CANTINA

Fluent: fluentbase & rwasm

Security Review

Cantina Managed review by:

Haxatron, Lead Security Researcher

Rikard Hjort, Lead Security Researcher

February 23, 2026

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

From Nov 24th to Jan 9th the Cantina team conducted a review of Fluent base and rwasm on commit hash 766c598d. The team identified a total of **47** issues:

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	3	3	0
High Risk	6	6	0
Medium Risk	14	14	0
Low Risk	12	11	1
Gas Optimizations	0	0	0
Informational	12	11	1
Total	**47**	**45**	**2**

Disclaimer: The total duration of the review amounted to 5 engineering weeks. Owing to the period during which the review was conducted, one additional week was allocated to account for observed holiday periods.

3 Findings

3.1 Critical Risk

3.1.1 Triggerable node crash via point above field modulus during `weierstrass_double` syscall

Severity: Critical Risk

Context: weierstrass_double.rs#L48-L63, weierstrass_double.rs#L124-L139

Description: Both the `weierstrass_double_handler` and `weierstrass_double_impl` do not verify that the provided input coordinates are not above the field modulus which can lead to a node crash that can be triggered trivially by providing a single invalid point to the syscall.

```rust
fn syscall_weierstrass_double_handler<E: EllipticCurve, const POINT_SIZE: usize>(
    ctx: &mut impl Store<RuntimeContext>,
    params: &[Value],
    _result: &mut [Value],
) -> Result<(), TrapCode> {
    let p_ptr: u32 = params[0].i32().unwrap() as u32;

    let mut p = [0u8; POINT_SIZE];
    ctx.memory_read(p_ptr as usize, &mut p)?;

    let result = syscall_weierstrass_double_impl::<E, POINT_SIZE>(p);
    ctx.memory_write(p_ptr as usize, &result)?;

    Ok(())
}
```

```rust
fn syscall_weierstrass_double_impl<E: EllipticCurve, const POINT_SIZE: usize>(
    p: [u8; POINT_SIZE],
) -> [u8; POINT_SIZE] {
    let (px, py) = p.split_at(p.len() / 2);
    let p_affine = AffinePoint::<E>::new(BigUint::from_bytes_le(px),
     ↪  BigUint::from_bytes_le(py));
    let result_affine = E::ec_double(&p_affine);
    let (rx, ry) = (result_affine.x, result_affine.y);
    let mut result = [0u8; POINT_SIZE];
    let mut rx = rx.to_bytes_le();
    rx.resize(POINT_SIZE / 2, 0);
    let mut ry = ry.to_bytes_le();
    ry.resize(POINT_SIZE / 2, 0);
    result[..POINT_SIZE / 2].copy_from_slice(&rx);
    result[POINT_SIZE / 2..].copy_from_slice(&ry);
    result
}
```

This is because it can cause an underflow of the `dashu::UBig` type in the underlying `sw_double` implementation in the SP1 code, which panics on an underflow.

crates/curves/src/weierstrass/mod.rs#L204-L230

```rust
pub fn sw_double(&self) -> AffinePoint<SwCurve<E>> {
//...
            let x_3n = (&slope * &slope + &p + &p - &self_x - &self_x) % &p;
//...
        }
    }
}
```

The underflow occurs on the above line, if `2 * self_x > slope * slope + 2 * p`, then an underflow can occur. Here, `p` is the field modulus of the curve which is slightly less than the maximum unsigned 48-bit integer (`0xfff..fff`). As such it is possible to specify an invalid coordinate `(0xfff...fff, 0x0)` which results in `self_x = 0xfff..fff` and `self_y = 0x0`. The result of `slope` when `self_y` is always 0 and this is because the result of `dashu_modpow` when the `slope_denominator` is 0 is 0.

Hence, the condition `2 * self_x > slope * slope + 2 * p` is trivially reachable via the point `(0xfff...fff, 0x0)` with the x-coordinate being above the field modulus `p` for any curve. The following is a proof-of-concept demonstrating the crash:

Proof of Concept: Paste the following test in `weierstrass_double.rs`

```
/// Reproduces chain halt when doubling an invalid BLS12-381 point with
/// x = 0xff..ff (above modulus) and y = 0x0.
#[test]
fn test_coords_above_modulus_halt() {
    // 96-byte affine point: [x || y], each 48 bytes.
    let mut coords_above_modulus_point = [0u8; 96];

    // Set x = 0xff..ff (48 bytes of 0xff), y stays all zeros.
    coords_above_modulus_point[..48].fill(0xff);

    let _result = syscall_bls12381_double_impl(coords_above_modulus_point);
}
```

Result:

```
---- syscall_handler::weierstrass::weierstrass_double::tests::test_coords_above_modulus_
↪  halt stdout ----

thread 'syscall_handler::weierstrass::weierstrass_double::tests::test_coords_above_modul
↪  us_halt' panicked at
↪  /home/ser/.cargo/registry/src/index.crates.io-1949cf8c6b5b557f/dashu-int-0.4.1/src/e
↪  rror.rs:20:5:
UBig result must not be negative
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace


failures:
    syscall_handler::weierstrass::weierstrass_double::tests::test_coords_above_modulus_h
     ↪  alt

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 44 filtered out;
↪  finished in 0.00s

error: test failed, to rerun pass `--lib`
```

As observed above, the test crashes because `UBig` result was negative which will cause a panic.

Recommendation: Validate that both x and y-coordinates are below the field modulus for the `weierstrass_double` syscall. We also recommend validating the same for the `weierstrass_decompress` as while there is no concrete exploit for a point above field modulus for the `weierstrass_decompress` syscall, addition validation woukd reduce attack surface.

Fluent Labs: Fixed in PR 265.

Cantina Managed: Fix verified.

3.1.2 Incorrect implementation of `i64_div_u` and `i64_rem_u` instruction in `rwasm`

Severity: Critical Risk

Context: i64_div_u.rs#L22-L43, i64_rem_u.rs#L52-L67, div_u.rs#L37-L352, rem_u.rs#L35-L307

Description: The `i64_div_u` instruction was incorrect for some inputs. One such input found by our differential fuzzer was. `args=[I64(9223372036854775807), I64(3707827967)]` which resulted in an output of `2155872256` by `rwasm` instead of the correct output of `2487540446` which can be verified by both a calculator and `wasmtime`.

Further testing the `i64_rem_u` instruction with the same inputs of `args=[I64(9223372036854775807), I64(3707827967)]` showed that the `i64_rem_u` implementation was also incorrect. `rwasm` yielded a value of `8388607` instead of the correct output of `2132322525` which can also be verified by both a calculator and `wasmtime`.

Analysing the `rust` implementation of the snippets will showcase the bug, the root cause is a subtle overflow issue in the fast path's `div_mod_64_by_32` function.

snippets/i64_div_u.rs#L22-L43:

```
fn div_mod_64_by_32(hi: u32, lo: u32, d: u32) -> (u32, u32) {
    let mut q = 0u32;
    let mut r = 0u32;
    for i in (0..64).rev() {
        // shift the remainder left, bring the next dividend bit
        r <<= 1;
        r |= if i >= 32 {
            (hi >> (i - 32)) & 1
        } else {
            (lo >> i) & 1
        };
        if r >= d {
            r -= d;
            if i >= 32 {
                q |= 1 << (i - 32);
            } else {
                q |= 1 << i;
            }
        }
    }
    (q, r)
}
```

The intermediate value of the remainder

$$r$$

at each step can be up to $ 2d - 1 $ before the subtraction, so with $ d = 4083285857 $ we get $ 2d - 1 = 8166571713 $, which is larger than the maximum value of an unsigned 32-bit integer representation (it can go up to 33 bits). As such the value of r in `div_mod_64_by_32` can overflow causing `div_mod_64_by_32` to return an incorrect remainder used for subsequent operations.

The same logic applies for the `rem_64_by_32` fast path, where r can also overflow:

snippets/i64_rem_u.rs#L52-L67:

```
fn rem_64_by_32(hi: u32, lo: u32, d: u32) -> u32 {
    let mut r = 0u32;
    for i in (0..64).rev() {
        r <<= 1;
        r |= if i >= 32 {
            (hi >> (i - 32)) & 1
        } else {
            (lo >> i) & 1
        };
        if r >= d {
            r -= d;
        }
    }
    r
}
```

Proof of Concept: When passing `args=[I64(9223372036854775807), I64(3707827967)]` (found by differential fuzzer) to the following module we observe different results for `rwasm` and `wasmtime`.

```
(module
  (type (;0;) (func (param i64 i64) (result i64)))
  (export "test" (func 0))
  (func (;0;) (type 0) (param i64 i64) (result i64)
    local.get 0
    local.get 1
    i64.div_u
  )
)
```

Results:

```
rwasm: [I64(2155872256)]
wasmtime: [I64(2487540446)]
```

For the same `args=[I64(9223372036854775807), I64(3707827967)]` but for the `i64_rem_u` instruction, we also obtain different results.

```
(module
  (type (;0;) (func (param i64 i64) (result i64)))
  (export "test" (func 0))
  (func (;0;) (type 0) (param i64 i64) (result i64)
    local.get 0
    local.get 1
    i64.rem_u
  )
)
```

Results:

```
rwasm: [I64(8388607)]
wasmtime: [I64(2132322525)]
```

We can see the same bug occurring for the `rust` implementation of `i64_div_u` before it gets translated to `rwasm` opcodes with the exact same incorrect output:

```
#[test]
fn test_div_u() {
    let numerator = 11082746836925402571u64;
    let denominator = 4083285857u64;
    // split numerator and denominator
    let n_lo = (numerator & 0xffffffff) as u32;
    let n_hi = (numerator >> 32) as u32;
    let d_lo = (denominator & 0xffffffff) as u32;
    let d_hi = (denominator >> 32) as u32;
    assert_eq!(i64_div_u(n_lo, n_hi, d_lo, d_hi), numerator / denominator);
}
```

Recommendation: Modify the fast paths `div_mod_64_by_32` and `rem_64_by_32` to account for the case where `r` can exceed 32 bits up to 33 bits or otherwise eliminate the fast paths.

Fluent Labs: Fixed in PR 96.

Cantina Managed: Fix verified.

3.1.3 `translate_locals` does not charge fuel for locals

Severity: Critical Risk

Context: func_builder.rs#L66-L73

Description: In Wasm, it is possible to very succinctly declare that a function has a large number of locals (see Proof of Concept for example).

Locals are turned into stack elements for rWasm, but the translation fails to insert fuel metering for these pushed values. Instead `translate_locals` directly emits `op_i32_const()` instructions, one for each `i32` local and two for each `i64` local.

The compilation currently fails if any function more than 2^{15} locals, due to a `DropKeepOutOfBounds` error, but Wasmtime by default supports up to 50,000 locals per function.

This means anyone can create a module which simply allocates the max amount of locals in a function and call it repeatedly, as a DoS attack. It forces nodes to perform huge amounts of work (pushing 0.25 MB of stack items) for the same fuel costs as calling an empty function. It's a cheap attack on the network.

Proof of Concept: The following test compares three modules, each with a single function. The first has no locals, the second 4096 locals (any more causes a runtime stack overflow), and one with the max number of locals. All cost a single unit of fuel to execute, but the 4k module requires around 2,000 times as much CPU time (in `release` mode), while the 32k module encounters StackOverflow immediately and exits.

```rust
use rwasm::{
    always_failing_syscall_handler, CompilationConfig, ExecutionEngine, FuelConfig,
     ↪  ImportLinker,
    RwasmModule, RwasmStore,
};
use std::time::Instant;

/// Empty function: (module (func (export "main")))
const EMPTY_WASM: &[u8] = &[
    0x00, 0x61, 0x73, 0x6d, 0x01, 0x00, 0x00, 0x00, // magic + version
    0x01, 0x04, 0x01, 0x60, 0x00, 0x00,             // type section: () -> ()
    0x03, 0x02, 0x01, 0x00,                         // function section: 1 func, type 0
    0x07, 0x08, 0x01, 0x04, 0x6d, 0x61, 0x69, 0x6e, 0x00, 0x00, // export "main"
    0x0a, 0x04, 0x01, 0x02, 0x00, 0x0b,             // code section: body_size=2, 0
     ↪  locals, end
];

/// 4096 i64 locals (4096 = 0x80 0x20 LEB128) - max before StackOverflow
/// Body: 1 local decl (1) + count leb128 (2) + type (1) + end (1) = 5 bytes
/// Section: func_count (1) + body_size (1) + body (5) = 7 bytes
const LOCALS_4096_WASM: &[u8] = &[
    0x00, 0x61, 0x73, 0x6d, 0x01, 0x00, 0x00, 0x00, // magic + version
    0x01, 0x04, 0x01, 0x60, 0x00, 0x00,             // type section: () -> ()
    0x03, 0x02, 0x01, 0x00,                         // function section: 1 func, type 0
    0x07, 0x08, 0x01, 0x04, 0x6d, 0x61, 0x69, 0x6e, 0x00, 0x00, // export "main"
    0x0a, 0x07, 0x01, 0x05, 0x01, 0x80, 0x20, 0x7e, 0x0b, // code: 4096 i64 locals
];

/// 32767 i64 locals (32767 = 0xFF 0xFF 0x01 LEB128) - max before DropKeepOutOfBounds
/// Body: 1 local decl (1) + count leb128 (3) + type (1) + end (1) = 6 bytes
/// Section: func_count (1) + body_size (1) + body (6) = 8 bytes
const LOCALS_32767_WASM: &[u8] = &[
    0x00, 0x61, 0x73, 0x6d, 0x01, 0x00, 0x00, 0x00, // magic + version
    0x01, 0x04, 0x01, 0x60, 0x00, 0x00,             // type section: () -> ()
    0x03, 0x02, 0x01, 0x00,                         // function section: 1 func, type 0
    0x07, 0x08, 0x01, 0x04, 0x6d, 0x61, 0x69, 0x6e, 0x00, 0x00, // export "main"
    0x0a, 0x08, 0x01, 0x06, 0x01, 0xff, 0xff, 0x01, 0x7e, 0x0b, // code: 32767 i64 locals
];

fn compile(wasm: &[u8]) -> RwasmModule {
    let config = CompilationConfig::default()
        .with_entrypoint_name("main".into())
        .with_consume_fuel(true);
    RwasmModule::compile(config, wasm).expect("compile").0
}

fn execute(module: &RwasmModule) -> u64 {
    let engine = ExecutionEngine::new();
    let mut store = RwasmStore::new(
        ImportLinker::default().into(),
        (),
        always_failing_syscall_handler,
        FuelConfig::default(),
    );
    let _ = engine.execute(&mut store, module, &[], &mut []);
    store.fuel_consumed()
}

fn benchmark(module: &RwasmModule, iterations: u32) -> std::time::Duration {
    let engine = ExecutionEngine::new();
    let mut store = RwasmStore::<()>::default();
    // Warmup
    for _ in 0..10 {
        let _ = engine.execute(&mut store, module, &[], &mut []);
    }

    // Timed runs
    let start = Instant::now();
    for _ in 0..iterations {
        let _ = engine.execute(&mut store, module, &[], &mut []);
    }
    start.elapsed()
}

#[test]
fn test_locals_fuel_and_runtime() {
    const ITERATIONS: u32 = 1000;

    let empty = compile(EMPTY_WASM);
    let loc4k = compile(LOCALS_4096_WASM);
    let loc32k = compile(LOCALS_32767_WASM);

    let empty_fuel = execute(&empty);
    let loc4k_fuel = execute(&loc4k);
    let loc32k_fuel = execute(&loc32k);

    let empty_time = benchmark(&empty, ITERATIONS) / ITERATIONS;
    let loc4k_time = benchmark(&loc4k, ITERATIONS) / ITERATIONS;
    let loc32k_time = benchmark(&loc32k, ITERATIONS) / ITERATIONS;

    eprintln!("\n=== Locals Fuel & Runtime ===");
    eprintln!("0 locals:     fuel={}, time={:?}", empty_fuel, empty_time);
    eprintln!("4096 locals:  fuel={}, time={:?}", loc4k_fuel, loc4k_time);
    eprintln!("32767 locals: fuel={}, time={:?}", loc32k_fuel, loc32k_time);

    eprintln!("Slowdown: {:.0}x",
        loc4k_time.as_nanos() as f64 / empty_time.as_nanos() as f64);
}
```

