code4rena

Succinct

Smart Contract Security Assessment

Audit dates: Sep 15 — Oct 15, 2025

Overview

About C4

Code4rena (C4) is a competitive audit platform where security researchers, referred to as Wardens, review, audit, and analyze codebases for security vulnerabilities in exchange for bounties provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the Succinct smart contract system. The audit took place from September 15 to October 15, 2025.

Final report assembled by Code4rena.

Summary

The C4 analysis yielded an aggregated total of 8 unique vulnerabilities. Of these vulnerabilities, 1 received a risk rating in the category of HIGH severity and 7 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 4 reports detailing issues with a risk rating of LOW severity or non-critical.

All of the issues presented here are linked back to their original finding, which may include relevant context from the judge and Succinct team.

Scope

The code under review can be found within the C4 Succinct repository, and is composed of 242 smart contracts written in the Rust programming language and includes 42,043 lines of Rust code.

The code in C4's Succinct repository was pulled from:

- Repository: https://github.com/succinctlabs/sp1-wip
- Commit hash: `0283d21f9f73d3de246d95621c591f1efaf8c597`

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges

- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on [the C4 website](), specifically our section on [Severity Categorization]().

High Risk Findings (1)

[[H-01] Range check fails to bind limbs to value, enabling invalid witnesses]()

Submitted by [Matte](), also found by [Almanax](), [Cryptor](), and [zubyoz]()

`crates/recursion/gnark-ffi/go/sp1/koalabear/koalabear.go` [#L79-L97]()

In @crates/recursion/gnark-ffi/go/sp1/koalabear/koalabear.go, the helper `KoalaBearRangeCheck()` splits an input `value` into two hinted parts, a 24-bit `lowLimb` and a 7-bit `highLimb`, and then only enforces per-limb bitlength constraints (via `RangeChecker.Check` or `ToBinary`, depending on `GROTH16`). It also adds a guard that when `highLimb` equals the maximal 7-bit value, `lowLimb` must be zero. Critically, the function never introduces a constraint that ties the hinted limbs back to the original variable; i.e., it does not assert `value == highLimb * 2^24 + lowLimb`. Without this recomposition equation, the limb checks do not constrain `value` at all.

Key nuance: gnark hints do not constrain the circuit by themselves. They are advice values. Any relation between a hint's inputs and outputs must be enforced with explicit constraints. Here, `SplitLimbsHint` computes limbs from `value` and even returns an error if `value >= modulus`, but this check exists only inside the hint function. Since `KoalaBearRangeCheck()` never asserts that the limbs recombine to `value`, a malicious prover could supply arbitrary limb values that merely satisfy bitlength checks, and the circuit would accept them. Consequently, `KoalaBearRangeCheck()` is insufficient as a standalone field-membership or range proof.

Contrast with the correct pattern already present in the code: in `reduceWithMaxBits()` (same file), after calling `SplitLimbsHint` on `remainder`, the code asserts `remainder == highLimb * 2^24 + lowLimb` before applying the bit checks and boundary guard. That recomposition constraint is exactly what's missing from `KoalaBearRangeCheck()` and should be replicated there.

This helper is invoked broadly in @crates/recursion/gnark-ffi/go/sp1/sp1.go over all scalar witness elements and each component of extension-field witnesses. Any path that relies on `KoalaBearRangeCheck()` to enforce validity—without subsequently reducing and asserting equalities involving the original `value`—is vulnerable to accepting out-of-range or inconsistent witnesses.

Exploitation sketch

- Pick any witness `value` that benefits later constraints (e.g., zeroing a product or aligning a digest preimage).
- Choose `highLimb`/`lowLimb` within their bit bounds, avoiding the boundary case `highLimb == 127` to bypass the conditional zero check.
- Since `KoalaBearRangeCheck()` lacks `value` recomposition, the check passes regardless of any relation between `value` and the limbs.
- Downstream gadgets then treat `value` as valid; if no subsequent reduction/equality ties `value` to canonical field form, this permits out-of-range or inconsistent inputs to drive logic and commitments, breaking soundness.

