

OPENVM WHITEPAPER

OPENVM CONTRIBUTORS

ABSTRACT. OpenVM is a performant and modular zkVM framework built for customization and extensibility. OpenVM introduces a new “no-CPU” zkVM design paradigm where the execution trace is not materialized in any centralized chip and is instead managed collectively by all instruction executor chips. This no-CPU design allows OpenVM to support all functionality through VM extensions, which are groups of instructions interoperating through common memory spaces. Developers can add custom VM support to OpenVM using custom VM extensions without forking or modifying the core OpenVM libraries, and all default functionality in OpenVM, including RISC-V and recursion support, is implemented through a set of default VM extensions.

This paper describes the core OpenVM architecture, including the arithmetization and zkVM design, modular ISA design via VM extensions, support for unbounded programs and on-chain verification via proof recursion and continuations, and Rust toolchain support targeting the 32-bit RISC-V ISA.

CONTENTS

1. Introduction	1
2. Arithmetization Framework and ZK Backend	3
3. Instruction Set Architecture	6
4. zkVM Design	10
5. Proof Architecture	14
6. RISC-V Support and Rust Toolchain	26
References	27

1. INTRODUCTION

Recent advancements in zero-knowledge (ZK) proofs have led to the introduction of new tools to enable developers to use ZK. Systems such as Circom [ide25], halo2 [EP25], Gnark [Con23], and plonky3 [Pol25] provide tools to encode computation in an intermediate representation designed specifically for ZK. This approach provides the most flexibility and theoretically best performance by allowing developers to directly access the inputs to a ZK backend. However, writing circuits directly has high developer overhead and requires new security work for each ZK circuit.

A more recent class of ZK virtual machines (zkVMs) addresses these developer experience concerns by introducing specialized ZK circuits verifying the execution of programs targeting an instruction set architecture (ISA). These tools allow developers to use standard imperative

Date: July 21, 2026.

Contact: info@openvm.dev.

programming models while enabling verifiability in ZK, but do so at the expense of developer overhead to use a ZK-native ISA (Cairo [GPR21], Valida [Val24]) or performance overhead to support a standard ISA such as RISC-V (Risc0 [RIS25], SP1 [Suc25]).

This paper introduces OpenVM, a zkVM framework which balances flexibility and performance with a modular, developer-defined ISA that enables customization of zkVMs to specific use cases. By introducing a novel “no-CPU” zkVM design that decouples opcode implementations, OpenVM enables developers to do this by building custom VM extensions without forking or modifying the core OpenVM libraries. We describe the core OpenVM architecture for a modular zkVM, how it can be used to support proving unbounded program execution via recursion and continuations, and Rust toolchain support via the RISC-V target.

1.1. Background and Motivation. zkVMs have emerged as the most ergonomic means for developers to write verifiable software. However, existing zkVMs have struggled with trade-offs between performance and developer experience depending on the computer architecture they wish to emulate. Those designed for custom ZK-native ISAs require developers to learn new programming models, while others must make a fixed choice of traditional ISA to emulate and incur performance overheads.

OpenVM addresses these limitations by introducing a framework that:

- enables familiar development workflows using standard languages like Rust,
- provides an extensible ISA designed for custom instruction sets across diverse use cases,
- facilitates seamless composition of VM extensions,
- embraces a modular architecture to accommodate future proof system advancements.

1.2. Design Principles. The architecture and features of OpenVM are guided by a set of core design principles:

- (1) **Modularity and Composability:** The framework is designed so that the ISA and zkVM circuit can be extended independently of the core system. We minimize the reliance on centralized components and ensure that new components can be added to the zkVM without requiring changes to the core system. This principle is embodied in the “no-CPU” design and the concept of VM extensions.
- (2) **Performance:** We believe that developers should not have to choose between modularity and performance. On the contrary, we believe that modularity, when incorporated correctly, can provide more opportunities for performance optimization.
- (3) **Developer Experience:** We ensure that customization of the zkVM integrates with existing developer workflows.
- (4) **Future Proof:** By maintaining modular abstraction layers, we ensure that different parts of the system can be updated independently. Notably, we choose intermediate representations that enable the underlying ZK proof system to be upgraded without completely rewriting the zkVM circuit logic.

1.3. OpenVM Architecture Overview. The OpenVM architecture is comprised of the following main components. The circuit arithmetization framework and ZK backend (§2) provides the interface to represent circuit logic. We keep the circuit logic abstracted away from the implementation of the proof system, allowing the latter to be upgraded without affecting the former. The instruction set architecture (§3) defines the virtual machine execution model and the foundations to support VM extensions. The zkVM design (§4) explains our circuit architecture which enables zkVMs to be extended to support custom instruction set extensions

without core system changes. In §5, we explain how we prove unbounded program execution using continuations and proof recursion (see §5 for definitions). The framework is designed so that custom extensions have first-class support in the Rust toolchain (§6), which is achieved by synchronizing OpenVM extensions with those in the RISC-V ISA.

Acknowledgments. OpenVM is a new zkVM design framework. In the process of building it, we studied and learned from the designs and implementations of many projects. We would like to especially thank StarkWare [Sta21] for their pioneering work on STARKs, Plonky3 [Pol25] for their modular design of polynomial IOPs, Valida [Val24] for new designs around circuit arithmetization and ZK-native ISAs, RISC Zero [RIS25] for their foundational work on Rust toolchain integration for zkVMs, and SP1 [Suc25] for their precompile-centric design and zkVM based recursion programs.

2. ARITHMETIZATION FRAMEWORK AND ZK BACKEND

We describe the arithmetization framework that we use to design ZK circuits. This arithmetization framework represents a *ZK frontend* which we maintain as an abstraction boundary between the representation of the circuit and the proof generation process. We then discuss how our *ZK backend* provides a general framework for generating proofs from circuits designed using this arithmetization. Our design allows the ZK backend to be upgraded for newer proof systems without changing the frontend.

2.1. Notation. For the rest of this paper, we fix the base field $\mathbb{F}$ used in the arithmetization and the proof system. We assume that $\mathbb{F}$ is a prime field of characteristic p where p has 31 bits. Our present implementation uses the BabyBear prime $p = 15 \cdot 2^{27} + 1 = 2^{31} - 2^{27} + 1$, but this choice is not fundamental to the design and may be changed in the future.

The proof system also requires a choice of extension field $\mathbb{F}_{\text{ext}}$ over $\mathbb{F}$. The extension field is used to boost the security of the proof system. The extension field is chosen so that the order $|\mathbb{F}_{\text{ext}}|$ is at least 2^{120} . Presently, the quartic extension field $\mathbb{F}[x]/(x^4 - 11)$ of BabyBear is used.

2.2. AIRs with interactions. Modern ZK systems use *arithmetization* to represent circuits, which may constrain arbitrary computation, in an intermediate representation that is more readily processed by the proof system. The choice of arithmetization is crucial as it determines how circuits will be designed and represented in the system.

Our arithmetization framework is an extension of the *algebraic intermediate representation* (AIR) framework, with extensions tailored for newer proving techniques such as LogUp [Hab22, PH23]. We refer to this framework as **AIRs with Interactions**.

AIRs were first introduced by [BBHR18] and have been prevalent in STARK proof systems since. The precise definition of an AIR varies slightly in the literature, so we provide a definition below to clarify our terminology.

Definition 2.2.1 (AIR). An algebraic intermediate representation (AIR) is a set of pairs (C_i, S_i) , where $C_i: \mathbb{F}[x_1, \dots, x_w, y_1, \dots, y_w]$ are *constraint polynomials* and w is a fixed width associated with the AIR. The S_i are special selectors, to be explained below, which can be one of All, First, Last, Transition.

