Aztec Packages
Security Review

Cantina Managed review by:
Andrei maiboroda, Lead Security Researcher

Cperezz, Lead Security Researcher
Ed255, Lead Security Researcher

October 4, 2024

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Aztec Labs was founded in 2017, and has a team of +50 leading zero-knowledge cryptographers, engineers, and business experts. Aztec Labs is developing its namesake products: AZTEC (a privacy-first L2 on Ethereum) and NOIR (the universal ZK language).

From Sep 4th to Sep 25th the Cantina team conducted a review of aztec-packages on commit hash 4ba553ba. The team identified a total of **17** issues in the following risk categories:

- Critical Risk: 2
- High Risk: 1
- Medium Risk: 2
- Low Risk: 2
- Gas Optimizations: 0
- Informational: 10

3 Findings

3.1 Critical Risk

3.1.1 Partially unconstrained `bigfield bit` in pow

Severity: Critical Risk

Context: bigfield_impl.hpp#L1030

Description: In `bigfield bit`, the limbs that are supposed to be 0 are not constrained to actually be 0. This is because the witness constructor will just assign the witness value without any copy constraint. This means that an attacker can choose the `limbs[1,2,3]` of the `bit` value thus making the circuit validate a different result than expected.

Recommendation: Use constants for the `limbs[1,2,3]`. If witness values are really needed, then copy constrain them to be zero.

- Option 1: This uses fields that set the constants in the `additive_constant` so they are part of the circuit:

  ```
  field_t<Builder>(0),
  field_t<Builder>(0),
  field_t<Builder>(0),
  ```

- Option 2: This will assign the witness but copy constraint it to constant 0:

```
field_t<Builder>(witness_t<Builder>::create_constant_witness(ctx, 0)),
field_t<Builder>(witness_t<Builder>::create_constant_witness(ctx, 0)),
field_t<Builder>(witness_t<Builder>::create_constant_witness(ctx, 0)),
```

3.1.2 Invariant broken for limb size while congruent result is produced within bigint constructor

Severity: Critical Risk

Context: bigfield_impl.hpp#L73-L83

Description: The constructor, when called with two limbs, can be exploited to produce two different exploits/bugs:

1. An invariant of the codebase can be broken, where we are able to generate limbs that have more than $2^1$36 bits. This, results on "valid" limbs that will overflow at the very first multiplication they do. Or even addition.
2. We can overpass the `range_check` for `low_bits_in` within the constructor function. Such that we pass the range check but in reality we're working with a much bigger number.

Proof of concept: Notice that we are range-checking `low_bits_in` to be less than 2^{136}. So 2^{B^2}. The problem, is that we don't `normalize()` it. This simple detail (which is handled in the `plookup` version), results on not accounting any of the `multiplicative_constant` nor `additive_constant` of the element.

After that, we can just generate the `limb_0` and `limb_1` and the curious thing, is that while they sum to the same value, they're malformed (`limb_0` indeed overpasses 2^{136}, but still added to `limb_1` gives a correct result).

This leads to the 2 problems mentioned above.

```
auto builder = Builder();
auto lo = field_ct(witness_ct(
    &builder, fr(uint256_t("0000000000000000000000000000000000000000000000100000000000000000")))); // 2^68
auto hi = field_ct(witness_ct(&builder, fr(uint256_t(0))));
auto ct = field_ct(fr(uint256_t(10)));
info("lo: wit_idx = ",
        lo.witness_index,
        ", mul_ct = ",
        lo.multiplicative_constant,
        ", add_ct = ",
        lo.additive_constant);

info("ct: wit_idx = ",
        ct.witness_index,
        ", mul_ct = ",
        ct.multiplicative_constant,
        ", add_ct = ",
        ct.additive_constant);

auto lo_ct = lo * ct; // 0xa00000000000000000 -> limb1= 0xa, limb0=0x0

info("lo_ct: wit_idx = ",
        lo_ct.witness_index,
        ", mul_ct = ",
        lo_ct.multiplicative_constant,
        ", add_ct = ",
        lo_ct.additive_constant);

fq_ct a(lo_ct, hi);
info("a = ", a);
dbg_bf("a", &a);

EXPECT_EQ(CircuitChecker::check(builder), true);
```

