nested if, else-if, and else statements:

```
if (is_constant() && other.is_constant()) {
    std::cerr << "bigfield: calling assert equal on 2 CONSTANT bigfield
    elements...is this intended?"
        << std::endl;
    return;
} else if (other.is_constant()) {
    // TODO(https://github.com/AztecProtocol/barretenberg/issues/998):
    Something is fishy here
    // evaluate a strict equality - make sure *this is reduced first, or an
    honest prover
    // might not be able to satisfy these constraints.
    field_t<Builder> t0
    = (binary_basis_limbs[0].element - other.binary_basis_limbs[0].element);
    field_t<Builder> t1
    = (binary_basis_limbs[1].element - other.binary_basis_limbs[1].element);
    field_t<Builder> t2
    = (binary_basis_limbs[2].element - other.binary_basis_limbs[2].element);
    field_t<Builder> t3
    = (binary_basis_limbs[3].element - other.binary_basis_limbs[3].element);
    field_t<Builder> t4 = (prime_basis_limb - other.prime_basis_limb);
    t0.assert_is_zero();
    t1.assert_is_zero();
```

[5] In the current implementation, the value of 2^{117} itself is also allowed, which is 118 bits.

```
    t2.assert_is_zero();
    t3.assert_is_zero();
    t4.assert_is_zero();
    return;
} else if (is_constant()) {
    other.assert_equal(*this);
    return;
} else {
    if (is_constant() && other.is_constant()) {
        std::cerr << "bigfield: calling assert equal on 2 CONSTANT bigfield
    elements...is this intended?"
                    << std::endl;
        return;
    } else if (other.is_constant()) {
        // evaluate a strict equality - make sure *this is reduced first, or an
    honest prover
        // might not be able to satisfy these constraints.
        field_t<Builder> t0
    = (binary_basis_limbs[0].element - other.binary_basis_limbs[0].element);
        field_t<Builder> t1
    = (binary_basis_limbs[1].element - other.binary_basis_limbs[1].element);
        field_t<Builder> t2
    = (binary_basis_limbs[2].element - other.binary_basis_limbs[2].element);
        field_t<Builder> t3
    = (binary_basis_limbs[3].element - other.binary_basis_limbs[3].element);
        field_t<Builder> t4 = (prime_basis_limb - other.prime_basis_limb);
        t0.assert_is_zero();
        t1.assert_is_zero();
        t2.assert_is_zero();
        t3.assert_is_zero();
        t4.assert_is_zero();
        return;
    } else if (is_constant()) {
        other.assert_equal(*this);
        return;
    }
```

After `else if (is_constant())`, in the `else` condition, both `this` and `other` must be nonconstants. So the following code is not reachable through either of the following conditions.

Unreachable branch in `unsafe_evaluate_square_add` function

In the `unsafe_evaluate_square_add` function, there is a first check:

```
void bigfield<Builder, T>::unsafe_evaluate_square_add(const bigfield& left,
    const std::vector<bigfield>& to_add, const bigfield& quotient,
    const bigfield& remainder)
{
    if (HasPlookup<Builder>) {
        unsafe_evaluate_multiply_add(left, left, to_add, quotient, { remainder
    });
        return;
    }
    ...
```

Then later in the function, there is another identical check:

```
if constexpr (HasPlookup<Builder>) {
        carry_lo = carry_lo.normalize();
        carry_hi = carry_hi.normalize();
        ctx->decompose_into_default_range(carry_lo.witness_index,
    static_cast<size_t>(carry_lo_msb));
        ctx->decompose_into_default_range(carry_hi.witness_index,
    static_cast<size_t>(carry_hi_msb));
}
```

This condition cannot be reached since the function would have returned during the previous check. To simplify the code, the second `if` block may be removed.

Unused variables

The variables `prime_basis_maximum_limb` and `negative_prime_modulus_mod_binary_basis` are defined but not used.

4.17. Misleading names of functions or variables

Some functions or variables have names suggesting a behavior that differs from the actual implementation.