A *trace matrix* is a matrix with entries in $\mathbb{F}$ where the number of rows is a power of two. We define the height of the matrix to be the number of rows and the width to be the number of columns. We say that a trace matrix $\mathbf{T}$ satisfies the AIR $\{(C_i, S_i)\}$ if the width of $\mathbf{T}$ equals

w and for each i the constraint polynomial $C_i(x_1, \dots, x_w, y_1, \dots, y_w)$ evaluates to zero on the following domain:

- If S_i is **All**, then this applies to all pairs of cyclically consecutive rows $(x_1, \dots, x_w), (y_1, \dots, y_w)$ of $\mathbf{T}$.
- If S_i is **First**, then this applies to the first pair of cyclically consecutive rows of $\mathbf{T}$.
- If S_i is **Last**, then this applies to the pair (last row, first row) of $\mathbf{T}$.
- If S_i is **Transition**, then this applies to all pairs of non-cyclically consecutive rows – that is, like **All** but except (last, first).

Note that in our definition, the height of the trace matrix is not specified by the AIR – trace matrices of different heights can satisfy the same AIR.

In our arithmetization, we also allow each AIR to specify a partition of the columns $\{1, \dots, w\}$ of the trace matrix into parts of different types:

- Preprocessed
- Cached
- Common

This partitioning specifies to the ZK backend how data is supplied for different parts of the trace matrix. The *preprocessed trace* is data that is shared and agreed upon ahead of time between the prover and verifier. The *cached trace* is data that is only available to the prover, but may be cached and reused across different proofs. Lastly, the *common trace* is the remaining data available only to the prover. We note that if an AIR requires a preprocessed trace, then the height of the trace matrix is fixed.

For greater flexibility, we allow multiple AIRs in the arithmetization of a single circuit. In other words, a circuit is proved by providing multiple trace matrices. We extend the AIR arithmetization framework with an intermediate representation for constraining relations between different AIRs. This intermediate representation, known as *interactions*, was first introduced by [Val24], building on previous interfaces for lookup tables and permutation arguments.

Definition 2.2.2 (Interactions and buses). An *interaction* of width w and message length ℓ on bus b is a triple (σ, m, b) , where:

- $\sigma \in \mathbb{F}[x_1, \dots, x_w, y_1, \dots, y_w]^\ell$ is a sequence of ℓ polynomials defining the **message**.
- $m \in \mathbb{F}[x_1, \dots, x_w, y_1, \dots, y_w]$ is a polynomial that determines the **multiplicity** of the corresponding message.
- $b \in \mathbb{F} \setminus \{0\}$ is the **bus index** specifying the bus. It must be nonzero.

Given a trace matrix $\mathbf{T}$ of width w with entry $\mathbf{T}_{ij}$ on row i and column j , we say that an interaction (σ, m, b) defined on $\mathbf{T}$ *sends* over bus b , for each row i , the image

$$\sigma(\mathbf{T}_{i1}, \dots, \mathbf{T}_{iw}, \mathbf{T}_{\text{next}(i)1}, \dots, \mathbf{T}_{\text{next}(i)w}) \in \mathbb{F}^\ell$$

with multiplicity

$$m(\mathbf{T}_{i1}, \dots, \mathbf{T}_{iw}, \mathbf{T}_{\text{next}(i)1}, \dots, \mathbf{T}_{\text{next}(i)w}) \in \mathbb{F}$$

where $\text{next}(i)$ is the cyclic next row in the trace matrix, i.e., $\text{next}(i) = i + 1$ if i is not the last row and $\text{next}(i) = 1$ if i is the last row.

In our arithmetization, an AIR is augmented with a set of interactions, where the AIR width and interaction width coincide. An AIR may have multiple interactions, where each interaction may have a different message length and/or bus index.

Definition 2.2.3 (Circuit). A *circuit* $\mathcal{C}$ is a collection of (A, I) where A is an AIR and I is a collection of interactions associated with A .

2.2.4. *$\mathbb{F}$ -multiset balancing.* In order to define what it means for a collection of trace matrices to satisfy a circuit, we introduce a notion of $\mathbb{F}$ -multiset balancing below.

Consider a circuit with t AIRs of widths $w_1, \dots, w_t$, where the i -th AIR has k_i interactions $(\sigma_k^{(i)}, m_k^{(i)}, b_k^{(i)})$ of length ℓ_{ik} for $k \in \{1, \dots, k_i\}$.

The set of possible messages is denoted $\mathbb{F}^+ = \bigsqcup_{i \geq 1} \mathbb{F}^i$ (disjoint union). An $\mathbb{F}$ -multiset is a function

$$\mathcal{M} : \mathbb{F}^+ \rightarrow \mathbb{F}$$

that assigns an $\mathbb{F}$ -valued “multiplicity” to each message $\mathbb{F}^+$.

Given a set of trace matrices $\mathbf{T}^{(1)}, \dots, \mathbf{T}^{(t)}$ with respective heights $h_1, \dots, h_t$, these traces together define a multiset $\mathcal{M}_b$ for each bus index b . To simplify notation, for $i \in \{1, \dots, t\}$ (per AIR), $j \in \{1, \dots, h_i\}$ (per trace row), and $k \in \{1, \dots, k_i\}$ (per interaction), define $\hat{m}_{j,k}^{(i)} \in \mathbb{F}$ by:

$$\hat{m}_{j,k}^{(i)} = m_k^{(i)}(T_{j,1}^{(i)}, \dots, T_{j,w}^{(i)}, T_{\text{next}(j),1}^{(i)}, \dots, T_{\text{next}(j),w}^{(i)})$$

and define $\hat{\sigma}_{j,k}^{(i)} \in \mathbb{F}^{\ell_{ik}}$ analogously. The multiset defined by the traces is then given by:

$$\mathcal{M}_b(\tau) = \sum_{i=1}^t \sum_{k=1}^{k_i} \sum_{j=1}^{h_i} \hat{m}_{j,k}^{(i)} \mathbf{1}(b = b_k^{(i)} \wedge \tau = \sigma_{j,k}^{(i)}), \quad \tau \in \mathbb{F}^+$$

where $\mathbf{1}(b = b_k^{(i)} \wedge \tau = \sigma_{j,k}^{(i)})$ is the indicator function that is 1 if $b = b_k^{(i)}$ and $\tau = \sigma_{j,k}^{(i)}$ and 0 otherwise. Here we implicitly embed $\sigma_{j,k}^{(i)}$ into $\mathbb{F}^+$.

We say that a bus b is *balanced* with respect to the trace matrices if the $\mathbb{F}$ -multiset $\mathcal{M}_b$ satisfies $\mathcal{M}_b(\tau) = 0$ for all messages $\tau \in \mathbb{F}^+$.

Under certain conditions which prevent integer overflow of the field characteristic, $\mathbb{F}$ -multiset balancing can be used to imply:

- Lookup table relations: a single AIR declares a set of messages and other AIRs may constrain that a given message belongs to this set.
- Permutation checks: AIRs may add messages with positive integer multiplicities to one of two multisets (a “send” and “receive” multiset), and the bus constrains that the two multisets are equal.

These constraints will play a key role in our zkVM design, cf. §4.

2.2.5. *Circuit satisfiability.* Given a circuit $\mathcal{C}$, we say that a collection of trace matrices $(\mathbf{T}_A)_A$, one per AIR A in $\mathcal{C}$, satisfies the circuit if each trace matrix satisfies the corresponding AIR and all interaction buses are balanced with respect to the trace matrices.