Recommendation

Augment `KoalaBearRangeCheck()` to add the missing recomposition constraint that binds the hinted limbs to the original variable for both proving systems. A minimal, local fix is to assert `value == highLimb * 2^24 + lowLimb` immediately after obtaining the hint, then keep the existing limb range checks and the boundary guard. Prefer integer shifts for powers of two (e.g., `1 << 24`, `1 << 7`) instead of `math.Pow` to avoid float conversions.

Add regression tests:

- A minimal circuit that only calls `KoalaBearRangeCheck(x)` should currently accept arbitrary `x`; after the fix, it must accept only when `x` equals the recomposed limbs.
- A negative test where `value` and `(highLimb, lowLimb)` disagree must be rejected.

```
--- a/crates/recursion/gnark-ffi/go/sp1/koalabear/koalabear.go
+++ b/crates/recursion/gnark-ffi/go/sp1/koalabear/koalabear.go
@@ func (p *Chip) KoalaBearRangeCheck(value frontend.Variable) {
-    lowLimb := new_result[0]
-    highLimb := new_result[1]
+    lowLimb := new_result[0]
+    highLimb := new_result[1]
+
+    // Bind limbs to the source variable.
+    p.api.AssertIsEqual(
+        p.api.Add(
+            p.api.Mul(highLimb, frontend.Variable(uint64(1<<24))),
+            lowLimb,
+        ),
+        value,
+    )
@@
-    shouldCheck := p.api.IsZero(p.api.Sub(highLimb, uint64(math.Pow(2,
7))-1))
```

```
+    // Optional: prefer shifts for 2^k constants.
+    shouldCheck := p.api.IsZero(p.api.Sub(highLimb, uint64((1<<7)-1)))
     p.api.AssertIsEqual(
         p.api.Mul(
             shouldCheck,
             lowLimb,
         ),
         frontend.Variable(0),
     )
```

Medium Risk Findings (7)

[M-01] PLONK/Groth16 verifiers accept proofs with untrusted recursion vk root

Submitted by Matte, also found by mohamedfahmy and niffylord

`crates/prover/src/verify.rs` #L565-L624

The BN254 final verification helpers `verify_plonk_bn254()` and `verify_groth16_bn254()` in @crates/prover/src/verify.rs parse `vk_root`, `proof_nonce`, and other public inputs from `proof.public_inputs`, invoke the scheme verifier (PLONK/Groth16), and then re-check only the verifying key hash and the SP1 public values hash via `verify_plonk_bn254_public_inputs()` / `verify_groth16_bn254_public_inputs()`. Neither function enforces that the public `vk_root` equals the trusted recursion verification-key allowlist root maintained by the prover (`self.recursion_prover.recursion_vk_root`).

Both `verify_plonk_bn254()` and `verify_groth16_bn254()` omit an equality check between the parsed `vk_root` and the prover's trusted `recursion_vk_root`. Because `vk_root` is carried as a public input and the SP1 public values hash does not attest the allowlist root against a trusted reference here, a prover can generate wrap + final proofs using an unauthorized recursion vk tree while keeping all internal cryptographic checks consistent. The helpers will still accept such proofs since only the vkey hash and public-values hash are re-validated.

A malicious could prover construct a recursion Merkle tree containing unauthorized verification keys (e.g., recursion circuits that omit critical constraints), produces wrap plus PLONK/Groth16 proofs referencing that altered `vk_root`, and submits them for verification. Since neither final verifier cross-checks `vk_root` against `recursion_vk_root`, these proofs are accepted, bypassing the recursion vk allowlist and allowing arbitrary, potentially invalid executions to be certified (e.g., for onchain verification).