Function `from_witness`

The function `from_witness` is implemented as follows:

```
static bigfield from_witness(Builder* ctx, const bb::field<T>& input)
```

```
{
    uint256_t input_u256(input);
    field_t<Builder> low(witness_t<Builder>(ctx, bb::fr(input_u256.slice(0,
    NUM_LIMB_BITS * 2))));
    field_t<Builder> hi(witness_t<Builder>(ctx,
    bb::fr(input_u256.slice(NUM_LIMB_BITS * 2, NUM_LIMB_BITS * 4))));
    return bigfield(low, hi);
}
```

As can be seen, it does not take a witness as argument, but instead it constructs one from a field element, making the function name confusing.

Variables in `bigfield` constructor taking a byte array

In the constructor `bigfield<Builder, T>::bigfield(const byte_array<Builder>& bytes)`, the naming scheme

```
const field_t<Builder> hi_8_bytes(bytes.slice(0, 6));
const field_t<Builder> mid_split_byte(bytes.slice(6, 1));
const field_t<Builder> mid_8_bytes(bytes.slice(7, 8));

const field_t<Builder> lo_8_bytes(bytes.slice(15, 8));
const field_t<Builder> lo_split_byte(bytes.slice(23, 1));
const field_t<Builder> lolo_8_bytes(bytes.slice(24, 8));

const auto [limb0, limb1] = reconstruct_two_limbs(ctx, lo_8_bytes,
    lolo_8_bytes, lo_split_byte);
const auto [limb2, limb3] = reconstruct_two_limbs(ctx, hi_8_bytes,
    mid_8_bytes, mid_split_byte);
```

is inconsistent, making it confusing. For example `high_high`, `high_mid`, `high_low`, `low_high`, `low_mid`, and `low_low` would be more consistent.

Switched names `negative_prime_modulus_mod_binary_basis` and `negative_prime_modulus`

In bigfield.hpp, `negative_prime_modulus_mod_binary_basis` and `negative_prime_modulus` are defined as follows:

```
static constexpr bb::fr negative_prime_modulus_mod_binary_basis
    = -bb::fr(uint256_t(modulus_u512));
static constexpr uint512_t negative_prime_modulus
    = binary_basis.modulus - target_basis.modulus;
```

The names would fit much better with the value of the other of the two; in particular, the binary basis is uninvolved in the value `negative_prime_modulus_mod_binary_basis`. The two names appear to have been mistakenly switched.

4.18. Improvements for comments/documentation

Some comments do not reflect the current implementation. This makes understanding the code more difficult and can cause mistakes when making changes to the code or writing code that uses the `bigfield` class.

Comment documenting `is_composite`

The constant `is_composite` in bigfield.hpp is documented as follows:

```
static constexpr bool is_composite = true; // false only when fr is native
```

The usage of `r` and `fr` here is potentially confusing, since elsewhere in bigfield, `r` and `Fr` are used to refer to the actual native scalar field over which the circuit is defined. An example is the following comment:

```
// the rationale of the expression is we should not overflow Fr when applying
    any bigfield operation (e.g. *) and
// starting with this max limb size
```

The comment documenting `is_composite` might thus be clearer if formulated for example as follows:

```
static constexpr bool is_composite = true; // false only when the modulus is
    the native one
```

In addition, note that in the documentation ↗, the native field is noted $\mathbb{F}_n$. A unified notation should be used throughout the code and the documentation to avoid confusion.

Misplaced comment in `operator-`

See this comment in `operator-`:

```
/**
* Plookup bigfield subtractoin
*
```

```
 * We have a special addition gate we can toggle, that will compute: (w_1 + w_4
     - w_4_omega + q_arith = 0)
 * This is in addition to the regular addition gate
 *
 * We can arrange our wires in memory like this:
 *
 * | 1 | 2 | 3 | 4 |
 * |-----|-----|-----|-----|
 * | b.p | a.0 | b.0 | c.p | (b.p + c.p - a.p = 0) AND (a.0 - b.0 - c.0 = 0)
 * | a.p | a.1 | b.1 | c.0 | (a.1 - b.1 - c.1 = 0)
 * | a.2 | b.2 | c.2 | c.1 | (a.2 - b.2 - c.2 = 0)
 * | a.3 | b.3 | c.3 | --- | (a.3 - b.3 - c.3 = 0)
 *
 **/
```

It appears to fit better with the method `evaluate_non_native_field_subtraction` of the `UltraCircuitBuilder_<Arithmetization>` class, since that is where the wires are actually set and used.