2.3. **ZK backend.** Our ZK backend provides a framework to separate the implementation of the proof system from the design of the circuit by using the AIRs with interactions arithmetization as an intermediate representation.

The goal of the ZK backend is to allow different proof system implementations, in different hardware environments, to take a common input format — a circuit described as AIRs with interactions, together with a collection of trace matrices — and produce a non-interactive proof of knowledge that the trace matrices satisfy the circuit.

The backend models the proof system as a non-interactive proof system derived from a polynomial IOP via the Fiat-Shamir transform. Notably, the backend does not force a choice of PIOP. The backend framework organizes the proof system implementation into the following components, which can be customized based on the proof system and hardware environment:

- **Trace Commitment:** The trace matrices are committed to by the prover, and the verifier only receives a commitment. This is typically chosen for compatibility with a polynomial commitment scheme where each column of the trace matrix is interpolated into a polynomial. The backend supports committing multiple matrices of different heights into a single commitment.
- **Constraint Proving with Verifier Randomness:** The prover may generate additional data or simulate verifier-injected randomness, via the Fiat-Shamir transform, to prove that constraints are satisfied. As an example, this component may be used to prove interactions using LogUp techniques [Hab22].
- **Commitment Opening:** In a proof system derived from a polynomial IOP, the prover generates all necessary polynomial opening proofs. Analogously, the verifier must verify these opening proofs.

We have designed the above components to maximize flexibility and support for future proof system developments.

We describe the implementations of the above components at the time of writing of this paper to serve as an example of how the framework can be used. The SWIRL proof system is used: The trace commitment is a combination of matrix stacking – with support for matrices of different heights, Reed-Solomon encoding, and Merkle tree commitments. The constraint proving step consists of customized sumcheck arguments to link ZeroCheck and LogUp-GKR with support for trace matrices of different heights. The commitment opening step consists of a stacked opening reduction and a batch-WHIR multilinear polynomial opening.

3. INSTRUCTION SET ARCHITECTURE

We introduce the OpenVM instruction set architecture (ISA). The goal of the ISA is to provide a common open and extensible architecture that can be used for:

- Emulation of existing computer architectures (e.g., RISC-V, WASM, x86, aarch, etc.),
- Custom programs that desire lower-level integration with the underlying ZK circuit arithmetization,
- Hybrid combinations of the above within *the same* execution environment.

The ISA is intentionally designed for virtual software execution which can be provably verified. While traditional ISAs must fix certain properties of the ISA (e.g., register size) to optimize for performance on a fixed set of hardware, the OpenVM ISA is freed from many of these restrictions and provides a platform where these traditionally conflicting properties can all coexist within the same architecture.

The ISA globally depends on a prime field $\mathbb{F}$, which is the base field used in the circuit arithmetization. We identify the elements of this field with integers in the range $[0, p)$ where p is the characteristic of the field.

3.1. Virtual machine execution environment. Programs in the OpenVM ISA are executed in a virtual machine (VM). The VM is a state machine where program execution occurs within a *guest* environment that modifies a guest state. The VM is executed on a physical *host* machine,

which maintains an external host state. The ISA provides certain interfaces that allow the guest program to request or modify the host state.

3.2. Virtual machine state. The state of the virtual machine consists of the following components:

Guest State	Host State
Program ROM (Read Only)	Input Stream
Program Counter <code>pc</code>	Hint Stream
Data Memory (Read/Write)	
User Public Outputs	

We describe these components in more detail below.

3.2.1. Program ROM. OpenVM operates under the Harvard architecture, where program code is stored separately from data memory. The program code is loaded as read-only memory (ROM) in the VM state prior to execution, and it remains immutable throughout the execution. Program code is a partially defined map

$$[0, 2^{\text{PC_BITS}}) \rightarrow \mathbb{F} \times \mathbb{F}^{\text{NUM_OPERANDS}}$$

where $\text{PC_BITS} = 30$ and the target is the space of instructions (see below). Instructions will typically only exist at a subset of the domain $[0, 2^{\text{PC_BITS}})$.

3.2.2. Instruction format. Instructions are encoded as a global opcode (field element) followed by $\text{NUM_OPERANDS} = 7$ operands (field elements):

$$\text{Inst} = (\text{opcode}, \text{operands}) \in \mathbb{F} \times \mathbb{F}^{\text{NUM_OPERANDS}}.$$

The maximum number of operands may be increased in the future without affecting existing instructions.

3.2.3. Program counter. There is a single special purpose register `pc` $\in \mathbb{F}$ for the program counter which stores the location of the instruction being executed.

3.2.4. Data memory. Data memory is a random access memory (RAM) which supports read and write operations. Memory consists of addressable cells which represent a single field element indexed by *address space* and *pointer*, where address space and pointer are each a field element. Memory may be viewed as a partially defined map

$$\mathbb{F} \times \mathbb{F} \rightarrow \mathbb{F} : (\text{addr_space}, \text{ptr}) \mapsto [\text{ptr}]_{\text{addr_space}}$$

VM instructions can access (read or write) a contiguous list of cells (called a *block*) in a single address space. The block size is fixed to 4 cells across the current ISA, and block accesses must be aligned to this size. Instructions whose logical semantics access fewer cells, such as byte or halfword loads and stores, are implemented using aligned 4-cell block accesses together with local constraints that select the relevant sub-block.

3.2.5. Address spaces. To support execution of programs with different data assumptions and properties within the same architecture, we separate data memory into different *address spaces*. While memory cells within each address space can always be viewed as field elements, the ISA may impose additional invariants on certain address spaces to optimize the performance of sub-classes of instructions.

The address spaces below are reserved and used by existing OpenVM instructions:

Address Space	Name	Description
1	Registers	Elements are constrained to lie in $[0, 2^8)$
2	User Memory	Elements are constrained to lie in $[0, 2^8)$
3	User IO	Used to expose user outputs
4	Deferral	Stores deferral accumulators as field elements (§5.5)

The address space 0 is reserved and not used by data memory¹.

3.2.6. Inputs and hints. To enable user input and non-determinism in OpenVM programs, the host state must maintain an input stream and hint stream. The input stream is a non-interactive queue of data that the host is instantiated with prior to program execution. The hint stream allows the host to provide auxiliary data during execution. The hint stream itself is unconstrained by the zkVM circuit, so any required relation between hint values and guest state must be enforced by the instructions and program logic that consume them.

The ISA allows the addition of custom instructions that can request modification of the host state by either moving data from the input stream to the hint stream or by specifying preferred operations that the host should perform to mutate its state (e.g., the host may perform a square root of a value in guest memory). The guest may also copy data from the host state to guest memory via explicit instruction calls.

3.2.7. User public outputs. To make program outputs public, OpenVM allows the guest program to specify a list of field elements to make public. This list can be modified by certain custom instructions.

3.3. VM execution model. The VM is a state machine, and we view VM execution as a series of state transitions

$$\text{State}_0 \xrightarrow{\text{Step}} \text{State}_1 \xrightarrow{\text{Step}} \dots \xrightarrow{\text{Step}} \text{State}_{\text{final}}$$

We define the conditions for a state transition to be valid below. Program execution in the VM is declared *successful* with respect to the initial state if all state transitions are valid and the instruction at the program counter pc_{final} in the final state is a special terminating instruction **TERMINATE** with exit code 0 (success).

