Fault Proof VM: Cannon

Cannon is Optimism's default Fault Proof Virtual Machine (FPVM). Cannon has two main components:

- OnchainMIPS.sol
- : EVM implementation to verify execution of a single MIPS instruction.
- Offchainmipsevm
- : Go implementation to produce a proof for any MIPS instruction to verify onchain.

Between these two modes of operation, thewitness data is essential, to reproduce the same instruction onchain as offchain.

OnchainMIPS.sol

This is an onchain implementation of big-endian 32-bit MIPS instruction execution. This covers MIPS III, R3000, as required by themips Go compiler/runtime target.

The syscall instruction is implemented to simulate a minimal subset of the Linux kernel, just enough to serve the needs of a basic Go program: allocate memory, read/write to certain file-descriptors, and exit.

Note that this does not include concurrency related system calls: when running Go programs, the GC has to be disabled, since it runs concurrently. This is done by patching out specific runtime functions that start the GC, by simply inserting jumps to the return-address, before the functions spin up any additional threads.

Offchainmipsevm

This is an instrumented emulator, also running big-endian 32-bit MIPS, that executes one step at a time, and maintains the same state as used for onchain execution. The difference is that it has access to the full memory, and pre-image oracle. And as it executes each step, it can optionally produce the witness data for the step, to repeat it onchain.

The Cannon CLI is used to load a program into an initial state, transition it to N steps quickly without witness generation, and 1 step while producing a witness.

mipsevm is instrumented for proof generation and handles delay-slots by isolating each individual instruction and trackingnextPC to emulate the delayedPC changes after delay-slot execution.

Witness Data

There are 3 types of witness data involved in onchain execution:

- Packed State
- Memory proofs
- Pre-image data

Packed State

The Packed State is provided in every executed onchain instruction. SeeCannon VM Specs(opens in a new tab) for details on the state structure.

The packed state is small! TheState data can be packed in such a small amount of EVM words, that it is more efficient to fully provide it, than to create a proof for each individual part involved in execution.

The program stops changing the state whenexited is set to true. The exit-code is remembered, to determine if the program is successful, or panicked/exited in some unexpected way. This outcome can be used to determine truthiness of claims that are verified as part of the program execution.

Memory Proofs

Simplicity is key here. Previous versions of Cannon used to overlap memory and register state, and represent it with a Merkle Patricia Trie. This added a lot of complexity, as there would be multiple dynamically-sized MPT proofs, with read and write operations, during a single instruction.

Instead, memory in Cannon is now represented as a binary merkle tree: The tree has a fixed-depth of 27 levels, with leaf values of 32 bytes each. This spans the full 32-bit address space:2**27 * 32 = 2**32 . Each leaf contains the memory at that part of the tree.

The tree is efficiently allocated, since the root of fully zeroed sub-trees can be computed without actually creating the full-subtree:zero_hash[d] = hash(zero_hash[d-1], zero_hash[d-1]) , until the base-case ofzero_hash[0] == bytes32(0) .

Nodes in this memory tree are combined as:out = keccak256(left ++ right) , where++ is concatenation, andleft andright are the 32-byte outputs of the respective sub-trees or the leaf values themselves. Individual leaf nodes are not hashed.

The proof format is a concatenated byte array of 28bytes32 . The firstbytes32 is the leaf value, and the remaining 27bytes32 are the sibling nodes up till the root of the tree, along the branch of the leaf value, starting from the bottom.

To verify the proof, start with the leaf value asnode , and combine it with respective sibling values:node = keccak256(node ++ sibling) ornode = keccak256(sibling ++ node) , depending the position ofnode at that level of the tree.

During the onchain execution, each instruction only executes 1 or 2 memory reads, followed by 0 or 1 writes, where the write is over the same memory as was last read.

The memory access is specifically:

- instruction (4 byte) read atPC
- load or syscall mem read, always aligned 4 bytes, read at anyaddr
- store or syscall mem write, always aligned 4 bytes, at the sameaddr

Writing only once, at the last read leaf, also means that the leaf can be safely updated and the same proof-data that was used to verify the read, can be used to reconstruct the newmemRoot of the memory tree, since all sibling data that is combined with the new leaf value was already authenticated during the read part.

Pre-Image Data

Pre-image data is accessed through syscalls exclusively. The OP-stack fault-proofPre-image Oracle specs(opens in a new tab) define the ABI for communicating pre-images.

This ABI is implemented by the VM by intercepting theread /write syscalls to specific file descriptors. SeeCannon VM Specs(opens in a new tab) for more details.

The data is loaded intoPreimageOracle.sol using the respective loading function based on the pre-image type. And then retrieved during execution of theread syscall.

Note that although the oracle provides up to 32 bytes of the pre-image, Cannon only supports reading at most 4 bytes at a time, to unify the memory operations with regular load/stores.

Usage in Dispute Game

The onchain MIPS execution functions as base-case of an interactive dispute game. The dispute game narrows down a sequence of disputed instructions to a single disputed instruction. This instruction can then be executed onchain to determine objective truth.

The Dispute Game itself is out of scope for Cannon: Cannon is just one example of a fault-proof VM that can be used to resolve disputes. SeeMIPS reference for an alternative FPVM that can be used.

Overview MIPS.sol