ZKSECURITY

Audit of Aztec Foreign Field Arithmetic (Bigfield)

July 1st, 2024

Introduction

zkSecurity was contracted by Aztec to audit their Bigfield implementation in the Barretenberg library. The audit was conducted between June 10th and June 28th. The specific commit that was audited is **`4ba553ba31`**.

Methodology

zkSecurity did line-by-line review of the Barretenberg **`bigfield`** and **`field`** modules. We used a SageMath script to symbolically simplify the custom gate expressions used throughout the Barretenberg **`bigfield`** implementation to ease the subsequent manual review of the simplified expressions.

Overview of Barretenberg Bigfield

The Barretenberg **`bigfield`** module implements arithmetic of a prime field different from the native field of the circuit, i.e. it implements $\mathbb{F}_p$ arithmetic using a (UltraPlonK) circuit over $\mathbb{F}_r$. In particular, it can be used to emulate a prime field which is (somewhat) larger than the native field.

Within Barretenberg, **`bigfield`** is primarily used inside the **`biggroup`** module, which implements constraints for elliptic curve operations over the emulated base field. Applications of this includes verifying a Bn254-based SNARK inside a BN254-based SNARK, or checking secp256k1-based ECDSA signatures inside a Bn254-based SNARK.

How to Embed a Field

For context, let's briefly outline the high-level strategy employed by the **`bigfield`** module to emulate a foreign field $\mathbb{F}_p$.

Motivation. Because the foreign field $\mathbb{F}_p = \mathbb{Z}/(p)$ is larger than the native field $\mathbb{F}_r = \mathbb{Z}/(r)$, it is not possible to "embed" an element of $\mathbb{F}_p$ into $\mathbb{F}_r$; representing it as an integer encoded in the native field. Furthermore, when we embed $\mathbb{F}_p$ into another ring, we must ensure that all arithmetic operations can be executed without overflow: the ring is "large enough" to contain the result. As long as this is achieved, the prover can then non-deterministically reduce the representative modulo p to enable further computation in the foreign field.

A Bigger Ring. To obtain a ring sufficiently large that this embedding is possible *and* to allow for executing all arithmetic operations without overflow, the Barretenberg **`bigfield`** module lifts elements of the field $\mathbb{F}_p = \mathbb{Z}/(p)$ into the ring $\mathbb{Z}_{r \cdot 2^k} = \mathbb{Z}/(r \cdot 2^k)$, where r is the modulus of the native field and k is sufficiently large i.e. $r \cdot 2^k \gg p$. This ring enables emulation of *integer arithmetic* ($\mathbb{Z}$) as long as the elements involved are smaller than $r \cdot 2^k$; such that to reduction modulo $r \cdot 2^k$ occurs. Emulation of $\mathbb{F}_p = \mathbb{Z}/(p)$ is done by emulating $\mathbb{Z}$ arithmetic (on bounded integers) inside $\mathbb{Z}_{r \cdot 2^k}$ and non-deterministically reducing the result modulo p (the prover witnesses the reduction and proves it correct).

CRT Decomposition. Since 2^k and r are coprime, the integer ring above is isomorphic to the product $\mathbb{Z}_{r \cdot 2^k} \cong \mathbb{Z}_r \times \mathbb{Z}_{2^k}$ and the isomorphism is given by the Chinese Remainder Theorem (CRT). Barretenberg **`bigfield`** uses this to represent an element of $\mathbb{Z}_{r \cdot 2^k}$ using its CRT decomposition: as a pair of elements in $\mathbb{Z}_r$ and $\mathbb{Z}_{2^k}$. This enables operating independently on the two components for much better performance. Representing and computing on the $\mathbb{Z}_r$-component is trivially achieved by using the native field arithmetic, while the computation on the $\mathbb{Z}_{2^k}$-component is achieved by decomposing into a set of small limbs such that limb-wise intermediate results never overflow the native field.

Bounds and Range Checks. It might be immediate to the reader, but a potential source of vulnerabilities in any such emulated field is "overflows": since the arithmetic is really done by emulating $\mathbb{Z}_p$ inside $\mathbb{Z}_{r \cdot 2^k}$ which itself is emulated inside $\mathbb{Z}_r$ using limb decomposition. Issue could arise in which the arithmetic is satisfied in $\mathbb{Z}/(r \cdot 2^k)$ but not in $\mathbb{Z}$ and thus not in $\mathbb{Z}_p$, or in which the emulation of $\mathbb{Z}_{r \cdot 2^k}$ is incorrect because a limb overflows the native field during emulated operations in $\mathbb{Z}_{2^k}$. To avoid such issues, Barretenberg **`bigfield`** keeps track of the maximum sizes of all involved operands and intermediate results when generating constraints. This approach allows Barretenberg to be "lazy" about enforcing reductions modulo p, for instance allowing multiple additions to be performed before reducing the result modulo p.

Findings

Below are listed the findings found during the engagement. High severity findings can be seen as so-called "priority 0" issues that need fixing (potentially urgently). Medium severity findings are most often serious findings that have less impact (or are harder to exploit) than high-severity findings. Low severity findings are most often exploitable in contrived scenarios, if at all, but still warrant reflection. Findings marked as informational are general comments that did not fit any of the other criteria.

ID	COMPONENT	NAME	RISK
00	ultra_circuit_builder.cpp	Deduplicating Cached Multiplications Leaves Outputs Unconstrained	High
01	bigfield_impl.hpp	Broken Bigfield Constructor for Non-Normalized Inputs	High
02	bigfield_impl.hpp	Unconstrained Exponent	High
03	bigfield_impl.hpp	Unconstrained Limbs in Exponentiation	High
04	bigfield.hpp	Maximum Limb Size Fails to Ensure Multiplication Soundness	Medium
05	bigfield_impl.hpp	Broken Bigfield Constructor for Fields of Odd Bitlength	Medium
06	bigfield_impl.hpp	Swapped Range Constraints on Carries	Medium
07	bigfield.hpp	Unsafe Constructors Allow Breaking Core Invariant	Medium
08	bigfield_impl.hpp	Incompleteness of Not-Equals Assertion	Low

ID	COMPONENT	NAME	RISK
09	bigfield_impl.hpp	Maximum High Carry Too Small	Low
0a	bigfield_impl.hpp	Handling of Constant Exponents	Low
0b	bigfield_impl.hpp	Unsound Edge Cases in Not-Equals Assertion	Low
0c	bigfield_impl.hpp	Reduction Check Allows Unsound Extreme Cases	Informational
0d	bigfield_impl.hpp	Maximum Carry Size Can Underflow 0	Informational
0e	bigfield_impl.hpp	Maximum Values Higher Than Necessary	Informational
0f	field.cpp	Redundant Normalize	Informational
10	bigfield_impl.hpp	Redundant Bigfield Reduction	Informational
11	bigfield_impl.hpp	Mixing Constant and Variable Limbs Is Not Supported	Informational
12	bigfield_impl.hpp	Moduli With 249 Bits Or Less Are Not Supported	Informational