Program execution may *fail* for the following reasons:

- (1) The sequence of state transitions is valid and the final program counter yields the **TERMINATE** instruction with a non-zero exit code. This is a requested trap by the guest program to exit unsuccessfully.
- (2) The sequence of state transitions results in an invalid state transition. This indicates that the VM state no longer satisfies the required properties of either the architecture or the specific instruction being executed.

¹Address space 0 is used as an operand flag to specify immediate values in instructions.

3.3.1. *Guest program execution.* We define *guest program execution* to be the subset of VM execution that only mutates the guest state:

- program counter
- data memory
- user public outputs.

Guest program execution can be modeled as a series of state transitions on the guest state, assuming oracle access to a non-deterministic host state.

3.3.2. *Initial state.* The initial state of the VM consists of:

- Program ROM – immutable throughout VM execution
- pc_0 – starting program counter
- Initial data memory
- Input stream

The user public outputs and hint stream are empty.

3.3.3. *State transition.* A state transition is a function

$$\text{Step} : \text{State}_{\text{from}} \rightarrow \text{State}_{\text{to}}.$$

It consists of the following stages:

Instruction Fetch: The instruction is fetched from the program ROM based on the current program counter $pc_{\text{from}} \in \text{State}_{\text{from}}$.

Instruction Routing: The opcode is parsed from the instruction and used to decide how instruction execution will be handled.

Instruction Execution: Execution of the instruction is custom according to the specification of the instruction. Instruction execution is not required to be atomic and has mutable access to the global VM state:

- Program Counter
- Data Memory
- User Public Outputs
- Input Stream
- Hint Stream

Local state may be maintained during execution of a single instruction, but the only state that is persisted and shared outside of the lifecycle of execution of a single instruction is the VM state above.

Instruction execution is expected to end with an update of the program counter to pc_{to} . Instructions that do not pertain to control flow will advance $pc_{\text{to}} = pc_{\text{from}} + \text{DEFAULT_PC_STEP}$ where $\text{DEFAULT_PC_STEP} = 4$ is set for compatibility with RISC-V conventions.

The state transition is valid if a valid instruction is fetched from the program ROM and the instruction's execution maintains the required invariants of the VM state. In particular, memory access bounds and required properties of memory address spaces must be respected. The pc_{to} program counter must also be set to a valid instruction.

3.3.4. Phantom sub-instructions. To facilitate hinting and debugging on the host, the ISA supports the notion of *phantom instructions*. These are instructions which are identical to a no-op at the level of the guest state, but which may be used to request mutations in the host state. Use cases of phantom instructions include interacting with the input or hint streams or displaying debug information on the host machine.

3.4. VM extensions. The ISA is designed to be maximally extensible. Outside of a few system instructions, there is no required “base” instruction set. Instead, instructions are grouped into composable **VM extensions** which are instruction set extensions to the ISA meant to provide the highest degree of customization and performance for VM execution.

A VM extension consists of a collection of instructions, which must adhere to the ISA invariants previously described in this section, and the specification for each instruction of:

- (1) The guest instruction execution,
- (2) The preferred host behavior during execution,
- (3) The implementation of the instruction within the circuit architecture.

We discuss item (3) in more detail in §4.

VM extensions allow new instructions to be easily introduced to accelerate custom workloads as composable “add-ons” to existing instruction sets. This takes full advantage of the virtual nature of the execution environment to dramatically lower the barrier to customizing computer architecture.

4. ZKVM DESIGN

We describe the design of our ZK circuit architecture that enables creation of zkVMs for the OpenVM ISA defined in §3. Since the OpenVM ISA is not a single instruction set but a framework to define custom instruction sets from composable VM extensions, we clarify our definition of a zkVM:

Definition 4.0.1 (zkVM). Given a fixed set of instructions within the OpenVM ISA, a ZK virtual machine (zkVM) is the collection of:

- (1) A host execution environment capable of executing every instruction in the instruction set.
- (2) A circuit $\mathcal{C}$ as defined in Definition 2.2.3 that constrains a finite sequence of guest state transitions to be valid with respect to a public commitment to an initial guest state. The circuit exposes a commitment of the final guest state as public output. The state transitions are presented in the form of trace matrices.
- (3) Methods to generate the trace matrices from the guest state transitions necessary to generate a proof for the circuit $\mathcal{C}$.

We note that the zkVM circuit only supports a *finite* sequence of state transitions, whereas program execution is *a priori* unbounded. We describe how we prove successful guest execution of unbounded programs using continuations in §5.2.

Further note that the zkVM circuit only constrains validity of guest state transitions. The circuit *does not* constrain host behavior. As such, the guest program always operates under the assumption that the host environment is *untrusted*. The guest program may use control flow to handle preferred host behavior, but it must ensure that program execution cannot be tampered with by a dishonest host.

In the rest of this section, we explain the framework which enables the creation of zkVMs for *any* instruction set defined within the OpenVM ISA. In particular, the framework enables zkVM circuits to be extended to support new VM extensions via composition.

4.1. Overview. At a high level, we organize the zkVM into *chips*, where a chip contains a single AIR and the methods to generate the trace matrix for that AIR. Our zkVM consists of a collection of system chips required by the architecture together with an *inventory* of chips defined by VM extensions.

Chips come in two types:

- Instruction executor
- Periphery

An instruction executor chip owns the execution methods for a subset of the instructions in the instruction set. Each instruction must be handled by exactly one chip, but the same chip may handle multiple instructions. Periphery chips provide auxiliary functionality such as lookup tables that may be shared by other chips. Each chip may expose interfaces for communicating with other chips, even those defined by other VM extensions.

Instruction execution is distributed across multiple chips. In the host execution environment, instruction parsing, routing, and the global VM state are managed by a centralized controller. However, in the zkVM circuit, there is *no centralized entity*: the constraints on valid state transitions are enforced entirely through interaction buses, which we explain below.

We will focus on the design of the circuit architecture, as that dictates the other aspects of the zkVM implementation.

4.2. Temporal execution in circuit. We revisit the VM execution model introduced in §3.3. Our VM execution implicitly incorporates temporal logic – execution is treated as a serial sequence of state transitions continually progressing forward in time. On the other hand, the circuit arithmetization does not inherently provide a representation of temporal constraints.

In order to represent constraints of temporal logic within the zkVM circuit, we discretize time and materialize a global timestamp t as a separate variable in circuit constraints. The execution trace must associate a timestamp t_i with each state State_i that appears in a state transition. Given state transitions

$$\text{State}_0 \xrightarrow{\text{Step}} \text{State}_1 \xrightarrow{\text{Step}} \dots \xrightarrow{\text{Step}} \text{State}_{\text{final}}$$

where each **Step** represents execution of a single instruction, the circuit must constrain:

- (1) **Continuity between instructions:** The timestamps must be monotonically increasing:

$$t_0 < t_1 < t_2 < \dots < t_{\text{final}}.$$

- (2) **Continuity within an instruction:** Execution of a single instruction consists of a finite sequence of atomic accesses to the guest state. The execution trace must associate an intermediate timestamp $t_{i,j}$ to each guest state access such that

$$t_i < t_{i,1} < t_{i,2} < \dots < t_{i,k} < t_{i+1},$$

and these discrete timestamps represent the only points at which the guest state is accessed. A state access includes any read or write.

