# Aztec Barretenberg Proof System

## Security Review

Cantina Managed review by:
**CPerezz**, Lead Security Researcher
**Jakub Heba**, Associate Security Researcher

November 10, 2025

# 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 level | Impact: High | Impact: Medium | Impact: Low |
|---|---|---|---|
| **Likelihood: high** | Critical | High | Medium |
| **Likelihood: medium** | High | Medium | Low |
| **Likelihood: low** | Medium | Low | Low |

### 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 are 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 28th to Oct 4th the Cantina team conducted a review of aztec-packages on commit hash 72f52e7b. The team identified a total of **15** issues:

**Issues Found**

| Severity | Count | Fixed | Acknowledged |
|---|---|---|---|
| Critical Risk | 2 | 2 | 0 |
| High Risk | 0 | 0 | 0 |
| Medium Risk | 3 | 3 | 0 |
| Low Risk | 5 | 5 | 0 |
| Gas Optimizations | 0 | 0 | 0 |
| Informational | 5 | 5 | 0 |
| **Total** | **15** | **15** | **0** |

## 2.1 Scope

The security review had the following components in scope for aztec-packages on commit hash 72f52e7b:

```
barretenberg/cpp/src/barretenberg/stdlib/primitives
├── bool
│   ├── bool.cpp
│   ├── bool.hpp
│   └── bool.test.cpp
└── byte_array
    ├── byte_array.cpp
    ├── byte_array.hpp
    └── byte_array.test.cpp
```

# 3 Findings

## 3.1 Critical Risk

### 3.1.1 Size Constructor: Unconstrained Byte Creation & assignment

**Severity:** Critical Risk

**Context:** byte_array.cpp#L21-L25

**Description:** The size constructor `byte_array(Builder*, size_t n)` creates n default field elements without applying any range constraints. This allows creation of ”byte” values that exceed the valid 8-bit range (0-255).

Attack Vector:

- Prover creates `byte_array` using size constructor: `byte_array_ct arr(&builder, 4)`.
- Prover assigns witness values > 255: `arr[0] = witness_ct(&builder, 256)`.
- No range constraint prevents this assignment.
- Circuit verification passes with invalid ”byte” values.
- Prover can inject arbitrary large values as ”bytes”.

It seems that the codebase assumes an invariant which is that byte arrays won't be internally modified or touched. Which is not documented nor told to the user. And it's also not a realistic expectation.

**Proof of Concept:**

```
TYPED_TEST(ByteArraySoundnessExploit, DISABLED_SizeConstructorBypassesRangeConstraints)
{
    using Builder = TypeParam;
    using field_ct = typename TestFixture::field_ct;
    using witness_ct = typename TestFixture::witness_ct;
    using byte_array_ct = typename TestFixture::byte_array_ct;

    Builder builder;

    // Create byte_array with size constructor - NO constraints applied
    byte_array_ct arr(&builder, 4);

    // EXPLOIT: Assign values > 255 without any range constraints
    arr[0] = witness_ct(&builder, 256);  // Should be invalid for a "byte"
    arr[1] = witness_ct(&builder, 512);  // Way larger than a byte
    arr[2] = witness_ct(&builder, 1024); // ditto
    arr[3] = witness_ct(&builder, 100);  // Valid byte

    // These "bytes" are NOT constrained to 8 bits!
    // No create_range_constraint(8) was ever called on them

    // Convert to field - uses these invalid "bytes" in computation
    field_ct result = static_cast<field_ct>(arr);

    // Expected if bytes were valid: (256*256^3 + 512*256^2 + 1024*256 + 100)
    // But these aren't valid bytes at all!
    uint256_t expected = uint256_t(256) * (uint256_t(1) << 24) +
                         uint256_t(512) * (uint256_t(1) << 16) +
                         uint256_t(1024) * (uint256_t(1) << 8) +
                         uint256_t(100);

    EXPECT_EQ(result.get_value(), fr(expected));

    // THE EXPLOIT: Circuit passes verification even with invalid "bytes"!
    // This should FAIL but it PASSES because no constraints were added
    bool verified = CircuitChecker::check(builder);
    EXPECT_TRUE(verified);  // □ VULNERABILITY: Should reject but accepts!
}
```

**Recommendation:**

1. Remove this constructor. It doesn't really add any value to have it. And allows circuit builder to bypass checks without knowing.
2. Apply Constraints to Defaults.

```
byte_array(Builder* parent_context, const size_t n)
    : context(parent_context)
    , values(n)
{
    for (auto& value : values) {
        // Default field_t() is constant 0, which is valid
        // But require explicit constraint to prevent misuse
        value.create_range_constraint(8, "byte_array: size constructor byte");
    }
}
```

3. Make Assignment Operator Apply Constraints.

```
field_t<Builder>& operator[](const%20size_t%20index) {
    BB_ASSERT_LT(index, values.size());
    // On assignment, ensure constraint exists
    values[index].create_range_constraint(8);
    return values[index];
}
```

**Aztec Labs:** Fixed in PR 17838.

**Cantina Managed:** Fix verified.

### 3.1.2 `bytes_t` constructors are underconstrained allowing to bypass the checks

**Severity:** Critical Risk

**Context:** byte_array.cpp#L194-L204

**Description:** The `bytes_t constructors (byte_array(Builder*, bytes_t const&)` and `byte_array(Builder*, bytes_t&&))` directly copy a vector of `field_t` elements without applying any range constraints. This allows creation of byte_array instances containing "byte" values that exceed the valid 8-bit range (0-255).