00 - Deduplicating Cached Multiplications Leaves Outputs Unconstrained

● ultra_circuit_builder.cpp

High

Description

The Ultra circuit builder uses a vector of `` `cached_partial_non_native_field_multiplication` `` structs to store bigfields a, b along with three outputs of a partial multiplication:

- $\mathrm{lo}_0 = \mathrm{a}_0\mathrm{b}_0 + (\mathrm{a}_0\mathrm{b}_1 + \mathrm{a}_1\mathrm{b}_0)2^\ell$
- $\mathrm{hi}_0 = \mathrm{a}_0\mathrm{b}_2 + \mathrm{a}_2\mathrm{b}_0 + (\mathrm{a}_0\mathrm{b}_3 + \mathrm{a}_3\mathrm{b}_0)2^\ell$
- $\mathrm{hi}_1 = \mathrm{hi}_0 + \mathrm{a}_1\mathrm{b}_1 + (\mathrm{a}_1\mathrm{b}_2 + \mathrm{a}_2\mathrm{b}_1)2^\ell$

Note that lo_0 and hi_1 are the low and high contributions to $\mathrm{ab} \bmod 2^{4\ell}$, while hi_0 is an intermediate value.

Partial multiplications are used by `` `unsafe_evaluate_multiple_multiply_add()` `` as an intermediate step to constraining a sum of products ab in the non-native field $\mathbb{F}_\mathrm{p}$. Instead of constraining them immediately, it calls `` `queue_partial_non_native_field_multiplication()` ``, which puts lo_0, hi_0 and hi_1 into fresh witnesses and adds a new cache entry.

When finalizing the circuit, Ultra circuit builder calls `` `process_non_native_field_multiplications()` `` to add the deferred constraints:

```cpp
void UltraCircuitBuilder_<Arithmetization>::process_non_native_field_multiplications()
{
    // [ZKSECURITY] replace every "input wire" with the "canonical wire" in the equivalence
class
    for (size_t i = 0; i < cached_partial_non_native_field_multiplications.size(); ++i) {
        auto& c = cached_partial_non_native_field_multiplications[i];
        for (size_t j = 0; j < 5; ++j) {
            c.a[j] = this->real_variable_index[c.a[j]];
            c.b[j] = this->real_variable_index[c.b[j]];
        }
    }

    // [ZKSECURITY] deduplicate the cached partial nnf multiplications based on the input wires
    cached_partial_non_native_field_multiplication
        ::deduplicate(cached_partial_non_native_field_multiplications);

    // iterate over the cached items and create constraints
    for (const auto& input : cached_partial_non_native_field_multiplications) {
        // [ZKSECURITY] elided: check deduplicated multiplications
    }
}
```

The problem lies in `` `deduplicate()` ``: it removes entries where a, b point to the same variables. However, **deduplicating ignores lo_0, hi_0 and hi_1**. These will in practice be different variables (with the same values) across deduplicated entries, because they are created in `` `queue_partial_non_native_field_multiplication()` `` independently each time that function is called.

To see why `` `deduplicate()` `` ignores lo_0, hi_0 and hi_1, observe that it is based on a `` `std::unordered_set` `` with the hash function and equality operator defined in `` `cached_partial_non_native_field_multiplication` ``. Both `` `Hash` `` and `` `operator==` `` ignore struct contents besides a and b.

```
struct cached_partial_non_native_field_multiplication {
    std::array<uint32_t, 5> a;
    std::array<uint32_t, 5> b;
    FF lo_0;
    FF hi_0;
    FF hi_1;

    bool operator==(const cached_partial_non_native_field_multiplication& other) const
    {
        // [ZKSECURITY] we only compare a and b
        bool valid = true;
        for (size_t i = 0; i < 5; ++i) {
            valid = valid && (a[i] == other.a[i]);
            valid = valid && (b[i] == other.b[i]);
        }
        return valid;
    }

    static void deduplicate(std::vector<cached_partial_non_native_field_multiplication>& vec)
        {
            std::unordered_set<cached_partial_non_native_field_multiplication,
                               Hash, std::equal_to<>> seen;
    // [ZKSECURITY] elided ...
};
```

Since NNF constraints are only added on the deduplicated cache entries, the lo_0, hi_0 and hi_1 in discarded cache entries end up being completely unconstrained. This means a prover can freely assign any value to them.

Nevertheless, unconstrained lo_0 and hi_1 are used as outputs of the partial multiplication in `` `unsafe_evaluate_multiple_multiply_add()` `` and are accumulated into the `` `remainder_terms` `` of the final call to `` `evaluate_non_native_field_multiplication()` ``.

Impact

Controlling the remainder term means the prover can freely adjust the overall bigfield multiplication result -- for example, `` `remainder` `` in `` `mult_madd` `` --, whenever a call to `` `unsafe_evaluate_multiple_multiply_add()` `` features a pair of factors a, b that is repeated across the same circuit.

Code comments indicate that such repeated pairs of factors occur in practice in the **`biggroup`** gadgets (out of scope of this audit), which could open the door to real-world attacks that break the protocol.

One caveat that makes attacks harder to pull off is that the prime limb of the remainder term is still constrained correctly, and would be inconsistent with the binary limbs when manipulating a multiplication result.

Recommendation

Before discarding lo_0, hi_0 and hi_1, add equality constraints ensuring they are equal to the multiplication results that end up being checked. This should be a small local change in the deduplication step, with no impact on the number of gates.

Side remark: As can be seen in the code excerpt above, **`cached_partial_non_native_field_multiplication`** struct defines **`lo_0`**, **`hi_0`** and **`hi_1`** as **`FF`** (field elements). This is likely a mistake, because they represent witness indices (**`uint32_t`**), and are implicitly cast back to **`uint32_t`** when used in gates. The type should be changed to **`uint32_t`** to avoid confusion.

Developer Response

The issue was fixed as recommended.

# 01 - Broken Bigfield Constructor for Non-Normalized Inputs

● bigfield_impl.hpp

High

Description

One of the main bigfield constructors creates a bigfield from two native field elements: the low and high halves.