Recall that the guest state consists of the program ROM, **pc**, data memory, and user public outputs. To simplify the design, we require that the program ROM is only read at the boundaries t_i, t_{i+1} of a **Step** : $\text{State}_i \rightarrow \text{State}_{i+1}$. Likewise, the program counter

is only accessed and updated at timestamps t_i, t_{i+1} . Given this, the only state accesses at intermediate timestamps $t_{i,j}$ are of the data memory and user public outputs. To summarize, a single instruction execution can be decomposed into a sequence of smaller state transitions

$$\begin{array}{c} \text{Step} \\ \overbrace{(\text{State}_i, t_i) \xrightarrow{f_{i,0}} (\text{State}_{i,1}, t_{i,1}) \xrightarrow{f_{i,1}} \dots \xrightarrow{f_{i,k}} (\text{State}_{i+1}, t_{i+1})} \end{array}$$

where each intermediate $t_{i,j}$ marks an atomic access to data memory or user public outputs, and the transitions functions $f_{i,j}$ contain no other state accesses.

4.3. System buses. The circuit architecture relies on three system buses to constrain valid state transitions during program execution:

- Program bus
- Execution bus
- Memory bus

We explain each bus’s message format and how it is used within the architecture below.

4.4. Program bus. The program bus ensures the correctness of instruction fetching during execution. A message on the program bus has the form:

$$(\text{pc}, \text{opcode}, \text{operands})$$

where pc is the program counter and $\text{opcode}, \text{operands}$ represents an instruction.

The program bus is used as a lookup bus for instruction executor AIRs to constrain that the instruction they are executing indeed exists in the program ROM at the given program counter. This is enabled by a *program chip*, whose trace matrix contains the program ROM as a cached trace with one instruction per row. The cached trace is committed to separately from the rest of the zkVM trace so that its commitment is a commitment of the program ROM. In this way, the program ROM is *not* a fixed part of the circuit constraints. Instead the prover can load different program ROMs, hence initializing the VM with different states, to generate proofs of execution for *the same* circuit.

4.5. Execution bus: the no-CPU design. The execution bus constrains the continuity of program execution between instructions and also constrains state accesses to the program counter. A message on the execution bus has the form:

$$(\text{pc}, t)$$

Chips that interact with the execution bus must maintain the invariant that a message (pc, t) appears in the bus if and only if the program counter equals pc in the guest state at timestamp t during program execution.

The execution bus is used to perform a permutation check between a “send” and “receive” set of messages. The architecture requires that every instruction executor AIR *must* constrain that it adds a message $(\text{pc}_{\text{from}}, t_{\text{from}})$ to the receive set and a message $(\text{pc}_{\text{to}}, t_{\text{to}})$ to the send set exactly once for each instruction that appears in the AIR trace. The AIR must also constrain that $t_{\text{from}} < t_{\text{to}}$.

A *connector chip* adds a message $(\text{pc}_0, 1)$ to the send set and a message $(\text{pc}_{\text{final}}, t_{\text{final}})$ to the receive set and exposes $(\text{pc}_0, \text{pc}_{\text{final}})$ as public outputs.

Assuming that the instruction executor AIRs enforce the constraints above, the send and receive sets are equal if and only if the read and write access to the program counter is consistent across all instructions executed. In other words, it constrains that the routing of instructions and the transition between instructions is consistent.

The execution bus allows the transcript of program execution to be distributed across the traces of multiple chips, and there does not need to be a single central chip that materializes the full transcript. Prior zkVM designs have traditionally relied on a single central processing unit (CPU) chip to materialize this complete transcript. The existence of a CPU chip leads to a trace matrix with rows that grow with the total number of clock cycles in program execution, which creates a performance bottleneck. Our use of the execution bus avoids this bottleneck, which is why we colloquially refer to our design as the “no-CPU” architecture.

4.6. Memory bus. The memory bus is an adaptation of the offline memory checking argument of [BEG⁺94]. A message on the memory bus has the form:

$$(\text{addr_space}, \text{ptr}, \text{data}, t)$$

where $\text{addr_space}, \text{ptr}, t \in \mathbb{F}$ and $\text{data} \in \mathbb{F}^B$ where B is the fixed memory block size of the ISA. In the current ISA, $B = 4$. Chips that interact with the memory bus must maintain the invariant that a message $(\text{addr_space}, \text{ptr}, \text{data}, t)$ appears in the bus if and only if at timestamp t the data memory had values data in address space addr_space at pointers $[\text{ptr}, \text{ptr} + B)$ during program execution.

4.6.1. Offline Memory Checking: Handling Read and Write Operations. The memory bus is used to perform a permutation check between a “send” and “receive” multiset of messages. Each instruction executor chip in the system independently maintains memory consistency by interacting with the send and receive multisets during both read and write operations.

Read Operation: To constrain a read operation, an AIR must

- (1) add a message $(\text{addr_space}, \text{ptr}, \text{data}, t_{\text{prev}})$ to the receive set, and
- (2) add a message $(\text{addr_space}, \text{ptr}, \text{data}, t)$ to the send set.

The AIR must constrain that $t_{\text{prev}} < t$. The value of t_{prev} should be the timestamp of the last access to the block at $(\text{addr_space}, \text{ptr})$ prior to timestamp t .

Write Operation: To constrain a write operation, an AIR must

- (1) add a message $(\text{addr_space}, \text{ptr}, \text{data}_{\text{prev}}, t_{\text{prev}})$ to the receive set, and
- (2) add a message $(\text{addr_space}, \text{ptr}, \text{data}, t)$ to the send set.

The AIR must constrain that $t_{\text{prev}} < t$. The value of t_{prev} should be the timestamp of the last access to the block at $(\text{addr_space}, \text{ptr})$ prior to timestamp t , and the value of $\text{data}_{\text{prev}}$ should be the value of that block in data memory immediately before timestamp t .

4.6.2. Initial and Final Memory States. A memory *boundary chip* is used to add messages to the send multiset at timestamp 0 and to the receive multiset at timestamp t_{final} . In order for the bus to balance, the messages at timestamp 0 (resp. t_{final}) must correspond to the initial (resp. final) memory state.

The memory boundary chip exposes a Merkle root of the initial memory state as a public value and constrains the messages at timestamp 0 to be consistent with the Merkle root via Merkle proofs, which are constrained by an auxiliary chip. It also exposes a Merkle root of the final memory state and constrains that the messages at timestamp t_{final} are consistent with this

Merkle root. These memory-root public values are used to check consistency between segment boundaries in the continuations setting (see §5).

4.7. Public values. The public values of a zkVM circuit are as follows:

- (1) The initial and final program counters $\text{pc}_0, \text{pc}_{\text{final}}$.
- (2) Merkle roots of the initial and final memory states.

User public outputs are stored within the data memory in a dedicated address space. They are not direct public values of the zkVM circuit; instead they are extracted using a Merkle proof from the final memory Merkle root during proof verification or aggregation.

4.8. Hinting and phantom sub-instructions. The system provides a *phantom chip* to handle phantom sub-instructions. The phantom chip’s AIR interacts only with the program and execution bus and advances the program counter by the default step size and performs no other state accesses. However the phantom chip’s host execution can register custom behavior for phantom sub-instructions. The host execution of these sub-instructions can read both guest and host state and modify the hint stream in the host state.

Special instructions may write to the guest data memory based on values read from the host state. The AIRs associated to these instructions must constrain the memory writes via the memory bus as required for all guest state accesses.

4.9. VM extension support. The zkVM design is centered around support for custom VM extensions. An existing zkVM circuit can be extended to support the instructions from a new VM extension (assuming they are compatible with existing instructions) by adding new chips – both instruction executor chips and periphery chips – to handle execution of the new instructions. The VM extension may specify new buses for cross-chip communication and also use existing buses from other extensions. The chips are provided with interfaces to access the full global VM state during instruction execution. The circuit architecture requires that the chips’ AIRs must interact with the three system buses as described above. Chip design is fully modular because the architecture gives each chip full control of the state transition for the instructions that it executes.

