CANTINA

Succinct sp1

Competition

July 19, 2024

Contents

1 Introduction

1.1 About Cantina

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

1.2 Disclaimer

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

SP1 is a performant, 100% open-source, contributor-friendly zero-knowledge virtual machine (zkVM) that verifies the execution of arbitrary Rust (or any LLVM-compiled language) programs.

From Jun 3rd to Jun 23rd Cantina hosted a competition based on sp1. The participants identified a total of **25** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 6
- Low Risk: 9
- Gas Optimizations: 0
- Informational: 10

***Note**: There have been no **High** severity findings reported during the initial judging phase of the competition. However, the Succinct team identified and rewarded findings `3.1.1` to `3.1.6` as undesired behaviour in the code and not as security vulnerabilities.*

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

3 Findings

3.1 Medium Risk

3.1.1 `ProgramChip` can receive all-zero columns

Submitted by georg

Severity: Medium Risk

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

Description: In mod.rs#L140, the program is padded with zeros (PC of 0, instruction flags of 0, opcode of 0, etc...). The CPU can look up these columns, because the `PragmChip` "*receives*" the program with arbitrary multiplicity. This effectively adds a line to the program.

To exploit this, the attacker would have to find a way to set the PC to 0. I don't see an obvious way to do that, but it would certainly be possible to write a custom assembly program that jumps to 0 using the `JALR` instruction.

Recommendation: The multiplicity should be enforced to be 0 in the padded rows.

3.1.2 Possible overflows of Syscall arguments

Submitted by georg

Severity: Medium Risk

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

Description: There are a few places where `Word::reduce()` is called operands `a`, `b` or `c`, which are not checked to be below the BabyBear prime:

- When calling `send_syscall` (see ecall.rs#L52-L53). Syscall arguments are mostly memory pointers, which would cause reading from a wrong address.
- When verifying the `deferred_proofs_digest` public value (ecall.rs#L170).
- When setting the exit code (ecall.rs#L193). Here, a exit code equal to the BabyBear prime would overflow to 0 (see proof of concept below).

Proof of concept: This patch demonstrates that even though we change the Fibonacci example to exit with a non-zero exit code, the claimed exit code in the public values is 0:

```
diff --git a/core/src/runtime/syscall.rs b/core/src/runtime/syscall.rs
index 7cb1cb33..71d1b34f 100644
--- a/core/src/runtime/syscall.rs
+++ b/core/src/runtime/syscall.rs
@@ -260,7 +260,7 @@ impl<'a> SyscallContext<'a> {
     }

     pub fn set_exit_code(&mut self, exit_code: u32) {
-        self.exit_code = exit_code;
+        self.exit_code = exit_code % 0x78000001;
     }
 }

diff --git a/examples/fibonacci/program/src/main.rs b/examples/fibonacci/program/src/main.rs
index b968e86b..551d757e 100644
--- a/examples/fibonacci/program/src/main.rs
+++ b/examples/fibonacci/program/src/main.rs
@@ -34,4 +34,13 @@ pub fn main() {
     // outputs to the prover.
     sp1_zkvm::io::commit(&a);
     sp1_zkvm::io::commit(&b);
+
+    unsafe {
+        core::arch::asm!(
+            "ecall",
+            in("t0") sp1_zkvm::syscalls::HALT,
+            // Overflows to 0
+            in("a0") 0x78000001
+        );
```

```
+    }
 }
diff --git a/prover/src/verify.rs b/prover/src/verify.rs
index f829f4d6..d70043b1 100644
--- a/prover/src/verify.rs
+++ b/prover/src/verify.rs
@@ -49,6 +49,7 @@ impl SP1Prover {
         // Verify shard transitions.
         for (i, shard_proof) in proof.0.iter().enumerate() {
             let public_values = PublicValues::from_vec(shard_proof.public_values.clone());
+            println!("public_values: {:?}", public_values);
             // Verify shard transitions
             if i == 0 {
                 // If it's the first shard, index should be 1.
diff --git a/sdk/src/provers/local.rs b/sdk/src/provers/local.rs
index 5eadf79a..f74e5d03 100644
--- a/sdk/src/provers/local.rs
+++ b/sdk/src/provers/local.rs
@@ -48,6 +48,7 @@ impl Prover for LocalProver {
         let proof = self.prover.prove_core(pk, &stdin)?;
         let deferred_proofs = stdin.proofs.iter().map(|p| p.0.clone()).collect();
         let public_values = proof.public_values.clone();
+        self.prover.verify(&proof.proof, &pk.vk).unwrap();
         let reduce_proof = self.prover.compress(&pk.vk, proof, deferred_proofs)?;
         Ok(SP1CompressedProof {
             proof: reduce_proof.proof,
```

Running `cd examples/fibonacci && cargo run -r`, one can see that:

- The public values claim that the exit code is 0.
- The proof generation succeeds and is accepted by the verifier.