Native field elements (`` `field_t` ``) are represented lazily as $\mathrm{m} \cdot \mathrm{x} + \mathrm{a}$, where m is a multiplicative constant, x is the witness, and a is an additive constant. The witness x is modeled as a `` `uint32_t` `` witness index.

```
template <typename Builder> class field_t {
    // [ZKSECURITY] elided ...
    mutable bb::fr additive_constant;
    mutable bb::fr multiplicative_constant;

    // [ZKSECURITY] elided ...
    mutable uint32_t witness_index = IS_CONSTANT;
};
```

When including field elements in gates by referring to their witness index, we also have to handle multiplicative and additive constants. For some gates, those constants can be embedded as gate coefficients. In other places, we don't have (or use) this opportunity and expect a raw witness index to be passed in.

Before using a method that expects a raw witness index, the code commonly calls `` `field_t::normalize()` ``. Normalizing means adding the necessary constraints to turn a `` `field_t` `` into one with the same value but `` `multiplicative_constant = 1` `` and `` `additive_constant = 0` ``. We can use the output's `` `.witness_index` `` directly because it fully represents the field element.

In the bigfield constructor, there are instances where the call to `` `normalize()` `` is *missing*. A raw witness index is constrained, coming from a field element which is not necessarily normalized:

```
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)
{
    // [ZKSECURITY] elided ...
    if constexpr (HasPlookup<Builder>) {
        const auto limb_witnesses =
            context->decompose_non_native_field_double_width_limb(
              low_bits_in.normalize().witness_index);
        // [ZKSECURITY] elided ...
    } else {
        size_t mid_index;
        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));
    }
    // [ZKSECURITY] elided ...
}
```

In the code above, there are two different branches depending on the **`Builder`** which bigfield is generic over. The **`HasPlookup<Builder>`** branch correctly uses **`low_bits_in.normalize().witness_index`** when range-constraining **`low_bits_in`** (the lower part of the bigfield).

However, in the "else" branch, which is relevant for **`StandardCircuitBuilder`**, the code directly passes **`low_bits_in.witness_index`** to **`decompose_into_base4_accumulators()`**.

An equivalent mistake is made in the same function when handling **`high_bits_in`**.

Impact

Missing **`normalize()`** in this instance has two consequences:

First, the value that is range-checked is not the actual value of **`low_bits_in`** or **`high_bits_in`**, because the multiplicative and additive constants are not accounted for.

Second, observe how **`limb_1`** is calculated to hold (essentially) the upper half of the bits of **`low_bits_in.witness_index`**, which is *not* necessarily the upper half of **`low_bits_in`**. Subsequently, **`limb_0`** is calculated as:

```
limb_0 = (low_bits_in - (limb_1 * shift_1));
```

This last line ensures that the two limbs will satisfy **`low_bits_in == limb_0 + limb_1 * shift_1`**. However, if **`limb_1`** was calculated inconsistently from **`low_bits_in.witness_index`**, the limb split must not be canonical. In particular, **`limb_0`** is not guaranteed to be at most **`NUM_LIMB_BITS`** in size.

In summary, the constructed bigfield is not guaranteed to satisfy any of the range assumptions we rely on:

- size of individual limbs
- size of the bigfield as a whole
- invariant that the actual value of a limb is bounded by its `` `maximum_value` ``

This could enable a prover to produce an overflow in multiplications where the bigfield is used, and pass the multiplication constraints with an invalid result.

The test below shows how to construct limbs that are larger than their maximum value.

```
static void test_field_construction_bug()
{
    auto builder = Builder();
    fr_ct zero = witness_ct::create_constant_witness(&builder, fr::zero());
    uint256_t two_to_68 = uint256_t(1) << fq_ct::NUM_LIMB_BITS;

    // construct bigfield where the low limb has a non-trivial `additive_constant`
    fq_ct z(zero + two_to_68, zero);

    // assert invariant for every limb: actual value <= maximum value
    // FAILS for StandardCircuitBuilder
    for (auto zi : z.binary_basis_limbs) {
        EXPECT_LE(uint256_t(zi.element.get_value()), zi.maximum_value);
    }
}
```

Recommendation

The fix is to call `` `normalize()` `` on `` `low_bits_in` `` and `` `high_bits_in` `` before accessing their `` `.witness_index` ``.

Developer Response

The issue was fixed, and the API was made easier to use correctly by adding a `` `get_normalized_witness_index()` `` method to `` `field_t` ``.

# 02 - Unconstrained Exponent

● bigfield_impl.hpp

High

Description

`` `bigfield::pow()` `` exponentiates the base, a bigfield, by an exponent of type `` `field_t` `` (native field element variable). The method also restricts the exponent to be at most 32 bits.

```
bigfield<Builder, T> bigfield<Builder, T>::pow(const field_t<Builder>& exponent) const
```

There is a method overload that allows passing in the exponent as a `` `size_t` `` instead, i.e. a constant:

```
template <typename Builder, typename T> bigfield<Builder, T> bigfield<Builder, T>::pow(const
size_t exponent) const
{
    auto* ctx = get_context() ? get_context() : nullptr;

    return pow(witness_t<Builder>(ctx, exponent));
}
```

There are various methods for turning constants of various types into `` `field_t` ``; the one used here is `` `witness_t<Builder>(ctx, exponent)` ``. This constructor simply *witnesses* the input: it creates a fresh witness that contains the `` `exponent` `` as value; no constraints are added on the new witness to connect it to the constant input.

Impact

Not constraining the constant is a mistake: A malicious prover can supply an arbitrary exponent without affecting the circuit.

This is certainly unexpected to users of `` `pow()` ``. Consistent with how constants are used in general in barretenberg, `` `y = x.pow(5)` `` is expected to prove the statement "$y = x^5$ in the bigfield". The statement proven instead is "there exists some exponent e (with at most 32 bits) such that $y = x^e$ in the bigfield".

This issue exposes any circuit that uses `` `pow()` `` with a `` `size_t` `` exponent to likely attacks.

Recommendation

A simple fix is to use a constructor that explicitly adds an equality constraint between the constant and the newly created witness:

```
return pow(witness_t<Builder>::create_constant_witness(ctx, exponent));
```

A better fix is to properly support constant exponents in the base `pow()` method, because then the exponentiation loop can hard-code which multiplications to perform based on the exponent bits, leading to fewer constraints.

In that case, the overload method should just pass in a constant `field_t`:

```
return pow(field_t<Builder>(exponent));
```

However, in the reviewed version of the code, passing in a constant causes the base method to error. See **our finding on unconstrained limbs** and on **constant exponent handling** for details and recommended fixes.