Example output of `cargo test --release --test locals_runtime_bench -- --nocapture`:

```
=== Locals Fuel & Runtime ===
0 locals:     fuel=1, time=11ns
4096 locals:  fuel=1, time=22.344µs
32767 locals: fuel=1, time=11ns
Slowdown: 2031x
test test_locals_fuel_and_runtime ... ok
```

Recommendation: The code currently contains a `fuel_for_locals()` function that by default charges 1 fuel for every 16 locals. Use it to emit fuel metering instructions before the locals are pushed to the stack.

Given the cost (on one system) of ~5.5 ns for each additional local, similar to the order of magnitude of calling an empty function, also consider charging more fuel per local, for example 1 fuel per local, same as pushing a value to the stack.

Fluent Labs: Fixed in PR 97.

Cantina Managed: Fix verified.

3.2 High Risk

3.2.1 Incorrect result from BLS12-381 point addition if the output is the infinity point

Severity: High Risk

Context: lib.rs#L325-L370

Description: For both BLS12-381 G1 and G2 point addition if the output is the infinity point, the result returned will be incorrect. This is because `to_uncompressed` is called on the result which internally sets the 2nd most significant bit of the encoded point if the point is the point at infinity. This is incorrect because the EVM expects the output of the infinity point to be all zeros.

```
PRECOMPILE_BLS12_381_G1_ADD => {
//...
    let p_aff = G1Affine::from_uncompressed(&p).unwrap_or(G1Affine::identity());
    let q_aff = G1Affine::from_uncompressed(&q).unwrap_or(G1Affine::identity());

    let result = p_aff.add_affine(&q_aff);
    let result_bytes = result.to_uncompressed();

    // Convert output from runtime format to EVM format
    let out = convert_g1_output_to_evm(&result_bytes);
    Ok(out.into())
}
PRECOMPILE_BLS12_381_G2_ADD => {
//...
    let p_aff = G2Affine::from_uncompressed(&p).unwrap_or(G2Affine::identity());
    let q_aff = G2Affine::from_uncompressed(&q).unwrap_or(G2Affine::identity());

    let result = G2Projective::from(p_aff) + G2Projective::from(q_aff);
    let result_aff = G2Affine::from(result);
    let result_bytes = result_aff.to_uncompressed();

    // Encode output: 256 bytes (x0||x1||y0||y1), each limb is 64-byte BE padded (16
    ↪  zeros + 48 value)
    let out = convert_g2_output_to_evm_rwasm(&result_bytes);
    Ok(out.into())
}
```

rwasm-patches/bls12_381/blob/rwasm/src/g1.rs#L260

```
/// Serializes this element into uncompressed form. See
↪  [`notes::serialization`](crate::notes::serialization)
/// for details about how group elements are serialized.
pub fn to_uncompressed(&self) -> [u8; 96] {
    let mut res = [0; 96];

    res[0..48].copy_from_slice(
        &Fp::conditional_select(&self.x, &Fp::zero(), self.infinity).to_bytes()[..],
    );
    res[48..96].copy_from_slice(
        &Fp::conditional_select(&self.y, &Fp::zero(), self.infinity).to_bytes()[..],
    );

    // Is this point at infinity? If so, set the second-most significant bit.
    res[0] |= u8::conditional_select(&0u8, &(1u8 << 6), self.infinity);

    res
}
```

Recommendation: Check if the result is identity and return all zeroes as per EVM specification.

```
// Check if the result is identity
let out = if result.is_identity().unwrap_u8() == 1 {
    [0u8; ...] // set padding size to G1 or G2
} else {
    ... // return the correctly encoded point
};
```

Fluent Labs: Fixed in PR 251.

Cantina Managed: Fix verified.

3.2.2 Invalid points are incorrectly converted to infinity points for BLS12-381 operations

Severity: High Risk

Context: lib.rs#L336-L338, lib.rs#L360-L361, lib.rs#L395, lib.rs#L446-L456

Description: For all BLS12-381 operations during point decoding if `from_uncompressed` fails the point gets incorrectly converted to the infinity point via `unwrap_or` instead of reverting the precompile as required by the EVM.

```
let p_aff = G1Affine::from_uncompressed(&p).unwrap_or(G1Affine::identity());
```

This would lead to an incorrect result for the BLS12-381 operation if an invalid point (ie. not on curve or subgroup) is provided.

rwasm-patches/bls12_381/blob/rwasm/src/g1.rs#L280:

```
/// Attempts to deserialize an uncompressed element. See
↪  [`notes::serialization`](crate::notes::serialization)
/// for details about how group elements are serialized.
pub fn from_uncompressed(bytes: &[u8; 96]) -> CtOption<Self> {
    Self::from_uncompressed_unchecked(bytes)
        .and_then(|p| CtOption::new(p, p.is_on_curve() & p.is_torsion_free()))
}
```

Recommendation: Revert if `from_uncompressed` fails instead of converting to infinity point. To support cases where the actual infinity point is provided as the input, consider the following:

- For point addition: if the infinity point is provided, then immediately return the other point as the result.
- For MSM: if the infinity point is provided, then skip the scalar-point pair in the MSM since the result of multiplying any scalar with the infinity point is the identity element (which is the infinity point).
- For pairing: if the infinity point is provided, then skip the pair, this is because if either P or Q is the infinity point then:

$$e(P, Q) = 1$$

  which causes it to cancel out when performing multiplication over all pairings.

Fluent Labs: Fixed in PR 251.

Cantina Managed: Fix verified.

3.2.3 BLS12-381 MSM operation incorrectly converts scalar to zero if above subgroup order

Severity: High Risk

Context: lib.rs#L294-L301

Description: For the BLS12-381 MSM operation during scalar decoding if `Scalar::from_bytes` fails, the scalar gets converted to zero.

```
/// Helper function to convert scalar from BE format to rwasm-patches Scalar
#[inline(always)]
fn convert_scalar_be_to_rwasm_patches(scalar_be: &[u8; SCALAR_SIZE]) -> Scalar {
    // Convert from BE bytes to LE bytes, then to Scalar
    let mut bytes = [0u8; 32];
    bytes.copy_from_slice(scalar_be);
    bytes.reverse(); // Convert BE to LE
    Scalar::from_bytes(&bytes).unwrap_or(Scalar::zero())
}
```

This can happen if the scalar is above the subgroup order

```
q = 0x73eda753299d7d483339d80809a1d80553bda402fffe5bfeffffffff00000001
```

rwasm-patches/bls12_381/blob/main/src/scalar.rs#L254-L280:

```
/// Attempts to convert a little-endian byte representation of
/// a scalar into a `Scalar`, failing if the input is not canonical.
pub fn from_bytes(bytes: &[u8; 32]) -> CtOption<Scalar> {
    let mut tmp = Scalar([0, 0, 0, 0]);

    tmp.0[0] = u64::from_le_bytes(<[u8; 8]>::try_from(&bytes[0..8]).unwrap());
    tmp.0[1] = u64::from_le_bytes(<[u8; 8]>::try_from(&bytes[8..16]).unwrap());
    tmp.0[2] = u64::from_le_bytes(<[u8; 8]>::try_from(&bytes[16..24]).unwrap());
    tmp.0[3] = u64::from_le_bytes(<[u8; 8]>::try_from(&bytes[24..32]).unwrap());

    // Try to subtract the modulus
    let (_, borrow) = sbb(tmp.0[0], MODULUS.0[0], 0);
    let (_, borrow) = sbb(tmp.0[1], MODULUS.0[1], borrow);
    let (_, borrow) = sbb(tmp.0[2], MODULUS.0[2], borrow);
    let (_, borrow) = sbb(tmp.0[3], MODULUS.0[3], borrow);

    // If the element is smaller than MODULUS then the
    // subtraction will underflow, producing a borrow value
    // of 0xffff...ffff. Otherwise, it'll be zero.
    let is_some = (borrow as u8) & 1;

    // Convert to Montgomery form by computing
    // (a.R^0 * R^2) / R = a.R
    tmp *= &R2;

    CtOption::new(tmp, Choice::from(is_some))
}
```

The correct behaviour in this case is to reduce the scalar by modulo `q` and not convert to 0, as per the spec EIP-2537#encoding-of-scalars-for-multiplication-operation:

> A scalar for the multiplication operation is encoded as 32 bytes by performing BigEndian encoding of the corresponding (unsigned) integer. The corresponding integer is not required to be less than or equal to main subgroup order q.

Recommendation: Left-pad the scalar by 32 zero bytes and then use `Scalar::from_bytes_wide` which reduces the scalar using modulo `q`.

Fluent Labs: Fixed in PR 251.

Cantina Managed: Fix verified.

3.2.4 Memory range operations undercharge for access by rounding down

Severity: High Risk

Context: memory.rs#L152-L156, memory.rs#L165-L170, memory.rs#L235-L240, table.rs#L34-L39, table.rs#L66-L71, table.rs#L77-L82, table.rs#L93-L98

Description: The operations `memory.fill`, `memory.copy` and `memory.init` all charge based on the range of memory bytes accessed. However, the cost is charged for every 64 bytes of memory access, rounding down. This means that, for example, a `memory.fill` of 63 bytes consumes no fuel.

```
if inject_fuel_check {
    self.op_local_get(1); // n
    self.op_i32_const(MEMORY_BYTES_PER_FUEL_LOG2); // 2^6=64
    self.op_i32_shr_u(); // delta/64
    self.op_consume_fuel_stack();
}
```

A similar issue exists for accessing table elements, where access is charged in groups of 16 and rounded down for the following operations: `table.init`, `table.grow`, `table.fill` and `table.copy`.

Recommendation: Charge for each full page accessed by simulating ceiling division: `ceildiv(a, b) == (a + (b - 1)) / b`:

```
  if inject_fuel_check {
      self.op_local_get(1); // n
+     self.op_i32_const((1 << MEMORY_BYTES_PER_FUEL_LOG2) - 1);
+     self.op_i32_add();
      self.op_i32_const(MEMORY_BYTES_PER_FUEL_LOG2); // 2^6=64
      self.op_i32_shr_u(); // delta/64
      self.op_consume_fuel_stack();
  }
```

Fluent Labs: Fixed in PR 89.

Cantina Managed: Fix verified.