Recommendation

Add explicit equality checks in both `verify_plonk_bn254()` and `verify_groth16_bn254()` to ensure the parsed public `vk_root` matches `self.recursion_prover.recursion_vk_root` (converted to the BN254 field representation). Optionally enforce caller-provided expectations for `exit_code` and `proof_nonce` before returning success.

```
--- a/crates/prover/src/verify.rs
+++ b/crates/prover/src/verify.rs
@@ -589,6 +589,13 @@ impl<C: SP1ProverComponents> SP1Prover<C> {
         )?;

+        // Enforce vk_root matches the trusted recursion allowlist root.
+        let expected_vk_root =
+
sp1_primitives::koalabears_to_bn254(&self.recursion_prover.recursion_vk_r
oot)
+                 .as_canonical_biguint();
+        if vk_root != expected_vk_root {
+            return Err(anyhow::anyhow!("vk_root mismatch"));
+        }
         verify_plonk_bn254_public_inputs(vk, public_values,
&proof.public_inputs)?;

         Ok(())
@@ -619,6 +626,13 @@ impl<C: SP1ProverComponents> SP1Prover<C> {
         )?;

+        // Enforce vk_root matches the trusted recursion allowlist root.
+        let expected_vk_root =
+
sp1_primitives::koalabears_to_bn254(&self.recursion_prover.recursion_vk_r
oot)
+                 .as_canonical_biguint();
+        if vk_root != expected_vk_root {
+            return Err(anyhow::anyhow!("vk_root mismatch"));
+        }
         verify_groth16_bn254_public_inputs(vk, public_values,
&proof.public_inputs)?;

         Ok(())
```

[M-02] Truncated `public_values` causes panic in `SP1Prover::verify`

Submitted by mohamedfahmy

- `crates/hypercube/src/air/public_values.rs` #L192-L201
- `crates/prover/src/verify.rs` #L71-L84
- `crates/hypercube/src/verifier/shard.rs` #L458-L466

A malformed proof can set `ShardProof.public_values.len()` to be smaller than the expected number of public value elements. The `Borrow` implementation for `PublicValues` slices the backing slice with `[0..size]` without a release mode length check, causing a panic before verification returns an error.

Borrow impl (unchecked slice): `crates/hypercube/src/air/public_values.rs` (around lines 192—201)

Early borrows in verifier before shard-level checks: `crates/prover/src/verify.rs` (for example: lines ~71, 108, 153, 199, 231, 266, 351, 398)

Although the shard verifier later checks `public_values.len() < self.machine.num_pv_elts()` in `crates/hypercube/src/verifier/shard.rs` around (lines 458-466), the core verifier borrows the slice earlier, so the panic can occur first.

Impact

Any service exposing `SP1Prover::verify` can be crashed by submitting a proof with a truncated `public_values` buffer causing remote DoS.

If built with `panic=abort`, this aborts the process, otherwise it unwinds and terminates the request path. No forged proof is required, only valid serialization with a short vector.

View detailed Proof of Concept

Recommendation

Add an eager length check in `SP1Prover::verify` before any `as_slice().borrow()` calls.

[M-03] Zero Config Values Can Crash or Stall the Prover Indefinitely

Submitted by johnyfwesh, also found by mohamedfahmy

`crates/core/executor/src/record.rs` #L201)

`ExecutionRecord::split` trusts `SplitOpts` values without guarding against zeros. The per-syscall divisor comes straight from the options (`threshold = opts.syscall_threshold[syscall_code]`) and is passed to `events.chunks_exact(threshold)` (`crates/core/executor/src/record.rs:201`). Rust panics on `chunks_exact(0)`, so any zero threshold crashes the prover during shard generation.

When `done == true`, the same routine computes `capacity = 2 * opts.page_prot` and `capacity = 2 * opts.memory` inside loops that slice `init_remaining/finalize_remaining` by `init_to_take` and `finalize_to_take` (crates/core/executor/src/record.rs:250 and crates/core/executor/src/record.rs:314). If either capacity is zero, both counts stay at zero, the slices never shrink, and the loop spins forever while holding the deferred-record mutex—a CPU-burning DoS.