Recommendation: The `BabyBearWordRangeChecker` gadget should be used in cases where `Word::reduce()` is called. Perhaps it's helpful to introduce a new type for a range checked word, so that the `reduce()` function is only available once the word has been passed through the `BabyBearWordRangeChecker` gadget.

3.1.3 Overflow possible in `eval_memory_access_slice`

Submitted by georg

Severity: Medium Risk

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

Description: In SP1, memory addresses are typically asserted to be smaller than the BabyBear prime $p = 15 \times 2^{27} + 1$. However, the `MemoryAirBuilder::eval_memory_access_slice` function adds to a (range-checked) address, possibly causing an overflow. This function is used by various precompiles to read / write memory slices.

For example, a malicious actor can pass $p-4$ as the address, causing the function to read / write addresses $p - 4$, 0, 4, ...

In the case of SP1, these addresses coincide with registers x0, x4, etc, which is a known issue and therefore out of scope for this competition. However, this issue would remain even if this was fixed (for example by introducing a separate memory for registers).

Proof of concept: The vulnerability can be demonstrated by applying the following patch:

```
diff --git a/core/src/air/builder.rs b/core/src/air/builder.rs
index d957f43c..f421c6db 100644
--- a/core/src/air/builder.rs
+++ b/core/src/air/builder.rs
@@ -490,6 +490,7 @@ pub trait MemoryAirBuilder: BaseAirBuilder {
                 shard,
                 channel.clone(),
                 clk.clone(),
+                // This computation can overflow.
                 initial_addr.clone().into() + Self::Expr::from_canonical_usize(i * 4),
                 access_slice,
                 verify_memory_access,
diff --git a/core/src/runtime/mod.rs b/core/src/runtime/mod.rs
```

```
index 3eb8c4ea..ee4b3a39 100644
--- a/core/src/runtime/mod.rs
+++ b/core/src/runtime/mod.rs
@@ -231,6 +231,7 @@ impl Runtime {

     /// Get the current value of a word.
     pub fn word(&self, addr: u32) -> u32 {
+        let addr = addr % 0x78000001;
         match self.state.memory.get(&addr) {
             Some(record) => record.value,
             None => 0,
@@ -261,6 +262,7 @@ impl Runtime {

     /// Read a word from memory and create an access record.
     pub fn mr(&mut self, addr: u32, shard: u32, timestamp: u32) -> MemoryReadRecord {
+        let addr = addr % 0x78000001;
         // Get the memory record entry.
         let entry = self.state.memory.entry(addr);

@@ -309,6 +311,7 @@ impl Runtime {

     /// Write a word to memory and create an access record.
     pub fn mw(&mut self, addr: u32, value: u32, shard: u32, timestamp: u32) -> MemoryWriteRecord {
+        let addr = addr % 0x78000001;
         // Get the memory record entry.
         let entry = self.state.memory.entry(addr);

diff --git a/core/src/syscall/precompiles/mod.rs b/core/src/syscall/precompiles/mod.rs
index bc08c685..332a8c84 100644
--- a/core/src/syscall/precompiles/mod.rs
+++ b/core/src/syscall/precompiles/mod.rs
@@ -42,9 +42,10 @@ pub fn create_ec_add_event<E: EllipticCurve>(
 ) -> ECAddEvent {
     let start_clk = rt.clk;
     let p_ptr = arg1;
-    if p_ptr % 4 != 0 {
-        panic!();
-    }
+    // This check would fail
+    // if p_ptr % 4 != 0 {
+    //     panic!();
+    // }
     let q_ptr = arg2;
     if q_ptr % 4 != 0 {
         panic!();
diff --git a/examples/fibonacci/program/src/main.rs b/examples/fibonacci/program/src/main.rs
index b968e86b..d4877bb1 100644
--- a/examples/fibonacci/program/src/main.rs
+++ b/examples/fibonacci/program/src/main.rs
@@ -8,6 +8,10 @@
 #![no_main]
 sp1_zkvm::entrypoint!(main);

+extern "C" {
+    fn syscall_secp256k1_add(p: *mut u32, q: *const u32);
+}
+
 pub fn main() {
     // Read an input to the program.
     //
@@ -18,6 +22,53 @@ pub fn main() {
     // Write n to public input
     sp1_zkvm::io::commit(&n);

+    let p: u32 = 0x78000001;
+
+    // Write to address p - 1 (happens to be divisible by 4)
+    // This is fine, because p - 1 is a BabyBear field element.
+    unsafe {
+        let addr = (p - 1) as *mut u8;
+        *addr = 0xFF;
+    }
+
+    // Write to address p + 3
+    // This would fail, because addresses are enforced to be < p.
+    // unsafe {
```

```
+    //     let addr = (p + 3) as *mut u8;
+    //     *addr = 0xFF;
+    // }
+
+    // Some EC point
+    let b: [u8; 64] = [
+        152, 23, 248, 22, 91, 129, 242, 89, 217, 40, 206, 45, 219, 252, 155, 2, 7, 11, 135, 206,
+        149, 98, 160, 85, 172, 187, 220, 249, 126, 102, 190, 121, 184, 212, 16, 251, 143, 208, 71,
+        156, 25, 84, 133, 166, 72, 180, 23, 253, 168, 8, 17, 14, 252, 251, 164, 93, 101, 196, 163,
+        38, 119, 218, 58, 72,
+    ];
+
+    // Let's store the value at memory address 4.
+    let value_at_addr4_before = unsafe { *(4 as *const u32) };
+
+    // This reads & writes the addresses p - 64, p - 60, ..., p - 4.
+    // No overflow yet.
+    // But notice that the addresses are not properly aligned, this should not be possible!
+    unsafe {
+        syscall_secp256k1_add((p - 64) as *mut u32, b.as_ptr() as *const u32);
+    }
+
+    // We have not changed the value at address 4.
+    let value_at_addr4_after = unsafe { *(4 as *const u32) };
+    assert_eq!(value_at_addr4_before, value_at_addr4_after);
+
+    // This reads & writes the addresses p - 56, p - 52, ..., p - 4, 0, 4.
+    unsafe {
+        syscall_secp256k1_add((p - 56) as *mut u32, b.as_ptr() as *const u32);
+    }
+
+    // Validate that the value at address 4 has changed.
+    let value_at_addr4_after = unsafe { *(4 as *const u32) };
+    assert_ne!(value_at_addr4_before, value_at_addr4_after);
+
     // Compute the n'th fibonacci number, using normal Rust code.
     let mut a = 0;
     let mut b = 1;
```

