Fault proof VM: Cannon

Cannon is an instance of a Fault Proof Virtual Machine (FPVM) that can be used as part of the Dispute Game for any OP Stack Blockchain. The Dispute Game itself is modular, allowing for any FPVM to be used in a dispute. However, 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.

Note that Cannon is just one example of a FPVM that can be used to resolve disputes.

This documentation will go into detail about the subcomponents that make up the offchain Cannon implementation as a whole. Additionally, we will explore the differences between Cannon and the onchainMIPS.sol . For more information about the onchain implementation, please refer toMIPS reference . Now for simplicity, when referring to Cannon in this documentation, we are referring to the offchain implementation.

Control flow

In the above diagram, recreated from theOP Fault Proof System video(opens in a new tab)by Clabby, we can see that Cannon interacts with OP-Challenger and OP-Program. However, this diagram is a simplification of the relationship between OP-Challenger <> Cannon, and OP-Program <> Cannon. In general, Cannon will not be run until an active fault dispute reaches the execution trace portion of the bisection game. This does not occur until the participants in the active fault dispute game reach a single L2 block state transition that they disagree on.

OP-Challenger <> Cannon

Once an active fault dispute game reaches a depth below attacking / defending L2 block state transitions,OP-Challenger will run Cannon to begin processing MIPS instructions within the FPVM. As part of processing MIPS instructions, Cannon will generate state witness hashes, which are the commitment to the results of the MIPS instructions' computation within the FPVM. Now, in the bisection game, OP-Challenger will provide the generated hashes until a single MIPS instruction is identified as the root disagreement between participants in the active dispute. Cannon will then generate the witness proof, which contains all the information required to run the MIPS instruction onchain. Running this single MIPS instruction onchain inMIPS.sol will be used to definitively prove the correct post state, which will then be used to resolve the fault dispute game.

OP-Program <> Cannon

Once the execution trace bisection begins and Cannon is run, an Executable and Linkable Format (ELF) binary will be loaded and run within Cannon. Within Cannon is themipsevm that is built to handle the MIPS R3000, 32-bit Instruction Set Architecture (ISA). The ELF file contains MIPS instructions, where the code that has been compiled into MIPS instructions is OP-Program.

OP-Program is golang code that will be compiled into MIPS instructions and run within the Cannon FPVM. OP-Program, whether run as standalone golang code or in Cannon, fetches all necessary data used for deriving the state of the L2. It is built such that the same inputs will produce not only the same outputs, but the same execution trace. This allows all participants in a fault dispute game to run OP-Program such that, given the same L2 output root state transition, can generate the same execution traces. This in turn generates the same witness proof for the exact same MIPS instruction that will be run onchain.

Overview of offchain Cannon components

Now, we will go over each major component that makes up Cannon. Components are grouped by what functionality is being performed for Cannon, and may be correlated to one or more Go files. For brevity, each Go file will be explained at a high level, with the most important features / considerations highlighted.

mipsevm

state and memory

As mentioned previously, themipsevm is 32-bit, which means the full addressable address range is[0, 2^32-1] . The memory layout uses the typical monolithic memory structure, and the VM operates as though it were interacting directly with physical memory.

For themipsevm , how memory is stored isn't important, as it can hold the entire monolithic memory within the Go runtime. In this way, how memory is represented is abstracted away from the VM itself. However, it is important for memory to be represented such that only small portions are needed in order to run a MIPS instruction onchain. This is because it is

infeasible to represent the entire 32-bit memory space onchain due to cost. Therefore, memory is stored in a binary Merkle tree data structure, with the implementation spread acrossmemory.go (opens in a new tab) andpage.go (opens in a new tab) . 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 for that part of the tree.

memory.go defines the data structure,Memory , which holds pointers tonodes andpages . A memorynode holds the calculated Merkle root of its parent node within the memory binary Merkle tree, where the 'location' of the node is determined by its generalized index. The index calculated for a Merkle root in the nodes mapping follows thegeneralized Merkle tree index specification(opens in a new tab) .

page.go , as the name implies, defines memory pages. EachPage is 4096 bytes, which is also specified as the minimum page allocation size for the Go runtime. APage represents the lowest depths of the memory binary Merkle tree, andpage.go performs a similar role tomemory.go , calculating Merkle roots for each level of the tree.