Nothing prevents these zeros from entering the function. `SplitOpts::new` divides the remaining trace area by various costs and then truncates to the nearest multiple of 32 via `trunc_32` (crates/core/executor/src/opts.rs:83-145, crates/core/executor/src/utils.rs:111). Whenever the quotient drops below 32, the threshold collapses to zero. The only assertion checks `available_trace_area >= fixed_trace_area`, so reaching exactly zero slack—or any positive slack insufficient for a single chunk—is accepted.

Attackers do not need privileged configuration access:

- `PrecompileEvents::default` seeds an entry for every `should_send()` syscall, even when the event list is empty. A zeroed threshold therefore triggers `chunks_exact(0)` immediately on the first shard (crates/core/executor/src/events/precompiles/mod.rs:246-279).
- The public prover APIs propagate `SP1CoreOpts` straight from environment variables or user-supplied configs. A large-yet-valid ELF (≈23.6M instructions) already causes `available_trace_area` to fall below the minimum cost, yielding zero thresholds under default settings. Users can also set `HEIGHT_THRESHOLD=1` or otherwise override the options to hit the same state.

Affected Code

```
// crates/core/executor/src/record.rs:188-360
for (syscall_code, events) in precompile_events.into_iter() {
    let threshold: usize = opts.syscall_threshold[syscall_code];

    let chunks = events.chunks_exact(threshold); // panic when threshold
== 0
    …
}

let capacity = 2 * opts.page_prot; // zero ⇒ init_to_take ==
finalize_to_take == 0
let init_to_take = init_remaining.len().min(capacity);
let finalize_to_take = finalize_remaining.len().min(capacity -
init_to_take);
…
init_remaining = &init_remaining[init_to_take..]; // never advances when
both counts are 0
```

Impact

- **Deterministic panic:** Any zero `syscall_threshold` index triggers `slice::chunks_exact(0)`, terminating the worker thread and aborting proof generation ([`crates/core/executor/src/record.rs:201`]()).
- **Livelock/DoS:** Zero `memory` or `page_prot` thresholds trap the splitter in infinite loops, monopolising the record mutex and stalling the prover ([`crates/core/executor/src/record.rs:250`](), [`crates/core/executor/src/record.rs:314`]()).
- **Attacker-controlled reachability:** `SplitOpts` is `Serialize`/`Deserialize`, exposed through SDK builder APIs, and populated from unvalidated environment variables ([`crates/core/executor/src/opts.rs:36-56`]()). Large programs or crafted configs make zeros unavoidable under default logic.

[View detailed Proof of Concept]()

Recommended Mitigation

- Enforce minimum values when constructing `SplitOpts` (e.g., clamp each threshold to at least `1`, or wrap them in `NonZeroUsize` and surface configuration errors early).
- Add defensive assertions in `ExecutionRecord::split` to fail fast if a zero sneaks through, ensuring future regressions are caught during testing.
- Consider distributing trace area using `usize::max(value, 1)` (or similar) instead of hard truncation via `trunc_32`.
- Cover these invariants with unit tests that instantiate `SplitOpts` near the boundary conditions.

References

- [`crates/core/executor/src/record.rs:201`]()
- [`crates/core/executor/src/record.rs:250`]()
- [`crates/core/executor/src/record.rs:314`]()
- [`crates/core/executor/src/opts.rs:36-145`]()
- [`crates/core/executor/src/utils.rs:111`]()
- [`crates/core/executor/src/events/precompiles/mod.rs:246-279`]()

[[M-04] Public verifier API panics on malformed inputs, enabling DoS]()

Submitted by [valarislife]()

- `crates/verifier/src/plonk/mod.rs` [#L120-L129]()
- `crates/verifier/src/converter.rs` [#L15-L30]()

The `PlonkVerifier::verify_gnark_proof` function unwraps fallible decoding operations for the PLONK verifying key, proof, and public inputs without proper error handling. When malformed inputs are provided, these `.unwrap()` calls cause the program to panic, enabling a denial-of-service (DoS) attack.