3.2.5 Missing fuel charge for computationally expensive precompiles

Severity: High Risk

Context: build.rs#L13-L116

Description: During genesis, all precompile contracts are compiled with the `with_consume_fuel` and `with_builtins_consume_fuel` to false, this means that operations in these precompiles must meter their own fuel or no fuel is charged for them.

```
const GENESIS_CONTRACTS: &[(Address, fluentbase_contracts::BuildOutput)] = &[
    (fluentbase_sdk::PRECOMPILE_BIG_MODEXP,
    ↪   fluentbase_contracts::FLUENTBASE_CONTRACTS_MODEXP),
    (fluentbase_sdk::PRECOMPILE_BLAKE2F,
    ↪   fluentbase_contracts::FLUENTBASE_CONTRACTS_BLAKE2F),
    (fluentbase_sdk::PRECOMPILE_BN256_ADD,
    ↪   fluentbase_contracts::FLUENTBASE_CONTRACTS_BN256),
    (fluentbase_sdk::PRECOMPILE_BN256_MUL,
    ↪   fluentbase_contracts::FLUENTBASE_CONTRACTS_BN256),
    (fluentbase_sdk::PRECOMPILE_BN256_PAIR,
    ↪   fluentbase_contracts::FLUENTBASE_CONTRACTS_BN256),
    (fluentbase_sdk::PRECOMPILE_UNIVERSAL_TOKEN_RUNTIME,
    ↪   fluentbase_contracts::FLUENTBASE_UNIVERSAL_TOKEN),
    (fluentbase_sdk::PRECOMPILE_EIP2935,
    ↪   fluentbase_contracts::FLUENTBASE_CONTRACTS_EIP2935),
    (fluentbase_sdk::PRECOMPILE_EVM_RUNTIME,
    ↪   fluentbase_contracts::FLUENTBASE_CONTRACTS_EVM),
    #[cfg(feature="svm")]
    (fluentbase_sdk::PRECOMPILE_SVM_RUNTIME,
    ↪   fluentbase_contracts::FLUENTBASE_CONTRACTS_SVM),
    (fluentbase_sdk::PRECOMPILE_FAIRBLOCK_VERIFIER,
    ↪   fluentbase_contracts::FLUENTBASE_CONTRACTS_FAIRBLOCK),
    (fluentbase_sdk::PRECOMPILE_IDENTITY,
    ↪   fluentbase_contracts::FLUENTBASE_CONTRACTS_IDENTITY),
    (fluentbase_sdk::PRECOMPILE_KZG_POINT_EVALUATION,
    ↪   fluentbase_contracts::FLUENTBASE_CONTRACTS_KZG),
    (fluentbase_sdk::PRECOMPILE_BLS12_381_G1_ADD,
    ↪   fluentbase_contracts::FLUENTBASE_CONTRACTS_BLS12381),
    (fluentbase_sdk::PRECOMPILE_BLS12_381_G1_MSM,
    ↪   fluentbase_contracts::FLUENTBASE_CONTRACTS_BLS12381),
    (fluentbase_sdk::PRECOMPILE_BLS12_381_G2_ADD,
    ↪   fluentbase_contracts::FLUENTBASE_CONTRACTS_BLS12381),
    (fluentbase_sdk::PRECOMPILE_BLS12_381_G2_MSM,
    ↪   fluentbase_contracts::FLUENTBASE_CONTRACTS_BLS12381),
    (fluentbase_sdk::PRECOMPILE_BLS12_381_PAIRING,
    ↪   fluentbase_contracts::FLUENTBASE_CONTRACTS_BLS12381),
    (fluentbase_sdk::PRECOMPILE_BLS12_381_MAP_G1,
    ↪   fluentbase_contracts::FLUENTBASE_CONTRACTS_BLS12381),
    (fluentbase_sdk::PRECOMPILE_BLS12_381_MAP_G2,
    ↪   fluentbase_contracts::FLUENTBASE_CONTRACTS_BLS12381),
    (fluentbase_sdk::PRECOMPILE_NATIVE_MULTICALL,
    ↪   fluentbase_contracts::FLUENTBASE_CONTRACTS_MULTICALL),
    (fluentbase_sdk::PRECOMPILE_NITRO_VERIFIER,
    ↪   fluentbase_contracts::FLUENTBASE_CONTRACTS_NITRO),
    (fluentbase_sdk::PRECOMPILE_OAUTH2_VERIFIER,
    ↪   fluentbase_contracts::FLUENTBASE_CONTRACTS_OAUTH2),
    (fluentbase_sdk::PRECOMPILE_RIPEMD160,
    ↪   fluentbase_contracts::FLUENTBASE_CONTRACTS_RIPEMD160),
    (fluentbase_sdk::PRECOMPILE_WASM_RUNTIME,
    ↪   fluentbase_contracts::FLUENTBASE_CONTRACTS_WASM),
    (fluentbase_sdk::PRECOMPILE_SECP256K1_RECOVER,
    ↪   fluentbase_contracts::FLUENTBASE_CONTRACTS_ECRECOVER),
    (fluentbase_sdk::PRECOMPILE_SHA256,
    ↪   fluentbase_contracts::FLUENTBASE_CONTRACTS_SHA256),
    (fluentbase_sdk::PRECOMPILE_WEBAUTHN_VERIFIER,
    ↪   fluentbase_contracts::FLUENTBASE_CONTRACTS_WEBAUTHN),
];
//...
fn compile_all_contracts() -> HashMap<&'static [u8], (B256, Bytes)> {
//...
    let config = default_compilation_config()
        .with_consume_fuel(false)
        .with_builtins_consume_fuel(false);
    for (_, contract) in GENESIS_CONTRACTS {
        i...
        let rwasm_bytecode =
            compile_wasm_to_rwasm_with_config(contract.wasm_bytecode, config.clone())
                .expect(format!("failed to compile ({}), because of: ",
                ↪   contract.name).as_str());
//...
    }
//...
}
```

However, certain precompiles, particularly for non-EVM precompiles do not charge any fuel, and are free to use, the precompiles are `PRECOMPILE_EIP2935`, `PRECOMPILE_WEBAUTHN_VERIFIER`, `PRECOMPILE_NITRO_VERIFIER`, `PRECOMPILE_NATIVE_MULTICALL`, `PRECOMPILE_UNIVERSAL_TOKEN_RUNTIME`, `PRECOMPILE_WASM_RUNTIME`, `PRECOMPILE_FAIRBLOCK_VERIFIER`.

Recommendation: Charge an adequate amount of fuel for these precompiles.

Fluent Labs: Fixed in PR 276.

Cantina Managed: Fix verified.

3.2.6 Reachable code branch reverts with compiler hint `unreachable_unchecked()`

Severity: High Risk

Context: allocator.rs#L69-L103, shared.rs#L152

Description: The global allocator reverts if it runs out of memory. It would be easy to make it run out of memory through over-allocation, by choosing a high enough pointer to allocate. However, the code chooses to revert by the `core::hints::unreachable_unchecked()` function. The purpose of this function is as follows:

> As the compiler assumes that all forms of Undefined Behavior can never happen, it will eliminate all branches in the surrounding code that it can determine will invariably lead to a call to `unreachable_unchecked()`.
> If the assumptions embedded in using this function turn out to be wrong - that is, if the site which is calling `unreachable_unchecked()` is actually reachable at runtime - the compiler may have generated nonsensical machine instructions for this situation, including in seemingly unrelated code, causing difficult-to-debug problems.

The stated reason for using `unreachable_unchecked()` is because the generated code is much smaller, simply issuing an `unreachable` Wasm instruction, whereas a regular `unreachable!()` macro generates up to 2300 instructions.

```rust
pub const unsafe fn unreachable_unchecked() -> ! {
    ub_checks::assert_unsafe_precondition!(
        check_language_ub,
        "hint::unreachable_unchecked must never be reached",
        () => false
    );
    // SAFETY: the safety contract for `intrinsics::unreachable` must
    // be upheld by the caller.
    unsafe { intrinsics::unreachable() }
}
```

However, as stated in the documentation, the purpose of `unreachable_unchecked()` is to help the compiler make optimizations, and if the code is in fact reachable, this could cause nonsensical code. Any future compiler upgrade, or use of feature flags, could generate vulnerable code that could even lead to remote code execution.

The same issue is also found in `shared.rs` when checking input size.

Proof of Concept: The following test from the finding "Module size is not explicitly bounded" runs into the `unreachable_unchecked` invocation and emits an error in debug mode. It can be added to the `fluentbase/e2e/src/deployer.rs` file.

```rust
#[test]
fn test_locals_amplification_find_limit() {
    //let mut ctx = EvmTestingContext::default().with_full_genesis();
    let owner: Address = Address::ZERO;

    // Test various function counts to find limits
    try_deploy(owner, 16);
    try_deploy(owner, 17);
}

fn try_deploy(owner: Address, num_funcs: u32) {
    let wasm = build_max_locals_module(num_funcs);
    let wasm_len = wasm.len();

    // Create fresh context for each test to avoid state interference
    let mut ctx = EvmTestingContext::default().with_full_genesis();

    match ctx.deploy_evm_tx_with_gas_result(owner, wasm.into()) {
        Ok((addr, gas)) => {
            let size = ctx.get_code(addr).unwrap().len();
            println!("funcs #{:>2}: initcode {:>4} bytes; {:>12} gas; deployed {:>9}
            ↪  bytes; OK", num_funcs, wasm_len, gas, size);

        }
        Err(result) => {
            println!("funcs #{}: initcode {:>12} bytes; {:>12} gas; deployed {:>12}
            ↪  bytes; Err", num_funcs, wasm_len, "-", 0);
        }
    }
}

fn leb128(mut n: u32) -> Vec<u8> {
    let mut out = Vec::new();
    loop {
        let mut byte = (n & 0x7F) as u8;
        n >>= 7;
        if n != 0 {
            byte |= 0x80;
        }
        out.push(byte);
        if n == 0 {
            break;
        }
    }
    out
}

/// Build WASM module with N functions, each with 32767 i64 locals
fn build_max_locals_module(num_funcs: u32) -> Vec<u8> {
    let num_funcs_leb = leb128(num_funcs);

    let func_section_size = num_funcs_leb.len() + num_funcs as usize;
    let func_section_size_leb = leb128(func_section_size as u32);

    // Each function body: size=6, 1 local decl, 32767 (0xFF 0xFF 0x01), i64, end
    let body: &[u8] = &[0x06, 0x01, 0xff, 0xff, 0x01, 0x7e, 0x0b];
    let code_section_size = num_funcs_leb.len() + (num_funcs as usize * body.len());
    let code_section_size_leb = leb128(code_section_size as u32);

    let mut wasm = vec![
        0x00, 0x61, 0x73, 0x6d, 0x01, 0x00, 0x00, 0x00, // magic + version
        0x01, 0x04, 0x01, 0x60, 0x00, 0x00, // type section: () -> ()
    ];

    // Function section
    wasm.push(0x03);
    wasm.extend_from_slice(&func_section_size_leb);
    wasm.extend_from_slice(&num_funcs_leb);
    for _ in 0..num_funcs {
        wasm.push(0x00);
    }

    // Export section (export first func as "main")
    wasm.extend_from_slice(&[
        0x07, 0x08, 0x01, 0x04, 0x6d, 0x61, 0x69, 0x6e, 0x00, 0x00,
    ]);

    // Code section
    wasm.push(0x0a);
    wasm.extend_from_slice(&code_section_size_leb);
    wasm.extend_from_slice(&num_funcs_leb);
    for _ in 0..num_funcs {
        wasm.extend_from_slice(body);
    }

    wasm
}

/// Single function with 32767 i64 locals
const SINGLE_FUNC_MAX_LOCALS_WASM: &[u8] = &[
    0x00, 0x61, 0x73, 0x6d, 0x01, 0x00, 0x00, 0x00, // magic + version
    0x01, 0x04, 0x01, 0x60, 0x00, 0x00, // type section: () -> ()
    0x03, 0x02, 0x01, 0x00, // function section: 1 func, type 0
    0x07, 0x08, 0x01, 0x04, 0x6d, 0x61, 0x69, 0x6e, 0x00, 0x00, // export "main"
    0x0a, 0x08, 0x01, 0x06, 0x01, 0xff, 0xff, 0x01, 0x7e, 0x0b, // code: 32767 i64 locals
];
```

The output of running

```
cargo test -p fluentbase-e2e test_locals_amplification_find_limit -- --nocapture
```

is:

```
running 1 test
funcs #16: initcode  158 bytes; 55250 gas; deployed 12583233 bytes; OK
```

```
output: 0x756e7361666520707265636f6e646974696f6e2873292076696f6c617465643a2068696e743a3a
↪   756e72656163686162c655f756e636865636b6564206d757374206e65766572206265207265616368650
↪   640a0a5468697320696e64696361746573206120627567206e207468652070726f6772616d2e205468
↪   697320556e646566696e656420426568617669f7220636865636b206973206f7074696f6e616c2c2061
↪   6e642063616e6e6f742062652072656c696564206f6e20666f72207361666574792e (unsafe
↪   precondition(s) violated: hint::unreachable_unchecked must never be reached

This indicates a bug in the program. This Undefined Behavior check is optional, and
↪   cannot be relied on for safety.)
funcs #17: initcode  166 bytes;     - gas; deployed        0 bytes; Err
test deployer::test_locals_amplification_find_limit ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 62 filtered out; finished in
↪   38.17s
```

Adding the `--release` flag suppresses this warning.

Recommendation: Use the (safe) `core::arch::wasm32::unreachable()`. Instead of calling the `unreachable()` intrinsic, it calls `abort()`, which is safer and does not cause undefined behavior through compiler optimization.

```
/// Generates the [`unreachable`] instruction, which causes an unconditional [trap].
///
/// This function is safe to call and immediately aborts the execution.
///
/// [`unreachable`]:
↪   https://webassembly.github.io/spec/core/syntax/instructions.html#syntax-instr-control
/// [trap]: https://webassembly.github.io/spec/core/intro/overview.html#trap
#[cfg_attr(test, assert_instr(unreachable))]
#[inline]
#[stable(feature = "unreachable_wasm32", since = "1.37.0")]
pub fn unreachable() -> ! {
    crate::intrinsics::abort()
}
```

Fluent Labs: Fixed in PR 268.

Cantina Managed: Fix verified.

3.3 Medium Risk

3.3.1 BLS12-381 point decoding do not revert if left padding bytes are non-zero

Severity: Medium Risk

Context: lib.rs#L122-L155, lib.rs#L157-L204, lib.rs#L238-L248, lib.rs#L250-L268, lib.rs#L270-L291

Description: The EVM represents an (x or y)-coordinate in 64-bytes with 16-bytes left padding of zeroes. When decoding a point for any BLS12-381 operations the EVM must revert if any of the byte in the 16-byte left padding are non-zero, as discussed in the specs EIP-2537#field-elements-encoding:

> Note on the top 16 bytes being zero: it is required that an encoded element is ”in a field”, which means strictly < modulus. In BigEndian encoding it automatically means that for a modulus that is just 381 bit long the top 16 bytes in 64 bytes encoding are zeroes and this must be checked even if only a subslice of input data is used for actual decoding.

And also in revm crates/precompile/src/bls12_381/utils.rs#L16:

```
pub(super) fn remove_fp_padding(input: &[u8]) -> Result<&[u8; FP_LENGTH],
↪   PrecompileError> {
//...
    let (padding, unpadded) = input.split_at(FP_PAD_BY);
    if !padding.iter().all(|&x| x == 0) {
        return Err(PrecompileError::Other(format!(
            "{FP_PAD_BY} top bytes of input are not zero",
        )));
    }
    Ok(unpadded.try_into().unwrap())
}
```

The problem is that Fluent does not check that the 16-byte left padding are all non-zero during any point decoding operation and revert which is non-conformant to the EVM implementation.

Recommendation: Check that the 16-byte left padding are all non-zero.

Fluent Labs: Fixed in PR 251.

Cantina Managed: Fix verified.

3.3.2 Out-of-bounds access during `SYSCALL_ID_METADATA_COPY`

Severity: Medium Risk

Context: syscall.rs#L942-L970

Description: The syscall `SYSCALL_ID_METADATA_COPY`, copies the minimum of of the length of the bytecode and the specified length from a specified offset of the bytecode of an address to the return buffer.

As a result, a user can specify a large offset such that the `offset + length` of the bytes to copy exceeds the length of the bytecode slice, or alternatively a high `offset + length` that that results in an overflow. Both of which results in an attempted out-of-bounds access and a subsequent panic in the node process, thereby halting the node.

```
SYSCALL_ID_METADATA_COPY => {
//...
    let offset = LittleEndian::read_u32(&input[20..24]);
    let length = LittleEndian::read_u32(&input[24..28]);
    // take min
    let length = length.min(ownable_account_bytecode.metadata.len() as u32);
    let metadata = ownable_account_bytecode
        .metadata
        .slice(offset as usize..(offset + length) as usize);
    return_result!(metadata, Ok)
}
```

This function is called by system runtime bytecode which is a privileged context, but is is still recommended to fix this in case any runtime implementation accidentally triggers this issue, or a malicious user can cause the runtime code to trigger this issue.

Recommendation: Copy the minimum of the `offset + length` against the length of the bytecode slice rather than just the length. Also ensure that `offset + length` computation cannot overflow.