Unlike the size constructor which creates default fields, these constructors accept pre-existing unconstrained fields, making the error chance higher. Thus even more direct and dangerous.

Root Cause: The constructors assume that the caller has already applied proper 8-bit range constraints to all `field_t` elements in the input vector. However:

- This assumption is not documented in the API.
- There is no runtime validation to enforce it.
- The type system provides no protection (a `vector<field_t>` can contain any field values).
- Circuit developers may not realize this requirement.

This violates the fundamental invariant: All elements in a `byte_array` must be 8-bit constrained.

Comparison with Safe Constructors:

- SECURE: `vector<uint8_t>` constructor:

```
byte_array(Builder* parent_context, std::vector<uint8_t> const& input) {
    for (size_t i = 0; i < input.size(); ++i) {
        field_t<Builder> value(witness_t(context, input[i]));
        value.create_range_constraint(8, "...");  // CONSTRAINT APPLIED
        values[i] = value;
    }
}
```

- VULNERABLE: `bytes_t` constructor:

```
byte_array(Builder* parent_context, bytes_t const& input)
    : values(input)  // NO VALIDATION OR CONSTRAINTS
{}
```

Direct Exploitation:

1. Prover creates unconstrained witness values: `witness_ct(&builder, 1000)`.
2. Prover places them in a vector:

```
std::vector<field_ct> malicious = {witness_ct(&builder, 1000), ...}
```

3. Prover passes to `bytes_t` constructor: `byte_array_ct arr(&builder, malicious)`.
4. No constraints are applied - values > 255 are accepted.
5. Circuit verification passes with invalid "bytes".

**Proof of Concept:**

1. Direct Unconstrained Field Injection:

```
TYPED_TEST(ByteArraySoundnessExploit, DISABLED_BytesConstructorBypassesConstraints)
{
    using Builder = TypeParam;
    using field_ct = typename TestFixture::field_ct;
    using witness_ct = typename TestFixture::witness_ct;
    using byte_array_ct = typename TestFixture::byte_array_ct;

    Builder builder;

    // Create unconstrained field elements (values > 255)
    std::vector<field_ct> unconstrained_fields;
    unconstrained_fields.push_back(witness_ct(&builder, 1000));  // NOT a byte!
    unconstrained_fields.push_back(witness_ct(&builder, 2000));  // NOT a byte!
    unconstrained_fields.push_back(witness_ct(&builder, 3000));  // NOT a byte!

    // EXPLOIT: Create byte_array from unconstrained fields
    byte_array_ct arr(&builder, unconstrained_fields);

    EXPECT_EQ(arr.size(), 3UL);

    // These "bytes" have NO range constraints!
    EXPECT_EQ(arr[0].get_value(), fr(1000));  // Not a valid byte
    EXPECT_EQ(arr[1].get_value(), fr(2000));  // Not a valid byte
    EXPECT_EQ(arr[2].get_value(), fr(3000));  // Not a valid byte

    // THE EXPLOIT: Circuit accepts invalid "bytes"
    bool verified = CircuitChecker::check(builder);
    EXPECT_TRUE(verified);  // VULNERABILITY: Accepts unconstrained values!

    // PROOF OF VULNERABILITY:
    // The bytes_t constructor does not apply create_range_constraint(8) to input
    ↪  fields.
    // Callers must manually ensure fields are constrained, which is error-prone.
}
```