Typo in `operator-` comment

In `operator-`, there is the following comment:

```
/**
 * Update the maximum possible value of the result. We assume here that
    (*this.value) = 0
**/
```

This should say `Update the maximum possible value of the result. We assume here that other.value = 0` to be correct.

Typo in `mul_product_overflows_crt_modulus` functions

Both the function `mul_product_overflows_crt_modulus` functions refer to the `ctf modulus`:

```
/**
     * Check that the maximum value of a bigfield product with added values
    overflows ctf modulus.
     *
     * @param a_max multiplicand maximum value
     * @param b_max multiplier maximum value
     * @param to_add vector of field elements to be added
     *
     * @return true if there is an overflow, false otherwise
     **/
```

It should be the CRT modulus.

Documentation comment for `operator*`

The `operator*` documentation comment states the following:

```
/**
 * Evaluate a non-native field multiplication: (a * b = c mod p) where p ==
    target_basis.modulus
 *
 * We compute quotient term `q` and remainder `c` and evaluate that:
 *
 * a * b - q * p - c = 0 mod modulus_u512 (binary basis modulus, currently
    2**272)
 * a * b - q * p - c = 0 mod circuit modulus
 **/
```

However, the value stored in `modulus_u512` is the target prime. What is intended in the second-to-last line is `mod binary_basis.modulus (binary basis modulus, currently 2**272)`.

Misleading comment in `unsafe_evaluate_multiply_add`

The `unsafe_evaluate_multiply_add` function has the following comment:

```
// N.B. this method also evaluates the prime field component of the non-native
    field mul
const auto [lo_idx, hi_idx]
    = ctx->evaluate_non_native_field_multiplication(witnesses, false);

bb::fr neg_prime = -bb::fr(uint256_t(target_basis.modulus));
field_t<Builder>::evaluate_polynomial_identity(left.prime_basis_limb,
                                               to_mul.prime_basis_limb,
                                               quotient.prime_basis_limb *
    neg_prime,
                                               -remainder_prime_limb);
```

The comment could be read to suggest that the method `evaluate_non_native_field_multiplication` called immediately below it evaluates the prime field component of the non-native field multiplication. This is, however, not the case. This evaluation is instead done by the call to `evaluate_polynomial_identity` below.

Typo in `decompose_into_bits` documentation comments

There is a typo in the documentation comments for `decompose_into_bits` in `cpp/src/barretenberg/stdlib/primitives/field/field.cpp`: the names `y_lo` and `y_hi` are swapped in the subtraction table.

Typo in Plonk-gate explanation in field.hpp

Similarly, at the bottom of `cpp/src/barretenberg/stdlib/primitives/field/field.hpp`, there is the following comment,

```
* A Plonk gate is a mix of witness values and selector values. e.g.
    the regular PLONK arithmetic gate checks that:
*
*      w_1 * w_2 * q_m + w_1 * q_1 + w_2 * w_2 + w_3 * q_3 + q_c = 0
```

where it seems there is a typo, and the `w_2 * w_2` summand should be `w_2 * q_2`.

Meaning of `num_products` argument for `reduction_check`

The `reduction_check` function parameter `num_products` is documented as follows:

```
/**
* REDUCTION CHECK

...

*
* @param num_products The number of products a*b in the parent function that
    calls the reduction check. Needed to
*limit overflow
**/
```

However, the meaning of the parameter `num_products`, needed to limit overflows, is unclear. It may be interpreted as the number of products $a \cdot b \cdot c \cdot \ldots$ or the number of products in a sum $\sum a_i \cdot b_i$.

The code only calls the function with `num_products=1`, where the two interpretations do not differ. We recommend to either document more clearly how `num_products` is intended or remove that argument.

Endianness documentation

Different functions in the codebase make different endianness assumptions. For example, the `bigfield(const byte_array<Builder>& bytes)` constructor interprets the byte array as big endian. In contrast, the binary limbs of `bigfield` elements are stored in little endian, and other functions, such as `UltraCircuitBuilder_<Arithmetization>::decompose_non_native_field_double_width_limb` return little-endian values as well.

We thus recommend to document for each function, in which both little- or big-endian order for arguments or return values would be possible, which of the two is used.