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 subtrees or the leaf values themselves. Individual leaf nodes are not hashed.

In bothmemory.go andpage.go , there are a few optimizations designed to reduce the computationally expensive Merkle root calculations. One such optimization is that the Merkle root of zeroed-out regions of memory are calculated for each depth. This means the tree is efficiently allocated, since the root of fully zeroed subtrees 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) . So, pre-calculating zeroed Merkle roots initially allows unused memory regions to be cached. Additionally, non-zero Merkle roots are cached in bothmemory.go andpage.go , and used so long as the memory region the Merkle root covers has not been written to. Otherwise, the cache is invalidated, which requires the Merkle root to be calculated over its entire subtree. Another optimization specifically inmemory.go is caching the last two pages that have been used. Two pages are cached because themipsevm uses up to two memory addresses per instruction: one address is thePC , which contains the instruction to run, and the other address may be the target of a load or store instruction.

Another important implementation detail is that the endianness of themipsevm , which is the ordering of bytes in a word, is Big-Endian. The assumption of Cannon is that it is operating on a Little-Endian machine, and as such all reads / writes have the endianness swapped so that the internalmipsevm always handles Big-Endian words and the machine running Cannon always handles Little-Endian words. Endianness is also an important factor for the onchainMIPS.sol , where the EVM itself is Big-Endian. Therefore,MIPS.sol does not have to do any endianness swapping and can assume all data uses Big-Endian ordering. This reduces complexity within the smart contract itself.

The last major component is located instate.go (opens in a new tab) . TheState struct instate.go holds all the execution state that is required for themipsevm . The information stored is largely identical to theVM execution state forMIPS.sol . The key differences are:

- Instead of storing just the memory Merkle root, there is aMemory
- Struct pointer for the binary Merkle tree representation of the entire 32-bit memory space.
- There is an optionalLastHint
- bytes variable, which can be used to communicate a Pre-image hint to avoid having to load in multiple prior Pre-images.

Generating the witness proof

Cannon handles two major components in the dispute game: generating state witness hashes for OP-Challenger to post during the execution trace bisection game, and generating the witness proof once a single MIPS instruction is reached as the root of disagreement in the fault dispute game. The witness proof, as mentioned previously, contains all the necessary information forMIPS.sol to be able to run the same instruction onchain, and derive the post state that will be used to resolve the fault dispute game. The post state of the instruction run byMIPS.sol should be exactly the same as the post state generated by themipsevm .

The top-levelwitness.go (opens in a new tab) incannon/cmd initiates the witness proof generation. The internalwitness.go (opens in a new tab) incannon/mipsevm defines the struct that holds all the relevant information for the particular MIPS instruction. The information that is encoded for theMIPS.sol calldata can be seen in theMIPS.sol documentation .

Additionally, if a Pre-image is required for the MIPS instruction,witness.go will communicate the relevant Pre-image key and offset to OP-Challenger so that it can be posted onchain toPreimageOracle.sol .

An important note about generating the witness proof: it is imperative that all relevant information about the instruction to be run onchain is generatedbefore themipsevm execution state changes as a result of processing the MIPS instruction. Otherwise, if the witness proof is generated after running the instruction offchain, the state that will be encoded will be the post state.

Loading the ELF file

Once the execution trace portion of the bisection game begins, the ELF file containing OP-Program compiled into MIPS instructions will be run within Cannon. However, getting OP-Program into Cannon so that it can be run requires a binary

loader. The binary loader is composed ofload_elf.go (opens in a new tab) andpatch.go (opens in a new tab) .load_elf.go parses the top-level arguments and reads, loads, and patches the ELF binary such that it can be run by Cannon.

patch.go is responsible for actually parsing the headers of the ELF file, which among other information specifies what programs exist within the file and where they are located in memory. The loader uses this information to instantiate the execution state for themipsevm and load each program into memory at the expected location. Additionally, as part of instantiating the execution state,patch.go sets the values ofPC ,NextPC , the initial Heap location of0x20000000 , the initial stack location of0x7fffd000 , and also sets the arguments required for the Go runtime above the stack pointer location.

While loading the ELF file into Cannon,metadata.go (opens in a new tab) is used to parse all the symbols stored in the ELF file. Understanding which ELF symbols exist and at which regions of memory they are located is important for other functionality, such as understanding if the currentPC is running within a specific function.