Developer Response

The issue was fixed by adding a dedicated method to exponentiate by a constant. The fix also addresses the finding on **constant exponent handling**.

# 03 - Unconstrained Limbs in Exponentiation

- bigfield_impl.hpp

High

Description

The main loop in `` `bigfield::pow()` `` constructs a bigfield `` `bit` `` from the exponent bit, which determines how we update the accumulated power.

The idea is that we precompute $\text{base} - 1$, and then, in every iteration update the accumulator using

$$\text{acc} = \text{acc} \cdot ((\text{base} - 1) \cdot \text{bit} + 1)$$

Assuming bit is either 0 or 1, the right hand side is either acc or $\text{acc} \cdot \text{base}$. Thus, the exponent bit selects whether to multiply the accumulator with the base.

This is the code:

```
bigfield accumulator(ctx, 1);
bigfield mul_coefficient = *this - 1;
for (size_t digit_idx = 0; digit_idx < 32; ++digit_idx) {
    accumulator *= accumulator;
    const bigfield bit(field_t<Builder>(exponent_bits[digit_idx]),
                       field_t<Builder>(witness_t<Builder>(ctx, 0)),
                       field_t<Builder>(witness_t<Builder>(ctx, 0)),
                       field_t<Builder>(witness_t<Builder>(ctx, 0)),
                       /*can_overflow=*/true);
    accumulator *= (mul_coefficient * bit + 1);
}
```

As can be seen, `` `bit` `` is constructed by using the given exponent bit for the low limb, and zeros for the high limbs.

The problem lies in how those zeros are constructed: `` `witness_t<Builder>(ctx, 0)` `` will simply create a fresh witness with no constraints that force it to be 0.

Impact

The prover is free to choose different values for the high limbs of `` `bit` ``, which enables them to manipulate the exponentiation result with many degrees of freedom. This leaves any circuit that uses `` `pow()` `` highly vulnerable to attacks.

Another remark is that the present way of selecting base or 1 is inefficient: instead of bigfield multiplication **`mul_coefficient * bit`**, we can use **`bigfield::conditional_select()`** which is uses much fewer constraints.

Furthermore, the current code produces an error when the exponent is constant. The error occurs because **`exponent_bits[digit_idx]`** is a constant **`bool_t`**, and so the **`bigfield bit`** mixes constant and variable limbs, which is not supported as documented in another finding.

Recommendation

The following code fixes the vulnerability:

```
bigfield accumulator(1);
bigfield one(1);
for (size_t digit_idx = 0; digit_idx < 32; ++digit_idx) {
    accumulator *= accumulator;
    accumulator *= one.conditional_select(*this, exponent_bits[digit_idx]);
}
```

This version is also more efficient: With **`UltraCircuitBuilder`**, the number of gates in a circuit that does one **`bigfield::pow()`**, where both base and exponent are witnesses, decreases from 8748 to 6455 compared to the original.

The code above works when the exponent is constant, but doesn't handle this case optimally. See our finding on constant exponent handling for details.

Developer Response

The issue was fixed as recommended.

# 04 - Maximum Limb Size Fails to Ensure Multiplication Soundness

- bigfield.hpp

Medium

Description

The bigfield header defines a constant called `` `MAX_UNREDUCED_LIMB_SIZE` ``, which is used by `` `reduction_check()` `` to detect when individual limbs of a bigfield have maximum values too large to safely use them in multiplication.

```
// 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;
```

The formula for `` `MAX_REDUCED_LIMB_SIZE` `` is wrong, and using it as upper bound does not ensure soundness of multiplications.

When computing a bigfield multiplication $a \cdot b$, we sum up terms of the form $2^\ell a_i b_j$ where a_i, b_j are limbs and $\ell = 68$ is the default limb size. One of the conditions for soundness is that the sum of terms does not overflow the native modulus n.

In numbers, `` `MAX_REDUCED_LIMB_SIZE` `` works out to be $(253 + 1)/2 - 10 = 117$.

For two limbs a_i and b_j of 117 bits,

$$2^\ell a_i b_j \approx 2^{68+2\cdot 117} = 2^{302}$$

does overflow the native modulus ($\approx 2^{253.5}$).

Problems in the current formula are that it fails to take into account the 2^ℓ factor, incorrectly doesn't divide `` `MAX_ADDITION_LOG` `` by 2, and rounds up instead of down with `` `+ 1` ``.

Let's derive the correct formula: We search for the maximum M that satisfies

$$K 2^\ell 2^{2M} < n$$

where K is the maximum number of terms we add. Taking logs and reordering, we get

$$M < (\log(n) - \log(K) - \ell)/2$$

In numbers, we get $(253 - 10 - 68)/2 = 87$.

Impact

A local impact is that when going into `*` and other methods that do multiplication, limbs greater than 87 bits will not be detected by `reduction_check()`, and cause overflow.

Furthermore, `max_lo_bits` and `max_hi_bits` in `unsafe_evaluate_multiply_add()` can obtain values where $2^{2\ell}$ times the carry can overflow, because they are computed from max limb sizes.

The impact is significantly mitigated because normal bigfields would overflow the CRT modulus after many additions, even *before* individual limbs go beyond 87 bits. Overflowing the CRT modulus is correctly detected by `reduction_check()`, so these bigfields would get reduced anyway.

However, in special cases where the high limbs of a bigfield are hard-coded to zero, its individual limbs might overflow 87 bits without being caught by the CRT modulus check.

Recommendation

Use a corrected formula for the maximum limb size:

```
MAX_UNREDUCED_LIMB_SIZE = (bb::fr::modulus.get_msb() - MAX_ADDITION_LOG - NUM_LIMB_BITS) / 2
```

Developer Response

The issue was fixed as recommended.

# 05 - Broken Bigfield Constructor for Fields of Odd Bitlength

● bigfield_impl.hpp

Medium

This finding concerns the bigfield constructor from two field elements (high and low bits), that we already discussed in our finding on missing normalization.

Two input field elements are each split into two limbs themselves, giving four in total. The output bigfield is constructed to have as many bits as the target modulus p (but not necessarily be smaller than the modulus), by constructing three lower limbs of $\ell := 68$ bits and a highest limb of $\texttt{num\_last\_limb\_bits} := \lceil \log \mathrm{p} \rceil - 3\ell$ bits.