2. Corruption via `write_at()`:

```
TYPED_TEST(ByteArraySoundnessExploit, DISABLED_WriteAtOverwritesWithUnconstrained)
{
    using Builder = TypeParam;
    using field_ct = typename TestFixture::field_ct;
    using witness_ct = typename TestFixture::witness_ct;
    using byte_array_ct = typename TestFixture::byte_array_ct;
```

```
    Builder builder;

    // Step 1: Create properly constrained byte array
    std::vector<uint8_t> valid_bytes = { 0x01, 0x02, 0x03, 0x04 };
    byte_array_ct valid_arr(&builder, valid_bytes);

    // These bytes ARE properly constrained (via vector<uint8_t> constructor)
    // Each has create_range_constraint(8) applied

    // Step 2: Create malicious byte array with unconstrained fields
    std::vector<field_ct> malicious_fields;
    malicious_fields.push_back(witness_ct(&builder, 500));  // NOT a byte!
    malicious_fields.push_back(witness_ct(&builder, 600));  // NOT a byte!

    byte_array_ct malicious_arr(&builder, malicious_fields);

    // Step 3: EXPLOIT - Overwrite constrained bytes with unconstrained ones
    valid_arr.write_at(malicious_arr, 1);  // Overwrite at index 1

    // Now valid_arr contains:
    // [0] = 0x01 (still constrained)
    // [1] = 500  (UNCONSTRAINED!)
    // [2] = 600  (UNCONSTRAINED!)
    // [3] = 0x04 (still constrained)

    EXPECT_EQ(valid_arr[0].get_value(), fr(0x01));
    EXPECT_EQ(valid_arr[1].get_value(), fr(500));  // Invalid "byte"
    EXPECT_EQ(valid_arr[2].get_value(), fr(600));  // Invalid "byte"
    EXPECT_EQ(valid_arr[3].get_value(), fr(0x04));

    // THE EXPLOIT: Circuit accepts corrupted array
    bool verified = CircuitChecker::check(builder);
    EXPECT_TRUE(verified);  // VULNERABILITY: Should reject but accepts!

    // PROOF OF VULNERABILITY:
    // write_at() preserves field element references, including their (lack of)
    ↪  constraints.
    // A properly constrained byte_array can be corrupted by writing unconstrained
    ↪  fields.
}
```

**Recommendations:**

1. Apply Constraints in Constructor (SAFEST):

```
template <typename Builder>
byte_array<Builder>::byte_array(Builder* parent_context, bytes_t const& input)
    : context(parent_context)
    , values(input)
{
    // Validate that all input fields are properly constrained
    for (auto& value : values) {
        value.create_range_constraint(8, "byte_array: bytes_t constructor -
        ↪  unconstrained input");
    }
}

template <typename Builder>
byte_array<Builder>::byte_array(Builder* parent_context, bytes_t&& input)
    : context(parent_context)
    , values(std::move(input))
{
    // Validate that all input fields are properly constrained
    for (auto& value : values) {
        value.create_range_constraint(8, "byte_array: bytes_t constructor -
        ↪  unconstrained input");
```

```
    }
}
```

2. Apply assignment constraints:

```
field_t<Builder>& operator[](const%20size_t%20index) {
    BB_ASSERT_LT(index, values.size());
    // On assignment, ensure constraint exists
    values[index].create_range_constraint(8);
    return values[index];
}
```

If no changes want to be done, I'd at least heavily document that these constructors don't apply constraints. As otherwise the circuit developer is completelly mercyless when calling it.

**Aztec Labs:** Fixed in PR 17838.

**Cantina Managed:** Fix verified.

## 3.2 Medium Risk

### 3.2.1 32-byte conversion blocked by C++ assertion (not circuit constraint)

**Severity:** Medium Risk

**Context:** byte_array.cpp#L242

**Description:** Operator `field_t()` has `ASSERT(bytes < 32)` preventing valid 32-byte conversions.

- Documentation claims ”injective when size < 32”.
- 32-byte case IS supported by `field_t` constructor (with modulus check).
- But conversion FROM 32-byte array is blocked by C++ assertion.
- Creates asymmetry: can construct 32-byte array from `field`, but can't convert back.
- Blocks valid round-trip: `field` → 32-byte array → `field` in the most critical security code (modulus boundary check) which cannot be tested via fuzzer creating a dead zone in test coverage for most security-critical functionality.

**Proof of Concept:**

```
TYPED_TEST(ByteArraySoundnessExploit, DISABLED_ThirtyTwoByteConversionBlocked)
{
    using Builder = TypeParam;
    using field_ct = typename TestFixture::field_ct;
    using witness_ct = typename TestFixture::witness_ct;
    using byte_array_ct = typename TestFixture::byte_array_ct;

    Builder builder;

    // Create a valid field value at modulus boundary
    uint256_t valid_value = fr::modulus - 1;  // Largest valid field element

    field_ct field_val = witness_ct(&builder, valid_value);

    // STEP 1: Create 32-byte array from field - THIS WORKS
    // This triggers the complex modulus boundary check (lines 157-189)
    byte_array_ct arr_32_bytes(field_val, 32);

    EXPECT_EQ(arr_32_bytes.size(), 32UL);

    // All bytes are properly constrained to 8 bits
    // Modulus boundary was validated with borrow bit technique
    // This is the MOST security-critical code in the entire implementation

    // STEP 2: Try to convert back to field - THIS FAILS
    field_ct result = static_cast<field_ct>(arr_32_bytes);
```

```
    // ASSERTION FAILURE: ASSERT(bytes < 32) on line 242

    // PROOF OF CORRECTNESS ISSUE:
    // 1. The assertion blocks valid 32-byte to field conversion
    // 2. This creates asymmetry: field->32bytes works, 32bytes->field doesn't
    // 3. The most critical security code (modulus check) is untestable via fuzzer
    // 4. Fuzzer's to_field_t() cannot test 32-byte case (see byte_array.fuzzer.hpp:391)
    //
    // SECURITY IMPACT:
    // - Zero continuous fuzzer coverage for most critical constraint logic
    // - Regression bugs in modulus boundary check would not be caught
    // - Lines 157-189 are mathematically correct but have NO ongoing validation
}
```