Affected code (lines 120-129 in `crates/verifier/src/plonk/mod.rs`):

```
pub fn verify_gnark_proof(
    proof: &[u8],
    public_inputs: &[[u8; 32]],
    plonk_vk: &[u8],
) -> Result<(), PlonkError> {
    let plonk_vk =
load_plonk_verifying_key_from_bytes(plonk_vk).unwrap(); // ⚠️ panic on
malformed VK
    let proof = load_plonk_proof_from_bytes(proof,
plonk_vk.qcp.len()).unwrap(); // ⚠️ panic on malformed proof

    let public_inputs =
        public_inputs.iter().map(|input|
Fr::from_slice(input).unwrap()).collect::<Vec<_>>(); // ⚠️ panic on
malformed input
    verify_plonk_algebraic(&plonk_vk, &proof, &public_inputs)
}
```

Similar patterns exist in `crates/verifier/src/converter.rs` when parsing compressed points.

Impact:

1. **Denial of Service**: Any service or application that forwards user-supplied proof/VK bytes to `verify_gnark_proof` can be crashed by malformed inputs.
2. **Repeated attacks:** Malicious actors can repeatedly submit invalid inputs to maintain persistent DoS.
3. **Process termination**: The panic terminates the entire process, not just the verification operation.
4. **Public API exposure**: This is a public API function that may be called from web services, APIs, or other untrusted contexts.

Exploit scenario:

An attacker identifies a web service that accepts ZK proofs for verification. The service uses `PlonkVerifier::verify_gnark_proof` to validate submissions. The attacker submits:

- A truncated or malformed verifying key (e.g., only 3 bytes instead of expected size)
- Invalid proof bytes
- Malformed public input arrays

The service panics and crashes. By repeatedly sending malformed inputs, the attacker can keep the service offline, preventing legitimate users from verifying valid proofs.

Recommended mitigation steps

1. **Replace all `.unwrap()` calls with proper error propagation:**

```
pub fn verify_gnark_proof(
    proof: &[u8],
    public_inputs: &[[u8; 32]],
    plonk_vk: &[u8],
) -> Result<(), PlonkError> {
    let plonk_vk = load_plonk_verifying_key_from_bytes(plonk_vk)
        .map_err(|_| PlonkError::InvalidVerifyingKey)?;
    let proof = load_plonk_proof_from_bytes(proof,
plonk_vk.qcp.len())
        .map_err(|_| PlonkError::InvalidProof)?;

    let public_inputs: Result<Vec<_>, _> = public_inputs
        .iter()
        .map(|input| Fr::from_slice(input))
        .collect();
    let public_inputs = public_inputs
        .map_err(|_| PlonkError::InvalidPublicInput)?;

    verify_plonk_algebraic(&plonk_vk, &proof, &public_inputs)
}
```

2. **Validate input lengths and structure** before attempting to parse.
3. **Add fuzzing tests** for proof/VK decoders in CI to catch similar issues.
4. **Document expected input formats** and size constraints in the API documentation.
5. **Add error variants** to `PlonkError` enum:

```
pub enum PlonkError {
    // existing variants...
    InvalidVerifyingKey,
    InvalidProof,
    InvalidPublicInput,
}
```

View detailed Proof of Concept

[M-05] SP1 host verifier rejects PLONK and Groth16 proofs with BLAKE3 hashed public values

Submitted by mohamedfahmy

- `crates/prover/src/verify.rs` #L631-L647
- `crates/prover/src/verify.rs` #L654-L672

The host side verifier in `crates/prover/src/verify.rs` validates public values using only SHA256 for PLONK, and for Groth16 it returns early on SHA256 mismatch, making the intended SHA256 or BLAKE3 fallback unreachable