In the code branch that applies to `` `StandardCircuitBuilder` ``, the two high limbs are split into $\texttt{num\_high\_limb\_bits} = \ell + \texttt{num\_last\_limb\_bits}$ as follows:

```
high_accumulator = context->decompose_into_base4_accumulators(high_bits_in.witness_index,
    static_cast<size_t>(num_high_limb_bits), "bigfield: high_bits_in too large.");

if constexpr (!IsSimulator<Builder>) {
    limb_3.witness_index = high_accumulator[static_cast<size_t>((num_last_limb_bits / 2) - 1)];
}
limb_2 = (high_bits_in - (limb_3 * shift_1));
```

The highest limb `` `limb_3` `` is set to the output of `` `decompose_into_base4_accumulators()` `` at index `` `(num_last_limb_bits / 2) - 1` ``. The lower limb `` `limb_2` `` is then computed from the input and `` `limb_3` ``.

The accumulator array returned by `` `decompose_into_base4_accumulators()` `` is the result of accumulating 1- or 2-bit chunks of the input, starting from the most significant bits:

- The 0th accumulator contains the highest 1 or 2 bits, depending on whether the bit length is odd or even
- Every subsequent accumulator is the result of scaling the previous accumulator by 4 and adding the next two input bits

In summary, the ith accumulator consists of the highest k input bits, where $\mathrm{k} = 2\mathrm{i} + 1$ if the input bit length is odd, and $\mathrm{k} = 2\mathrm{i} + 2$ if the bit length is even.

In the computation of `` `limb_3` ``, we have $\mathrm{i} = \texttt{num\_last\_limb\_bits}/2 - 1$. Note also that $\texttt{num\_last\_limb\_bits}$ is odd iff $\texttt{num\_high\_limb\_bits}$ is odd iff $\lceil \log \mathrm{p} \rceil$ is odd.

If $\texttt{num\_last\_limb\_bits}$ is even, `` `limb_3` `` contains the highest

$$2 \cdot (\texttt{num\_last\_limb\_bits}/2 - 1) + 2 = \texttt{num\_last\_limb\_bits}$$

bits of `high_bits_in`, as expected.

If $\texttt{num\_last\_limb\_bits}$ is odd, the division by 2 rounds down, so `limb_3` contains the highest

$$2 \cdot (\texttt{num\_last\_limb\_bits}/2 - 0.5 - 1) + 1 = \texttt{num\_last\_limb\_bits} - 2$$

bits of `high_bits_in`.

The result in the odd case is incorrect: The high limb contains 2 bits less than intended. This case is relevant whenever the target modulus p has an odd bit length.

Because `limb_3` is 2 bits shorter than it should be, `limb_2 = (high_bits_in - (limb_3 * shift_1))` is not of size ℓ bits as intended, but of size up to $\lceil \log \mathrm{p} \rceil - 2\ell$ bits (same as `high_bits_in`, because we only subtract about 1/4 of its value). The two limbs still sum to the input value as intended, but the split is non-canonical and violates the target range.

Impact

The impact is similar as in our finding on missing normalization, as `limb_2` will violate assumptions on its range, and its `maximum_value` will not be a bound for the actual value.

For a concrete example, for a non-native field of 255 bits (e.g. the Pallas and Vesta fields), `limb_2` has up to $255 - 2 \cdot 68 = 119$ bits. This is true for typical bigfields, since the two-native-fields constructor is the most common way to create bigfields, especially for `StandardCircuitBuilder`. Thus, a typical bigfield multiplication encounters terms of the form $2^{\ell}(\mathrm{a}_2\mathrm{b}_1 + \mathrm{a}_1\mathrm{b}_2)$ which have $68 + 119 + 68 + 1 = 256$ bits and overflow the native field.

However, the issue only affects fields of odd bitlengths. The most prominent use case for bigfield in barretenberg is the Bn254 base field, which has 254 bits and is not affected. Also, only `StandardCircuitBuilder` is affected, as the `HasPlookup<Builder>` branch correctly computes the limb split.

Recommendation

The index for `limb_3` should be computed as `(num_last_limb_bits + 1) / 2 - 1`. This correctly rounds up in the case `num_last_limb_bits` is odd.

Developer Response

The issue was fixed as recommended.

# 06 - Swapped Range Constraints on Carries

● bigfield_impl.hpp

Medium

The core multiplication routine, `unsafe_evaluate_multiply_add()`, constrains multiplication modulo the "binary modulus" $2^{4\ell}$ using two equations (low and high), which both involve a carry for the bits that overflow $2^{2\ell}$. Here, $\ell = 68$ is the bit length of individual limbs.

For soundness, carries c have to be range-constrained such that $2^{2\ell}c$ plus (comparatively small) remainder terms cannot overflow the native field. Concretely, a 116-bit range check on both carries would suffice to ensure soundness.

However, range checks get more expensive with increasing number of bits, so it makes sense to use the smallest bit range that contains the checked values in practice. Thus, the code computes the maximum possible bit length of low and high carries, `carry_lo_msb` and `carry_hi_msb`, and adds range checks using the following code:

```
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));
```

Look closely at the arguments to `range_constrain_two_limbs()` and you can probably guess that something is wrong.

The arguments are swapped: `hi.witness_index` is constrained to the `carry_lo_msb` range, and `lo.witness_index` is constrained to the `carry_hi_msb` range.

The exact same mistake is made in `unsafe_evaluate_multiple_multiply_add()`.

Impact

The issue does not affect soundness, because both ranges are bounded by 70 bits and easily sufficient to prevent overflow, as pointed out above.

However, a completeness issue is reasonably likely: For example, if the low carry's actual value is larger than the `carry_hi_msb` range, creating the proof will fail. Since fixing this issue likely has to involve changing the circuit, such a completeness failure can cause significant disruption to any protocol that relies on the bigfield library.

Recommendation

The fix is straightforward: Swap the arguments to `range_constrain_two_limbs()` in both places where this bug occurs.

Developer Response

The issue was fixed as recommended.

07 - Unsafe Constructors Allow Breaking Core Invariant

● bigfield.hpp

Medium

This issue was identified by the Aztec team during the audit.

Description

The `bigfield` class has two constructors which allow to pass in the limbs that make up a bigfield directly.

```
// 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[0].element);
};
```

Limbs are constructed to have the default `maximum_value`. However, as the comment also communicates, the constructor does not add constraints to prove that the limbs are actually contained in this range.

Impact

It's fairly easy to overlook the external assumption made by this constructor (and the similar constructor below it), which is inconsistent with other bigfield constructors that add all necessary checks themselves. Misusing the constructor can lead to badly underconstrained circuits.

Recommendation

The Aztec team proposed two potential ways of fixing this issue:

- Rename the constructors to mark them as "unsafe", so they are not accidentally misused
- Ensure that the constructors can only be used by `biggroup` by declaring it as friend class

Developer Response

The issue was addressed by making the constructors normal methods, marked as unsafe by their name. A safe variant was added as well. Three misuses of the original constructor in the wider codebase were fixed in the process.