**Recommendation:** Just remove the assertion. This is safe because:

- All bytes in the array already have 8-bit range constraints (applied in constructor).
- The conversion logic itself is mathematically valid for all sizes.

**Aztec Labs:** Fixed in PR 17838.

**Cantina Managed:** Fix verified.

### 3.2.2 `write_at()` can propagate unconstrained fields and fail at `field_t` reconstruction from bytes

**Severity:** Medium Risk

**Context:** byte_array.cpp#L262-L273

**Description:** The `write_at()` function overwrites a portion of the `byte_array` with contents from another `byte_array`. It performs no validation that the source array's fields are properly 8-bit constrained.

This is a pure assignment operation - it directly copies `field_t` references from `other` into `this`, including any constraint references (or lack thereof). If `other` was created using an unsafe constructor (size or bytes_t), its fields may be unconstrained. When copied via `write_at()`, these unconstrained fields corrupt the target array, even if it was previously properly constrained.

`write_at()` assumes the invariant that all byte_arrays contain only 8-bit constrained fields. This assumption is violated when:

1. Source array created via size constructor → fields may be unconstrained.
2. Source array created via `bytes_t` constructor → fields may be unconstrained.
3. Source array modified via `operator[]` after `size` constructor → fields may be unconstrained.

The function trusts its input without validation, making it a propagation vector for the constructor vulnerabilities.

**Proof of Concept:**

```
// Start with SAFE constructor
byte_array_ct arr(&builder, {0x01, 0x02, 0x03, 0x04});  // All constrained

// Get unconstrained witness
field_ct bad_input = witness_ct(&builder, 500);  // No constraint!

// Corrupt via operator[] assignment
arr[1] = bad_input;  // Overwrites constrained byte!

// Now arr[1] is unconstrained, even though we used safe constructor!

// write_at() propagates this corruption
byte_array_ct target(&builder, {0xAA, 0xBB});
target.write_at(arr, 0);  // Copies the unconstrained field!
```

**Recommendation:** No good fix exists for this. As even if all the constructors are fixed, and the `operator[]` applies a constraint. The API still allows to make this mistake. The only way would be to risk duplicating range-constraints for bytes sometimes (as assignation operations would duplicate constraints in some cases but prevent foreign `field_t` unconstraint assignments).

**Aztec Labs:** Fixed in PR 17838.

**Cantina Managed:** Fix verified.

### 3.2.3 No checks nor documentation that warns the circuit developer about the non-addition of any constraints when using `field_t` constructor

**Severity:** Medium Risk

**Context:** byte_array.cpp#L234-L252

**Description:** The conversion to `field_t` assumes bytes are properly constrained, but doesn't verify anything. Looking at the conversion (`byte_array.cpp#L243-L257`):

```
template <typename Builder> byte_array<Builder>::operator field_t<Builder>() const
{
    const size_t bytes = values.size();
    // Commented out: ASSERT(bytes < 32);

    std::vector<field_t<Builder>> scaled_values;
    for (size_t i = 0; i < bytes; ++i) {
        const field_t<Builder> scaling_factor(one << (8 * (bytes - i - 1)));
        scaled_values.push_back(values[i] * scaling_factor);  // No constraint check here
    }
    return field_t<Builder>::accumulate(scaled_values);
}
```

The conversion trusts that bytes are already constrained. This creates a constraint assumption propagation issue:

- Constructor assumes: "I don't need to constrain, someone else did".
- Conversion assumes: "Bytes are already constrained by constructor" or operator that assigned, wrote, sliced...
- Result: NOTHING constrains the bytes.

This is a clear issue that makes the whole codebase quite fragile. And it's the fact that it's almost impossible to prevent the user from introducing bytes within a byte array that come from other places of the circuit. These, won't be constrained to be a byte. User needs to know that the conversion fn won't actually apply any constraints which is counter intuitive.

And this combined with the constructors that don't constrain + the operators for arrays which propagate the misscontstraints, it makes it quite easy to mess up when writing a circuit.