The patch slightly modifies witness generation code so that the memory overflow is simulated correctly, but does not change any constraints. Then, the fibonacci example is amended with code that demonstrates the bug when using the `syscall_secp256k1_add` function.

To run: `cd examples/fibonacci && cargo run -r`

Recommendation: One way to fix is to force addresses to be smaller than `p - <some small constant>` in the `BabyBearBitDecomposition` gadget. Then `MemoryAirBuilder::eval_memory_access_slice` could assert that the length of the slice is such that an overflow cannot happen.

3.1.4 `uint256` multiplication syscall cannot succeed when `x_ptr == y_ptr`

Submitted by cergyk

Severity: Medium Risk

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

Description: We can see during the `UINT256_MUL` syscall, that the access being done on the section of memory after `x_ptr` and the one after `y_ptr` is done on the same `clk` value (air.rs#L374 and air.rs#L385).

This means that the interactions with memory cannot be successful if `x_ptr == y_ptr`, because each of those interactions would need a previous value which has a strictly lower clock (as checked in `eval_memory_access_timestamp`; builder.rs#L543-L554).

Recommendation: Please consider offsetting the clock of the `x_ptr` memory access by 1, as done in other precompiles (see weierstrass_add.rs#L480-L495). This would allow for the use of the same `x_ptr` and `y_ptr` during `uint256` multiplication.

3.1.5 Unaligned memory accesses are possible in MemoryChip and precompiles, breaking correctness

Submitted by cergyk

Severity: Medium Risk

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

Description: In sp1 zkVm, memory is divided in words of 4 bytes. Accesses for read and writes must access 1 word of memory, so checks for alignment are enforced when an instruction is `is_memory_instruction` in the cpu air during `eval_memory_address_and_access`.

