Execution Proof Aggregator (EPA)

zkVM execution Models and Constraints

There are 2 models that can be applied to ZK program invocation:

- Dual-layer VMs (aka Nested VMs)
- Single-layer VMs

In order to achieve full ZK program composability while maintaining zkVM platform independence, *zkVM invocation must occur once per transaction*.

zkVM execution Models and Constraints

Dual-layer zkVM

- ⚠️ x5-x50 slower execution times
- ⚠️ Requires ELF to be loaded into zkVM as a private input (for inner VM interpretation)
- ⚠️ Flexible native RISC-V or other ISA interpretations, but programs themselves have no proofs (everything is a single VM proof - does not fit within the goals of ZK composability - not standalone ZK programs)
- ⚠️ Faster CPI composability, but for each CPI, each program ELF has to be loaded into ZK context as ZK input
- ⚠️ Doesn't allow composability between different zkVM platforms
- Provable termination condition and gas accounting (albeit gas accounting by itself increases cycle consumption by x5)

Single-layer zkVM

- ✓ Significantly faster (x10 slower than bare metal (native CPU) - very acceptable)
- ⚠️ ZK program is "naked" and runs directly under zkVM - requires external validation, handling of panic and metering overrun conditions.
- ✓ Single Layer zkVM allows for dual-layer nested zkVM execution as well.
- ⚠️ Most zkVM frameworks (RISC-0, SP1, JOLT) are *incapable of proving actual compute consumption* (GAS metering), however gas can be statically limited proving the termination due to the limit overflow.

Conclusion

Single-layer zkVMs are significantly faster and match design and performance goals of composable ZK programs.

Single-layer zkVMs allow for broader ecosystem compatibility (this is harder to implement but provides a better opportunity for the long-term ecosystem scalability).

Since a Single-layer zkVM is an *untrusted guest program* created by a 3rd-party, its execution must be externally validated and proven.

This requires an additional ZK program that checks and creates a validation proof of the untrusted user program execution.

This component is called an Execution Proof Aggregator (EPA)

zkVM executes USER PROGRAM, capturing it's proof (containing public inputs and outputs), then supplies that proof to the EPA PROGRAM for validation. EPA generates a proof proving that the USER PROGRAM *conforms to platform protocols and execution rules*.

EPA is a *system program* that validates data processing and protocol conformity of *user programs*. Unlike user programs (vProgs), EPA acts on behalf of the L1 and is the only program that is integrated directly within L1.

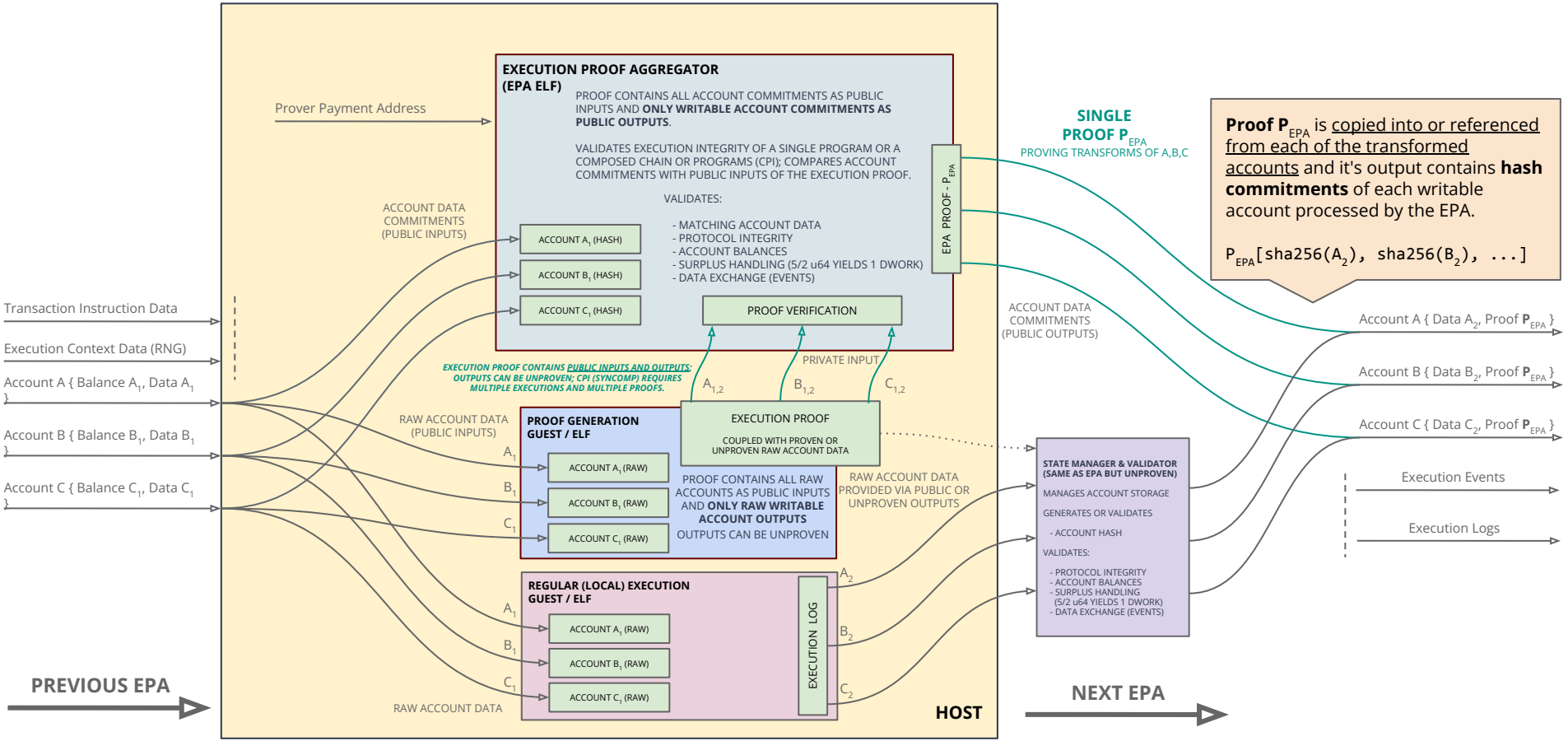
EPA is the only ZK program that requires knowledge of different ZK framework verification schemes.

Program executor subsystem that runs user programs needs to be able to run user programs for supported frameworks, but this concern is related only to L2.

L1 needs to track vProg program hashes and handle vProg execution interpretation within it's CDAG, however, in the EPA model, it only needs to verify EPA proofs. This, however, requires that the EPA program is upgradeable.

EPA

Execution Proof Aggregator



EPA provides external validation for the program execution.

It is important to note that it is the EPA that reduces inputs and outputs down to commitments.

EPA does not negate support for dual-layer VMs - Dual-layer VMs can still be used if desirable and possibly simplified since the EPA provides additional top-level validation of the execution.

NOTE: EPA does not prevent use of Dual-layer zkVMs. If desirable, EPA can be replaced with a Dual-layer zkVM that performs EPA-like inner VM logic validation. Such design would require a registry of multiple system programs, where a vProg can specify the type of execution environment it requires in it's metadata.

NOTE: vProg design assumes per-program proofs, while EPA provides a single aggregated proof for multiple programs executed during symcomp/CPI. As long as each vProg can discern each account within the proof, same proof can be reused to represent each vProg invocation.