Which will output:

```
lo: wit_idx = 2, mul_ct = 0x0000000000000000000000000000000000000000000000000000000000000001, add_ct =
↪  0x0000000000000000000000000000000000000000000000000000000000000000
ct: wit_idx = 4294967295, mul_ct = 0x0000000000000000000000000000000000000000000000000000000000000001, add_ct
↪  = 0x000000000000000000000000000000000000000000000000000000000000000a
lo_ct: wit_idx = 2, mul_ct = 0x000000000000000000000000000000000000000000000000000000000000000a, add_ct =
↪  0x0000000000000000000000000000000000000000000000000000000000000000
a = 0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000⌋
↪  000000a00000000000000000
a.p: 0x0000000000000000000000000000000000000000000000a00000000000000000
  a.b0: { 0x0000000000000000000000000000000000000000000000900000000000000000 <=
  ↪  0x00000000000000000000000000000000000000000000000fffffffffffffffff }
  a.b1: { 0x0000000000000000000000000000000000000000000000000000000000000001 <=
  ↪  0x00000000000000000000000000000000000000000000000fffffffffffffffff }
  a.b2: { 0x0000000000000000000000000000000000000000000000000000000000000000 <=
  ↪  0x00000000000000000000000000000000000000000000000fffffffffffffffff }
  a.b3: { 0x0000000000000000000000000000000000000000000000000000000000000000 <=
  ↪  0x00000000000000000000000000000000000000000000000000003fffffffffff }
```

Pay special attention to:

```
a.b0: { 0x0000000000000000000000000000000000000000000000900000000000000000 <=
↪  0x00000000000000000000000000000000000000000000000fffffffffffffffff }
```

Here is clear that we've created a limb that is bigger than the allowed range. But that this limb added to the rest still gives a congruent value as a result once the limbs are accumulated.

Recommendation: We simply need to call `normalize()` before we input our `low_bits_in` in the decomposition check. Also, the current bug doesn't only affect the `low_bits_in` but also the `hi_limbs_in`. So this needs to be fixed in both ends. The bug is related to soundness and not just correctness.

3.2 High Risk

3.2.1 Constant exponent in pow is unconstrained

Severity: High Risk

Context: bigfield_impl.hpp#L982

Description: The types used by this method makes the user think that they are doing an exponentiation with a constant, but this constant is not part of the circuit, it's just a value that is used to instantiate a witness at prove creation.

This means that whenever we have a call to `pow` with a `const size_t` exponent, a malicious prover can replace that exponent by any value.

Recommendation: Create a witness value that is copy constrained to have the constant value `exponent` like this:

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

3.3 Medium Risk

3.3.1 `to_byte_array` can have aliases

Severity: Medium Risk

Context: bigfield.hpp#L191

Description: The `to_byte_array()` method only guarantees that the underlying representation fits in 256 bits. Since the emulated modulus p is below $2^{256} - 1$ this means that the byte array representation can have aliases (multiple array representations correspond to the same element modulo p).

Recommendation: Document that this method can return different aliases for the same element. For example, if this byte array is used to hash the element, it would be easy for a malicious prover to generate different hashes for the same value.

Document that to avoid aliases the developer should call `assert_is_in_field()` before calling `to_byte_array()`.

3.3.2 Overflow in `unsafe_evaluate_multiply_add` mod `T` constraints

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: `unsafe_evaluate_multiply_add` is supposed to verify that the solution to a multiply and add operation is valid in both $\mod n$ and $\mod T$ where n is the native modulus of the circuit arithmetic and $T = 2^{272}$.

When used in `operator*` the inputs are only reduced if they exceed 117 bits.

If the inputs are 117 bits, the constraints that calculates `carry_lo` and `carry_hi` can overflow in a way that allow different solutions of the reminder (that are not congruent modulo T) to pass the constraints while all the range checks are still satisfied. Nevertheless finding such an alternative solution will break the $\mod n$ constraints, so it's not clear whether this is exploitable; but this means that the reasoning behind applying the CRT with $\mod n$ and $\mod T$ is invalid.

