Challenges: Interactive Fraud Proofs

Suppose the rollup chain looks like this:

RBlocks 93 and 95 are siblings (they both have 92 as predecessor). Alice is bonded on 93 and Bob is bonded on 95.

At this point we know that Alice and Bob disagree about the correctness of RBlock 93, with Alice committed to 93 being correct and Bob committed to 93 being incorrect. (Bob is bonded on 95, and 95 implicitly claims that 92 is the last correct RBlock before it, which implies that 93 must be incorrect.)

Whenever two bonders are bonded on sibling RBlocks, and neither of those bonders is already in a Challenge , anyone can start a challenge between the two. The rollup protocol will record the challenge and referee it, eventually declaring a winner and confiscating the loser's bond. The loser will be removed as a bonder.

The challenge is a game in which Alice and Bob alternate moves, with an Ethereum contract as the referee. Alice, the defender, moves first.

The game will operate in two phases: Dissection , followed by one-step proof. Dissection will narrow down the size of the dispute until it is a dispute about just one instruction of execution. Then the one-step proof will determine who is right about that one instruction.

We'll describe the dissection part of the protocol twice. First, we'll give a simplified version which is easier to understand but less efficient. Then we'll describe how the real version differs from the simplified one.

Dissection Protocol: Simplified Version

Alice is defending the claim that starting with the state in the predecessor RBlock, the state of the Virtual Machine can advance to the state specified in RBlock A. Essentially she is claiming that the Virtual Machine can execute N instructions, and that that execution will consume M inbox messages and transform the hash of outputs from H' to H.

Alice's first move requires her to dissect her claims about intermediate states between the beginning (0 instructions executed) and the end (N instructions executed). So we require Alice to divide her claim in half, and post the state at the half-way point, after N/2 instructions have been executed.

Now Alice has effectively bisected her N-step Assertion into two (N/2)-step assertions. Bob has to point to one of those two half-size assertions and claim it is wrong.

At this point we're effectively back in the original situation: Alice having made an assertion that Bob disagrees with. But we have cut the size of the assertion in half, from N to N/2. We can apply the same method again, with Alice bisecting and Bob choosing one of the halves, to reduce the size to N/4. And we can continue bisecting, so that after a logarithmic number of rounds Alice and Bob will be disagreeing about a single step of execution. That's where the dissection phase of the protocol ends, and Alice must make a one-step proof which will be checked by the EthBridge.

Why Dissection Correctly Identifies a Cheater

Before talking about the complexities of the real Challenge protocol , let's stop to understand why the simplified version of the protocol is correct. Here correctness means two things: (1) if Alice's initial claim is correct, Alice can always win the challenge, and (2) if Alice's initial claim is incorrect, Bob can always win the challenge.

To prove (1), observe that if Alice's initial claim is correct, she can offer a truthful midpoint claim, and both of the implied half-size claims will be correct. So whichever half Bob objects to, Alice will again be in the position of defending a correct claim. At each stage of the protocol, Alice will be defending a correct claim. At the end, Alice will have a correct one-step claim to prove, so that claim will be provable and Alice can win the challenge.

To prove (2), observe that if Alice's initial claim is incorrect, this can only be because her claimed endpoint after N steps is incorrect. Now when Alice offers her midpoint state claim, that midpoint claim is either correct or incorrect. If it's incorrect, then Bob can challenge Alice's first-half claim, which will be incorrect. If Alice's midpoint state claim is correct, then her second-half claim must be incorrect, so Bob can challenge that. So whatever Alice does, Bob will be able to challenge an incorrect half-size claim. At each stage of the protocol, Bob can identify an incorrect claim to challenge. At the end, Alice will have an incorrect one-step claim to prove, which she will be unable to do, so Bob can win the challenge.

(If you're a stickler for mathematical precision, it should be clear how these arguments can be turned into proofs by induction on N.)

The Real Dissection Protocol

The real dissection protocol is conceptually similar to the simplified one described above, but with several changes that improve efficiency or deal with necessary corner cases. Here is a list of the differences.