Fluent Labs: Fixed in PR 249.

Cantina Managed: Fix verified.

3.3.3 Using `wrapping_mul` when converting gas limit to fuel introduces EVM behavioural difference

Severity: Medium Risk

Context: opcodes.rs#L443-L444, opcodes.rs#L468-L469, opcodes.rs#L493-L494, opcodes.rs#L518-L519

Description: A caller can pass in a `u64` gas limit to `*CALL` opcodes. Before passing the gas limit to the respective syscall, the `local_gas_limit` is converted to fuel units via `wrapping_mul` with the `FUEL_DENOM_RATE`.

```
fn call<
    WIRE: InterpreterTypes<Extend = InterruptionExtension, Stack = Stack>,
    H: Host + HostWrapper + ?Sized,
>(
    context: InstructionContext<'_, H, WIRE>,
) {
```

```
//...
    interrupt_into_action(context, |context, sdk| {
        let input = global_memory_from_shared_buffer(&context, in_range);
        // TODO(dmitry123): I know that wrapping mul works here, but why?
        let fuel_limit = Some(local_gas_limit.wrapping_mul(FUEL_DENOM_RATE));
        sdk.call(to, value, input.as_ref(), fuel_limit)
    });
}
```

Under normal circumstances, passing in such a high gas limit causes the EVM to just use `63/64` of the current gas left to determine the gas to forward to the child call frame

```
let mut gas_limit = min(
    frame.interpreter.gas.remaining_63_of_64_parts(),
    inputs.syscall_params.fuel_limit / FUEL_DENOM_RATE,
);
```

But because the `fuel_limit` can now overflow and wrap around to a smaller number, even to a number lower than `63/64` of the current gas left, the actual gas forwarded to the call is now different than what it would be on the EVM.

This behavioural difference could allow a caller to pass in a false gas limit to bypass some checks but the gas charged would be lower than the passed in gas limit.

Recommendation: Consider using `saturating_mul` over `wrapping_mul` to clamp the passed in gas limit to `u64::MAX` since a high gas limit would already be clamped down to `63/64` of the current gas left anyway.

Fluent Labs: Fixed in PR 255.

Cantina Managed: Fix verified.

3.3.4 Missing check for certificate timestamp validity in `nitro` precompile

Severity: Medium Risk

Context: attestation.rs#L133-L172

Description: When checking each certificate in the certificate chain (consisting of all certificates in the certificate bundle and the attestation document certificate), we must also verify that the current timestamp is within the validity period as per Section 3.2.3.1 of the specification:

> 3.2.3.1. Certificates validity
> For all the certificates in the newly generated list of X509 certificates we must ensure that they are still valid: if the current date and time are within the validity period given in the certificate. This also applies for the root certificate we are checking the chain against.

The nitro precompile does not perform this making it non-conformant with the specification:

```
fn verify_attestation_doc(doc: &AttestationDoc, root_certificate: &Certificate) {
    let mut chain = Vec::new();
    assert!(!doc.cabundle.is_empty());
    assert_eq!(doc.cabundle[0], root_certificate.to_der().unwrap());
    for cert in &doc.cabundle {
        chain.push(Certificate::from_der(cert).unwrap());
    }
    chain.push(Certificate::from_der(&doc.certificate).unwrap());
    for i in 0..chain.len() - 1 {
        verify_certificate(&chain[i + 1], &chain[i]);
    }
}
```

This allows an expired certificate to be used in the certificate chain which weakens the security guarantees of the `nitro` validation precompile.

Recommendation: For each certificate, ensure the current block timestamp is within the validity period of as defined by the certificate. The Solidity-based `base/nitro-validator` could serve as a useful reference.

Fluent Labs: Fixed in PR 261.

Cantina Managed: Fix verified.

3.3.5 Missing check for certificate critical extensions in `nitro` precompile

Severity: Medium Risk

Context: attestation.rs#L133-L172

Description: For each certificate in the chain (consisting of all certificates in the certificate bundle and the attestation document certificate), the certificate must be checked to consist of two critical extensions: the basic constraints extension and the key usage extension, see Section 3.2.3.2 and 3.2.3.3 of the specification.

Basic Constraints: For each certificate's basic constraints extension, it must be checked that the CA flag is set to true for CA certificates (non-leaf certificates in the chain) and vice versa and the `pathLenConstraint`, if defined, is also respected. The `pathLenConstraint` determines the maximum number of descendants a particular certificate can have - unlimited if undefined.

Key Usage: This defines the purpose of the key contained in the certificate and is represented by a bitmap.

- For CA certificates, `keyCertSign` (bit 5) must be set.
- For the leaf certificate (attestation document certificate), `digitalSignature` (bit 0) must be set.

Section 3.2.3.2 and 3.2.3.3 of the specification:

> We have the following restrictions for the root certificate:
> - `pathLenConstraint` is greater or equal than the chain size;
> - Key usage: `keyCertSign` bit is present.
>
> For all the certificates in the chain excepting the target certificate (the. first one) we must ensure:
> - no certificate was used to create a chain longer than its `pathLenConstraint`;
> - Key usage: `keyCertSign` bit is present.
>
> For the target certificate (the first one) we must ensure:
> - Key usage: `digitalSignature` bit is present;
> - `pathLenConstraint` must be undefined since this is a client certificate.

The nitro precompile present neither validates the existence of both critical extensions and nor performs any of the respective checks defined in both critical extensions which weakens the security guarantees of the `nitro` validation precompile.

Recommendation: Perform the two critical extension validations for each certificate in the chain. The Solidity-based `base/nitro-validator` could serve as a useful reference.

Fluent Labs: Fixed in PR 261.

Cantina Managed: Fix verified.

3.3.6 `universal-token` uses incorrect `SIG_ALLOWANCE` function signature

Severity: Medium Risk

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

Description: The `universal-token` uses an incorrect `allowance()` function signature. The correct function signature should be `allowance(address,address)` as per the EIP-20 spec. This is because it takes in two addresses, the owner and the spender and determine the total amount of allowance the spender is given by the owner.

However, the incorrect function signature is used an defined in the `consts.rs` file, taking only one address:

universal-token/src/consts.rs#L28:

```
pub const SIG_ALLOWANCE: EvmExitCode = derive_keccak256_id!("allowance(address)");
```

These can cause actual calls to the `allowance` function via Solidity / Vyper to be broken.

Recommendation: Change `SIG_ALLOWANCE` to use the correct function signature `allowance(address,address)`.

Fluent Labs: Fixed in PR 258.

Cantina Managed: Fix verified.

3.3.7 Return values from `name()` and `symbol()` in `universal_token` are not ABI encoded

Severity: Medium Risk

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

Description: Context: universal-token/lib.rs#L35-L52,

Solidity strings are ABI-encoded as a bytes array, with an initial `uint256` representing the offset, then a `uint256` to represent the length in bytes, and then the string as UTF-8 bytes (not null-terminated), right-padded with 0s to a multiple of 32 bytes.

However, the current implementation of `name()` and `symbol()` only returns the bytes representation of the string, without offset and length-integers and without right-padding:

```
/// Returns the ERC-20 `symbol()` as a short string stored at `SYMBOL_STORAGE_SLOT`.
fn erc20_symbol_handler<SDK: SharedAPI>(
    sdk: &mut SDK,
    _input: &[u8],
) -> Result<EvmExitCode, ExitCode> {
    let value = sdk.storage_short_string(&SYMBOL_STORAGE_SLOT)?;
    sdk.write(value.as_bytes());
    Ok(0)
}

/// Returns the ERC-20 `name()` as a short string stored at `NAME_STORAGE_SLOT`.
fn erc20_name_handler<SDK: SharedAPI>(
    sdk: &mut SDK,
    _input: &[u8],
) -> Result<EvmExitCode, ExitCode> {
    let value = sdk.storage_short_string(&NAME_STORAGE_SLOT)?;
    sdk.write(value.as_bytes());
    Ok(0)
}
```

Proof of Concept: Here is how ABI encoding would render the string ”hello”:

```
cast abi-encode "foo(string)" "hello"

0x00000000000000000000000000000000000000000000000000000000000000200000000000000000000000⌟
↪   000000000000000000000000000000000000000005686566c6c6f00000000000000000000000000000000⌟
↪   0000000000000000000000
```

Breaking the result down:

```
0x0000000000000000000000000000000000000000000000000000000000000020 # offset to start
0x0000000000000000000000000000000000000000000000000000000000000005 # length in bytes
0x68656c6c6f000000000000000000000000000000000000000000000000000000 # data
```

Here is the resulting encoding using `as_bytes()`:

```
let s = String::from("hello");

assert_eq!(&[104, 101, 108, 108, 111], s.as_bytes());
```

Recommendation: Use the existing `SolidityABI::encode()` function to create the return string for `name()` and `symbol()`.

Fluent Labs: Fixed in PR 258.

Cantina Managed: Fix verified.

3.3.8 EIP-2935 incorrect expected block number endianness

Severity: Medium Risk

Context: lib.rs#L36, lib.rs#L52

Description: EIP-2935 specifies that when calling the system contract's `get` operation:

> - Callers provide the block number they are querying in a big-endian encoding.
> - If calldata is not 32 bytes, revert.
> - For any request outside the range of `[block.number-HISTORY_SERVE_WINDOW, block.number-1]`, revert.

However, the current implementation in `fluentbase/` processes the input as little-endian. A program passing big-endian encoded input, as for Ethereum, will result in a panic, because of the third condition above. The condition is implemented as follows:

```
let user_requested_block_number =
↪  U256::try_from_le_slice(&input[..U256::BYTES]).unwrap();
let block_number = sdk.context().block_number();
if block_number <= 0 {
    sdk.native_exit(ExitCode::Panic);
}
let block_number = U256::from(block_number);
let block_number_prev = block_number - U256::from(1);
if user_requested_block_number > block_number_prev {
    sdk.native_exit(ExitCode::Panic);  // <--- This panic will trigger
}
```

Currently, all block numbers fit in 4 bytes, and thus will have one of the 32 last bits set in a big-endian encoding. Interpreting this as little-endian will mean one of the 32 *most* significant bits will be set, in a `U256`, which makes it larger than 2^{224}, a number far larger than any block number that will likely ever be produced.

For example, taking a recent block number, $241191061 \approx 2.4 \cdot 10^7$, converting it to big-endian bytes, and interpreting as little-endian, results in $\approx 9.5 \cdot 10^{75}$.

Recommendation: Switch to big-endian encoding, using `try_from_be_slice()` and `as_be_slice()`.

Fluent Labs: Fixed in PR 263.

Cantina Managed: Fix verified.

3.3.9 Memory intensive operations cost minimum fuel amount

Severity: Medium Risk

Context: fuel_costs.rs#L21-L24

Description: Table, memory and calling operations may need to visit arbitrary memory locations. These all charge `FuelCosts::ENTITY`, `FuelCost::LOAD`, `FuelCosts::STORE`, or `FuelCosts::CALL` as a base charge. Then, dynamic memory operations like `fill` charge for the number of element accesses, calls charge for the gas in the call frame etc... (However, also note the finding Memory range operations undercharge for access by rounding down).

Memory operations need to access arbitrary memory, tables produce lookups in arbitrarily sized tables.

Calls need to look up a function. Fluent supports Wasm modules up to ~1 MB, and a function can be defined with only 3 bytes. The rWasm compiler does not merge "equivalent" functions (and should not do so, since this is expensive to test for and often impossible), so a module such as the following could fit $2^{20}/3 \approx 350,000$ functions:

```
(module
  (func)
  (func)
  ;; ...
  ;; up to 1 MiB module size
  (func)
)
```

All the above `FuelCosts` crate constants resolve to `1`. This is a very small gas fee for accessing arbitrary memory locations, and such a cost might reasonably be reserved for access to e.g. stack elements and perhaps local variables.

Recommendation: Increase the base charge for accessing table and memory, perhaps to 10 fuel or more, to reflect worst-case computational costs.

Fluent Labs: Fixed in PR 112.

Cantina Managed: Fix verified.

3.3.10 `ref.null` tracked as `i32` on the translator type stack, causing compile-time type assertion

Severity: Medium Risk

Context: translator.rs#L1611-L1616

Description: During compilation, `ref.null ExternRef` or `FuncRef` was being translated as an `i32` on the emulated type stack.

src/compiler/translator.rs#L1644-L1649:

```
fn visit_ref_null(&mut self, _ty: ValType) -> Self::Output {
    // Since `rwasm` bytecode is untyped, we have no special `null` instructions
    // but simply reuse the `constant` instruction with an immediate value of 0.
    // Note that `FuncRef` and `ExternRef` are encoded as 64-bit values in `rwasm`.
    self.visit_i32_const(0i32)
}
```

This can cause an issue during `call` type-checks for the following example pattern:

```
(func (;0;) (type 0) (param externref)
//...
    ref.null extern
    call 0
//...
```

This is because the following assertion is triggered - the program expects the parameter type to be an `ExternRef` / `FuncRef` but encounters an `i32` instead.

src/compiler/translator.rs#L620:

```
/// Adjusts the emulated value stack given the [`rwasm_legacy::FuncType`] of the call.
fn adjust_value_stack_for_call(&mut self, func_type_idx: FuncTypeIdx) {
//...
    for func_type in func_type.params().iter().rev() {
        let popped_type = self.alloc.stack_types.pop().unwrap();
        assert_eq!(*func_type, popped_type)
    }
//...
}
```

Proof of Concept: The following test case found by our fuzzer demonstrates the panic due to assertion triggered in `rwasm` compilation process.

```
(module
  (type (;0;) (func (param externref)))
  (type (;1;) (func (result i32)))
  (table (;0;) 760 763 funcref)
  (memory (;0;) 8 9)
  (global (;0;) (mut i32) i32.const 1000)
  (export "" (func 0))
  (export "1" (table 0))
  (export "2" (memory 0))
  (func (;0;) (type 0) (param externref)
    global.get 0
    i32.eqz
    if ;; label = @1
      unreachable
    end
    global.get 0
    i32.const 1
    i32.sub
    global.set 0
    table.size 0
    ref.null extern
    call 0
    table.size 0
    table.size 0
    drop
    drop
    drop
  )
)
```

Results:

```
thread '<unnamed>' (991) panicked at /home/ser/rwasm/src/compiler/translator.rs:620:13:
assertion `left == right` failed
  left: ExternRef
 right: I32
```

Recommendation: For `visit_ref_null` function, the correct WASM `ExternRef` / `FuncRef` type must still be tracked on the emulated type stack:

```rust
fn visit_ref_null(&mut self, ty: ValType) -> Self::Output {
    self.translate_if_reachable(|builder| {
        builder.bump_fuel_consumption(|| FuelCosts::BASE)?;
        // Since `rwasm` bytecode is untyped, we have no special `null` instructions
        // but simply reuse the `constant` instruction with an immediate value of 0.
        //
        // IMPORTANT: We still must track the correct Wasm type on the emulated type
        ↪  stack,
        // otherwise later type checks (e.g. for `call`) will panic.
        match ty {
            ValType::FuncRef | ValType::ExternRef => {
                builder.alloc.stack_types.push(ty);
                builder.stack_height.push1();
                builder.alloc.instruction_set.op_i32_const(0);
                Ok(())
            }
            ty => panic!("encountered an invalid value type for RefNull: {ty:?}"),
        }
    })
}
```

Fluent Labs: Fixed in PR 103.