Recommendations: In order to keep the CRT reasoning valid we need to be more strict with the maximum number of bits that values can take to prevent overflows in the `mod T` constraints. In particular we need to make sure that no side of this equality overflows (which also guarantees that the equality involving carry_lo doesn't overflow):

```
lhs_hi := (a[1]*b[1] + q[1]*neg_p[1] + a[2]*b[0] + q[2]*neg_p[0] + a[0]*b[2] + q[0]*neg_p[2] + carry_lo) +
        (a[3]*b[0] + q[3]*neg_p[0] + a[2]*b[1] + q[2]*neg_p[1] + a[1]*b[2] + q[1]*neg_p[2] + a[0]*b[3] +
↪  q[0]*neg_p[3])*B
rhs_hi := r[2] + r[3]*B + carry_hi*B^2
lhs_hi = rhs_lo
```

1. The `rhs_hi` expression (which contains `carry_hi*B^2`) doesn't overflow. This means that both carries (`carry_hi` and `carry_lo`) need to satisfy `max(rhs_hi) < n`, which is fulfilled when `carry_lo` and `carry_hi` are at most 117 bits if $s = 254$ bits.
2. The `lhs_hi` expression (which contains terms like `a3*b0*B` and `a0*b3*B`) doesn't overflow. This means that each limb needs to satisfy `max(lhs_hi) < n` which is fulfilled with each limb is at most 91 bits if $s = 254$ bits.

Proof of concept:

Let:

- `p`: emulated modulus (assume `2^253 < p < 2^254`).
- `n`: native modulus.
- `t = 272`.
- `T = 2^t = 2^272`.
- `B = 2^(t/4) = 2^68`.
- `xp in Fp`: value `x` congruent modulo `p`.
- `xn in Fn`: value `x` congruent modulo `n`.
- `x in ST`: value `x` congruent modulo `T`.
- `x_i in SB`: limb `i` of value `x` in modulo `T`.

We want: `ap*bp = rp mod p`.

Equivalent to solution for `ap*bp = qp*pp + rp mod n*T`. Equivalent to solutions for (using CRT):

1. `an*bn = qn*pn + rn mod n`
2. `a*b = q*p + rn mod T`

We do `2.` via limbs in native arithmetic:

```
(a0*b0 + q0*neg_p0) + (a1*b0 + q1*neg_p0 + a0*b1 + q0*neg_p1)*B =
  r0 + r1*B + carry_lo*B^2 mod n
```

With the following range checks:

- `ai, bi in 117 bits` (this is the max that skips the `self_reduce`).
- `qi, ri in 68 bits` (because `q, r` is freshly built from witness).
- `carry_lo in 168 bits` (that's what `unsafe_evaluate_multiply_add` calcualtes on the previous `a, b`).

This identity has terms that overflow:

- `a1*b0*B`: `117 + 117 + 68 bits = 302 bits`.
- `a0*b1*B`: `117 + 117 + 68 bits = 302 bits`.
- `carry_lo*B^2`: `168 + 2*68 bits = 304 bits`.

With this we can find multiple `r` values that satisfy the identity and the range checks:

```
NUM_LIMB_BITS=68
-  Native modulus
n=21888242871839275222246405745257275088548364400416034343698204186575808495617
-  Emulated modulus
p=21888242871839275222246405745257275088696311157297823662689037894645226208583

t=272
T=2**272
B=2**68

def limbs_T(v):
    b=68
    limb0 = (v & (B-1)       ) >> 0
    limb1 = (v & ((B-1) << 1*b)) >> 1*b
    limb2 = (v & ((B-1) << 2*b)) >> 2*b
    limb3 = (v & ((B-1) << 3*b) )>> 3*b
    return [limb0, limb1, limb2, limb3]

neg_p = limbs_T(T - p)
[p0, p1, p2, p3] = limbs_T(p)

B1_inv =  pow(B, -1, n)
B2_inv =  pow(B**2, -1, n)
```

```
k = 0 # pick a different integer to get a different `r` value
res_lo = ((a[0]*b[0] + q[0]*neg_p[0]) + (a[1]*b[0] + q[1]*neg_p[0] + a[0]*b[1] + q[0]*neg_p[1])*B) % n
res_lo_t = limbs_T(res_lo + k*n)

r[0] = res_lo_t[0]
r[1] = res_lo_t[1]

carry_lo = ((a[0] * b[0] + q[0] * neg_p[0]) * B2_inv + (a[1] * b[0] + q[1] * neg_p[0] + a[0] * b[1] + q[0] *
↪  neg_p[1] - r[1]) * B1_inv - r[0] * B2_inv) % n

-  With k=0 this carry_lo is 118 bits
```