5. PROOF ARCHITECTURE

We describe the proof architecture used to prove correct execution of unbounded OpenVM programs. As noted in Definition 4.0.1, a zkVM circuit constrains only a finite sequence of guest state transitions. OpenVM therefore splits a long execution into finite *segments*, proves each segment independently, and recursively aggregates the resulting proofs into the proof format requested by the client.

The architecture has two goals. First, each segment proof should be self-contained, so that segment proving can be parallelized with minimal communication between provers. Second, recursive aggregation should expose enough public data to check segment adjacency and successful termination, while hiding intermediate execution state from the final verifier.

5.1. Topology. The proof pipeline consists of a sequence of layers. Each layer generates proofs for a particular circuit. For all except the static layer, the proofs are STARK proofs. The static layer generates a SNARK proof intended for on-chain verification.

App layer. The App VM produces one **app** proof per executed segment. The corresponding relation is denoted $\mathcal{R}_{\text{app}}$.

Leaf layer. A leaf aggregation circuit verifies up to N_{leaf} adjacent App segment proofs and produces one proof of relation $\mathcal{R}_{\text{leaf}}$.

Internal-for-leaf layer. An internal-for-leaf aggregation circuit verifies up to N_{internal} leaf proofs and preserves the same boundary-state public value interface. Its relation is denoted $\mathcal{R}_{\text{ifl}}$.

Internal-recursive layer. An internal-recursive aggregation circuit verifies up to N_{internal} internal-for-leaf proofs, or up to N_{internal} internal-recursive proofs. Its relation is denoted $\mathcal{R}_{\text{irec}}$.

Root layer. The root aggregation circuit verifies one internal-recursive proof, checks successful termination, and exposes the user public values. Its relation is denoted $\mathcal{R}_{\text{root}}$.

Static layer. The static layer re-encodes the root proof into a fixed, constant-sized format for on-chain verification. Its relation is denoted $\mathcal{R}_{\text{static}}$.

The client-facing proof formats correspond to truncations of this pipeline. An **app** proof consists only of non-aggregated App VM segment proofs. A (final) STARK proof runs the **leaf**, **internal-for-leaf**, and **internal-recursive** layers until a single aggregated proof remains. An EVM proof additionally runs the **root** and **static** layers.

Each aggregation layer consumes child proofs and produces a new proof with public values determined by the public values of its children. From the point of view of the proof architecture, the essential interface is therefore the public value vector exposed at each layer and the consistency checks that a parent performs on the public values of its children.

Figure 1 shows the topology schematically.

5.2. App segment proofs and continuations. We call the zkVM and associated instruction set for the unbounded program the “App” VM. If the execution has segments $S_0, \dots, S_{m-1}$, then each segment produces an **app** proof of the same App VM circuit, but with different trace matrices and different boundary public values.

The App public values propagated through aggregation are

$$\text{AppPVs} = (\text{pc}_{\text{init}}, \text{pc}_{\text{final}}, \text{exit}, b_{\text{term}}, R_{\text{init}}, R_{\text{final}}),$$

where R_{init} and R_{final} are Merkle roots of the initial and final data memory states. The value b_{term} indicates whether the segment terminates the program. A non-terminating segment has $b_{\text{term}} = 0$ and uses the default suspend exit code; a successfully terminating segment has $b_{\text{term}} = 1$ and uses the success exit code 0.

In order for proof aggregation to check consistency of boundary states between segments, the segment proof must expose commitments to the relevant guest state:

- (1) The program ROM does not change between segments and is recorded as a separate cached trace commitment in the proof (cf. §4.4).
- (2) The initial and final program counters of the segment are exposed by the connector chip (cf. §4.5), together with the exit code and termination flag.
- (3) The memory boundary chip exposes Merkle roots of the initial and final data memory states (cf. §4.6.2).

To expand on (3), we store the data memory state as a binary Merkle trie, where a path in the trie corresponds to the binary encoding of a memory address $(a - 1, p)$, where a is the address space and p is the pointer. The consistency of memory accesses within a segment is constrained by the memory bus, while the memory boundary chip adds messages to the memory

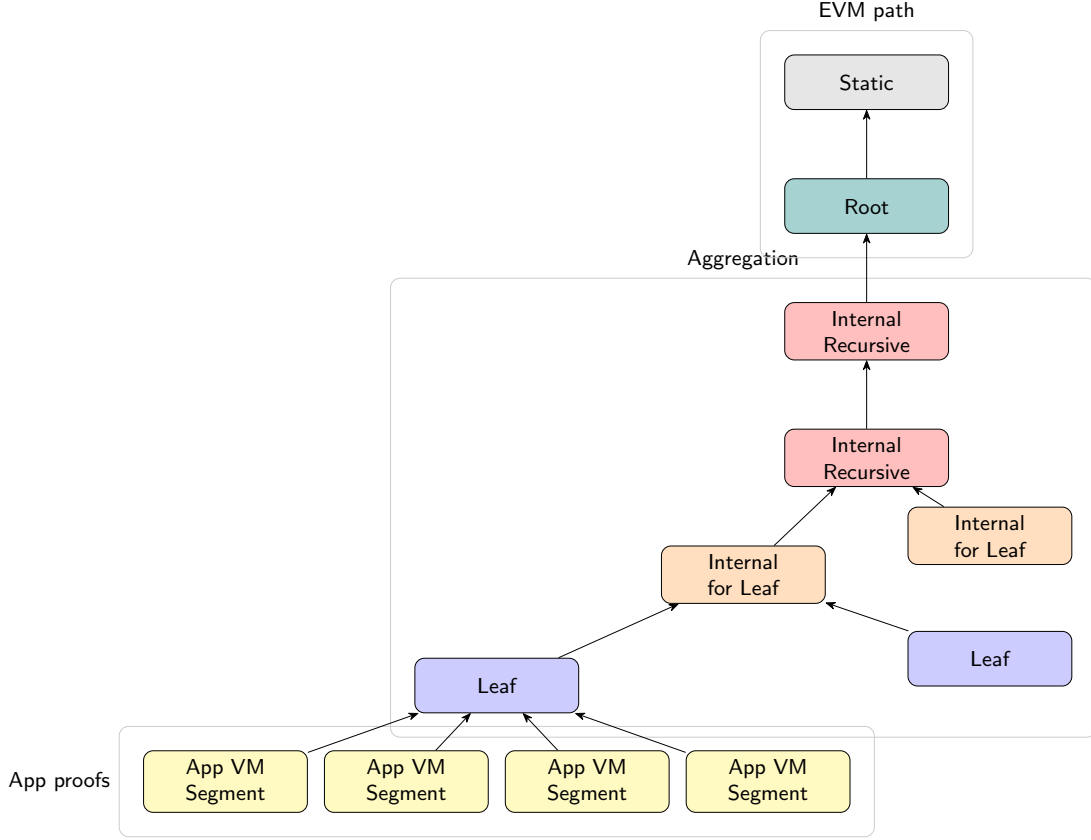


FIGURE 1. Proof aggregation topology. A leaf aggregates several adjacent App VM segment proofs. Internal-for-leaf and internal-recursive layers continue aggregation until one proof remains. The root and static layers are added for the EVM proof path.

bus for the initial and final memory states. These messages must be consistent with the public Merkle roots, where consistency is constrained by Merkle proof verifications in circuit.