Cantina Managed: Fix verified.

3.3.11 `ExternRef` reference type is erroneously not supported as a local during compilation

Severity: Medium Risk

Context: func_builder.rs#L57-L64

Description: `rwasm` supports the reference types WASM extension which adds the `FuncRef` and `ExternRef` types. However, during `translate_locals`, the `ExternRef` type is erroneously left out and hence a `CompilationError::NotSupportedLocalType` is returned during compilation if a local is the `ExternRef` type.

src/compiler/func_builder.rs#L58-L65:

```
fn translate_locals(&mut self) -> Result<(), CompilationError> {
//...
    self.validator.define_locals(offset, amount, value_type)?;
    match value_type {
        ValType::I32 | ValType::I64 => {}
        // TODO(dmitry123): "make sure this type is not allowed with floats disabled"
        ValType::F32 | ValType::F64 => {}
        ValType::V128 => return Err(CompilationError::NotSupportedLocalType),
        ValType::FuncRef => {}
        _ => return Err(CompilationError::NotSupportedLocalType),
    }
//...
}
```

This could lead to valid WASM programs not being able to compile.

Proof of Concept: The following module demonstrates the compilation error:

```
(module
  (type (;0;) (func))
  (global (;0;) (mut i32) i32.const 1000)
  (export "" (func 0))
  (func (;0;) (type 0)
    (local f64 externref)
    global.get 0
    i32.eqz
    if ;; label = @1
      unreachable
    end
    global.get 0
    i32.const 1
    i32.sub
    global.set 0
  )
)
```

When ran with our differential fuzzer, we obtain the following results:

```
thread '<unnamed>' (1380) panicked at fuzz_targets/differential.rs:455:13:
compile-diff: export= rwasm_compilation_error=NotSupportedLocalType (not supported local
↪  type) args=[] result_tys=[]
```

Recommendation: Add support for `ValType::ExternRef`, as shown:

```
fn translate_locals(&mut self) -> Result<(), CompilationError> {
//...
    self.validator.define_locals(offset, amount, value_type)?;
    match value_type {
        ValType::I32 | ValType::I64 => {}
        // TODO(dmitry123): "make sure this type is not allowed with floats disabled"
        ValType::F32 | ValType::F64 => {}
        ValType::V128 => return Err(CompilationError::NotSupportedLocalType),
        ValType::FuncRef | ValType::ExternRef => {}
        _ => return Err(CompilationError::NotSupportedLocalType),
    }
//...
}
```

Fluent Labs: Fixed in PR 101.

Cantina Managed: Fix verified.

3.3.12 `FuelCosts::costs_per` rounds down fuel consumption rather than up

Severity: Medium Risk

Context: fuel_costs.rs#L33-L39

Description: The `FuelCosts::costs_per` present in the `fuel_costs.rs` file rounds down fuel consumption rather than up, in favour of the user.

src/compiler/fuel_costs.rs#L26-L31:

```
/// Returns the fuel consumption of the number of items with costs per items.
pub fn costs_per(len_items: u32, items_per_fuel: u32) -> u32 {
    NonZeroU32::new(items_per_fuel)
        .map(|items_per_fuel| len_items / items_per_fuel)
        .unwrap_or(0)
}
```

This function is used to determine the number of fuel to charged based on the number of `DropKeep` for example. However it rounds down in favour of the user and not the protocol, for instance if there are less than the `items_per_fuel` being charged (for `DropKeep`, this is the `DROP_KEEP_PER_FUEL` constant, which is `16`), the total fuel consumption for performing the `DropKeep` would run down to zero essentially performing free work.

Recommendation: While it is more difficult to exploit this particular issue, we still recommend rounding up the `costs_per` fuel consumption in favour of the protocol and not the user.

Fluent Labs: Fixed in PR 102.

Cantina Managed: Fix verified.

3.3.13 EVM-specific post-`CREATE` processing checks potentially apply to all runtimes

Severity: Medium Risk

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

Description: After `CREATE` frame is executed, the `revm-rwasm` performs the following post-processing checks in the snippet below.

handler/src/frame.rs#L722-L755:

```
// Host error if present on execution
// If ok, check contract creation limit and calculate gas deduction on output len.
//
// EIP-3541: Reject new contract code starting with the 0xEF byte
if !is_eip3541_disabled
    && spec_id.is_enabled_in(LONDON)
    && interpreter_result.output.first() == Some(&0xEF)
{
    journal.checkpoint_revert(checkpoint);
    interpreter_result.result = InstructionResult::CreateContractStartingWithEF;
    return;
}

// EIP-170: Contract code size limit to 0x6000 (~25kb)
// EIP-7907 increased this limit to 0xc000 (~49kb).
if spec_id.is_enabled_in(SPURIOUS_DRAGON) && interpreter_result.output.len() >
↪   max_code_size {
    journal.checkpoint_revert(checkpoint);
    interpreter_result.result = InstructionResult::CreateContractSizeLimit;
    return;
}
let gas_for_code = interpreter_result.output.len() as u64 * gas::CODEDEPOSIT;
if !interpreter_result.gas.record_cost(gas_for_code) {
    // Record code deposit gas cost and check if we are out of gas.
    // EIP-2 point 3: If contract creation does not have enough gas to pay for the
    // final gas fee for adding the contract code to the state, the contract
    // creation fails (i.e. goes out-of-gas) rather than leaving an empty contract.
    if spec_id.is_enabled_in(HOMESTEAD) {
        journal.checkpoint_revert(checkpoint);
        interpreter_result.result = InstructionResult::OutOfGas;
        return;
    } else {
        interpreter_result.output = Bytes::new();
    }
}
```

It performs EIP-3541, EIP-170 and charges `CODEDEPOSIT` gas from the `CREATE` operation. This applied to all runtimes, including non-EVM ones which can cause unexpected behaviour. For instance the `CODEDEPOSIT` gas can be charged twice for EVM runtimes, or the wrong `max_code_size` can be enforced for the non-EVM runtimes.

Recommendation: Gate these three post-processing checks / operations behind the `legacy_bytecode_enabled` flag.

Fluent Labs: Fixed in PR 56.

Cantina Managed: Fix verified.

3.3.14 Compile-time type assertion failure when using an `if...else...` with `i64` parameters leading to node crash

Severity: Medium Risk

Context: translator.rs#L804-L817

Description: The compiler panics with a type assertion failure when compiling valid WebAssembly modules that use `if`...`else`... blocks with `i64` parameters.

```
assertion `left == right` failed
  left: I64
 right: I32
```

The crash is triggered at `src/compiler/translator.rs:4031`, in the `translate_to_snippet_call` function:

```
fn translate_to_snippet_call(&mut self, snippet: Snippet) -> Result<(),
↪  CompilationError> {
 self.translate_if_reachable(|builder| {
     builder.bump_fuel_consumption(|| FuelCosts::BASE)?;
     builder
         .stack_height
         .pop_n(snippet.func_type().params().len() as u32);
     builder
         .stack_height
         .push_n(snippet.func_type().results().len() as u32);
     for func_type in snippet.orig_func_type().params().iter().rev() {
         let popped_type = builder.alloc.stack_types.pop().unwrap();
         assert_eq!(*func_type, popped_type)   // <--- This assertion fails
     }
 // ...
```

The root cause is that in `visit_else`, when restoring the `if` block parameters onto the type stack for the `else` branch, the translator was using the adjusted/expanded function type `resolve_func_type` where `i64` params are represented as two `i32`s. That meant the type stack got `I32, I32` instead of `I64`, and later if the snippet, for instance calls `i64.add` the translator will assert it was popping `I64` params and fails.

src/compiler/translator.rs#L800:

```rust
match if_frame.block_type() {
    BlockType::FuncType(func_type_idx) => {
        let func_type = self
            .alloc
            .func_type_registry
            .resolve_func_type(func_type_idx); // BUG
        func_type.params().iter().for_each(|param| {
            if *param == ValType::I64 || *param == ValType::F64 {
                self.stack_height.push_n(2);
            } else {
                self.stack_height.push1();
            }
            self.alloc.stack_types.push(*param);
        });
    }
    _ => {}
}
```

Proof of Concept: Add the following test. This is a manually minified example based on one found by direct fuzzing.

```rust
use rwasm::{CompilationConfig, RwasmModule};

const BUG_WAT: &str = r#"
(module
  (type (;0;) (func))
  (func (;0;) (type 0)
    i64.const 0
    i64.const 0
    i32.const 0
    if (param i64 i64)
      drop
      i64.const 0
      i64.add
      drop
    else
      drop
      i64.const 0
      i64.add
      drop
    end)
  (export "!" (func 0)))
"#;

#[test]
fn test_type_assertion_bug() {
    let wasm = wat::parse_str(BUG_WAT).expect("valid WAT");

    let config = CompilationConfig::default();

    let _ = RwasmModule::compile(config, &wasm);
}
```

The `WebAssembly` Text module compiles and validates with standard tools (`wat2wasm`).

Recommendation: Use `resolve_original_func_type` so that the correct `i64` type can be pushed to the type stack instead of two `i32`s.

```rust
match if_frame.block_type() {
    BlockType::FuncType(func_type_idx) => {
        let func_type = self
            .alloc
            .func_type_registry
            .resolve_original_func_type(func_type_idx);
```

Fluent Labs: Fixed in PR 100.

Cantina Managed: Fix verified.

3.4 Low Risk

3.4.1 BLS12-381 point addition incorrectly checks if the point is not in the subgroup during point decoding

Severity: Low Risk

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

Description: During point decoding, BLS12-381 point addition incorrectly checks if the point is not in the subgroup. It calls `from_uncompressed` which calls `is_torsion_free` to check if the point is on the subgroup.

rwasm-patches/bls12_381/blob/rwasm/src/g1.rs#L280:

```
pub fn from_uncompressed(bytes: &[u8; 96]) -> CtOption<Self> {
    Self::from_uncompressed_unchecked(bytes)
        .and_then(|p| CtOption::new(p, p.is_on_curve() & p.is_torsion_free()))
}
```

Due to a separate bug, the invalid point gets converted to the infinity point instead of reverting. The correct behaviour should be that the point gets decoded successfully, as subgroup checks are not performed during point addition as per EVM spec:

EIP-2537#abi-for-operations:

> There is no subgroup check for the G1 addition precompile.
> There is no subgroup check for the G2 addition precompile.

As per `revm` implementation:

fluentlabs-xyz/revm-rwasm/blob/v82-patched/crates/precompile/src/bls12_381/g1_add.rs#L32:

```
// NB: There is no subgroup check for the G1 addition precompile because the time to do
↪  the subgroup
// check would be more than the time it takes to do the g1 addition.
//
// Users should be careful to note whether the points being added are indeed in the right
↪  subgroup.
let a_aff = &read_g1_no_subgroup_check(a_x, a_y)?;
let b_aff = &read_g1_no_subgroup_check(b_x, b_y)?;
let p_aff = p1_add_affine(a_aff, b_aff);
```

Recommendation: For both the `G1` and `G2` addition, create and use another path that skips the subgroup `is_torsion_free` checks.

Fluent Labs: Fixed in PR 251.

Cantina Managed: Fix verified.

3.4.2 `SYSCALL_ID_METADATA_CREATE` and `SYSCALL_ID_METADATA_WRITE` can be accessed from static contexts

Severity: Low Risk

Context: syscall.rs#L877-L940

Description: `SYSCALL_ID_METADATA_CREATE` and `SYSCALL_ID_METADATA_WRITE` can be accessed from static contexts. The two syscalls do not check whether they are called from a static context such as `assert_halt!(!is_static, StateChangeDuringStaticCall)`.

```rust
SYSCALL_ID_METADATA_CREATE => {
    let Some(account_owner_address) = account_owner_address else {
        return_halt!(MalformedBuiltinParams);
    };
    let (input, lazy_metadata_input) = get_input_validated!(>= 32);
    let salt = U256::from_be_slice(&input);
    let derived_metadata_address =
        calc_create_metadata_address(&account_owner_address, &salt);
    let account = ctx
        .journal_mut()
        .load_account_code(derived_metadata_address)?;
    // Verify no deployment collision exists at derived address.
    // Check only code_hash and nonce - intentionally ignore balance to prevent
    // front-running DoS where attacker funds address before legitimate creation.
    // This matches Ethereum CREATE2 behavior: accounts can be pre-funded.
    if account.info.code_hash != KECCAK_EMPTY || account.info.nonce != 0 {
        return_result!(CreateContractCollision);
    }
    // create a new derived ownable account
    let Ok(metadata_input) = lazy_metadata_input() else {
        return_halt!(MemoryOutOfBounds);
    };
    ctx.journal_mut().set_code(
        derived_metadata_address,
        Bytecode::OwnableAccount(OwnableAccountBytecode::new(
            account_owner_address,
            metadata_input.into(),
        )),
    );
    return_result!(Bytes::new(), Ok)
}

SYSCALL_ID_METADATA_WRITE => {
    let Some(account_owner_address) = account_owner_address else {
        return_halt!(MalformedBuiltinParams);
    };
    let (input, lazy_metadata_input) = get_input_validated!(>= 20 + 4);
    // read an account from its address
    let address = Address::from_slice(&input[..20]);
    let _offset = LittleEndian::read_u32(&input[20..24]) as usize;
    let mut account = ctx.journal_mut().load_account_code(address)?;
    // to make sure this account is ownable and owner by the same runtime, that allows
    // a runtime to modify any account it owns
    let ownable_account_bytecode = match account.info.code.as_mut() {
        Some(Bytecode::OwnableAccount(ownable_account_bytecode))
            if ownable_account_bytecode.owner_address == account_owner_address =>
        {
            ownable_account_bytecode
        }
        _ => {
            return_halt!(MalformedBuiltinParams)
        }
    };
    let Ok(new_metadata) = lazy_metadata_input() else {
        return_halt!(MemoryOutOfBounds);
    };
    // code might change, rewrite it with a new hash
    let new_bytecode = Bytecode::OwnableAccount(OwnableAccountBytecode::new(
        ownable_account_bytecode.owner_address,
        new_metadata.into(),
    ));
    ctx.journal_mut().set_code(address, new_bytecode);
    return_result!(Bytes::new(), Ok)
}
```

This can allow modification of state during static contexts which is non-conformant EVM behaviour.