Documentation comments for `evaluate_non_native_field_multiplication`

The function `UltraCircuitBuilder_<Arithmetization>::evaluate_non_native_field_multiplicatio` in `cpp/src/barretenberg/stdlib_circuit_builders/ultra_circuit_builder.cpp` is documented as follows:

```
/**
 * @brief Queue up non-native field multiplication data.
 *
 * @details The data queued represents a non-native field multiplication
   identity a * b = q * p + r,
 * where a, b, q, r are all emulated non-native field elements that are each
   split across 4 distinct witness variables.
 *
 * Without this queue some functions, such as
   bb::stdlib::element::multiple_montgomery_ladder, would
 * duplicate non-native field operations, which can be quite expensive. We
   queue up these operations, and remove
 * duplicates in the circuit finishing stage of the proving key computation.
 *
 * The non-native field modulus, p, is a circuit constant
 *
 * The return value are the witness indices of the two remainder limbs `lo_1,
   hi_2`
 *
 * N.B.: This method does NOT evaluate the prime field component of non-native
   field multiplications.
 **/
```

This suggests the function queues up multiplication checks. However, the function actually directly handles the relevant constraints and does not involve any queueing.

Documenting assumptions for `to_byte_array`

The `bigfield::to_byte_array` function can be used in circuit to serialize a `bigfield` element to a byte array of 32 bytes. It has a counterpart in a `bigfield` constructor taking a `byte_array<Builder>` as an argument, which deserializes a byte array to a `bigfield` again. The `to_byte_array` function is implemented as follows:

```
byte_array<Builder> to_byte_array() const
{
    byte_array<Builder> result(get_context());
    field_t<Builder> lo = binary_basis_limbs[0].element +
    (binary_basis_limbs[1].element * shift_1);
    field_t<Builder> hi = binary_basis_limbs[2].element +
    (binary_basis_limbs[3].element * shift_1);
    // n.b.  this only works if NUM_LIMB_BITS * 2 is divisible by 8
    //
    // We are packing two bigfield limbs each into the field elements `lo` and
    `hi`.
    // Thus, each of `lo` and `hi` will contain (NUM_LIMB_BITS * 2) bits. We
    then convert
    // `lo` and `hi` to `byte_array` each containing ((NUM_LIMB_BITS * 2) / 8)
    bytes.
    // Therefore, it is necessary for (NUM_LIMB_BITS * 2) to be divisible by 8
    for correctly
    // converting `lo` and `hi` to `byte_array`s.
    ASSERT((NUM_LIMB_BITS * 2 / 8) * 8 == NUM_LIMB_BITS * 2);
    result.write(byte_array<Builder>(hi, 32 - (NUM_LIMB_BITS / 4)));
    result.write(byte_array<Builder>(lo, (NUM_LIMB_BITS / 4)));
    return result;
}
```

Note that this function only produces a byte array from which the `bigfield` can be recovered using the mentioned constructor if `lo` and `hi` are at most 136 and 120 bits wide, respectively. This is also ensured by the `byte_array` constructor being used, which is documented in parts as follows:

```
* @brief Create a byte_array out of a field element.
*
* @details The length of the byte array will default to 32 bytes, but shorter
    lengths can be specified.
* If a shorter length is used, the circuit will NOT truncate the input to fit
    the reduced length.
* Instead, the circuit adds constraints that VALIDATE the input is smaller than
    the specified length
*
* e.g. if this constructor is used on a 16-bit input witness, where
    `num_bytes` is 1, the resulting proof will fail.
```

However, the `bigfield::to_byte_array` has no documenting comments indicating the requirements on the limbs' maximum size. Note that it does not suffice to have called `reduction_check` on the `bigfield` element before, as this, for example, allows up to 117-bit wide values for the second limb, so `lo` could be up to 185 bits wide. We recommend to document the requirements for `to_byte_array` to avoid a caller inadvertently creating completeness issues by failing to reduce the input when necessary. It could also be considered to create a variant of `reduction_check`, which reduces the element whenever it does not have reduced form (so whenever the lower three limbs are possibly more than 68 bits wide, rather than the 117 for `reduction_check`), and to call that at the start of `to_byte_array`.

5. Assessment Results

At the time of our assessment, the reviewed code was not deployed.

During our assessment on the scoped Barretenberg Bigfield circuits, we discovered 17 findings. Five critical issues were found. Two were of high impact, eight were of low impact, and the remaining findings were informational in nature.

5.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.