# 08 - Incompleteness of Not-Equals Assertion

● bigfield_impl.hpp

Low

Description

The **`assert_is_not_equal()`** method on the **`bigfield`** class enables the assertion that two bigfield elements are not equal. This is achieved by checking that the two elements are not equal modulo the native modulus r. To make this *sound*, namely to avoid this case where $\textsf{lhs} \neq \textsf{rhs} \mod \mathrm{r}$, yet $\textsf{lhs} = \textsf{rhs} \mod \mathrm{p}$ for the emulated modulus p, it is checked that:

$$\forall \mathrm{m} \in [-\textsf{neg}, \textsf{pos}]. \quad (\textsf{lhs} - \textsf{rhs}) + \mathrm{m} \cdot \mathrm{p} \not\equiv 0 \mod \mathrm{r}$$

For suitably large interval $[-\textsf{neg}, \textsf{pos}]$. This procedure is sound i.e. you can only pass **`assert_is_not_equal()`** if $\textsf{lhs} \not\equiv \textsf{rhs} \mod \mathrm{p}$. However it is not complete i.e. there exists $\textsf{lhs} \not\equiv \textsf{rhs} \mod \mathrm{p}$ such that the assertion fails, e.g. consider the case where the native field $\mathbb{F}_\mathrm{r}$ is the Bn254 scalar field and the emulated field $\mathbb{F}_\mathrm{p}$ is the Bn254 base field. Because $\mathrm{p} > \mathrm{r}$, even when fully reduced, there exists bigfield elements $\textsf{lhs} \not\equiv \textsf{rhs} \mod \mathrm{p}$ yet $\textsf{lhs} \equiv \textsf{rhs} \mod \mathrm{r}$, e.g. $\textsf{lhs} = 0$ and $\textsf{rhs} = \mathrm{r}$.

The comments surrounding the **`assert_is_not_equal()`** method note:

```
// WARNING: This method doesn't have perfect completeness -
// for points equal mod r (or with certain difference kp mod r) but different mod p,
// you can't construct a proof. The chances of an honest prover running afoul of this
// condition are extremely small (TODO: compute probability)
```

The comment is correct that for a scalar/base field pair on a *prime order* curve, like Bn254 above, the probability of this condition for random elements of $\mathbb{F}_\mathrm{p}$ is small: $(\mathrm{p} - \mathrm{r})/\mathrm{p} < (2\sqrt{\mathrm{r}})/\mathrm{p}$; roughly 2^{-126} for Bn254.

However, the condition is *not probabilistic*: a scenario could arise in which a malicious party can construct a statement which no honest prover could prove.

Developer Response

The issue was acknowledged by the Aztec team, pointing out that:

> ***"The incompleteness of 08 is known, however at this time we don't see how it could be practically abused."***

# 09 - Maximum High Carry Too Small

● bigfield_impl.hpp

Low

Description

In `` `unsafe_evaluate_multiply_add()` ``, we compute the max values that $2^{2\ell}\mathrm{lo}$ and $2^{2\ell}\mathrm{hi}$ can take.

```
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;
```

In the actual constraints, the computation of hi involves the lo carry:

$$2^{2\ell}\mathrm{hi} = \mathrm{lo} + \mathrm{z}_2 + 2^{\ell}\mathrm{z}_3$$

where z_2 and z_3 are the combined second and third limbs of all terms to be multiplied and added.

However, for `` `max_hi` `` we ignore the low carry and only add up the maximum of $\mathrm{z}_2 + 2^{\ell}\mathrm{z}_3$.

The same issue exists in `` `unsafe_evaluate_multiple_multiply_add()` ``.

Impact

In edge cases, the missing carry in `` `max_hi` `` might cause a completeness issue. If lo pushes the actual hi past a range check threshold, hi can become larger than its supposed maximum and the range check for `` `max_hi_bits` `` might fail.

Recommendation

Update the computation of `` `max_hi` `` to include `` `max_lo` ``:

```
const uint512_t max_lo = max_r0 + (max_r1 << NUM_LIMB_BITS) + max_a0;
const uint512_t max_lo_carry = max_lo >> (2 * NUM_LIMB_BITS);
const uint512_t max_hi = max_r2 + (max_r3 << NUM_LIMB_BITS) + max_a1 + max_lo_carry;
```

Developer Response

The issue was fixed as recommended.

# 0a - Handling of Constant Exponents

● bigfield_impl.hpp

Low

Description

As highlighted in our finding on unconstrained exponents, the `` `bigfield::pow()` `` method is vulnerable when passing in the exponent as a `` `size_t` ``. The recommended fix there -- converting the exponent to a constant `` `field_t` `` -- doesn't work in the current version of `` `pow()` ``, because the method fails on constant inputs.

Our finding on unconstrained limbs in the exponentiation incidentally showed a way of fixing the case of constant exponents. However, fixing that error uncovers other problems in how `` `pow()` `` handles constant exponents.

Before its main loop, `` `pow()` `` parses the exponent into 32 bits, which are then constrained to accumulate to the original exponent:

```
// [ZKSECURITY] create vector of 32 exponent bits
std::vector<bool_t<Builder>> exponent_bits(32);
for (size_t i = 0; i < exponent_bits.size(); ++i) {
    uint256_t value_bit = exponent_value & 1;
    bool_t<Builder> bit;
    bit = exponent_constant ? bool_t<Builder>(ctx, value_bit.data[0]) : witness_t<Builder>(ctx,
value_bit.data[0]);
    exponent_bits[31 - i] = (bit);
    exponent_value >>= 1;
}

// [ZKSECURITY] check that the exponent bits sum to the exponent
if (!exponent_constant) {
    field_t<Builder> exponent_accumulator(ctx, 0);
    for (const auto& bit : exponent_bits) {
        exponent_accumulator += exponent_accumulator;
        exponent_accumulator += bit;
    }
    exponent.assert_equal(exponent_accumulator, "field_t::pow exponent accumulator incorrect");
}
```

However, if the exponent is constant, the latter check is skipped. This means we silently only use the lowest 32 bits of the exponent. For a constant exponent larger than 32 bits, the method will simply return a wrong exponentiation result without violating any constraint or any other indication of error.

Another (non-security) issue is that the implementation does not exploit constant exponents for efficiency gains. It does a fixed-size loop of 32 times 3 bigfield multiplications. Instead, we could hard-code how many multiplications to do based on the statically known exponent bits. For a small exponent, this would vastly reduce the number of constraints.