Recommendation: For both syscalls, assert that they cannot be called from static contexts via `assert_halt!(!is_static, StateChangeDuringStaticCall)`.

Fluent Labs: Fixed in PR 249.

Cantina Managed: Fix verified.

3.4.3 Quadratic compilation time in the number of Wasm module types

Severity: Low Risk

Context: translator.rs#L151-L154

Summary: WebAssembly compilation consolidates the types in a module by a naive search algorithm, storing all types present in the original module and searching for the earliest instance. This means that the compilation time becomes quadratic in the number of types.

Description: For every function in the Wasm module, `init_func_body_block()` calls `resolve_func_type_signature()` which returns the index of the first type in the original module which matches the function being compiled. `resolve_func_type_signature()` does this by iterating over all the types in the original module. This is obviously linear in the number of function types, and thus becomes quadratic as it's applied to all functions and imports.

This enables anyone to cause nodes to perform large computations in relation to the fuel costs. Current max calldata size is 1 MB, so a a degenerate module could be essentially 1 MB of `func` declarations with unique types.

Proof of Concept: A degenerate module meant to cause maximal burden could look like this:

```
(module
  (func)
  (func (param i32))
  (func (param f32))
  (func (param i64))
  (func (param f64))
  (func (param funcref))
  (func (param externref))
  (func (param i32 i32))
  (func (param i32 f32))
  (func (param i32 i64))
  (func (param i32 f64))
  (func (param i32 funcref))
  (func (param i32 externref))
  (func (param f32 i32))
  (func (param f32 f32))
  (func (param f32 i64))
  ;; ...
  ;; up to 1 MB module size
)
```

Which each function type def being 3 bytes + `n` where `n` is number of params, and 6 native types, it's possible to get to the order of 100,000 types into a 1 MB module. ($ 6^6 \approx 50,000 $, the number of function types with 6 parameters, and $ 6^7 \approx 300,000 $. Therefore, 6-7 params per `func` is enough, that's 9-10 bytes per function, so $\approx 100,000$ fit in 1 MB.).

Recommendation: Switch to a hash map lookup of types by performing a single iteration over all the original types and mapping each type to the index of the first declaration of that type.

Fluent Labs: Acknowledged.

Cantina Managed: Acknowledged.

3.4.4 Missing validation of attestation document for `nitro` precompile

Severity: Low Risk

Context: attestation.rs#L51-L125

Description: When deserializing the attestation document via `AttestationDoc::from_slice`, the code does not perform any of the following validation as per section 3.2.2.2 of the specification:

> *3.2.2.2. Check content*
> For every value present in the CBOR containing the attestation document, we adhere to the. following restrictions (**Note:** we talk about CBOR standard types):
>
> - `module_id`.
>
> ```
> Type: Text String
> $length != 0 /* Module ID must be non-empty */
> ```
>
> - `digest`
>
> ```
> Type: Text String
> $value ∈ {"SHA384"} /* Digest can be exactly one of these values */
> ```
>
> - `timestamp`
>
> ```
> Type: Integer
> $value > 0   /* Timestamp must be greater than 0 */
> ```
>
> - `pcrs`
>
> ```
> Type: Map
> $size ∈ [1, 32] /* We must have at least one PCR and at most 32 */
> /* The following rules apply for EACH PCR existing in the `pcrs` map.
>  * We'll use an additional notation: pcrIdx for PCR index.
>  * Note: CBOR can manage several types of keys, hence we must ensure that
>  ↪  keys
>  * also have the right type.
>  */
> Type pcrIdx: Integer /* The type of the key must be integer */
> $pcrIdx ∈ [0, 32) /* A PCR index can be in this interval [0, 32) */
> Type pcrs[pcrIdx]: Byte String /* The type of a PCR content must be Byte
> ↪  String */
> $length pcrs[pcrIdx] ∈ {32, 48, 64} /* Length of PCR can be one of this
> ↪  values {32, 48, 64} */
> ```
>
> - `cabundle`
>
> ```
> Type: Array
> $length > 0 /* CA Bundle is not allowed to have 0 elements */
> Type cabundle[idx]: Byte String /* CA bundle entry must have Byte String
> ↪  type */
> $length cabundle[idx] ∈ [1, 1024] /* CA bundle entry must have length
> ↪  between 1 and 1024 */
> ```
>
> - `public_key`
>
> ```
> Type: Byte String
> $length ∈ [1, 1024]
> ```
>
> - `user_data`
>
> ```
> Type: Byte String
> $length ∈ [0, 512]
> ```
>
> - `nonce`
>
> ```
> Type: Byte String
> $length ∈ [0, 512]
> ```

Recommendation: Perform the above validation to conform to the specification. The Solidity-based `base/nitro-validator` could serve as a useful reference.

Fluent Labs: Fixed in PR 261.

Cantina Managed: Fix verified.

3.4.5 `Paused` and `Unpaused` events for `universal-token` do not left-pad address resulting in non-compliance with ABI standards

Severity: Low Risk

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

Description: `Paused` and `Unpaused` events for `universal-token` do not left-pad address resulting in non-compliance with ABI standards. The raw 20-byte `pauser` address is used with no `B256::left_padding_from` function to left-pad to 32-bytes.

universal-token/src/events.rs#L21-L27:

```
pub fn emit_pause_event(sdk: &mut impl SharedAPI, pauser: &Address) {
    sdk.emit_log(&[EVENT_PAUSED], pauser.as_slice());
}
```

```
pub fn emit_unpause_event(sdk: &mut impl SharedAPI, pauser: &Address) {
    sdk.emit_log(&[EVENT_UNPAUSED], pauser.as_slice());
}
```

This results in these `Paused` and `Unpaused` events being emitted that are non-compliant with ABI standards which make these events harder to ingest and consume.

Recommendation: Use `B256::left_padding_from` to left-pad the 20-byte addresses to 32-byte blocks.

```
pub fn emit_pause_event(sdk: &mut impl SharedAPI, pauser: &Address) {
    sdk.emit_log(&[EVENT_PAUSED], B256::left_padding_from(pauser.as_slice()));
}
```

```
pub fn emit_unpause_event(sdk: &mut impl SharedAPI, pauser: &Address) {
    sdk.emit_log(&[EVENT_UNPAUSED], B256::left_padding_from(pauser.as_slice()));
}
```

Fluent Labs: Fixed in PR 258.

Cantina Managed: Fix verified.

3.4.6 State-changing `universal-token` operations such as mint, transfers, approvals, pauses can be accessed from static contexts

Severity: Low Risk

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

Description: State-changing `universal-token` operations do not use any syscalls to read or modify storage. Instead, the slots to be modified are prefetched and provided directly to the precompile to be modified, bypassing any syscalls.

As such they are now possible to be accessed from static contexts as they directly bypass any static checks that are carried out when performing the syscall.

The following state-changing `universal-token` operations can be accessed from static contexts:

- Minting.
- Transfers (`transfer` / `transferFrom`).
- Approvals.
- Pausing / Unpausing.

Recommendation: Revert inside the precompile if the any state-changing `universal-token` operations are accessed from static contexts.

Fluent Labs: Fixed in PR 258.

Cantina Managed: Fix verified.

3.4.7 Constructor does not emit `Transfer` event

Severity: Low Risk

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

Description: Context: universal-token/lib.rs#L344-L393.

The universal token contract allows a deployer to mint an initial supply, which is sent to the the deployer. However, it does not emit any event indicating this initial transfer. This means indexers and other external observers might miss the initial mint. It also violates best practice of emitting events for all mints, burns and transfers.

Recommendation: Emit a regular transfer event from the zero address when minting in the constructor.

```
  if initial_supply > 0 {
      let caller = sdk.context().contract_caller();
      // Assign caller balance
      BalanceStorageMap::new(BALANCE_STORAGE_SLOT)
          .entry(caller)
          .set_checked(sdk, initial_supply)?;
      // Increase token supply
      sdk.write_storage(TOTAL_SUPPLY_STORAGE_SLOT, initial_supply)
          .ok()?;
+     emit_transfer_event(sdk, &Address::ZERO, &caller, &initial_supply)?;
  }
```

Fluent Labs: Fixed in PR 258.

Cantina Managed: Fix verified.

3.4.8 `rwasm` multi-return value functions return in the wrong order

Severity: Low Risk

Context: executor.rs#L213-L216

Description: A function in WASM can return multiple values. When populating the `return` buffer, the expected behaviour is that the first value corresponds to the first (deepest) value in the stack, and so on.

However, `rwasm` incorrectly fills the `result` buffer in stack pop order, which reverses the expected order of return values.

src/vm/executor.rs#L213-L216:

```
// copy output values in case of successful execution
for x in result {
    *x = self.sp.pop_value(x.ty());
}
```

This does not appear reachable in Fluentbase as it uses an empty return buffer (runtime/src/runtime/contract_runtime.rs#L83) when executing modules, however we recommend fixing this in `rwasm` for WASM specification correctness.

Proof of Concept: The following test case was found by differential fuzzing and different results were observed when comparing `wasmtime` against `rwasm`:

```
(module
  (type (;0;) (func (result i64 i64)))
  (global (;0;) (mut i32) i32.const 1000)
  (export "\u{a}++" (func 0))
  (func (;0;) (type 0) (result i64 i64)
    global.get 0
    i32.eqz
    if ;; label = @1
      unreachable
    end
    global.get 0
    i32.const 1
    i32.sub
    global.set 0
    i64.const 2251799813685248
    i64.const 0
  )
)
```

Results:

```
wasmtime=[I64(2251799813685248), I64(0)]
rwasm=[I64(0), I64(2251799813685248)]
```

Recommendation: Fill the `result` buffer in reverse order.

```
// Note: For multi-value returns, the last result is on the top of the value stack.
// Therefore we must pop in reverse order so that `result[0]` receives the first
// (deeper) result value.
for x in result.iter_mut().rev() {
    *x = self.sp.pop_value(x.ty());
}
```

Fluent Labs: Fixed in PR 95.

Cantina Managed: Fix verified.

3.4.9 `rwasm` value stack underflow via module if it contains both a `start` function and a requested entrypoint

Severity: Low Risk

Context: parser.rs#L94-L100

Description: If a WASM module contains a `start` function and the embedder requests execution of a specific export entrypoint via `CompilationConfig::with_entrypoint_name(...)`), the correct semantics are:

- Run the `start` function first (instantiation-time side effects).
- Then invoke the requested export and return its results.

The bug is the `rwasm` compiler prioritizes the `start` function and did not invoke the requested entrypoint when a `start` function existed.

src/compiler/parser.rs#L95-L105:

```
pub fn finalize(
    mut self,
    wasm_binary: &[u8],
) -> Result<(RwasmModule, ConstructorParams), CompilationError> {
    if let Some(start_func) = self.allocations.translation.start_func {
        // for the start section we must always invoke even if there is a main function,
        // otherwise it might be super misleading for devs
        self.allocations
            .translation
            .emit_function_call(start_func, true, false);
    }
```

The VM still attempts to read the requested entrypoint's return values from the value stack, which could be empty, leading to a value stack underflow `ValueStackPtr::dec_by`.

This does not appear reachable in Fluentbase, since WASM compilation does not use `CompilationConfig::with_entrypoint_name(...)` anywhere. However we recommend fixing this in `rwasm` for WASM specification correctness.

Proof of Concept: The following module test triggers a debug assert when compiled and ran.

```
use rwasm::{CompilationConfig, ExecutionEngine, RwasmModule, RwasmStore, Value};

#[test]
```

```
fn start_entrypoint_stack_underflow() {
    let wasm = wat::parse_str(r#"
(module
  (global $g (mut i32) (i32.const 0))
  (func $start
    i32.const 7
    global.set $g
  )
  (start $start)

  ;; entrypoint returns one value
  (func (export "main") (result i32)
    global.get $g
  )
)
"#).unwrap();

    // Buggy behavior: start runs, but requested entrypoint "main" is NOT invoked.
    let config = CompilationConfig::default().with_entrypoint_name("main".into());
    let (module, _) = RwasmModule::compile(config, &wasm).unwrap();

    let engine = ExecutionEngine::new();
    let mut store = RwasmStore::<()>::default();

    // Non-empty result buffer => executor will pop 1 value from the stack.
    let mut result = [Value::I32(0); 1];

    // On the buggy version this hits stack underflow (debug_assert) when popping
    ↪  result[0].
    engine.execute(&mut store, &module, &[], &mut result).unwrap();
}
```

The following debug assert was triggered:

src/vm/value_stack.rs#L458:

```
/// Decreases the [`ValueStackPtr`] of `self` by one.
#[inline]
fn dec_by(&mut self, delta: usize) {
    // SAFETY: Within Wasm bytecode execution we are guaranteed by
    //         Wasm validation and `rwasm` codegen to never run out
    //         of valid bounds using this method.
    self.ptr = unsafe { self.ptr.sub(delta) };
    debug_assert!(self.ptr >= self.src, "stack underflow");
}
```

Recommendation: If the start function exists, we must invoke any requested entrypoint to conform to WASM semantics:

```
pub fn finalize(
    mut self,
    wasm_binary: &[u8],
) -> Result<(RwasmModule, ConstructorParams), CompilationError> {
    if let Some(start_func) = self.allocations.translation.start_func {
        // for the start section we must always invoke even if there is a main function,
        // otherwise it might be super misleading for devs
        self.allocations
            .translation
            .emit_function_call(start_func, true, false);
        // If a specific entrypoint was requested, invoke it after the start function.
        //
        // This matches Wasm semantics: the start function runs at instantiation time and
        ↪  then
        // the exported function is invoked by the embedder. For rwasm's "entrypoint
        ↪  bytecode"
        // model we need to encode both.
        if let Some(entrypoint_name) = self.config.entrypoint_name.as_ref() {
            let func_idx = self
                .allocations
                .translation
                .exported_funcs
                .get(entrypoint_name)
                .copied()
                .ok_or(CompilationError::MissingEntrypoint)?;
            self.allocations
                .translation
                .emit_function_call(func_idx, true, true);
        }
    }
```

Fluent Labs: Fixed in PR 113.

Cantina Managed: Fix verified.

