# Fault Proof Overview

The Fault Proof System is comprised of three main components: a Fault Proof Program (FPP), a Fault Proof Virtual Machine (FPVM), and a dispute game protocol. The system is designed to eventually enable secure bridging without central fallback. The modular design of the fault proof system lays the foundation for a multi-proof future, inclusive of ZK proofs, and significantly increases the opportunities for ecosystem contributors to build alternative fault proof components to secure the system.

Visit theImmunefi bug bounty page(opens in a new tab)for details on testing and helping to build a robust fault proof system.

## System Design & Modularity

The Fault Proof System is comprised of three main components: a Fault Proof Program (FPP), a Fault Proof Virtual Machine (FPVM), and a dispute game protocol. These components will work together to challenge malicious or faulty activity on the network to preserve trust and consistency within the system. See the video below for a full technical walkthrough of the OP Stack's first fault proof system.

The OP Stack's unique, modular design allows the decoupling of the FPP and FPVM, resulting in the following:

- development of multiple proof systems, unique dispute games, and a variety of FPVMs in the future.
- custom-built fault proof systems comprised of any combination of these isolated components—including validity proofs, attestation proofs, or ZKVM.
- dispute games in the dispute protocol backed by multiple security mechanisms.

## Fault Proof Program

The default for this system component isop-program , which implements a fault proof program that runs through the rollup state-transition to verify an L2 output from L1 inputs. This verifiable output can then resolve a disputed output on L1. The FPP is a combination ofop-node andop-geth , so it has both the consensus and execution "parts" of the protocol in a single process. This means Engine API calls that would normally be made over HTTP are instead made as direct method calls to the op-geth code.

The FPP is designed so that it can be run in a deterministic way such that two invocations with the same input data will result in not only the same output, but the same program execution trace. This allows it to be run in an on-chain VM as part of the dispute resolution process.

All data is retrieved via the Preimage Oracle API. The preimages could be provided via the FPVM when on-chain or by a native "host" implementation that can download the required data from nodes via JSON-RPC requests. The native host implementation is also provided byop-program but doesn't run as part of the on-chain execution. Basically op-program has two halves: the "client" Fault Proof Program part covered in this section and the "host" part used to fetch required preimages.

## Fault Proof Virtual Machine

The Fault Proof Virtual Machine (FPVM) is one of the modules in the OP Stack's fault proof system. OP Stack's modularity decouples the Fault Proof Program (FPP) from the Fault Proof Virtual Machine (FPVM) to enable next-level composability and efficient parallelized upgrades to both components. The FPP (client-side) that runs within the FPVM is the part that expresses the L2 state-transition, and the interface between FPVM and FPP is standardized and documented in thespecs(opens in a new tab) .

Through this separation, the VM stays ultra-minimal: Ethereum protocol changes, like EVM op-code additions, do not affect the VM. Instead, when the protocol changes, the FPP can simply be updated to import the new state-transition components from the node software. Similar to playing a new version of a game on the same game console, the L1 proof system can be updated to prove a different program.

The FPVM is tasked with lower-level instruction execution. The FPP needs to be emulated. The VM requirements are low: the program is synchronous, and all inputs are loaded through the same pre-image oracle, but all of this still has to be proven in the L1 EVM onchain. To do this, only one instruction is proven at a time. The bisection game will narrow down the task of proving a full execution trace to just a single instruction. Proving the instruction may look different for each FPVM, but generally it looks similar to Cannon, which proves the instruction as follows:

- To execute the instruction, the VM emulates something akin to an instruction-cycle of a thread-context: the instruction is read from memory, interpreted, and the register-file and memory may change a little.
- To support the pre-image oracle, and basic program runtime needs like memory-allocation, the execution also supports a subset of linux syscalls. Read/write syscalls allow interaction with the pre-image oracle: the program writes a hash as request for a pre-image, and then reads the value in small chunks at a time.

Cannon is the default FPVM used in all disputesMIPS is the on-chain smart contract implementation of Cannon that can be implemented due to the modularity of the dispute game.

## Dispute Game Protocol

In the Dispute protocol, different types of dispute games can be created, managed, and upgraded through theDisputeGameFactory(opens in a new tab) . This opens the door to innovative features, like aggregate proof systems and the ability to expand the protocol to allow for disputing things apart from the state of L2, such as aFaultDisputeGame geared towards on-chain binary verification.

A dispute game is a core primitive to the dispute protocol. It models a simple state machine, and it is initialized with a 32 byte commitment to any piece of information of which the validity can be disputed. They contain a function to resolve this commitment to be true or false, which is left for the implementor of the primitive to define. Dispute games themselves rely on two fundamental properties:

- Incentive Compatibility: The system penalizes false claims and rewards truthful ones to ensure fair participation.
- Resolution: Each game has a mechanism to definitively validate or invalidate the root claim.

The standard is the bi-section game. This is a specific type of dispute game, and the first game built in the OP Stack's dispute protocol. We bisect over output roots (which each correspond to single L2 blocks), until we get to a single blockn -> n+1 state transition. Then, we bisect over a single block state transition's execution trace as described before. This is an optimization to reduce the runtime of the off-chain VM. After bisection has reached commitments to the state at individual trace instructions, theFaultDisputeGame executes a single instruction step on chain using a generic VM. The VM's state transition function, which we'll callT , can be anything, so long as it adheres to the formT(s, i) -> s' , wheres = the agreed upon prestate,i = the state transition inputs, ands' = the post state.

The first full implementation of the VM generic in the bisection game includes a single MIPS thread context on top of the EVM to execute single instructions within an execution trace generated byCannon and theop-program .

## Next Steps

- SeeMips.sol
- to learn more about the on-chain smart contract implementation of Cannon.
- SeeCannon FPVM Specification(opens in a new tab)
- for more detail.

Withdrawal Flow FPVM: Cannon