However there are no such checks during memory initialization/finalization in `MemoryChip` (only checks for all memory addresses to be different), and no such checks during evaluation of precompiles. This means that a prover can generate a program where some unaligned accesses receive incorrect memory values, breaking program correctness/riscv compliance.

- **Memory Initialization**: During memory initialization, it is checked that all of the addresses accessed are different global.rs#L234-L275. However it is not checked that these addresses are word aligned. This means that a prover can initialize unaligned addresses 'in-between' aligned addresses:

```
|       aligned-word 1        |       aligned-word 2        |
            |       unaligned-word 1        |       unaligned-word 2

N           N+2               N+4           N+6             N+8
```

  Same will be done for Finalization.

- **Precompile accesses**: In sp1 zkVM, precompile make a memory-access to fetch their arguments. This means that if the received pointer points to unaligned addresses which have been initialized as seen before, the program will be able to verify.

  > Note that alignment is checked during witness/trace generation.

  If we take the example of a `uint256` multiplication operation, the use of a read/write to such unaligned addresses would leave `aligned` memory unchanged, and thus when later accessing that section of memory in main program with regular `is_memory_instruction` instructions, the read would yield stale values, and altering program flow.

Recommendation:

- Enforce that all addresses are word-aligned during memory initialization and finalization.
- Optionally enforce that all pointers passed to precompiles are word-aligned as well

3.1.6 `uint256` precompile is underconstrained, result is not reduced by modulo

Submitted by cergyk

Severity: Medium Risk

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

Description: `uint256` syscall (air.rs) aims at emulating the mulmod operation on `uint256`.

Given 3 operands `a, b, modulus` and a `result` all of type `uint256` it aims to prove that `a * b = result mod(modulus)`

`uint256` syscall uses the field_op.rs::eval_with_modulus utility to check the following formula is verified:

```
a * b - result - modulus*carry = 0
```

However this formula fails to constraint that `result` is reduced modulo `modulus` (e.g `result < modulus`). As a result the prover can create a proof of a wrong `mulmod` result when modulus is not `2**256`.

> This limitation of the field operation utility has been noted in starkyx audit. This means that it is the reponsibility of the caller to `field_op` to check the result is reduced. For reference this is done correctly in field_sqrt.rs by doing a range check on the computed value.

Proof of concept: It suffices to modify the witness generation code in field_op.rs:

```
  pub fn populate_carry_and_witness(
      &mut self,
      a: &BigUint,
      b: &BigUint,
      op: FieldOperation,
      modulus: &BigUint,
  ) -> BigUint {
      let p_a: Polynomial<F> = P::to_limbs_field::<F, _>(a).into();
      let p_b: Polynomial<F> = P::to_limbs_field::<F, _>(b).into();
      let (result, carry) = match op {
          FieldOperation::Add => ((a + b) % modulus, (a + b - (a + b) % modulus) / modulus),
          FieldOperation::Mul => {
+             if (a * b > modulus.clone()
+                 && a * b + modulus.clone() < (BigUint::from(1u32) << 256))
+             {
+                 let r = ((a * b) % modulus) + modulus;
+                 (r.clone(), (a * b - r.clone()) / modulus)
+             } else {
            ((a * b) % modulus, (a * b - (a * b) % modulus) / modulus)
+             }
          }
          FieldOperation::Sub | FieldOperation::Div => unreachable!(),
      };
      // debug_assert!(&result < modulus);
      // debug_assert!(&carry < modulus);
      match op {
          FieldOperation::Add => debug_assert_eq!(&carry * modulus, a + b - &result),
          FieldOperation::Mul => debug_assert_eq!(&carry * modulus, a * b - &result),
              FieldOperation::Sub | FieldOperation::Div => unreachable!(),
      }
```

Instead of returning the `a*b % p` as a result, it will return `a*b % p + p`, and also reduces `carry` by `1` so the equation:

```
a*b - carry*p - result = 0
```

still holds.

We can then run the tests in mod.rs to ensure the alternative solution is also accepted.

Recommendation: Add a range check to the uint256 precompile, similarly to what is done in field_sqrt.rs.

A new gadget may need to be implemented, because `field_sqrt` checks that the result is in range of the prime of the field which is static.