The boundary chip only needs to add messages for addresses accessed in the segment. Moreover, the Merkle proofs for the initial memory state, respectively the final memory state, can be combined into a single multi-proof that proves inclusion of multiple leaf nodes using shared witness data. When memory accesses can be grouped into large continuous ranges, which is often the case for programs using standard memory allocators, the multi-proof gives significant performance benefits.

User public values are not direct public values of the App segment proof. Instead, the guest stores them in a dedicated public values address space inside data memory. Their presence in the final memory state is checked only later by either the native host verifier or by the root aggregation circuit.

5.3. Inner aggregation layers. The leaf, internal-for-leaf, and internal-recursive layers all use the same public value layout. For any of these layers we write the aggregation relation as

$$\mathcal{R}_{\text{inner}}((\text{PVs}_{\text{out}}); \pi_0, \dots, \pi_{k-1}), \quad 1 \leq k \leq N,$$

where N is N_{leaf} for a leaf aggregation and N_{internal} for an internal aggregation. This relation holds when the child proofs are valid proofs of the preceding layer, their public values describe adjacent execution ranges, and the output public values are computed from the child public values.

5.3.1. Public value layout. The public values of an inner aggregation proof are

$$\begin{aligned} \text{PVs}_{\text{out}} = & (\text{pc}_{\text{init}}^{\text{out}}, \text{pc}_{\text{final}}^{\text{out}}, \text{exit}^{\text{out}}, b_{\text{term}}^{\text{out}}, R_{\text{init}}^{\text{out}}, R_{\text{final}}^{\text{out}}, C_{\text{prog}}^{\text{out}}, f_{\text{int}}^{\text{out}}, f_{\text{rec}}^{\text{out}}, \\ & (C_{\text{app}}^{\text{vk}})^{\text{out}}, (C_{\text{leaf}}^{\text{vk}})^{\text{out}}, (C_{\text{ifl}}^{\text{vk}})^{\text{out}}, (C_{\text{irec}}^{\text{vk}})^{\text{out}}). \end{aligned}$$

Here the first six entries are the **app** public values for the aggregated execution range and C_{prog} is the **app** program commitment. The flags record which inner layer produced the proof and whether the proof is not yet internal-recursive, is the first internal-recursive layer, or is a later internal-recursive layer. Deferral-enabled proofs carry additional public values described in the deferral integration subsection.

Each $C_{\bullet}^{\text{vk}}$ entry is a **VkCommit**, not a single digest. It is a pair

$$C_{\ell}^{\text{vk}} = (C_{\ell, \text{cached}}^{\text{vk}}, h_{\ell, \text{pre}}^{\text{vk}}), \quad \ell \in \{\text{app}, \text{leaf}, \text{ifl}, \text{irec}\}.$$

The first component $C_{\ell, \text{cached}}^{\text{vk}}$ is the PCS commitment to the cached trace obtained by serializing the ℓ circuit's AIR constraints and interactions. The second component $h_{\ell, \text{pre}}^{\text{vk}}$ is a hash digest of parts of the verifying key to protect against weak Fiat–Shamir attacks. Together C_{ℓ}^{vk} is a full commitment to the verifying key.

When an aggregation circuit verifies a child proof π , the circuit will apply relations to public values from π as well as public values of the circuit itself. These public values include $C_{\ell, \text{cached}}^{\text{vk}}$. We write

$$\text{VkCheck}(C_{\text{cached}}, \pi_i)$$

for the check that the public values C_{cached} match the corresponding cached trace PCS commitment in π_i .

The companion public value $h_{\ell, \text{pre}}^{\text{vk}}$ is handled only by equality. If a child proof already exposes C_{ℓ}^{vk} , then the parent equates the child's $h_{\ell, \text{pre}}^{\text{vk}}$ public value with the corresponding parent public value. If the current layer first introduces C_{ℓ}^{vk} , then the $h_{\ell, \text{pre}}^{\text{vk}}$ component is re-exposed as a parent public value; there is no child public value for that component yet.

Within any one aggregation layer, all child proofs are proofs of the same child circuit. The layer's relation enforces that all child proofs have the same cached trace commitment. This single cached commitment is re-exposed as public values in the parent proof.

5.3.2. Common range constraints. The superscript out always denotes the public values of the parent proof being produced by the aggregation circuit. For example, $\text{pc}_{\text{init}}^{\text{out}}$ is the initial program counter in the parent's public values. For each child proof π_i , with $0 \leq i < k$, write the same symbols with superscript (i) for that child's public values; for example, $\text{pc}_{\text{init}}^{(i)}$ is the initial program counter in the public values of the child with index i . At the leaf layer, where the children are **app** segment proofs, this child notation applies only to the **app** public values and to the **app** program commitment. Let e_{susp} denote the default suspend exit code.

All inner aggregation layers impose the same execution-range composition constraints:

$$\begin{aligned}
\text{pc}_{\text{init}}^{\text{out}} &= \text{pc}_{\text{init}}^{(0)}, & R_{\text{init}}^{\text{out}} &= R_{\text{init}}^{(0)}, \\
\text{pc}_{\text{final}}^{\text{out}} &= \text{pc}_{\text{final}}^{(k-1)}, & R_{\text{final}}^{\text{out}} &= R_{\text{final}}^{(k-1)}, \\
\text{exit}^{\text{out}} &= \text{exit}^{(k-1)}, & b_{\text{term}}^{\text{out}} &= b_{\text{term}}^{(k-1)}, \\
\text{pc}_{\text{final}}^{(i)} &= \text{pc}_{\text{init}}^{(i+1)}, & R_{\text{final}}^{(i)} &= R_{\text{init}}^{(i+1)} \quad (0 \leq i < k-1), \\
(\text{exit}^{(i)}, b_{\text{term}}^{(i)}) &= (e_{\text{susp}}, 0) \quad (0 \leq i < k-1), \quad (\text{exit}^{(k-1)}, b_{\text{term}}^{(k-1)}) \in \{(e_{\text{susp}}, 0), (0, 1)\}.
\end{aligned}$$

Thus the output range starts at the first child, ends at the last child, and no child before the last can be terminal.

5.3.3. *Leaf layer.* The children are **app** segment proofs. The leaf relation verifies those **app** proofs and checks

$$\begin{aligned}
C_{\text{prog}}^{(0)} &= \dots = C_{\text{prog}}^{(k-1)} = C_{\text{prog}}^{\text{out}}, \\
f_{\text{int}}^{\text{out}} &= 0, & f_{\text{rec}}^{\text{out}} &= 0, \\
(C_{\text{leaf}}^{\text{vk}})^{\text{out}} &= (C_{\text{ifl}}^{\text{vk}})^{\text{out}} = (C_{\text{irec}}^{\text{vk}})^{\text{out}} = 0.
\end{aligned}$$

For each **app** child proof, the **app** cached commitment relation is

$$\text{VkCheck}((C_{\text{app, cached}}^{\text{vk}})^{\text{out}}, \pi_i) \quad (0 \leq i < k).$$

The public values $(h_{\text{app, pre}}^{\text{vk}})^{\text{out}}$ are unconstrained and meant to be re-exposed in later aggregation layers. The honest prover sets them equal to the **app** pre-vkey hash digest.