- Missing piece to achieve exploit:

  `q` and `r` are built modulo `n` and modulo `T` by constraints (they are tied together). So even if we manage to plug in an invalid `r` in the modulo `T` constraints we still need to satisfy the modulo `n` constraint which is:

  `an*bn = qn*pn + rn mod n` ⇔

  ```
  (a0 + a1*B + a2*B^2 + a3*B^3)*(b0 + b1*B + b2*B^2 + b3*B^3) =
  (q0 + q1*B + q2*B^2 + q3*B^3)*(p0 + p1*B + p2*B^2 + p3*B^3) +
    (r0 + r1*B + r2*B^2 + r3*B^3) mod n
  ```

  The alternative solutions we found for `r` previously will not work on this constraint. Can we find a way to make this constraint pass?

  The same exact issue happens within `unsafe_evaluate_square_add`. Notice that in this case, we have:

  ```
  a_n * a_n = q_n*p_n - r_n mod n
  ```

  We now operate with the limbs here.

  ```
  (a0*a0 + q0*neg_p0) + (a1*a0 + q1*neg_p0 + a0*a1 + q0*neg_p1)*B =
  r0 + r1*B + carry_lo*B^2 mod n
  ```

  With the following range checks:

  - `a_i in 117 bits` (this is the max that skips the `self_reduce`).
  - `q_i, r_i in 68 bits` (because `q, r` is freshly built from witness).
  - `carry_lo in 168 bits` (that's what `unsafe_evaluate_multiply_add` calcualtes on the previous `a, b`).

  This identity also has terms that overflow:

  - `a1*a0*B`: `117 + 117 + 68 bits = 302 bits`.

- a0*b1*B: 117 + 117 + 68 bits = 302 bits.
- carry_lo*B^2: 168 + 2*68 bits = 304 bits.

The same questions and recommendations from above extend to this function too.

3.4 Low Risk

3.4.1 Swapped bit sizes in `range_constrain_two_limbs`

Severity: Low Risk

Context: bigfield_impl.hpp#L2691, bigfield_impl.hpp#L2253

Description/Recommendation: The bit size parameters in this range constraint are swapped. The correct code should be:

```
ctx->range_constrain_two_limbs(
    hi.witness_index, lo.witness_index, (size_t)carry_hi_msb, (size_t)carry_lo_msb);
```

3.4.2 Assertion needed to prevent bypass range-checks in `unsafe_evaluate_multiply_add`

Severity: Low Risk

Context: bigfield_impl.hpp#L2116-L2121

Description: In `unsafe_evaluate_multiply_add`, nothing is asserting that `to_add` and `remainders` is limited to an amount of elements that doesn't cause an overflow when maximums are being computed. We just sum and accumulate all values:

```
for (size_t i = 0; i < to_add.size(); ++i) {
max_a0 += to_add[i].binary_basis_limbs[0].maximum_value +
           (to_add[i].binary_basis_limbs[1].maximum_value << NUM_LIMB_BITS);
max_a1 += to_add[i].binary_basis_limbs[2].maximum_value +
           (to_add[i].binary_basis_limbs[3].maximum_value << NUM_LIMB_BITS);
}
```

Notice that this is important since later we have:

```
// We can push the max here to actually be 511 bits with `to_add` abuse.
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;
```

These 511 bits will be then operated to obtain `carry_lo/hi_msb`:

```
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);
```

Resulting on a range_check that can't be performed. Basically because it isn't sound for such an amount of bits. All carry values would pass it.