**Proof of Concept:**

```
TYPED_TEST(ByteArraySoundnessExploit,
↪   DISABLED_ConversionToFieldDoesntVerifyConstraints_UnconstrainedConstructor)
{
    using Builder = TypeParam;
    using field_ct = typename TestFixture::field_ct;
    using witness_ct = typename TestFixture::witness_ct;
    using byte_array_ct = typename TestFixture::byte_array_ct;

    Builder builder;

    // BUG: Create byte_array with size constructor - NO constraints
    byte_array_ct arr(&builder, 4);

    // Assign invalid "bytes" (values > 255)
    arr[0] = witness_ct(&builder, 1000);  // NOT a byte!
    arr[1] = witness_ct(&builder, 2000);  // NOT a byte!
```

```
    arr[2] = witness_ct(&builder, 3000);  // NOT a byte!
    arr[3] = witness_ct(&builder, 4000);  // NOT a byte!

    // Convert to field_t - this SHOULD fail but doesn't
    // The conversion blindly multiplies by scaling factors:
    // result = 1000*(256^3) + 2000*(256^2) + 3000*256 + 4000
    field_ct malicious_field = static_cast<field_ct>(arr);

    // Calculate what the value WOULD be if these were valid bytes
    uint256_t expected_if_unchecked = uint256_t(1000) * (uint256_t(1) << 24) +
                                      uint256_t(2000) * (uint256_t(1) << 16) +
                                      uint256_t(3000) * (uint256_t(1) << 8) +
                                      uint256_t(4000);

    EXPECT_EQ(malicious_field.get_value(), fr(expected_if_unchecked));

    // THE EXPLOIT: Circuit accepts this invalid conversion!
    bool verified = CircuitChecker::check(builder);
    EXPECT_TRUE(verified);  // VULNERABILITY: Should reject but accepts!

    // PROOF OF VULNERABILITY:

    // WITHOUT verifying that values[i] is constrained to [0, 255]
    //.
    // The conversion assumes byte constraints exist but doesn't enforce them

}
```

```
TYPED_TEST(ByteArraySoundnessExploit,
↪   DISABLED_ConversionToFieldDoesntVerifyConstraints_ForeignFieldAssignment)
{
    using Builder = TypeParam;
    using field_ct = typename TestFixture::field_ct;
    using witness_ct = typename TestFixture::witness_ct;
    using byte_array_ct = typename TestFixture::byte_array_ct;

    Builder builder;

    // Step 1: Create a properly constrained byte_array
    std::vector<uint8_t> valid_bytes = { 0x01, 0x02, 0x03, 0x04 };
    byte_array_ct arr(&builder, valid_bytes);

    // Verify initial array is properly constrained
    for (size_t i = 0; i < 4; ++i) {.
        EXPECT_LT(arr[i].get_value(), fr(256));
    }.

    // Step 2: Create "foreign" field_t elements (unconstrained, different context)
    // Simulate unconstrained field elements that represent invalid "bytes"
    field_ct foreign_byte1 = witness_ct(&builder, 5000);  // NOT a byte!
    field_ct foreign_byte2 = witness_ct(&builder, 6000);  // NOT a byte!

    // These are NOT constrained to be bytes - they're just raw witnesses
    // In a real attack, these could come from a different part of the circuit

    // Step 3: BUG - Assign foreign fields to byte_array positions
    arr[1] = foreign_byte1;  // Overwrite constrained byte with unconstrained field
    arr[2] = foreign_byte2;  // Overwrite constrained byte with unconstrained field

    // Now arr contains:
    // [0] = 0x01 (constrained)
    // [1] = 5000 (UNCONSTRAINED)
    // [2] = 6000 (UNCONSTRAINED)
    // [3] = 0x04 (constrained)

    // Step 4: Convert to field_t - uses unconstrained values in computation!
```

```
    field_ct malicious_field = static_cast<field_ct>(arr);

    // Expected value if these were treated as valid inputs:
    // result = 1*(256^3) + 5000*(256^2) + 6000*256 + 4.
    uint256_t expected_malicious = uint256_t(1) * (uint256_t(1) << 24) +
                                   uint256_t(5000) * (uint256_t(1) << 16) +
                                   uint256_t(6000) * (uint256_t(1) << 8) +
                                   uint256_t(4);

    EXPECT_EQ(malicious_field.get_value(), fr(expected_malicious));

    bool verified = CircuitChecker::check(builder);
    EXPECT_TRUE(verified);  // BUG: Should reject but accepts!
    // NO VERIFICATION that values[i] has create_range_constraint(8) applied!
}
```

**Recommendation:**

- Force all constructors (and maybe assignment operators too) to apply range-constraints. Even at the risk of duplication.
- Introduce the range-constraint in this fn. Thus removing any options to screw up to the circuit developer.
- If we don't want to duplicate constraints. The suggestion would be to heavily document the fact that this fn assumes an invariant of "All elements within a byte array are constrained to be bytes already". Which as we have seen, is not true. And explain the exceptions and how to deal with them.