Dissection over L2 blocks, then over instructions: Alice's assertion is over an RBlock, which asserts the result of creating some number of Layer 2 Nitro blocks. Dissection first occurs over these Layer 2 blocks, to narrow the dispute down to a dispute about a single Layer 2 Nitro block. At this point, the dispute transforms into a dispute about a single execution of theState Transition Function or in other words about the execution of a sequence of WAVM instructions. The protocol then executes the recursive dissection sub-protocol again, this time over WAVM instructions, to narrow the dispute to a single instruction. The dispute concludes with a one-step proof of a single instruction (or a party failing to act and losing by timeout).

K-way dissection: Rather than dividing a claim into two segments of size N/2, we divide it into K segments of size N/K. This requires posting K-1 intermediate claims, at points evenly spaced through the claimed execution. This reduces the number of rounds by a factor of log(K)/log(2).

Answer a dissection with a dissection: Rather than having each round of the protocol require two moves, where Alice dissects and Bob chooses a segment to challenge, we instead require Bob, in challenging a segment, to post his own claimed endpoint state for that segment (which must differ from Alice's) as well as his own dissection of his version of the segment. Alice will then respond by identifying a subsegment, posting an alternative endpoint for that segment, and dissecting it. This reduces the number of moves in the game by an additional factor of 2, because the size is cut by a factor of K for every move, rather than for every two moves.

Deal With the Empty-Inbox Case : The real AVM can't always execute N units of gas without getting stuck. The machine might halt, or it might have to wait because its inbox is exhausted so it can't go on until more messages arrive. So Bob must be allowed to respond to Alice's claim of N units of execution by claiming that N steps are not possible. The real protocol thus allows any response (but not the initial claim) to claim a special end state that means essentially that the specified amount of execution is not possible under the current conditions.

Time Limits: Each player is given a time allowance. The total time a player uses for all of their moves must be less than the time allowance, or they lose the game. Think of the time allowance as being about a week.

It should be clear that these changes don't affect the basic correctness of the challenge protocol. They do, however, improve its efficiency and enable it to handle all of the cases that can come up in practice.

Efficiency

The challenge protocol is designed so that the dispute can be resolved with a minimum of work required by the protocol (via its Layer 1 Ethereum contracts) in its role as referee. When it is Alice's move, the protocol only needs to keep track of the time Alice uses, and ensure that her move does include K-1 intermediate points as required. The protocol doesn't need to pay attention to whether those claims are correct in any way; it only needs to know whether Alice's move "has the right shape".

The only point where the protocol needs to evaluate a move "on the merits" is at the one-step proof, where it needs to look at Alice's proof and determine whether the proof that was provided does indeed establish that the virtual machine moves from the before state to the claimed after state after one step of computation.

ChallengeManager

This section is a technical deep dive into theChallengeManager and will walk through the arbitration of a challenge game in great detail. TheChallengeManager plays the role of the arbiter of challenge games. Here's a diagram of the challenge state machine:

Block challenge

The challenge begins by bisecting over global states (including block hashes). Before actual machine execution is disputed, the dispute is narrowed down to an individual block. Once the challenge has been bisected down to an individual block,challengeExecution can be called by the current responder. This operates similarly to a bisection in that the responder must provide a competing global state and machine state, but it uses that information to transition to the execution challenge phase.

Execution challenge

Once narrowed down to an individual block, the actual machine execution can be bisected. Once the execution has been bisected down to an individual step,oneStepProveExecution can be called by the current responder. The current responder must provide proof data to execute a step of the machine. If executing that step ends in a different state than was previously asserted, the current responder wins the challenge.

General bisection protocol

Note: the term bisection in this document is used for clarity but refers to a dissection of any degree.

TheChallengeLib helper library contains ahashChallengeState method which hashes a list of segment hashes, a start

position, and a total segments length, which generates theChallengeLib.Challenge 'schallengeStateHash . This is enough information to infer the position of each segment hash. The challenge "degree" refers to the number of segment hashes minus one. The distance (in steps) between one segment and the next isfloor(segmentsLength / degree) , except for the last pair of segments, wheresegmentsLength % degree is added to the normal distance, so that the total distance issegmentsLength .