Recommendation: The solution proposal is to simply include an assertion that prevents anyone to create a circuit where `to_add` can be abused to achieve a bypass on the range-checks of the carries.

We've also revisited all the calls to `unsafe_evaluate_multiply_add` and the ones that call it internally with multiple `to_add` or `remainder` values like `internal_div` and confirm that none of them suffer from this issue.

In any case, it's a cheap prevention for possible future errors.

3.5 Informational

3.5.1 Consider removing vector terms to sum from some methods

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The current `bigfield` API has many methods that take as argument a vector of `bigfield` that will be added to the result of the main operation:

- `sqradd` squares a number and then adds all entries of `to_add`.
- `div_without_denominator_check` and `div_check_denominator_nonzero` first add all entries of `numerators` and then perform a division.
- `madd` performs a multiplication and then adds all the entries of `to_add`.
- `mult_madd` performs multiple multiplications and then adds all the entries of `to_add`.
- `dual_madd` performs two multiplications and then adds all the entries of `to_add`.
- `msub_div` performs multiple multiplications, adds all the entries of `to_sub` and then divides.

In most of these methods, the advantage of adding a vector of elements to add is that we get a result that is already reduced. On the contrary, if we just perform the main operation and then add elements using `operator+` the final result would not be reduced and the binary limbs would not be minimal.

Nevertheless since we have bit space in the binary limbs to perform many additions without reduction, in most cases doing the main operation and then adding elements with `operator+` leads to the same number of constraints. For example, the following two cases produce the same amount of constraints:

- Case 1:

```
fq value("0000000000000fffffffffffffffffffffffffffffffffffffffffffffffffff");
fq_ct a = fq_ct_from_witness(&builder, &value);
fq_ct c = a.sqradd({ a, a, a, a });
```

- Case 2:

```
fq value("0000000000000fffffffffffffffffffffffffffffffffffffffffffffffffff");
fq_ct a = fq_ct_from_witness(&builder, &value);
fq_ct c = a.sqradd({}) + a + a + a + a;
```

Recommendation: Removing all the method arguments that introduce vectors of elements to be added after the main operation would simplify the implementation of the `bigfield` library without growing the number of generated constraints in most cases.

This simplification can be combined with the usage of an optimized `sum` method to do the sum of elements in an optimized way with a simpler API.

Aztec: Acknowledged. We won't follow this suggestion, but we'll limit the number of elements that can be used in the vectors (in a fix for a different issue).

3.5.2 Simplification / optimization of `sum`

Severity: Informational

Context: bigfield_impl.hpp#L754

Description: The `sum` method builds a binary tree of additions from its `terms` using the `operator+`.

A binary tree has $n - 1$ non-leaf nodes, which means this approach leads to $n - 1$ `bigfield` additions. This is the same number of additions we would get with a sequential sum like this (which as a simpler implementation):

```
auto result = terms[0];
for (size_t i = 1; i < terms.size(); i++) {
        result = result + terms[i];
}
```

Moreover, through the code we can see functions that take vectors of `bigfield` to be summed which use the `field::add_two` method to reduce the amount of constraints.

Recommendation: The first option is to keep the same level of optimization but simplify the `sum` method by using a sequential approach like the one shown above.

The second option is to reimplement an optimized `sum` by using `field::add_two` to reduce the amount of constraints following the pattern used in other methods like `unsafe_evaluate_multiply_add`. Moreover, since this pattern is used in various places of the code base there's an opportunity of code deduplication by writing reusable functions that sum fields.

3.5.3 Missing `self_reduce` in `assert_equal`

Severity: Informational

Context: bigfield_impl.hpp#L1900

Description: The constraints `assert_equal` aren't guaranteed to pass when `other` is a constant and `this` hasn't been reduced. This happens because the constraints only pass if this is in "*canonical form*" (all the binary limbs are small and they encode a value smaller than the emulated field).

Recommendations: The first option is to call `self_reduce` on this to guarantee that the prover can encode `this` in the "canonical form" that allows the constraints to pass.

The second option would be to document this method explaining that when `other` is constant, `this` should be in "*canonical form*".