3.4.10 Value stack overflow when pushing large number of function parameters onto stack

Severity: Low Risk

Context: executor.rs#L196-L200

Description: `rwasm` can run into a value stack overflow when pushing a large number of function parameters onto stack. This is because `rwasm` pushed function parameters onto the stack without reserving more stack capacity if the function parameters took more than the default stack size of 32 cells.

src/vm/executor.rs#L196-L200:

```
pub fn run(&mut self, params: &[Value], result: &mut [Value]) -> Result<(), TrapCode> {
    // copy input params
    for x in params {
        self.sp.push_value(x);
    }
//...
```

Hence, if a module loaded in more than the default stack size of 32 cells, `rwasm` would run into a value stack overflow resulting in undefined behaviour.

However, this does not appear reachable in Fluentbase as it calls `rwasm` with no parameters when executing modules (data is passed via memory and syscalls instead), however we recommend fixing this in `rwasm` for WASM specification correctness.

Proof of Concept: The following test case uses a function with parameters that occupy 39 stack cells and thus overflows the default stack size of 32 cells. This was found by differential fuzzing.

```
(module
  (type (;0;) (func (param f64 f64 i32 f64 f64 f64 f64 f64 i32 f64 f64 f64 f64 f64 f64
↪   f64 f64 f64 f64 f32) (result i64)))
  (type (;1;) (func (param i32)))
  (global (;0;) (mut i32) i32.const 1000)
  (export "" (func 0))
  (func (;0;) (type 0) (param f64 f64 i32 f64 f64 f64 f64 f64 i32 f64 f64 f64 f64 f64 f64
↪   f64 f64 f64 f64 f32) (result i64)
    global.get 0
    i32.eqz
    if ;; label = @1
      unreachable
    end
    global.get 0
    i32.const 1
    i32.sub
    global.set 0
    i64.const 0
  )
)
```

Results:

```
wasmtime=Ok
rwasm_trap=StackOverflow
```

When the program is ran on both, `wasmtime` runs with no trap while `rwasm` traps with `TrapCode::StackOverflow`.

Recommendation: Reserve stack capacity for incoming function parameters

```
pub fn run(&mut self, params: &[Value], result: &mut [Value]) -> Result<(), TrapCode> {
    // Ensure the value stack has enough capacity for the incoming parameters.
    //
    // Important: parameters are pushed onto the value stack before the function's
     ↪  prologue
    // `StackCheck` executes, so we must reserve for them here. Otherwise, a function
     ↪  with many
    // parameters (especially `i64`/`f64` which occupy 2 i32 cells) can write past the
     ↪  initial
    // stack backing storage (N_DEFAULT_STACK_SIZE) and corrupt memory.
    self.value_stack.sync_stack_ptr(self.sp);
    let params_cells: usize = params
        .iter()
        .map(|v| match v {
            Value::I64(_) | Value::F64(_) => 2,
            _ => 1,
        })
        .sum();
    if params_cells > 0 {
        self.value_stack.reserve(params_cells)?;
        // rewrite SP after reserve because of potential reallocation
        self.sp = self.value_stack.stack_ptr();
    }
    // copy input params
    for x in params {
        self.sp.push_value(x);
    }
```

Fluent Labs: Fixed in PR 110.

Cantina Managed: Fix verified.

3.4.11 Module size is not explicitly bounded

Severity: Low Risk

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

Description: Ethereum and other chains limit the max EVM contract size for performance reasons (see EIP-170EIP-170). There is no such cap in here, and the actual largest module deployable is controlled by implicit limits.

Fluent supports transactions up to 1 MB in size. However, rWasm modules can be much larger due to the ability to succinctly express a large number of local variables to a function, and the fact that rWasm compilation transforms each local into stack push operations (see the finding "`translate_locals` does not charge fuel for locals" for more details).

Compilation currently fails if any function has more than 2^{15} locals, due to a `DropKeepOutOfBounds` error. Furthermore, runtime limitations give a `MemoryOutOfBounds` error when deploying a large module.

Experimentally we've been able to deploy a module which is 12.58 MB serialized, which contains 16 functions with the maximum number of locals, without running into `MemoryOutOfBounds`. The cost for deploying that module is ~55k gas.

If memory limits were not an issue due to future changes, then the largest module we can create -- as many functions as will fit in the 1 MB calldata limitation, each with the max number of locals -- is about 25.77 GB serialized. See "Quadratic compilation time in the number of Wasm module types" for rough calculations of size.

In essence, the max module size is a function of several interacting limits, none of which address module size directly, and thus the max module size may fluctuate as parameters are tweaked and capacities upgraded.

Proof of Concept: Note that in the tests below, we can not use the more readable `.wat` format (Wasm Text) because it does not support expressing local declarations succinctly, and thus become prohibitively large and unreadable. Hence we need to construct the bytecode manually.

Largest compilable module: The following test compiles the largest known possible rWasm module that fits within the 1 MB transaction limit. It can be executed in the `rwasm` crate. The resulting module is 27.55 GB.

```rust
use rwasm::{CompilationConfig, RwasmModule};

/// 32767 i64 locals (32767 = 0xFF 0xFF 0x01 LEB128) - max before DropKeepOutOfBounds
const LOCALS_32767_WASM: &[u8] = &[
    0x00, 0x61, 0x73, 0x6d, 0x01, 0x00, 0x00, 0x00, // magic + version
    0x01, 0x04, 0x01, 0x60, 0x00, 0x00,             // type section: () -> ()
    0x03, 0x02, 0x01, 0x00,                         // function section: 1 func, type 0
    0x07, 0x08, 0x01, 0x04, 0x6d, 0x61, 0x69, 0x6e, 0x00, 0x00, // export "main"
    0x0a, 0x08, 0x01, 0x06, 0x01, 0xff, 0xff, 0x01, 0x7e, 0x0b, // code: 32767 i64 locals
];

fn leb128(mut n: u32) -> Vec<u8> {
    let mut out = Vec::new();
    loop {
        let mut byte = (n & 0x7F) as u8;
        n >>= 7;
        if n != 0 { byte |= 0x80; }
        out.push(byte);
        if n == 0 { break; }
    }
    out
}

/// Build module with N functions, each with 32767 i64 locals
fn build_max_locals_module(num_funcs: u32) -> Vec<u8> {
    let num_funcs_leb = leb128(num_funcs);

    // Function section: num_funcs × type index 0
    let func_section_size = num_funcs_leb.len() + num_funcs as usize;
    let func_section_size_leb = leb128(func_section_size as u32);

    // Code section: each function body is 6 bytes (1 local decl, 3-byte count, type,
↪   end)
    let body: &[u8] = &[0x06, 0x01, 0xff, 0xff, 0x01, 0x7e, 0x0b]; // size=6, 1 decl,
↪   32767, i64, end
    let code_section_size = num_funcs_leb.len() + (num_funcs as usize * body.len());
    let code_section_size_leb = leb128(code_section_size as u32);

    let mut wasm = vec![
        0x00, 0x61, 0x73, 0x6d, 0x01, 0x00, 0x00, 0x00, // magic + version
        0x01, 0x04, 0x01, 0x60, 0x00, 0x00,             // type section
    ];

    // Function section
    wasm.push(0x03);
    wasm.extend_from_slice(&func_section_size_leb);
    wasm.extend_from_slice(&num_funcs_leb);
    for _ in 0..num_funcs { wasm.push(0x00); }

    // Export section (export first func as "main")
    wasm.extend_from_slice(&[0x07, 0x08, 0x01, 0x04, 0x6d, 0x61, 0x69, 0x6e, 0x00,
↪   0x00]);

    // Code section
    wasm.push(0x0a);
    wasm.extend_from_slice(&code_section_size_leb);
    wasm.extend_from_slice(&num_funcs_leb);
    for _ in 0..num_funcs { wasm.extend_from_slice(body); }

    wasm
}

#[test]
fn test_max_locals_single_func() {
    let config = CompilationConfig::default()
        .with_entrypoint_name("main".into())
        .with_consume_fuel(true);

    let (module, _) = RwasmModule::compile(config, LOCALS_32767_WASM).expect("compile");

    let input_size = LOCALS_32767_WASM.len();
    let output_size = module.serialize().len();

    eprintln!("\n=== Single Function, 32767 Locals ===");
    eprintln!("Input:  {} bytes", input_size);
    eprintln!("Output: {} bytes ({:.2} MB)", output_size, output_size as f64 /
    ↪   1_000_000.0);
}

#[test]
fn test_max_locals_max_funcs() {
    let num_funcs: u32 = 32768; // max before compiler panic
    let wasm = build_max_locals_module(num_funcs);

    let config = CompilationConfig::default()
        .with_entrypoint_name("main".into())
        .with_consume_fuel(true);

    let (module, _) = RwasmModule::compile(config, &wasm).expect("compile");

    let input_size = wasm.len();
    let output_size = module.serialize().len();

    eprintln!("\n=== {} Functions × 32767 Locals ===", num_funcs);
    eprintln!("Input:  {} bytes ({:.2} MB)", input_size, input_size as f64 /
    ↪   1_000_000.0);
    eprintln!("Output: {} bytes ({:.2} GB)", output_size, output_size as f64 /
    ↪   1_000_000_000.0);
}
```

Output:

```
=== Single Function, 32767 Locals ===
Input:  38 bytes
Output: 786533 bytes (0.79 MB)
test test_max_locals_single_func ... ok

=== 32768 Functions × 32767 Locals ===
Input:  262182 bytes (0.26 MB)
Output: 25770459225 bytes (25.77 GB)
test test_max_locals_max_funcs ... ok
```

Largest deployable module: The following test finds an approximate limit for deployable module size by trying to deploy a module with as many functions with the max locals as possible, and finds that the maximum is ~13 MB. It can be added to the `fluentbase/e2e/src/deployer.rs` file.

```
#[test]
fn test_locals_amplification_find_limit() {
    //let mut ctx = EvmTestingContext::default().with_full_genesis();
    let owner: Address = Address::ZERO;
    // Test various function counts to find limits
    try_deploy(owner, 16);
    try_deploy(owner, 17);
}

fn try_deploy(owner: Address, num_funcs: u32) {
    let wasm = build_max_locals_module(num_funcs);
    let wasm_len = wasm.len();

    // Create fresh context for each test to avoid state interference
    let mut ctx = EvmTestingContext::default().with_full_genesis();

    match ctx.deploy_evm_tx_with_gas_result(owner, wasm.into()) {
        Ok((addr, gas)) => {
            let size = ctx.get_code(addr).unwrap().len();
            println!("funcs #{:>2}: initcode {:>4} bytes; {:>12} gas; deployed {:>9}
            ↪  bytes; OK", num_funcs, wasm_len, gas, size);

        }
        Err(result) => {
            println!("funcs #{}: initcode {:>12} bytes; {:>12} gas; deployed {:>12}
            ↪  bytes; Err", num_funcs, wasm_len, "-", 0);
        }
    }
}

fn leb128(mut n: u32) -> Vec<u8> {
    let mut out = Vec::new();
    loop {
        let mut byte = (n & 0x7F) as u8;
        n >>= 7;
        if n != 0 {
            byte |= 0x80;
        }
        out.push(byte);
        if n == 0 {
            break;
        }
    }
    out
}

/// Build WASM module with N functions, each with 32767 i64 locals
fn build_max_locals_module(num_funcs: u32) -> Vec<u8> {
    let num_funcs_leb = leb128(num_funcs);

    let func_section_size = num_funcs_leb.len() + num_funcs as usize;
    let func_section_size_leb = leb128(func_section_size as u32);

    // Each function body: size=6, 1 local decl, 32767 (0xFF 0xFF 0x01), i64, end
    let body: &[u8] = &[0x06, 0x01, 0xff, 0xff, 0x01, 0x7e, 0x0b];
    let code_section_size = num_funcs_leb.len() + (num_funcs as usize * body.len());
    let code_section_size_leb = leb128(code_section_size as u32);

    let mut wasm = vec![
        0x00, 0x61, 0x73, 0x6d, 0x01, 0x00, 0x00, 0x00, // magic + version
        0x01, 0x04, 0x01, 0x60, 0x00, 0x00, // type section: () -> ()
    ];

    // Function section
    wasm.push(0x03);
    wasm.extend_from_slice(&func_section_size_leb);
    wasm.extend_from_slice(&num_funcs_leb);
    for _ in 0..num_funcs {
        wasm.push(0x00);
    }
    // Export section (export first func as "main")
    wasm.extend_from_slice(&[
        0x07, 0x08, 0x01, 0x04, 0x6d, 0x61, 0x69, 0x6e, 0x00, 0x00,
    ]);

    // Code section
    wasm.push(0x0a);
    wasm.extend_from_slice(&code_section_size_leb);
    wasm.extend_from_slice(&num_funcs_leb);
    for _ in 0..num_funcs {
        wasm.extend_from_slice(body);
    }

    wasm
}

/// Single function with 32767 i64 locals
const SINGLE_FUNC_MAX_LOCALS_WASM: &[u8] = &[
    0x00, 0x61, 0x73, 0x6d, 0x01, 0x00, 0x00, 0x00, // magic + version
    0x01, 0x04, 0x01, 0x60, 0x00, 0x00, // type section: () -> ()
    0x03, 0x02, 0x01, 0x00, // function section: 1 func, type 0
    0x07, 0x08, 0x01, 0x04, 0x6d, 0x61, 0x69, 0x6e, 0x00, 0x00, // export "main"
    0x0a, 0x08, 0x01, 0x06, 0x01, 0xff, 0xff, 0x01, 0x7e, 0x0b, // code: 32767 i64 locals
];
```

Output:

```
funcs #16: initcode  158 bytes; 55250 gas; deployed 12583233 bytes; OK
funcs #17: initcode  166 bytes;     - gas; deployed        0 bytes; Err
```

Recommendation: There is currently no known issues with the size, but there are also no explicit limitations. Consider implementing an explicit maximum module size as a countermeasure.

Fluent Labs: Fixed in PR 269.

Cantina Managed: Fix verified.

3.4.12 Gas charged after cold state operations

Severity: Low Risk

Context: syscall.rs#L209-L210

Description: For syscalls which perform cold reads to state (`SLOAD`, `SSTORE`, `*CALL`, `EXTCODE*`, `BALANCE`, `SELFDESTRUCT`), the gas is charged only after the cold state is read and I/O operation are done.

Hence, work can be done here for free by forwarding small amounts of gas to functions that execute the I/O-intensive operations which eventually revert via OOG. But the work (cold state read) is still done only consuming limited gas in the process.