**Aztec Labs:** Fixed in PR 17838.

**Cantina Managed:** Fix verified.

## 3.3 Low Risk

### 3.3.1 Const `operator[]`: Insufficient bounds checking

**Severity:** Low Risk

**Context:** byte_array.hpp#L54-L65

**Description:** The const version of `operator[]` has inconsistent and insufficient bounds checking compared to the non-const version.

- Const version: Only checks `assert(values.size() > 0)` - validates array is non-empty.
- Non-const version: Checks `BB_ASSERT_LT(index, values.size())` - validates index is in bounds.

This asymmetry means:

- Reading from a const byte_array with an out-of-bounds index → Undefined Behavior.
- Writing to a non-const byte_array with an out-of-bounds index → Assertion failure (caught).

The const `operator[]` was likely written assuming "if the array is non-empty, any access is valid", which is incorrect. The assertion only prevents accessing empty arrays, not out-of-bounds indices.

**Proof of Concept:**

```
TYPED_TEST(ByteArraySoundnessExploit, DISABLED_ConstOperatorIndexOutOfBounds)
{
    using Builder = TypeParam;
    using byte_array_ct = typename TestFixture::byte_array_ct;

    Builder builder;

    // Create 1-element array
    std::vector<uint8_t> single_byte = { 0x42 };
    const byte_array_ct arr(&builder, single_byte);  // const reference

    EXPECT_EQ(arr.size(), 1UL);
```

```
    // Valid access
    auto byte0 = arr[0];  // This works
    EXPECT_EQ(byte0.get_value(), fr(0x42));

    // EXPLOIT: Out of bounds access via const operator[]
    // The assertion only checks `assert(values.size() > 0)`
    // It does NOT check `assert(index < values.size())`

    auto byte5 = arr[5];  // UNDEFINED BEHAVIOR!
    // Assertion passes: size() > 0 is true (size = 1)
    // But index 5 is out of bounds for size 1 array
    // Result: Reads garbage memory or crashes

    // Const operator[] has weaker bounds checking than non-const version.
    // This is a code quality issue, not a circuit soundness issue.

    // What SHOULD happen:
    // assert(5 < 1) → false → assertion failure → program terminates

    // What ACTUALLY happens:
    // assert(1 > 0) → true → assertion passes → UB!
}
```

**Recommendation:** Just add the missing assertion in the const operator.

**Aztec Labs:** Fixed in PR 17838.

**Cantina Managed:** Fix verified.

### 3.3.2 `slice()`: Empty Array Edge Case Bug

**Severity:** Low Risk

**Context:** byte_array.cpp#L278-L296

**Description:** The `slice()` functions use `BB_ASSERT_LT(offset, values.size())` to validate the offset parameter. This assertion fails for the valid case of slicing an empty array at offset 0.

The Problem:

- Assertion: `BB_ASSERT_LT(offset, values.size())`.
- When array is empty: `size() == 0`.
- When slicing from start: `offset == 0`.
- Evaluation: `BB_ASSERT_LT(0, 0)` → FALSE → Assertion fails.

Expected Behavior: Slicing an empty array at offset 0 should return an empty array (valid operation).

Actual Behavior: Assertion triggers and program terminates (invalid). This means we have a correctness bug.

Any circuit compiled with an operation that might slice an empty array, it's going to produce an un-probable circuit instance.

**Proof of Concept:**

```
TYPED_TEST(ByteArraySoundnessExploit, DISABLED_EmptySliceTriggersAssertion)
{
    using Builder = TypeParam;
    using byte_array_ct = typename TestFixture::byte_array_ct;

    Builder builder;

    // Create empty byte array
    std::vector<uint8_t> empty_vec;
    byte_array_ct empty_arr(&builder, empty_vec);
```

```
    EXPECT_EQ(empty_arr.size(), 0UL);

    // ========================================
    // BUG: Attempt to slice at offset 0
    // ========================================
    // This is a semantically VALID operation:
    // - Slicing empty array from start → should return empty array
    // - Equivalent to: empty_array[0..0] → []

    // But it triggers: BB_ASSERT_LT(0, 0) which evaluates to FALSE
    auto sliced = empty_arr.slice(0);  // ASSERTION FAILURE!

    // What SHOULD happen:
    // - Returns empty byte_array (size 0)
    // - No assertion failure

    // This is a CORRECTNESS bug (blocks valid operations), not a soundness
    // issue (no circuit impact).
}
```

**Recommendation:** Just update the assertion:

```
template <typename Builder>
byte_array<Builder> byte_array<Builder>::slice(size_t offset) const
{
    BB_ASSERT_LTE(offset, values.size());  // Allow offset == size
    return byte_array(context, bytes_t(values.begin() + static_cast<ptrdiff_t>(offset),
    ↪  values.end()));
}

template <typename Builder>
byte_array<Builder> byte_array<Builder>::slice(size_t offset, size_t length) const
{
    BB_ASSERT_LTE(offset, values.size());  // Allow offset == size
    BB_ASSERT_LTE(length, values.size() - offset);
    auto start = values.begin() + static_cast<ptrdiff_t>(offset);
    auto end = values.begin() + static_cast<ptrdiff_t>((offset + length));
    return byte_array(context, bytes_t(start, end));
}
```