A challenge begins with only two segments (a degree of one), which is the asserter's initial assertion. Then, the bisection game begins on the challenger's turn. In each round of the game, the current responder must choose an adjacent pair of segments to challenge. By doing so, they are disputing their opponent's claim that starting with the first segment and executing for the specified distance (number of steps) will result in the second segment. At this point the two parties agree on the correctness of the first segment but disagree about the correctness of the second segment. The responder must provide a bisection with a start segment equal to the first segment, but an end segment different from the second segment. In doing so, they break the challenge down into smaller distances, and it becomes their opponent's turn. Each bisection must have degreemin(40, numStepsInChallengedSegment) , ensuring the challenge makes progress.

In addition, a segment with a length of only one step cannot be bisected. What happens there is specific to the phase of the challenge, as either achallengeExecution oroneStepProveExecution .

Note that unlike in a traditional bisection protocol, where one party proposes segments and the other decides which to challenge, this protocol is symmetric in that both players take turns deciding where to challenge and proposing bisections when challenging.

Winning the challenge

Note that for the time being, winning the challenge isn't instant. Instead, it simply makes the current responder the winner's opponent, and sets the state hash to 0. In that state the party does not have any valid moves, so it will eventually lose by timeout. This is done as a precaution, so that if a challenge is resolved incorrectly, there is time to diagnose and fix the error with a contract upgrade.

One Step Proof Assumptions

TheOne Step Proof (OSP) implementation makes certain assumptions about the cases that can arise in a correct execution. This documents those assumptions about what's being executed.

If a case is "unreachable", that is, the case is assumed to never arise in correct execution, then the OSP can implement any instruction semantics in that case.

- In a challenge between malicious parties, any case can arise. The challenge protocol must do
- something safe in every case. But the instruction semantics can be weird in such cases because
- if both parties to a challenge are malicious, the protocol doesn't care who wins the challenge.
- In a challenge with one honest party, the honest party will never need to one-step prove an
- unreachable case. The honest party will only assert correct executions, so it will only have to
- prove reachable cases.
- In a challenge with one honest party, the dishonest party could assert an execution that transitions
- into an unreachable case, but such an execution must include an invalid execution of a reachable case
- earlier in the assertion. Because a challenge involving an honest party will eventually require an OSP
- over the first instruction where the parties disagree, the eventual OSP will be over the earlier point
- of divergence, and not over the later execution from an unreachable case.

In general, some unreachable cases will be detectable by the OSP checker and some will not. For safety, the detectable unreachable cases should be defined by transition the machine into an error state, allowing governance to eventually push an upgrade to recover from the error. An undetectable unreachable case, if such a case were reached in correct execution, could lead to a security failure.

The following assumptions, together, must prevent an unreachable case from arising in correct execution.

The WAVM code is generated by Arbitrator from valid WASM

WAVM is the name of the custom instruction set similar toWASM used for proving. Arbitrator transpiles WASM code into WAVM. It also invokes wasm-validate fromwabt (the WebAssembly Binary Toolkit) to ensure the input WASM is valid. WAVM produced otherwise may not be executable, as it may try to close a non-existent block, mismatch types, or do any other number of invalid things which are prevented by WASM validation.

WAVM code generated from by Arbitrator from valid WASM is assumed to never encounter an unreachable case.

Inbox messages must not be too large

The current method of inbox hashing requires the full inbox message be available for proving. That message must not be too large as to prevent it from being supplied for proving, which is enforced by the inboxes.

The current length limit is 117,964 bytes, which is 90% of themax transaction size Geth will accept, leaving 13,108 bytes for other proving data.

Requested preimages must be known and not too large

WAVM has an opcode which resolves the preimage of a Keccak-256 hash. This can only be executed if the preimage is already known to all nodes, and can only be proven if the preimage isn't too long. Violations of this assumption are undetectable by the OSP checker.