Recommendation

An option which solves both issues is to handle the constant case separately, right after determining **`exponent.is_constant()`**, in a way that hard-codes constraints based on the exponent bits:

```
bool exponent_constant = exponent.is_constant();

if (exponent_constant) {
    bigfield accumulator(1);
    uint64_t i = exponent_value.get_msb();
    ASSERT(i < 32); // could be removed, left here for consistency with original code

    for (; i + 1 != 0; i--) {
        accumulator *= accumulator;
        if ((exponent_value >> i) & 1) {
            accumulator *= *this;
        }
    }
    accumulator.self_reduce();
    return accumulator;
}

// ... variable exponent case
```

Further efficiency improvements could be made with a windowed exponentiation algorithm.

Developer Response

The issue was fixed by adding a dedicated method to exponentiate by a constant.

# 0b - Unsound Edge Cases in Not-Equals Assertion

● bigfield_impl.hpp

Low

Description

To assert that two bigfield elements a and b are not equal, the `assert_is_not_equal()` method constructs an interval $[-R, L]$ and checks that

$$\forall k \in [-R, L], \quad (a-b)+kp \not\equiv 0 \mod n$$

where n is the native modulus and p is the bigfield modulus.

The code which constructs the interval has a bug that causes it to be too small in edge cases:

```
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());
```

In `get_overload_count()`, we are supposed to compute the smallest k (called `overload_count`) such that the input a_{max} (called `maximum_value`) satisfies $a_{\text{max}} < (k+1)p$.

The condition $a \leq a_{\text{max}} < (k+1)p$ ensures that if $a = b \bmod p$ for any $b \geq 0$, then $a - b$ (being a multiple of p) is at most kp.

To reach a `target` $= (k+1)p$ such that `target > maximum_value`, we should be incrementing `target` as long as `target <= maximum_value`. However, the while condition in the code uses a strict inequality: `target < maximum_value`. The logic is wrong in the edge case where `maximum_value` is a multiple of p.

As a concrete example, if $a = p$ and $b = 0$, and `a.get_maximum_value()` also happens to be p, then the current logic will not include $k = 1$ in the list of multiples to check, and the assertion `a.assert_is_not_equal(b)` succeeds even though $a = b \bmod p$. In other words, the method is unsound.

Impact

Exploiting this issue seems unlikely, since it is hard for `` `maximum_value` `` to accidentally become an exact multiple of p, except for constants; and for constants created in a normal way, the maximum value of every limb is currently `` `+1` `` the actual value, which mitigates the issue. Note that maximum values are hard-coded in the circuit at development time, and can't be chosen by a malicious party at proving time.

Recommendation

Fix the while condition:

```
while (target <= maximum_value) {
```

Developer Response

The issue was fixed as recommended.

# 0c - Reduction Check Allows Unsound Extreme Cases

● bigfield_impl.hpp

Informational

The method that is used throughout bigfield to ensure inputs are safe to multiply is **`reduction_check()`**. It works as follows:

- Check if any of the limbs are larger than the maximum allowed limb size
- Check if the bigfield as a whole is larger than the maximum allowed size
- If either of these conditions are met, call **`self_reduce()`**

The bounds on individual limbs and the bigfield as a whole are designed to ensure soundness of multiplication, and **`self_reduce()`** is supposed to bring the input back within those bounds.

The way **`self_reduce()`** works is by multiplying the bigfield by 1, and replacing itself with the remainder (which has default maximum values on all limbs).

On first glance, there is circularity in this logic: We do a multiplication to ensure a bigfield is safe to multiply.

However, the soundness of multiplication by 1 is much easier to control than arbitrary multiplication, and only extremely large unreduced bigfields would pose a problem for soundness of **`self_reduce()`**: limbs larger than ~185 bits or bigfields larger than ~526 bits.

How likely are those extreme cases? With the help of the Aztec team, we came to the conclusion that there is currently no way to externally create bigfields that fall into those cases. Thus, this issue should have no security impact in practice.

Recommendation

Add an assertion to **`self_reduce()`** that the bigfield satisfies bounds sufficient to multiply it by 1.

Developer Response

The issue was fixed as recommended, with a more conservative bound of 100 bits on each individual limb.

# 0d - Maximum Carry Size Can Underflow 0

● bigfield_impl.hpp

Informational

When `` `unsafe_evaluate_multiply_add()` `` computes the maximum sizes of its carries, it unconditionally subtracts `` `2 * NUM_LIMB_BITS` `` from the computed maximum sizes.

```
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 either `` `max_lo_bits` `` or `` `max_hi_bits` `` is less than `` `2 * NUM_LIMB_BITS` ``, this subtraction will underflow the uint64 range. The result would likely crash the program as `` `carry_lo_msb` `` and `` `carry_hi_msb` `` are passed to `` `decompose_into_default_range()` `` which would attempt to create an extremely large number of range constraints.

We found that the underflow is theoretically possible in contrived scenarios, and `` `self_reduce()` ``, with its custom small quotient and small multiplicand of 1, comes fairly close to triggering it. Overall, it seems unlikely that the underflow will happen in practice.

Recommendation

Clamp the subtraction to 0 if it goes negative.

Developer Response

The issue was fixed as recommended.

# 0e - Maximum Values Higher Than Necessary

● bigfield_impl.hpp

Informational

There are several places where the maximum value of a limb is set higher than necessary. We picked three to highlight here.

Unintended default maximum value for quotient

At least one case looks like this was done by mistake: In `self_reduce()`, the quotient bigfield is constructed in a custom way where the three highest limbs are hard-coded to zero.

```
quotient.binary_basis_limbs[0] = Limb(quotient_limb, uint256_t(1) << maximum_quotient_bits);
quotient.binary_basis_limbs[1] = Limb(field_t::from_witness_index(context, context->zero_idx),
0);
quotient.binary_basis_limbs[2] = Limb(field_t::from_witness_index(context, context->zero_idx),
0);
quotient.binary_basis_limbs[3] = Limb(field_t::from_witness_index(context, context->zero_idx),
0);
```

For limbs 1, 2 and 3, the `Limb()` constructor is used with a maximum value (second argument) of 0. The intention is probably to set the maximum value to 0, but in fact `Limb()` treats 0 as "no maximum value input", and the default maximum value will be used.

The only negative impact that we could find is that the range check for carries subsequently added by `unsafe_evaluate_multiply_add()` will likely be for a larger range (and thus, be less efficient) than necessary.

Maximum value strictly larger for constants

Another observation is that for constants, the maximum value of limbs is set to the actual value `+1`. This obviously seems to be intentional, even if inconsistent with the case of variables, where a tight maximum value is used. We could not find a reason for the `+1` to be necessary, but it seems fine to be slightly over-conservative in setting maximum values, so we don't advise to change this.