**Aztec Labs:** Fixed in PR 17838.

**Cantina Managed:** Fix verified.

### 3.3.3 `slice()` can propagate unconstrained fields and fail at `field_t` reconstruction from bytes

**Severity:** Low Risk

**Context:** byte_array.cpp#L278-L296

**Description:** The `slice()` function extracts a contiguous sub-range of bytes from a byte array, creating a new `byte_array` object. The function performs no constraint validation on the extracted bytes, as explicitly documented in the code comments:

```
/**
 * @brief Slice `length` bytes from the byte array, starting at `offset`.
 * Does not add any constraints
 **/
```

The `slice()` operation blindly preserves the constraint state of the source bytes:

| Source Array State | After slice() | Security Issue |
|---|---|---|
| Properly constrained (8-bit) | Stays constrained | None - safe |
| Unconstrained (arbitrary values) | Stays unconstrained | **CRITICAL** |

| | | |
|---|---|---|
| Mix of constrained/unconstrained | Preserves both | **CRITICAL** |

While this might be the intended way for the fn to be, it would be interesting to either use these kinds of functions as filters for un-constrained bytes (since the lookup is almost free). Or, otherwise, leave clearly documented that these kinds of error-propagation issues aren't tackled here.

**Proof of Concept:**

```
TEST(byte_array, slice_propagates_unconstrained_from_size_constructor)
{
    auto builder = Builder();

    // Step 1: Create unconstrained array via size constructor (vulnerability #1)
    byte_array_ct arr(&builder, 32);

    // Step 2: Assign out-of-range value via operator[] (vulnerability #3)
    arr[0] = witness_ct(&builder, 500);   // 500 > 255, but no constraint prevents this
    arr[1] = witness_ct(&builder, 1000);  // 1000 > 255

    // Step 3: Slice extracts the corrupted bytes (vulnerability #4)
    byte_array_ct sliced = arr.slice(0, 2);

    // Step 4: Verify the corruption persists
    auto value0 = uint256_t(sliced[0].get_value());
    auto value1 = uint256_t(sliced[1].get_value());

    EXPECT_EQ(value0, 500);   // Unconstrained value preserved
    EXPECT_EQ(value1, 1000);  // Unconstrained value preserved

    // Step 5: Proof verifies despite invalid byte values
    EXPECT_TRUE(CircuitChecker::check(builder));  // Should fail but passes
}
```

**Recommendation:**

- Constraint constructors.
- Constraint the slice operation itself (though at the risk of adding more lookup entries to the permutation which should not be that bad).

**Aztec Labs:** Fixed in PR 17838.

**Cantina Managed:** Fix verified.

### 3.3.4 Missing builder context validation in bool operations

**Severity:** Low Risk

**Context:** bool.cpp#L166-L168

**Finding Description:** In `&`, `|`, `^` and `==` operators, the boolean operators select context using `context ? context : other.context` but don't validate when both operands have different non-null contexts.

When both operands are non-constant, the code picks one builder (context) and then emits a gate using both operands witness indices without verifying they belong to that builder.

**Recommendation:** Before creating a gate, if both sides are non-constant, check:

```
if (!is_constant() && !other.is_constant()) {
ASSERT(context == other.context);
}
```

**Aztec Labs:** Fixed in PR 17838.

**Cantina Managed:** Fix verified.

### 3.3.5 Potential underflow in byte-array reverse on empty input

**Severity:** Low Risk

**Context:** byte_array.cpp#L301-L305

**Finding Description:** The reverse routine computes an index as `bytes.size -1`. When the array is empty, this computation underflows the index variable. Although the loop does not execute and no immediate crash occurs, the pattern is problematic, can confuse readers and sanitizers, and could become a bug if the loop body or guards change.

**Recommendation:** Return early for empty input or guard before subtracting one.

**Aztec Labs:** Fixed in PR 17838.

**Cantina Managed:** Fix verified.

## 3.4 Informational

### 3.4.1 Division by `shift` has an implicit safety assumption that isn't documented anywhere

**Severity:** Informational

**Context:** byte_array.cpp#L186-L188

**Description:** This division assumes `reconstructed_hi` is always divisible by $2^{128}$. Why does this matter?

1. If `reconstructed_hi` is NOT a multiple of $2^{128}$, the division will perform field modular division.
2. Field division computes `a / b = a * b^(-1) mod r`.
3. This would give a completely different value than integer division.