The current length limit is 117,964 bytes for the reasons mentioned above. Here's a list of which preimages may be requested by Nitro, and why they're known to all parties, and not too large:

Block headers

Nitro may request up to the last 256L2 Block headers. The last block header is required to determine the current state, and blocks before it are required to implement theBLOCKHASH evm instruction.

This is safe as previous block headers are a fixed size, and are known to all nodes.

State trie access

To resolve state, Nitro traverses the state trie by resolving preimages.

This is safe as validators retain archive state of unconfirmed blocks, each trie branch is of a fixed size, and the only variable sized entry in the trie is contract code, which is limited by EIP-170 to about 24KB.

WASM to WAVM

Not all WASM instructions are 1:1 with WAVM opcodes. This document lists those which are not, and explains how they're expressed in WAVM. Many of the WAVM representations use opcodes not in WASM, which are documented inwavm-custom-opcodes .

block

andloop

In WASM, a block contains instructions. Branch instructions exit a fixed number of blocks, jumping to their destination. A normalblock 's destination is the end of the block, whereas aloop 's destination is the start of the loop.

In WAVM, instructions are flat. At transpilation time, any branch instructions are replaced with jumps to the corresponding block's destination. This means that WAVM interpreters don't need to track blocks, and thus block instructions are unnecessary.

if

andelse

These are translated to a block with anArbitraryJumpIf as follows:

begin block with endpoint end conditional jump to else [instructions inside if statement] branch else: [instructions inside else statement] end

br

andbr_if

br andbr_if are translated intoArbitraryJump andArbitraryJumpIf respectively. The jump locations can be known at transpilation time, making blocks obsolete.

br_table

br_table is translated to a check for each possible branch in the table, and then if none of the checks hit, a branch of the default level.

Each of the non-default branches has a conditional jump to a section afterwards, containing adrop for the selector, and then a jump to the target branch.

local.tee

local.tee is translated to a WAVMDup and then aLocalSet .

return

To translate a return, the number of return values must be known from the function signature. A WAVMMoveFromStackToInternal is added for each return value. Then, a loop checksIsStackBoundary (which implicitly pops a value) until it's true and the stack boundary has been popped. Next, aMoveFromInternalToStack is added for each return value to put the return values back on the stack. Finally, a WAVMReturn is added, returning control flow to the caller.

Floating point instructions

A floating point library module must be present to translate floating point instructions. They are translated by bitcastingf32 andf64 arguments toi32 s andi64 s, then a cross module call to the floating point library, and finally bitcasts of any return values fromi32 s andi64 s tof32 s andf64 s.

WAVM Custom opcodes not in WASM

In addition to the MVP WASM specification, WAVM implements the multi value and sign extension ops WASM proposals.

WAVM also implements the following unique opcodes, which are not part of WASM nor any WASM proposal.

Invariants

Many of these opcodes have implicit invariants about what's on the stack, e.g. "Pops an i32 from the stack" assumes that the top of the stack has an i32. If these conditions are not satisfied, execution is generally not possible. These invariants are maintained by WASM validation and Arbitrator codegen. (SeeOne Step Proof Assumptions .)

Codegen internal

These are generated when breaking down a WASM instruction that does many things into many WAVM instructions which each do one thing. For instance, a WASMlocal.tee is implemented in WAVM withdup and thenlocal.set , the former of which doesn't exist in WASM.

Other times, these opcodes help out an existing WASM opcode by splitting out functionality. For instance, the WAVMreturn opcode by itself does not clean up the stack, but its WASM->WAVM codegen includes a loop that utilizesIsStackBoundary to perform the stack cleanup specified for WASM'sreturn .