- PLONK: `verify_plonk_bn254_public_inputs` compares `public_values.hash_bn254()` against the expected digest and errors on mismatch, with no fallback. File: `crates/prover/src/verify.rs` lines 631—647
- Groth16: `verify_groth16_bn254_public_inputs` does the same SHA256-only check and returns before calling `verify_public_values`, so the fallback is never tried. File: `crates/prover/src/verify.rs` lines 654—672
- The comment and helper `verify_public_values` state that both SHA2 and BLAKE3 digests should be accepted, but they are not used in the PLONK path and are unreachable in the Groth16 path. File: `crates/prover/src/verify.rs` lines 676—699

The separate `crates/verifier` path correctly implements a dual hash fallback, so BLAKE3 paths pass there, but host verification in `crates/prover` fails.

Impact

Deterministic failure (DoS) for legitimate proofs when the producer uses BLAKE3 for public values (for example: `PLONK_BLAKE3_ELF`).

Interoperability and UX issues because no soundness reduction (still strict equality on a collision resistant digest).

View detailed Proof of Concept

Recommendation

Accept both SHA256 and BLAKE3 by adding a dual hash check and remove the early return and use the existing helper to accept both.

[M-06] Proof nonce clobbered when packing deferred page protection events into last shard

Submitted by mohamedfahmy

- `crates/core/executor/src/record.rs` #L288-L291

- `crates/core/machine/src/executor.rs` [#L300-L309]
- `crates/prover/src/verify.rs` [#L223-L237]

When finishing execution and packing deferred page protection (page_prot) events into the already-produced last CPU shard, the code overwrites the last shard's `public_values.proof_nonce` with the deferred accumulator's nonce. Because the deferred accumulator is initialized with default public values and never populated with the real proving nonce, its `proof_nonce` remains `[0; 4]`. This clobbers the correct nonce set on CPU shards and causes a mismatch across shards.

Overwrite site: `crates/core/executor/src/record.rs` in `ExecutionRecord::split` (around line 289) assigns: `last_record_ref.public_values.proof_nonce = self.public_values.proof_nonce;`

Deferred accumulator creation (nonce stays default): `crates/core/machine/src/executor.rs` constructs `ExecutionRecord::new(program.clone())` for `deferred` and never sets its `public_values.proof_nonce`.

Verifier check: `crates/prover/src/verify.rs` requires all shards to have identical `proof_nonce`.

This path is reached when execution is done (`done == true`), no precompile shards were emitted (so the memory/page_prot events can be packed into `last_record`) and the deferred sets are below packing thresholds.

Impact

Valid executions that end with a small batch of deferred page-prot events fail verification because the final shard's nonce becomes `[0;4]` while earlier shards retain the real nonce.

Breaks the entire proof for affected runs.

Does not enable acceptance of invalid proofs it only causes valid proofs to be rejected.

View detailed Proof of Concept

Recommendation

Do not overwrite the nonce when packing into an existing last shard.

[M-07] Unbounded decimal public inputs enable verifier-side DoS via oversized numeric parsing

Submitted by EV_om

- `crates/recursion/gnark-ffi/go/sp1/verify.go` #L39-L51
- `crates/recursion/gnark-ffi/go/sp1/sp1.go` #L28-L37

In the Go gnark FFI verifier, `VerifyPlonk` and `VerifyGroth16` accept the five SP1 public inputs (`vkey_hash`, `committed_values_digest`, `exit_code`, `vk_root`, `proof_nonce`) as arbitrary strings and directly assign them to the circuit's public fields as `frontend.Variables` without any length or canonicalization checks. See `VerifyPlonk` constructing the `circuit` with `VkeyHash`, `CommittedValuesDigest`, `ExitCode`, `VkRoot`, and `ProofNonce` set from caller-provided strings, followed by `frontend.NewWitness`:

- `VerifyPlonk`
- Circuit public fields typed as `frontend.Variable` with `gnark:",public"`: `Circuit` **struct**

gnark will parse these strings into field elements internally when building the witness. If an attacker supplies extremely long decimal strings (e.g., millions of digits) that still reduce to the correct field values modulo BN254, `frontend.NewWitness` performs proportional big-integer parsing/coercion, imposing significant CPU and memory overhead before any constraint checks or early rejection. There is no pre-bound on length nor canonical encoding enforced at this boundary.