Considering that other "assert" methods like `assert_less_than` unconditionally call `self_reduce`, the first option would be preferable for consistency.

Note that the code already contains a `TODO` and an open issue about this.

3.5.4 Duplicate code in `assert_less_than` and `assert_is_in_field`

Severity: Informational

Context: bigfield_impl.hpp#L1812

Description: The `assert_less_than` and `assert_is_in_field` methods contain the same logic (implemented by very similar code) with the only difference that the upper limit in `assert_less_than` is an argument and the upper limit in `assert_is_in_field` is fixed.

Recommendations: Rewrite `assert_is_in_field` as a call to `assert_less_than(modulus_u512.lo)`. This will reduce the amount of code which improves the reviewability of the code base.

Note that the code already contains a `TODO` comment about this suggestion.

3.5.5 Operator / overloading should default to checking `divisor != 0`

Severity: Informational

Context: bigfield_impl.hpp#L730-L735

Description: Defaulting the / operator to not include the `divisor_non_zero` check is concerning in our opinion.

It's so easy for a lib user to not take this into account while writing circuits. Thus ending in situations where honest prover can't prove a circuit, or a bug can be originated.

Also, this is only used in one place:

```
bool_t<Builder> ecdsa_verify_signature(const stdlib::byte_array<Builder>& message,
    const G1& public_key,
    const ecdsa_signature<Builder>& sig)
{
```

Where we perform the following:

```
Fr z(hashed_message);
z.assert_is_in_field();

Fr r(sig.r);
// force r to be < secp256k1 group modulus, so we can compare with `result_mod_r` below
r.assert_is_in_field();

Fr s(sig.s);

// r and s should not be zero
r.assert_is_not_equal(Fr::zero());
s.assert_is_not_equal(Fr::zero());

// s should be less than |Fr| / 2
// Read more about this at: https://www.derpturkey.com/inherent-malleability-of-ecdsa-signatures/amp/
s.assert_less_than((Fr::modulus + 1) / 2);

Fr u1 = z / s;
Fr u2 = r / s;
```

Here, we check non-zero equality. So all is ok. But considering that / operator overloading is only used within this function. And, that it "*hides*" the fact that the user needs to check or not the 0 on the divisor.

Recommendation: It's better to either not implement operator overloading for division or do it with the `0` check ON by default.

In this way, all the possible mistakes are prevented.

3.5.6 `BigField` constructors can create invariant-broken instances if not constrained

Severity: Informational

Context: bigfield.hpp#L98-L132

Description: These constructors are highly insecure. And should likely be defined with `unsafe` tag within the name or something similar.

The basic explanation is that the whole `bigfield` construction (and specially the ops functions associated to it) heavily rely on the invariant that a `bigfield` cannot have limbs that overflow when an operation is done. Neither one that can overflow the CRT.

This, is ensured through the fact that users that create circuits, cannot build `bigfield` variables unless they use constructors (which ensure that indeed, these invariants are not broken).

But there's one exception to this. And are the two constructors that take the limbs as input. If the user isn't aware, it is possible to break the invariants by creating bigfields that are > `k*p` or similar such that we can trigger misbehaviors.

One example is for instance this code in `recursion_constraint.cpp#L119`:

```
aggregation_elements[i] =
    bn254::BaseField(field_ct::from_witness_index(&builder, aggregation_input[4 * i]),
                     field_ct::from_witness_index(&builder, aggregation_input[4 * i + 1]),
                     field_ct::from_witness_index(&builder, aggregation_input[4 * i + 2]),
                     field_ct::from_witness_index(&builder, aggregation_input[4 * i + 3]));
    aggregation_elements[i].assert_is_in_field();
```

In here, a user can mistakenly create an element, and if `assert_is_in_field` isn't called, the bug is created.

Recommendation: A simple and easy fix would be to just add a warning for the end user or to hide this constructors from the public API.

3.5.7 Unnecessary duplicated code might lead to refactor issues

Severity: Informational

Context: bigfield_impl.hpp#L2246-L2247

Description: These 2 lines of code are duplicated for the `Plookup` and `Non-Plookup` branches of the function.

Considering that the code is common to both, and these can be computed just once, this could help prevent errors when refactoring code or similar. As both branches need the same exact values here.