Opcode Name Description 0x8000 EndBlock Pops an item from the block stack. 0x8001 EndBlockIf Peeks the top value on the stack, assumed an i32. If non-zero, pops an item from the block stack. 0x8002 InitFrame Pops a caller module index i32, then a caller module internals offset i32, and finally a return InternalRef from the stack. Creates a stack frame with the popped info and the locals merkle root in proving argument data. 0x8003 ArbitraryJumpIf Pops an i32 from the stack. If non-zero, jumps to the program counter in the argument data. 0x8004 PushStackBoundary Pushes a stack boundary to the stack. 0x8005 MoveFromStackToInternal Pops an item from the stack and pushes it to the internal stack. 0x8006 MoveFromInternalToStack Pops an item from the internal stack and pushes it to the stack. 0x8007 IsStackBoundary Pops an item from the stack. If a stack boundary, pushes an i32 with value 1. Otherwise, pushes an i32 with value 0. 0x8008 Dup Peeks an item from the stack and pushes another copy of that item to the stack. The above opcodes eliminate the need for the following WASM opcodes (which are transpiled into other WAVM opcodes):

- loop
- if/else
- br_table
- local.tee

Linking

This is only generated to link modules together. Each import is replaced with a local function consisting primarily of this opcode, which handles the actual work needed to change modules.

Opcode Name Description 0x8009 CrossModuleCall Pushes the current program counter, module number, and module's internals offset to the stack. Then splits its argument data into the lower 32 bits being a function index, and the upper 32 bits being a module index, and jumps to the beginning of that function.

Host calls

These are only used in the implementation of "host calls". Each of these has an equivalent host call method, which can be invoked from libraries. The exception isCallerModuleInternalCall , which is used for the implementation of all of thewavm_caller_* host calls. Those calls are documented inwavm-modules.mdx .

For these instruction descriptions, all pointers and offsets are represented as WASM i32s.

Opcode Name Description 0x800A CallerModuleInternalCall Pushes the current program counter, module number, and module's internals offset (all i32s) to the stack. Then, it retrieves the caller module internals offset from the current stack frame. If 0, errors, otherwise, jumps to the caller module at function (internals offset + opcode argument data) and instruction 0. 0x8010 GetGlobalStateBytes32 Pops a pointer and then an index from the stack. If the index is greater than or equal to the number of global state bytes32s, errors. If the pointer mod 32 is not zero, errors. If the pointer + 32 is outside the programs memory, errors. Otherwise, writes the global state bytes32 value of the specified index to the specified pointer in memory. 0x8011 SetGlobalStateBytes32 Pops a pointer and then an index from the stack. If the index is greater than or equal to the number of global state bytes32s, errors. If the pointer mod 32 is not zero, errors. If the pointer + 32 is outside the programs memory, errors. Otherwise, reads a bytes32 from the specified pointer in memory and sets the global state bytes32 value of the specified index to it. 0x8012 GetGlobalStateU64 Pops a pointer and then an index from the stack. If the index is greater than or equal to the number of global state u64s, errors. If the pointer mod 32 is not zero, errors. If the pointer + 8 is outside the programs memory, errors. Otherwise, writes the global state u32 value of the specified index to the specified pointer in memory. 0x8013 SetGlobalStateU64 Pops a pointer and then an index from the stack. If the index is greater than or equal to the number of global state u64s, errors. If the pointer mod 32 is not zero, errors. If the pointer + 8 is outside the programs memory, errors. Otherwise, reads a u64 from the specified pointer in memory and sets the global state u64 value of the specified index to it. 0x8020 ReadPreImage Pops an offset and then a pointer from the stack. If the pointer mod 32 is not zero, errors. If the pointer + 32 is outside the programs memory, errors. Reads a 32 byte Keccak-256 hash from the specified pointer in memory. Writes up to 32 bytes of the preimage to that hash, beginning with theoffset byte of the preimage. Ifoffset is greater than or equal to the number of bytes in the preimage, writes nothing. Pushes the number of bytes written to the stack as an i32. 0x8021 ReadInboxMessage Pops an offset, then a pointer, and then an i64 message number from the stack. If the pointer mod 32 is not zero, errors. If the pointer + 32 is outside the programs memory, errors. Attempts to read an inbox message from the inbox identifier contained in the argument data (0 for theSequencer inbox, 1 for theDelayed Inbox) at the specified message number. If this exceeds the machine's inbox limit, enters the "too far" state. Otherwise, writes up to 32 bytes of the specified inbox message, beginning with theoffset byte of the message. Ifoffset is greater than or equal to the number of bytes in the preimage, writes nothing. Pushes the number of bytes written to the stack as an i32. 0x8022 HaltAndSetFinished Sets the machine status to finished, halting execution and marking it as a success.