```
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;
```

Generous maximum value in conditional negation

`conditional_negate()` called on a bigfield x sets the output to either x or $kp - x$ for a large enough k that ensures all limbs can be made positive.

The output limbs can never be larger than the maximum of those two values i.e. $\max(x_i, (kp)_i)$ on every limb. However, the code is less tight and sets the maximum value to the *sum* of the two possible maximum limb values:

```
uint256_t max_limb_0 = binary_basis_limbs[0].maximum_value + to_add_0_u256 + t0;
uint256_t max_limb_1 = binary_basis_limbs[1].maximum_value + to_add_1_u256 + t1;
uint256_t max_limb_2 = binary_basis_limbs[2].maximum_value + to_add_2_u256 + t2;
uint256_t max_limb_3 = binary_basis_limbs[3].maximum_value + to_add_3_u256 - t3;
```

Again, the only impact is likely generating a few more range constraints and/or having to reduce bigfields a bit more often than necessary.

Developer Response

The first issue was addressed by supporting a maximum value of 0 in the `Limb` constructor. The third issue was addressed as well.

# 0f - Redundant Normalize

- field.cpp

Informational

In:

```
std::vector<bool_t<Builder>> field_t<Builder>::decompose_into_bits
```

The `**normalize**` method is called on `**y_lo**` but the result is unused:

```
// We always borrow from 2**128*p_hi. We handle whether this was necessary later.
// y_lo = (2**128 + p_lo) - sum_lo
field_t<Builder> y_lo = (-sum) + (p_lo + shift);
y_lo += shifted_high_limb;
y_lo.normalize();
```

Developer Response

The call to `**normalize()**` was removed and the method was marked as `**[[nodiscard]]**` to ensure developers don't mistakenly assume that it has side effects:

```
 [[nodiscard]] field_t normalize() const;
```

The fix also uncovered a significant soundness bug in a `**bool_t**` constructor, where `**normalize()**` was needed but called without effect; it was fixed as well.

# 10 - Redundant Bigfield Reduction

● bigfield_impl.hpp

Informational

In `perform_reductions_for_mult_madd()`, called by `mult_madd()`, reductions are performed on a list of pairs of input factors to ensure the full sum of products cannot overflow the CRT modulus.

In the main loop, after picking an element and reducing it, we recompute `reduction_required` *before* updating the max values it is based on. This causes one extra iteration after we already made it below the threshold.

The effect is that `self_reduce()` is called one time too many, costing unnecessary constraints.

Also, `compute_updates()` and `sort()` would be more naturally placed at the beginning of the while block, so we don't do them another time when they are no longer needed.

Developer Response

The issue was fixed as recommended.

# 11 - Mixing Constant and Variable Limbs Is Not Supported

● bigfield_impl.hpp

Informational

Mixing both constant and variable limbs in the same bigfield causes multiplications to crash.

The current API allows creating mixed bigfields by calling **`bigfield::add_to_lower_limb()`** on a constant.

The following test demonstrates different behaviours for arithmetic operations on various combinations of constant and variable limbs. As the comments describe, the case **`mixed * var`** does work for the Standard circuit builder, but not for Ultra; while the cases **`mixed * mixed`** and **`mixed * constant`** cause segfaults for both circuit builders.

```
static void test_mixed_limbs_bug()
{
    auto builder = Builder();

    fq_ct constant = fq_ct(1);
    fq_ct var = fq_ct::create_from_u512_as_witness(&builder, 1);
    fr_ct small_var = witness_ct(&builder, fr(1));
    fq_ct mixed = fq_ct(1).add_to_lower_limb(small_var, 1);

    fq_ct r;

    // WORKS: add and sub
    r = mixed + mixed;
    r = mixed - mixed;
    r = mixed + var;
    r = mixed + constant;
    r = mixed - var;
    r = mixed - constant;
    r = var - mixed;

    // WORKS: mul and div on constant and var
    r = var * constant;
    r = constant / var;
    r = constant * constant;
    r = constant / constant;

    // FAILS: mul and div involving mixed

    // standard works, ultra throws assert
    r = mixed * var;
    r = mixed / var;
    // standard and ultra both segfault
    r = mixed * mixed;
    r = mixed * constant;

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

Developer Response

The issue was fixed for `` `bigfield::add_to_lower_limb()` `` specifically, by reinitializing the high limbs to witnesses when adding a witness in the low limb to a constant.

# 12 - Moduli With 249 Bits Or Less Are Not Supported

● bigfield_impl.hpp

Informational

In the `*` operator and other methods that do multiplication, `num_quotient_bits` is obtained by calling `get_quotient_reduction_info()` and then used to define how many bits to constrain the `quotient` to.

```
auto [reduction_required, num_quotient_bits] =
    get_quotient_reduction_info({ get_maximum_value() }, { other.get_maximum_value() }, {});
// [ZKSECURITY] elided ...
quotient = create_from_u512_as_witness(ctx, quotient_value, false, num_quotient_bits);
```

`create_from_u512_as_witness()` determines the range constraint for the highest limb by subtracting `3 * NUM_LIMB_BITS` $= 3 \cdot 68$ from the input bit length. The result is passed to `range_constrain_two_limbs()`, which asserts that the range is at most 70 bits.

Thus, there is an implicit maximum of $3 \cdot 68 + 70$ imposed on `num_quotient_bits`.

`num_quotient_bits` is ultimately computed by `get_quotient_max_bits()` to be

$$4 \cdot 68 + \lfloor \log \mathrm{n} - \log \mathrm{p} \rfloor - 1$$

where n is the native modulus and p is the bigfield modulus, and $\log \mathrm{n} \approx 253.5$.

Since `num_quotient_bits` can be at most $3 \cdot 68 + 70$, we obtain an inequality on the size of p:

$$4 \cdot 68 + \lfloor 253.5 - \log \mathrm{p} \rfloor - 1 \leq 3 \cdot 68 + 70$$
$$\lfloor 253.5 - \log \mathrm{p} \rfloor \leq 3$$

When the bigfield modulus has 249.5 bits or less, the inequality is not satisfied: The assertion in `range_constrain_two_limbs()` will prevent those moduli from being used for bigfield multiplications.

Since this limitation seems unintended, we wanted to highlight it even though support for those moduli might not be a pressing concern.

Developer Response

The Aztec team added a static assertion to the bigfield class to make it explicit that only bigfields with 250-256 bits are currently supported.