Is `reconstructed_hi` guaranteed to be a multiple of $2^{128}$?

```
std::vector<field_t> accumulator_hi;
for (size_t i = 0; i < num_bytes; ++i) {
    size_t bit_start = (num_bytes - i - 1) * 8;
    const field_t scaling_factor = one << bit_start;

    if (i < midpoint) {  // midpoint = 16 for num_bytes = 32
        accumulator_hi.push_back(scaling_factor * byte);
    }
}
const field_t reconstructed_hi = field_t::accumulate(accumulator_hi);
```

For `num_bytes = 32`, `midpoint = 16`:

- Bytes 0-15 go into `accumulator_hi`.
- Scaling factors: $2^{248}$, $2^{240}$, ..., $2^{136}$, $2^{128}$.

Rewording:

```
reconstructed_hi = Σ(i=0 to 15) byte[i] * 2^(248 - 8i)
                 = 2^128 * Σ(i=0 to 15) byte[i] * 2^(120 - 8i)
                 = 2^128 * [high_value]
```

So, YES, `reconstructed_hi` is ALWAYS a multiple of `2^128` by construction. But there's an issue with this. This is a dangerous implicit assumption that:

1. Is not documented in comments.
2. Could break if the accumulator logic changes.

**Recommendation:** So while this depends clearly on a refactor of the accumulator being done, it would be more cautious to add a bit of justification on why this divisibility is always assumed to be existing. As well as the invariants this assumes such that the whole implementation works and is sound. This is mentioned here because there's NO CONSTRAINT on the divisibility of the scaling factors.

```
// SAFETY: reconstructed_hi is always a multiple of 2^128 by construction
// (bytes 0-15 all have scaling factors ≥ 2^128)
const field_t diff_hi = (-reconstructed_hi / shift).add_two(s_hi, -overlap);
```

**Aztec Labs:** Fixed in PR 17838.

**Cantina Managed:** Fix verified.

#### 3.4.2 `get_string()`: Undefined behavior with non-ASCII

**Severity:** Informational

**Context:** byte_array.cpp#L324-L337

**Description:** If `get_value()` returns bytes > 127 (i.e., non-ASCII values):

- `std::string` constructor interprets as `char` (signed).
- Values 128-255 become negative `char` values.
- This is introduces a correctness issue.

Example:

```
byte_array_ct arr(&builder, {0x00, 0x7F, 0x80, 0xFF});
std::string s = arr.get_string();

s[0] = 0x00 (char)  // OK
s[1] = 0x7F (char)   // OK
s[2] = -128 (char)   // Negative!
s[3] = -1 (char)       // Negative!
```

As you can see here, the fn doesn't actually mention that non-ASCII characters aren't valid. And will end up producing a correctness issue depending on what's passed to the circuit builder.

**Proof of Concept:** Since this is only used in tests, we just wanted to inform. The proof of concept would be trivial but lacks any interest.

**Recommendation:** Simply document the function correctly or restrict it to ASCII characters only.

**Aztec Labs:** Fixed in PR 17838.

**Cantina Managed:** Fix verified.

#### 3.4.3 Unnecessary witness multiplied by 0

**Severity:** Informational

**Context:** bool.cpp#L523-L530

**Finding Description:** The normalization constraint for booleans wires the same witness into two gate inputs while multiplying one of them by zero. The created gate sets both `a` and `b` to `witness_index`, with `q_r = 0`. It's correct but confusing and marginally heavier to parse.

**Recommendation:** Assigning `context->zero_idx` to the `b` term would make the code more clear as the. reader doesn't have to figure out why the `witness_index` is used twice in the gate, only to realize that in one instance it's being multiplied by 0.

**Aztec Labs:** Fixed in PR 17838.

**Cantina Managed:** Fix verified.

#### 3.4.4 `rvalue` byte-array constructor performs a copy instead of a move

**Severity:** Informational

**Context:** byte_array.cpp#L200-L204

**Finding Description:** The byte-array constructor that accepts an `rvalue` container initializes the internal storage via copy construction rather than move construction. This silently incurs extra allocations and data copies, degrading performance, especially for large arrays, and violates caller expectations about move semantics.

**Recommendation:** Construct the internal storage with move semantics. Consider adding a test that checks for zero additional allocations when constructing from an rvalue.

**Aztec Labs:** Fixed in PR 17838.

**Cantina Managed:** Fix verified.

#### 3.4.5 Selector order mismatch in boolean equality gate

**Severity:** Informational

**Context:** bool.cpp#L378-L386

**Finding Description:** The `create_poly_gate` gate constructed for boolean equality passes the linear selectors in the opposite order from the conventional and documented (`q_l`, `q_r`) arrangement.

Presently both selectors happen to be equal, so behavior is unaffected, but this is fragile - if coefficients diverge in a future refactor or tools assume canonical ordering, it can lead to subtle errors.

**Recommendation:** Align the selector ordering with the codebase's standard and the derivation, and document the mapping of wires to selectors.

**Aztec Labs:** Fixed in PR 17838.

**Cantina Managed:** Fix verified.