WAVM Floating point implementation

Implementing correct, consistent, and deterministic floating point operations directly in WAVM (meaning both a Rust Arbitrator implementation and Solidity OSP implementation) would be an extremely tricky endeavor. WASM specifies floating point operations as being compliant to IEEE 754-2019, which is not deterministic, and full of edge cases.

Instead, floating point operations (apart from trivial bit-casts like i32 <-> f32) are implemented using the C Berkeley SoftFloat-3e library running inside WAVM. Arbitrator links other WAVM guests against this, by replacing float point operations with cross module calls to the library.

Berkeley SoftFloat does not implement all necessary floating point operations, however. Most importantly, it does not provide a min function, despite IEEE 754-2019 specifying one. The implementation of these operations, along with the export of convenient APIs for WASM opcode implementations, are contained in bindings32.c for 32 bit integers and bindings64.c for 64 bit integers.

This ensures that floating point operations are deterministic and consistent between Arbitrator and the OSP, as they are implemented exclusively using operations already known to be deterministic and consistent. However, it does not ensure that the floating point operations are perfectly compliant to the WASM specification. Go uses floating points in its JS<->Go WASM interface, and floating points may be used outside core state transition code for imprecise computations, but the former is well exercised as used in Nitro, and the latter generally doesn't rely on details like the minimum of NaN and infinity.

Known divergences from the WASM specification

Floating point to integer truncation will saturate on overflow, instead of erroring. This is generally safer, because on x86, overflowing simply produces an undefined result. A WASM proposal exists to add new opcodes which are defined to saturate, but it's not widely adopted.

WAVM Modules

WASM natively has a notion of modules. Normally, in WASM, a module is the entire program. A.wasm file represents one module, and generally they aren't combined. An exception to this is C compiled via Clang, where wasm files are also used as object files, butits linking scheme is not supported in other languages.

In WAVM this is extended to make the executing program composed of multiple modules. These may call each other, and library modules may write to their caller's memory to return results.

The entrypoint module

The entrypoint module is where execution begins. It calls modules'start functions if specified, and then calls the main module's main function, which is language specific. For Go it sets argv to["js"] to match the JS environment, and callsrun . For Rust it callsmain with no arguments.

Library exports

Libraries may export functions with the name patternmodule__name , which future libraries or the main module can import as"module" "name" .

For instance, this is used for wasi-stub to provide functions rust imports according to the WebAssembly System Interface.

Floating point operations

To provide floating point operations for future libraries, the soft float library exports functions which perform floating point ops. These have the same name as the WASM instruction names, except. is replaced with_ . Their type signature is also the same, except allf32 s andf64 s are bitcasted toi32 s andi64 s.

Future modules can implicitly use these by using WASM floating point operations, which are replaced at the WASM->WAVM level with bitcasts and cross module calls to these functions.

WAVM guest calls

Libraries may call the main module's exports via"env" "wavm_guest_call__*" .

For instance, go-stub calls Go's resume function when queueing async events viawavm_guest_call_resume() , and then retrieves the new stack pointer withwavm_guest_call_getsp() .

Caller module internals call

Every stack frame retains its caller module and its caller module's "internals offset", which is the first internal function index. WAVM appends 4 "internal" functions to each module, which perform a memory load or store of 1 or 4 bytes.

Viawavm_caller_{load,store}{8,32} , a library may access its caller's memory, which is implemented by calling these internal functions of the caller's module. Only libraries can access their caller's memory; the main module cannot.

For instance, this is used to read arguments from and write return values to the Go stack, when Go calls into go-stubEdit this page Last updatedonJan 27, 2025 Previous Optimistic Rollup Next AnyTrust protocol