As an example, the `SYSCALL_ID_STORAGE_READ` performs the `SLOAD` first before `charge_gas` is called.

crates/interpreter/src/instructions/host.rs#L189-L211:

```
SYSCALL_ID_STORAGE_READ => {
    let input = get_input_validated!(== 32);
    let slot = U256::from_le_slice(&input[0..32]);
    let value = ctx.journal_mut().sload(current_target_address, slot)?;
    charge_gas!(sload_cost(spec_id, value.is_cold));
    inspect!(opcode::SLOAD, [slot], [value.data]);
    let output: [u8; 32] = value.to_le_bytes();
    return_result!(output, Ok)
}
```

Recommendation: Use the `skip_cold_load` functionality present in `revm v102` for the relevant syscalls listed.

See the `SLOAD`, `SSTORE`, `EXTCODE*`, `BALANCE`, `SELFDESTRUCT` implementations in crates/interpreter/src/instructions/host.rs#L189-L211 and the `*CALL` implementation in crates/interpreter/src/instructions/contract.rs#L100-L122.

Fluent Labs: Fixed in PR 267.

Cantina Managed: Fix verified.

3.5 Informational

3.5.1 Off-by-one error when comparing code size against the `HARD_CAP` in `CREATE` syscalls

Severity: Informational

Context: syscall.rs#L457-L465

Description: When comparing `inputs.syscall_params.input.len()` against the `HARD_CAP` in `CREATE` syscalls, it uses he < operator as opposed to the <= operator creating an off-by-one error.

```
SYSCALL_ID_CREATE | SYSCALL_ID_CREATE2 => {
    assert_halt!(!is_static, StateChangeDuringStaticCall);

    // Make sure input doesn't exceed hard cap at least
    const HARD_CAP: usize = WASM_MAX_CODE_SIZE + U256::BYTES + U256::BYTES;
    assert_halt!(
        inputs.syscall_params.input.len() < HARD_CAP,
        MalformedBuiltinParams
    );
```

Recommendation: Use <= over <.

Fluent Labs: Fixed in PR 249.

Cantina Managed: Fix verified.

3.5.2 Incorrect `MAX_INITCODE_SIZE` constant used

Severity: Informational

Context: syscall.rs#L20-L26, lib.rs#L48

Description: The following line in `syscall.rs` uses the wrong `MAX_INITCODE_SIZE` constant.

```
let max_initcode_size = wasm_max_code_size(&init_code).unwrap_or(MAX_INITCODE_SIZE);
```

The file uses `interpreter::MAX_INITCODE_SIZE` which exports `primitives::eip7907::MAX_INITCODE_SIZE`.

```
interpreter::{
    gas,
    gas::{sload_cost, sstore_cost, sstore_refund, warm_cold_cost},
    interpreter_types::InputsTr,
    CallInput, CallInputs, CallScheme, CallValue, CreateInputs, FrameInput, Gas, Host,
    MAX_INITCODE_SIZE,
}
```

revm-rwasm/crates/interpreter/src/lib.rs#L48:

```
pub use primitives::{eip7907::MAX_CODE_SIZE, eip7907::MAX_INITCODE_SIZE};
```

This uses the new value of 0x12000 defined in EIP-7907, not the original value of 0x0C000 defined in EIP-3860.

revm-rwasm/crates/primitives/src/eip7907.rs#L6:

~~~
pub const MAX_INITCODE_SIZE: usize = 0x12000;
```rust.
~~~

~~~
Currently this is not reachable, as the `evm` crate also has an additional check that
↪  checks the using the correct constant `EVM_MAX_INITCODE_SIZE`.
```rust
wasm_max_code_size(&*prefix).unwrap_or(EVM_MAX_INITCODE_SIZE)
~~~

Recommendation: Consider using the correct constant defined in types/src/lib.rs#L100.

Fluent Labs: Fixed in commit 3394e1e0.

Cantina Managed: Fix verified.

3.5.3 Redundant check for closing quote in `webauthn` precompile

Severity: Informational

Context: webauthn.rs#L128-L133

Description: In the `webauthn` precompile, after checking that the `challenge_str` is present in `client_data_json` via `contains_at`, there is an additional `expected_quote_pos` check for the closing quote.

```rust
// Encode challenge in base64url format
let encoded_challenge = base64url_encode(challenge);
let challenge_str = format!("\"challenge\":\"{encoded_challenge}\"");

// Verify challenge
if !contains_at(challenge_str.as_bytes(), client_data_json, challenge_idx) {
    return false;
}
// Verify that the challenge is followed by a closing quote
let expected_quote_pos = challenge_idx + challenge_str.len() - 1;
if expected_quote_pos >= client_data_json.len() || client_data_json[expected_quote_pos]
↪  != b'"'
{
    return false;
}
```

This check is redundant because the closing quote is present in the `challenge_str` and so `contains_at` already checks for the closing quote.

Recommendation: Remove the redundant `expected_quote_pos` check.

Fluent Labs: Fixed in PR 256.

Cantina Managed: Fix verified.

3.5.4 `FLAGS_STORAGE_SLOT` is unused and can be removed

Severity: Informational

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

Description: The `FLAGS_STORAGE_SLOT` defined for `universal-token` is unused in the actual precompile code and can be removed from the code.

Recommendation: It can be removed from both universal-token/src/consts.rs#L36 and universal-token/src/storage.rs (multiple references).

Fluent Labs: Fixed in PR 258.

Cantina Managed: Fix verified.

3.5.5 `MINTER_STORAGE_SLOT` uses non-intuitive seed for storage slot derivation

Severity: Informational

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

Description: `MINTER_STORAGE_SLOT` uses a non-intuitive seed `total_supply_slotminter_slot` for storage slot derivation.

universal-token/src/consts.rs#L37-L38

```
pub const MINTER_STORAGE_SLOT: U256 =
    U256::from_le_bytes(derive_keccak256!(total_supply_slotminter_slot));
```

Recommendation: Use `minter_slot` as the seed instead:

```
pub const MINTER_STORAGE_SLOT: U256 =
    U256::from_le_bytes(derive_keccak256!(minter_slot));
```

Fluent Labs: Fixed in PR 258.

Cantina Managed: Fix verified.

3.5.6 Missing or extra storage slot keys pre-fetched during `erc20_compute_deploy_storage_keys`

Severity: Informational

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

Description: The `erc20_compute_deploy_storage_keys` function prefetches extraneous or missing storage-slot keys.

1. An missing `DECIMALS_STORAGE_SLOT` key is missing in this function, this needs to be loaded in as the decimals slot of the is initialized during deployment.
2. An extra storage slot `minter.compute_slot(BALANCE_STORAGE_SLOT)` is pre-fetched. This is the balance storage slot belonging to the minter which is not needed as the balance of the minter is not modified during deployment (only the caller's balance storage slot is modified when the initial supply is set to be greater than 0).

- (`FLAGS_STORAGE_SLOT` is also extraneous but that is covered in finding ”FLAGS_STORAGE_SLOT is unused and can be removed”).

universal-token/src/storage.rs#L93-L125:

```
pub fn erc20_compute_deploy_storage_keys(input: &[u8], caller: &Address) ->
↪   Option<Vec<U256>> {
    if input.len() < SIG_LEN_BYTES {
        return None;
    }
    let mut result = Vec::with_capacity(7);
    let Some(InitialSettings {
        minter,
        pauser,
        initial_supply,
        ..
    }) = InitialSettings::decode_with_prefix(input)
    else {
        // If input is incorrect then no storage keys required
        return None;
    };
    result.push(NAME_STORAGE_SLOT);
    result.push(SYMBOL_STORAGE_SLOT);
    result.push(FLAGS_STORAGE_SLOT);
    result.push(TOTAL_SUPPLY_STORAGE_SLOT);
    if !initial_supply.is_zero() {
        let storage_slot = caller.compute_slot(BALANCE_STORAGE_SLOT);
        result.push(storage_slot);
    }
    if let Some(minter) = minter {
        result.push(MINTER_STORAGE_SLOT);
        let storage_slot = minter.compute_slot(BALANCE_STORAGE_SLOT);
        result.push(storage_slot);
    }
    if let Some(_pauser) = pauser {
        result.push(PAUSER_STORAGE_SLOT);
    }
    Some(result)
}
```

Recommendation: Two fixes to `erc20_compute_deploy_storage_keys`:

1. Add the missing storage slot `DECIMALS_STORAGE_SLOT`.
2. Remove the extra storage slot `minter.compute_slot(BALANCE_STORAGE_SLOT)`.

Fluent Labs: Fixed in PR 258.

Cantina Managed: Fix verified.

3.5.7 Transfers initiated during pause result in confusing `ERR_MINTING_PAUSED` error

Severity: Informational

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

Description: Transfers initiated during pause results in `ERR_MINTING_PAUSED` error (both `transfer` / `transferFrom`) which can be confusing.

universal-token/lib.rs#L65-L149:

```rust
/// Implements ERC-20 `transfer(to, amount)` using the caller as the sender.
fn erc20_transfer_handler<SDK: SharedAPI>(
    sdk: &mut SDK,
    input: &[u8],
) -> Result<EvmExitCode, ExitCode> {
    let is_contract_frozen = sdk.storage(&CONTRACT_FROZEN_STORAGE_SLOT).ok()?;
    if !is_contract_frozen.is_zero() {
        return Ok(ERR_MINTING_PAUSED);
    }

//...
}
```

```rust
/// Implements ERC-20 `transferFrom(from, to, amount)` using caller as the spender.
fn erc20_transfer_from_handler<SDK: SharedAPI>(
    sdk: &mut SDK,
    input: &[u8],
) -> Result<EvmExitCode, ExitCode> {
    let is_contract_frozen = sdk.storage(&CONTRACT_FROZEN_STORAGE_SLOT).ok()?;
    if !is_contract_frozen.is_zero() {
        return Ok(ERR_MINTING_PAUSED);
    }

//...
}
```

Recommendation: Define a new `ERR_TRASNFER_PAUSED` error and replace it in both `transfer` / `transferFrom` handlers.

Fluent Labs: Fixed in PR 258.

Cantina Managed: Fix verified.

3.5.8 Infinite allowance for `universal-token` unsupported

Severity: Informational

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

Description: Context: universal-token/lib.rs#L121-L125.

Standard ERC20 implementations such as OpenZeppelin and Solady treat an approval of `type(uint256).max` as an infinite approval, that is not deducted from during `transferFrom()`. This saves a storage write and is often expected behavior.

Recommendation:

```
   let allowance_accessor = allowance_storage_map.entry(from).entry(spender);
   let allowance = allowance_accessor.get_checked(sdk)?;
-  let Some(new_allowance) = allowance.checked_sub(amount) else {
-       return Ok(ERR_INSUFFICIENT_ALLOWANCE);
-  };
+  if allowance != U256::MAX {
+       let Some(new_allowance) = allowance.checked_sub(amount) else {
+           return Ok(ERR_INSUFFICIENT_ALLOWANCE);
+       };
+       allowance_accessor.set_checked(sdk, new_allowance)?;
+   }
  // ...
-  allowance_accessor.set_checked(sdk, new_allowance)?;
```

Fluent Labs: Fixed in PR 258.

Cantina Managed: Fix verified.

3.5.9 LRU module cache replacements fail instead of evicting least recently used

Severity: Informational

Context: module_factory.rs#L203

Description: When replacing a LRU module in the cache with a bigger-size for the same key, if the new total number of bytes in the cache is greater than the maximum capacity of the LRU cache: `self.current_bytes + diff > self.max_bytes`, then the replacement will fail and the replaced module will be evicted from LRU cache. (`schnellru` evicts the replaced module if `on_replace` returns false).

```
fn on_replace(
    &mut self,
    _length: usize,
    _old_key: &mut B256,
    _new_key: Self::KeyToInsert<'_>,
    old_value: &mut V,
    new_value: &mut V,
) -> bool {
//...
    if new_size > old_size {
        let diff = new_size - old_size;
        if self.current_bytes + diff > self.max_bytes {
            return false;
        }
```

The expected behaviour is that the LRU evicts the least-recently used modules instead, to make space for the replaced module. The impact of this issue is minor as it causes incorrect caching which only impacts performance.

Recommendation: Replace the check with the following:

```
if new_size > self.max_bytes {
    return false;
}
```

Fluent Labs: Fixed in PR 266.

Cantina Managed: Fix verified.

3.5.10 Actual `MSH_I64_SHL` does not match comment

Severity: Informational

Context: bitwise.rs#L10, bitwise.rs#L87-L88

Description: The actual `MSH_I64_SHL` constant used did not match the comment. For `op_i64_shl`, the comment denotes the max stack height to be 19.

src/instruction_set/bitwise.rs#L87:

```
/// Max stack height: 19
pub fn op_i64_shl(&mut self) {
```

But the defined `MSH_I64_SHL` constant is 10.

src/instruction_set/bitwise.rs#L10:

```
pub const MSH_I64_SHL: u32 = 10;
```

Recommendation: Match the `MSH_I64_SHL` constant to the comment indicating 19 to allow a buffer for error.

Fluent Labs: Fixed in PR 107.

Cantina Managed: Fix verified.

3.5.11 `syscall_weierstrass*` operations do not support infinity points

Severity: Informational

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

Description: The `syscall_weierstrass*` operations do not support infinity points (identity element). Technically, the infinity point does not have any actual coordinate on the curve and implementations commonly use alternative notation to represent them. For instance, the EVM uses a coordinate of all zeros to represent the infinity point.

For `syscall_weierstrass*` operations, they do not allow specifying the infinity point as an input as they do not use any alternative notation. The underlying SP1 also does not support point addition when the result would be the infinity point providing undefined behaviour as a result (eg. addition of a point and its negation, $ P + (-P) = O $). This is marked as informational because, technically SP1 also does not support infinity points in their `weierstrass` operations, so it could be seen as intended behaviour since these `runtime` syscalls are meant to maintain parity with the SP1-supported syscalls.

Recommendation: Add support for infinity points, if it is required.

Fluent Labs: Acknowledged. We can't support it, because of SP1. We should have the same behaviour as SP1 otherwise we will have to modify circuits, that we're trying to avoid. Also for ETH weierstrass support we use different library that is fully ETH compatible.

Cantina Managed: Acknowledged.

3.5.12 `max_drop_keep_fuel` is tracked but unused

Severity: Informational

Context: translator.rs#L1137

Description: The variable `max_drop_keep_fuel` is unused to actually bump fuel consumption as of commit d787689821ad963b916c30e6c29e42c2c1ecbb5f (it is part of PR 51). However, the `max_drop_keep_fuel` variable is still being updated and calculated. Since it is written to and referenced, the Rust compiler does not flag it as unused. However, it is no longer serving any clear purpose.

Recommendation: Delete the `max_drop_keep_fuel` and it's uses in `compute_instr()`.

Fluent Labs: Fixed in PR 105.

Cantina Managed: Fix verified.