Another step that occurs while loading the ELF file is patching the binary of any incompatible functions. This step is crucial, as a key design decision important**, as a key design decision in both the onchain and offchainmipsevm implementations is that neither implementation has access to a kernel. This is primarily due to the limitations within the EVM itself, and since the offchain Cannon implementation must match functionality exactly with its onchain counterpart, kernel access is also not available within Cannon. This means that the VMs cannot replicate behavior that would otherwise be performed by a kernel 1:1, which primarily impacts system calls (syscalls). So features like concurrency, memory management, I/O, etc. are either partially implemented or unimplemented. For the unimplemented functionality, any function within the ELF file that would require this functionality is patched out. Patching the binary of these functions involves identifying problematic functions, searching for their corresponding symbols within the ELF file, and effectively stubbing out the function by returning immediately, and adding anop to the delay slot.

Instruction stepping

Once the MIPS binary is loaded into Cannon, we can then begin to run MIPS instructions one at a time.run.go (opens in a new tab) contains the top-level code responsible for stepping through MIPS instructions. Additionally, before each MIPS instruction,run.go will determine whether separate action(s) needs to be performed. The actions to be performed are configured by the user, and can include logging information, stopping at a certain instruction, taking a snapshot at a certain instruction, or signaling to the VM to generate the witness proof. Actions to be performed are instantiated and checked bymatcher.go (opens in a new tab) , which generates a match function that triggers when the configured regular expression is true.

Withinrun.go , theStepFn is the wrapper that initiates the MIPS instruction to be runinstrumented.go (opens in a new tab) implementsStep as the interface to be initiated for each MIPS instruction. Additionally,instrumented.go handles encoding information for the witness proof and Pre-image information (if required for the MIPS instruction).

mips.go

mips.go (opens in a new tab) implements all the required MIPS instructions, and also tracks additional memory access for the instruction currently being run. This is important to make sure that the second memory proof is encoded correctly for instructions that use it, such as loads, stores, and certain syscalls. The full list of instructions supported can be foundhere .

mips.go

vs.MIPS.sol

The offchainmips.go and the onchain CannonMIPS.sol behave similarly when it comes to executing 32-bit, MIPS III instructions. In fact, they must produceexactly the same results given the same instruction, memory, and register state. Consequently, thewitness data is essential to reproduce the same instruction onchain and offchain. However, there are differences between the two:

1. A single instruction will be run onchain inMIPS.sol
2. , whereas the offchainmips.go
3. will run all MIPS instructions for all state transitions in the disputed L2 state.
4. Themipsevm
5. contains the entire 32-bit monolithic memory space, is responsible for maintaining the memory state based on the results of MIPS instructions, and generates the memory binary Merkle tree, Merkle root, and memory Merkle proofs.MIPS.sol
6. is mostly stateless, and does not maintain the full memory space. Instead, it only requires the memory Merkle root, and up to two memory Merkle proofs: 1 for the instruction and 1 for potential load, store, or certain syscall instructions.
7. UnlikeMIPS.sol
8. ,mips.go
9. is responsible for writing Pre-images to the PreimageOracle Server, and optionally writing hints to the Server.

PreimageOracle interaction

As mentioned previously, Cannon is responsible for setting up all state that may be required to run an instruction inMIPS.sol

. Cannon is also responsible for interacting with the PreimageOracle Server, and directing OP-Challenger to provide Pre-images to the onchainPreimageOracle.sol if necessary for the instruction that will be run inMIPS.sol .

The PreimageOracle Server connection is instantiated inrun.go , where the server itself is run locally with its own local Key-Value store.mips.go communicates with the PreimageOracle Server when reading and writing Pre-images as part of MIPS syscall instructions, as well as hinting to the PreimageOracle Server.

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.

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 32-bit load/stores.

Further Reading

- Cannon FPVM Specification(opens in a new tab)
- Merkle Proofs Specification(opens in a new tab)
- Executable and Linkable Format (ELF)(opens in a new tab)
- Keys in Mordor Summit: Cannon & Fault Proofs(opens in a new tab)
- MIPS IV ISA Specification(opens in a new tab)

FP system components OP-Challenger