This does not create a soundness break for SP1: the Rust side later compares the proof's public inputs to expected values using canonical `BigUint` equality. However, the Go verification entrypoints can be driven into heavy computation with large untrusted numeric strings, which is a verifier-side DoS/availability issue consistent with the contest's Medium severity category.

Worst case, an attacker can reliably induce multi-second CPU spikes or worse by inflating the decimal size of any of the five public inputs, even if the underlying proof and vk are otherwise valid. If this surface is exposed to untrusted users (e.g., a service verifying proofs pre-Rust checks), it can degrade availability.

Recommended mitigation steps

- Enforce canonical, bounded input encoding for public inputs at the Go boundary:
  - Prefer fixed-size hex strings (32 bytes) for each public input, with strict length validation, or
  - Parse decimal strings with a hard maximum length (sufficient for BN254 scalar range), reduce modulo p, and re-encode canonically (e.g., fixed-width hex) before constructing the gnark witness.
- Fail fast if inputs exceed expected size or do not match canonical formatting to avoid expensive parsing inside gnark.

View detailed Proof of Concept

Low Risk and Informational Issues

For this audit, 4 QA reports were submitted by wardens detailing low risk and informational issues. The QA report highlighted below by **KKKKK** received the top score from the judge. 2 Low-severity findings were also submitted individually, and can be viewed here.

The following wardens also submitted QA reports: Afriauditor, niffylord, and timijcool.

Incorrect Cost Rate Used for StoreDouble Operations

`crates/core/executor/src/cost.rs` #L137

The `estimate_trace_elements` function in `cost.rs` incorrectly uses `RiscvAirId::StoreWord`'s cost rate when calculating costs for `RiscvAirId::StoreDouble` operations.

Affected Code (Line 137):

cells += (num_events_per_air[RiscvAirId::StoreDouble]).next_multiple_of(32)

- costs_per_air[RiscvAirId::StoreWord]; // ← Should be StoreDouble

Root Cause:

StoreWord and StoreDouble have different column structures:

- `StoreWordColumns` has **9 fields** (includes `offset_bit` and `store_value`)
- `StoreDoubleColumns` has **7 fields** (does not have these extra fields)

This difference results in different `NUM_STORE_WORD_COLUMNS` vs `NUM_STORE_DOUBLE_COLUMNS` values, meaning they should have different cost rates.

Verification:

StoreWordColumns has 9 public fields

grep -A 50 "pub struct StoreWordColumns" crates/core/machine/src/memory/instructions/store/store_word.rs | grep "pub " | wc -l

Output: 9

StoreDoubleColumns has 7 public fields

grep -A 50 "pub struct StoreDoubleColumns" crates/core/machine/src/memory/instructions/store/store_double.rs | grep "pub " | wc -l

Output: 7

Impact

Informational - This causes cost estimation to be approximately 29% too high for programs using SD (store double) instructions. While this leads to inefficient resource allocation, it is conservative (over-estimates rather than under-estimates) and does not affect proof soundness or correctness.

Recommended Mitigation

Change line 137 to use the correct cost rate:

cells += (num_events_per_air[RiscvAirId::StoreDouble]).next_multiple_of(32)

- costs_per_air[RiscvAirId::StoreWord];
- costs_per_air[RiscvAirId::StoreDouble];

This ensures StoreDouble operations are costed according to their actual column count rather than StoreWord's larger column count.

Disclosures