Recommendation: It should be removed from inside the `if else` and placed after this chunk of code:

```
uint64_t max_lo_bits = (max_lo.get_msb() + 1);
uint64_t max_hi_bits = max_hi.get_msb() + 1;
// We want `max_lo_bits` to be even. But that means we increase the limit by one.
if ((max_lo_bits & 1ULL) == 1ULL) {
    ++max_lo_bits;
}
if ((max_hi_bits & 1ULL) == 1ULL) {
    ++max_hi_bits;
}
```

3.5.8 `get_maximum_unreduced_value` is not correctly adjusted and wrongly justified

Severity: Informational

Context: bigfield.hpp#L378-L386

Description/Recommendation: This code is quite convoluted and it also looses an extra bit of space when it's not needed. Also, the comments that appear on it are wrong.

The comment should be updated as follows:

```
diff --git a/barretenberg/cpp/src/barretenberg/stdlib/primitives/bigfield/bigfield.hpp
↪ b/barretenberg/cpp/src/barrete
nberg/stdlib/primitives/bigfield/bigfield.hpp
index 4ec56921c..a467c133f 100644
--- a/barretenberg/cpp/src/barretenberg/stdlib/primitives/bigfield/bigfield.hpp
+++ b/barretenberg/cpp/src/barretenberg/stdlib/primitives/bigfield/bigfield.hpp
@@ -377,12 +377,11 @@ template <typename Builder, typename T> class bigfield {

     static constexpr uint512_t get_maximum_unreduced_value(const size_t num_products = 1)
     {
-        // return (uint512_t(1) << 256);
+        // This = `T * n = 2^272 * |BN(Fr)|` So this equals n*2^t
```

Also, by using get_msb and later calling -1, we are 2 bits under the limit when we should just be one.

We need the max unreduced value to be able to multiplied by itself satisfying: $\max * \max$ T · n. For that, we can have the less restrictive bound:

```
diff --git a/barretenberg/cpp/src/barretenberg/stdlib/primitives/bigfield/bigfield.hpp
↪ b/barretenberg/cpp/src/barretenberg/stdlib/primitives/bigfield/bigfield.hpp
index 4ec56921c..a467c133f 100644
--- a/barretenberg/cpp/src/barretenberg/stdlib/primitives/bigfield/bigfield.hpp
+++ b/barretenberg/cpp/src/barretenberg/stdlib/primitives/bigfield/bigfield.hpp
@@ -377,12 +377,11 @@ template <typename Builder, typename T> class bigfield {

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);
+ uint64_t maximum_product_bits = maximum_product.get_msb();
+ return (uint512_t(1) << (maximum_product_bits >> 1));
}
```

Which in essence, looses 1 less bit. Specially since `get_msb` already returns a maximum bound 1 bit under the max.

3.5.9 Unnecessary duplicated code in `assert_equal`

Severity: Informational

Context: bigfield_impl.hpp#L1918

Description/Recommendation: The entire `if` block (lines 1918-1939) is duplicated from lines 1895-1916. The three conditions in this `if` block are already checked previously so this has no effect and could be removed.

3.5.10 Extra `+ 1` in Limb `maximum_value` when created from a constant

Severity: Informational

Context: bigfield.hpp#L32, bigfield.hpp#L41

Description: The `Limb` in `bigfield` has a constructor that can take different paths depending on whether the input is a constant or a witness. For the constant case, the `maximum_value` stored for the limb has an off by one error in bigfield.hpp#L32.

The `+ 1` is not necessary. This doesn't create any soundness issue because the `maximum_value` is used to decide when to reduce the `bigfield`, so in the worst case a reduction would happen earlier than strictly necessary.

Similarly the auxiliary formatter method shows `v < maximum_value` which can be misleading, as the `maximum_value` is a possible value that `v` can take, in bigfield.hpp#L41.

Recommendation:

1. Remove the `+1` in bigfield.hpp#L32.
2. Update the formatter method representation in bigfield.hpp#L41 to be like this:

```
os << "{ " << a.element << " <= " << a.maximum_value << " }";
```