MT64Cannon Security Review

Coinbase Protocol Security

Feb 6, 2025

Contents

Review Scope

The following contracts were reviewed from the optimism monorepo at commit b8c011f18c79d735e01168345fc1c6f02fac584f:

- MIPS64.sol
- MIPS64Memory.sol
- MIPS64Syscalls.sol
- MIPS64State.sol
- MIPS64Instructions.sol
- MIPS64Arch.sol

MT64Cannon Overview

Multithreaded (MT) and 64-bit Cannon, known as MT64Cannon in this document, is the newest version of the Cannon Fault Proof Virtual Machine (FPVM). As the name implies, multithreading related system calls have been implemented and Cannon's MIPS ISA implementation has been updated to support a 64-bit word size. This new version of Cannon has two main benefits over previous versions:

- Multithreading support allows the compiled op-program to use Go's Garbage Collector (GC). This also removes the fragile instruction patching that was necessary to run op-program previously.
- 64-bit support gives the op-program many orders of magnitude more address space when running, especially in the heap segment.

The above benefits primarily have to do with memory usage, which is a constraint of the current non-multithreaded, 32-bit version of Cannon. The current version of Cannon has an upper limit on the amount of memory available for op-program throughout the derivation pipeline. As OP-Stack chains like Base scale their throughput with gas limit increases, this places an increased memory burden on Cannon. Therefore, MT64Cannon is designed to prevent op-program from ever reaching a ceiling on memory usage by giving it access to the GC to free unused memory and running it with a significantly larger heap space.

Upgrade Process

This new Cannon implementation will replace the 32-bit, singlethreaded version that is currently in use for all OP-Stack fault proof systems. The FPVM used by the current FaultDisputeGame and

PermissionedDisputeGame types will change, however the dispute game logic itself does not change so there will not be a new game type number. Consequently, when the new FPVM is added there will be games of the same type running with different versions of Cannon. These versions of Cannon are differentiated by their absolute prestate, where the challenger software is capable of running different versions of Cannon based on this prestate value. This ensures continuity during the upgrade process, as there will be active dispute games with the older version of Cannon waiting to be finalized.

System Assumptions

Based on OP's security model that assumes a correctly-implemented op-program for derivation and a compliant Go MIPS compiler, there are several key assumptions with the implementation of MT64Cannon in Solidity:

- There is no exception handling for instructions, except for the cases that would cause the EVM to panic (division by 0, etc.). Additionally, instructions such as TRAP* are not implemented.
- Any MIPS ISA-related version constraints are ignored for instructions.
- For instructions that specify they should fail if the provided address is unaligned or includes certain non-zeroed bits, these errors are ignored in favor of silently updating the address.
- The only instructions that are implemented are the ones that the Go MIPS compiler would generate, which may or may not conform to a specific MIPS ISA version.
- The only system calls that are implemented are the ones that op-program and the GC need to run. Additionally, only a subset of the possible options (flag values, behaviors, etc.) for those system calls may be implemented.

Executive Summary

In total, 1 medium severity finding was identified along with several informational findings. The medium finding has already been fixed, with the fix verified by the reviewers. The informational findings are not required to be fixed, but could help improve documentation and clarity in the more complex sections of the code.

Medium Severity

M-01: Incorrect Implementation of SLL and SLLV

The MIPS opcodes SLL and SLLV are meant to left shift a 32-bit value, and then sign-extend it to a 64-bit value. However, the left shift is done on a 64-bit value, contrary to the MIPS ISA, allowing the sign-extension operation to return incorrect values.

In particular, the resulting value can be one that is not possible from sign-extending a 32-bit number. This could cause the output state to be incorrect as there are now two “valid” results from these opcodes.

Recommendation: SLL and SLLV should be applied to values of type uint32 or cast the value into 32-bits after applying the left shift.

Exploit Scenario: Applying SLL or SLLV on 0x80000000 with a left shift amount of 1 will produce (0xFFFFFFFF « 32) instead of the expected amount 0:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.15;

import { Test } from "forge-std/Test.sol";
import { MIPS64Instructions } from "src/cannon/libraries/
   MIPS64Instructions.sol";

contract MIPS64InstructionsTest is Test {

    uint32 internal constant U32_MASK = 0xFFffFFff;

    function setUp() public {
    }

    function testSLL() external {
        uint8 shiftAmt = 1;
        uint8 rtReg = 9;
        uint8 rdReg = 8;

        MIPS64Instructions.CoreStepLogicParams memory params;

        params.insn = encodespec(0, rtReg, rdReg, uint16(shiftAmt) << 6);
        params.registers[rtReg] = 0x80000000;
        params.opcode = 0; // SPECIAL
        params.fun = 0; // SLL

        uint32 shiftedVal = uint32(params.registers[rtReg] << shiftAmt);
        uint64 expected = MIPS64Instructions.signExtend(uint64(shiftedVal)
           , 32);

        MIPS64Instructions.execMipsCoreStepLogic(params);
```

```
        assertNotEq(expected, params.registers[rdReg]);
        assertEq(expected, 0);
        assertEq(params.registers[rdReg], uint64(U32_MASK) << 32);
    }

    function testSLLV() external {
        uint8 shiftAmt = 1;
        uint8 rsReg = 10;
        uint8 rtReg = 9;
        uint8 rdReg = 8;
        uint16 func = 0x04; // SLLV

        MIPS64Instructions.CoreStepLogicParams memory params;

        params.insn = encodespec(rsReg, rtReg, rdReg, func);
        params.registers[rtReg] = 0x80000000;
        params.registers[rsReg] = shiftAmt;
        params.opcode = 0; // SPECIAL
        params.fun = func; // SLLV

        uint32 shiftedVal = uint32(params.registers[rtReg] << shiftAmt);
        uint64 expected = MIPS64Instructions.signExtend(uint64(shiftedVal)
            , 32);

        MIPS64Instructions.execMipsCoreStepLogic(params);

        assertNotEq(expected, params.registers[rdReg]);
        assertEq(expected, 0);
        assertEq(params.registers[rdReg], uint64(U32_MASK) << 32);
    }

    function encodespec(uint8 rs, uint8 rt, uint8 rd, uint16 funct)
       internal pure returns (uint32 insn_) {
        insn_ = uint32(rs) << 21 | uint32(rt) << 16 | uint32(rd) << 11 |
           uint32(funct);
    }
}
```

Status: Fixed

Informational

I-01: The following code comments are incorrect and should be updated to avoid confusion:

- The cpu and opcode comments in the CoreStepLogicParams NatSpec are mixed up.
- The comment for syscall BRK indicates the 32-bit program break location instead of the 64-bit location.

I-02: The return value of preemptThread is never used.

I-03: The byte size and contents of the ThreadState struct in the specification are incorrect on account of the recent changes.

I-04: The SYS_CLONE flags that are not included in the VALID_SYS_CLONE_FLAGS constant are unused and not expected to be valid flags. Therefore, they should be removed.

I-05: When calculating the MEM_PROOF_OFFSET and expectedcalldatasize for the thread hash, a magic number is used. This magic number corresponds to the aggregated hash value of the active thread stack, exclusive of the witness hash of the current thread. Since this is an important value, it should be a named constant.

I-06: The non-atomic load / store operations (LWL, SWL, LWU etc.) are inconsistent with their use of multiplication and shifting for calculating byte values. They are also inconsistent with using helper functions selectSubWord and updateSubWord. For example, LWU could use selectSubWord without sign-extension rather than manually parsing the value. Consider making these implementations consistent with shifting / multiplying and using available helper functions when applicable.