5.3.4. *Internal-for-leaf layer.* The children are leaf proofs. The parent verifies each child proof and checks that each child has the leaf-layer public-value pattern:

$$\begin{aligned}
f_{\text{int}}^{(i)} &= 0, & f_{\text{rec}}^{(i)} &= 0, \\
(C_{\text{leaf}}^{\text{vk}})^{(i)} &= (C_{\text{ifl}}^{\text{vk}})^{(i)} = (C_{\text{irec}}^{\text{vk}})^{(i)} = 0 \quad (0 \leq i < k).
\end{aligned}$$

It also checks that the already-set commitments agree across children and are propagated to the output:

$$\begin{aligned}
C_{\text{prog}}^{(0)} &= \dots = C_{\text{prog}}^{(k-1)} = C_{\text{prog}}^{\text{out}}, \\
(C_{\text{app}}^{\text{vk}})^{(0)} &= \dots = (C_{\text{app}}^{\text{vk}})^{(k-1)} = (C_{\text{app}}^{\text{vk}})^{\text{out}},
\end{aligned}$$

where equality of $C_{\text{app}}^{\text{vk}}$ means equality of both $C_{\text{app, cached}}^{\text{vk}}$ and $h_{\text{app, pre}}^{\text{vk}}$. It then sets the layer markers and remaining unset verifying key commits as

$$\begin{aligned}
f_{\text{int}}^{\text{out}} &= 1, & f_{\text{rec}}^{\text{out}} &= 0, \\
(C_{\text{ifl}}^{\text{vk}})^{\text{out}} &= (C_{\text{irec}}^{\text{vk}})^{\text{out}} = 0.
\end{aligned}$$

For each leaf child proof, the leaf cached commitment relation is

$$\text{VkCheck}((C_{\text{leaf, cached}}^{\text{vk}})^{\text{out}}, \pi_i) \quad (0 \leq i < k).$$

The public values $(h_{\text{leaf, pre}}^{\text{vk}})^{\text{out}}$ are unconstrained and meant to be re-exposed in later aggregation layers. The honest prover sets them equal to the leaf pre-vkey hash digest.

5.3.5. *First internal-recursive layer.* The children are internal-for-leaf proofs. The parent checks the internal-for-leaf child pattern

$$\begin{aligned} f_{\text{int}}^{(i)} &= 1, & f_{\text{rec}}^{(i)} &= 0, \\ (C_{\text{ifl}}^{\text{vk}})^{(i)} &= (C_{\text{irec}}^{\text{vk}})^{(i)} = 0 \quad (0 \leq i < k), \end{aligned}$$

requires

$$\begin{aligned} C_{\text{prog}}^{(0)} &= \dots = C_{\text{prog}}^{(k-1)} = C_{\text{prog}}^{\text{out}}, \\ (C_{\text{app}}^{\text{vk}})^{(0)} &= \dots = (C_{\text{app}}^{\text{vk}})^{(k-1)} = (C_{\text{app}}^{\text{vk}})^{\text{out}}, \\ (C_{\text{leaf}}^{\text{vk}})^{(0)} &= \dots = (C_{\text{leaf}}^{\text{vk}})^{(k-1)} = (C_{\text{leaf}}^{\text{vk}})^{\text{out}}, \end{aligned}$$

where equality of each $C_{\bullet}^{\text{vk}}$ means equality of both its cached commitment and pre-vkey hash components. It then sets the layer markers and remaining unset verifying key commit as

$$\begin{aligned} f_{\text{int}}^{\text{out}} &= 2, & f_{\text{rec}}^{\text{out}} &= 1, \\ (C_{\text{irec}}^{\text{vk}})^{\text{out}} &= 0. \end{aligned}$$

For each internal-for-leaf child proof, the internal-for-leaf cached commitment relation is

$$\text{VkCheck}((C_{\text{ifl, cached}}^{\text{vk}})^{\text{out}}, \pi_i) \quad (0 \leq i < k).$$

The public values $(h_{\text{ifl, pre}}^{\text{vk}})^{\text{out}}$ are unconstrained and meant to be re-exposed in later aggregation layers. The honest prover sets them equal to the internal-for-leaf pre-vkey hash digest.

5.3.6. *Later internal-recursive layers.* The children are internal-recursive proofs. For some $r \in \{1, 2\}$, all children must satisfy

$$f_{\text{int}}^{(i)} = 2, \quad f_{\text{rec}}^{(i)} = r \quad (0 \leq i < k),$$

and the parent propagates the commitments that have already been set:

$$\begin{aligned} C_{\text{prog}}^{(0)} &= \dots = C_{\text{prog}}^{(k-1)} = C_{\text{prog}}^{\text{out}}, \\ (C_{\text{app}}^{\text{vk}})^{(0)} &= \dots = (C_{\text{app}}^{\text{vk}})^{(k-1)} = (C_{\text{app}}^{\text{vk}})^{\text{out}}, \\ (C_{\text{leaf}}^{\text{vk}})^{(0)} &= \dots = (C_{\text{leaf}}^{\text{vk}})^{(k-1)} = (C_{\text{leaf}}^{\text{vk}})^{\text{out}}, \\ (C_{\text{ifl}}^{\text{vk}})^{(0)} &= \dots = (C_{\text{ifl}}^{\text{vk}})^{(k-1)} = (C_{\text{ifl}}^{\text{vk}})^{\text{out}}. \end{aligned}$$

Again, equality of $C_{\bullet}^{\text{vk}}$ is componentwise equality of both $C_{\bullet, \text{cached}}^{\text{vk}}$ and $h_{\bullet, \text{pre}}^{\text{vk}}$. The output flags are always

$$f_{\text{int}}^{\text{out}} = 2, \quad f_{\text{rec}}^{\text{out}} = 2.$$

If $r = 1$, then this is the second recursive layer, so

$$(C_{\text{irec}}^{\text{vk}})^{(i)} = 0 \quad (0 \leq i < k).$$

The parent introduces the internal-recursive verifying key commit as an output public value. Its cached commitment relation is

$$\text{VkCheck}((C_{\text{irec, cached}}^{\text{vk}})^{\text{out}}, \pi_i) \quad (0 \leq i < k).$$

The public values $(h_{\text{irec, pre}}^{\text{vk}})^{\text{out}}$ are unconstrained and meant to be re-exposed in later aggregation layers. The honest prover sets them equal to the internal-recursive pre-vkey hash digest.

If $r = 2$, then the internal-recursive verifying key commitment has already been set and is propagated unchanged:

$$(C_{\text{irec}}^{\text{vk}})^{(0)} = \dots = (C_{\text{irec}}^{\text{vk}})^{(k-1)} = (C_{\text{irec}}^{\text{vk}})^{\text{out}},$$

which includes the pre-vkey hash equalities

$$(h_{\text{irec,pre}}^{\text{vk}})^{(0)} = \dots = (h_{\text{irec,pre}}^{\text{vk}})^{(k-1)} = (h_{\text{irec,pre}}^{\text{vk}})^{\text{out}}.$$

The cached commitment relation is also still checked:

$$\text{VkCheck}((C_{\text{irec,cached}}^{\text{vk}})^{\text{out}}, \pi_i) \quad (0 \leq i < k).$$

5.4. Root aggregation layer. The root layer verifies a single internal-recursive child proof. At this point segment adjacency has already been enforced, so the root proof no longer exposes the boundary-state public values used by the inner aggregation layers.

The root relation is also what a native host verifier must check for validity of a final STARK aggregation proof.

5.4.1. Root public value layout. The root relation is written as

$$\mathcal{R}_{\text{root}}((\text{PV}_{\text{user}}, C_{\text{exe}}, C_{\text{vm}}); \pi).$$

It holds when π is a valid internal-recursive proof whose public values claim successful App VM termination, the claimed user public values PV_{user} are contained in the final App memory root, and the root commitments C_{exe} and C_{vm} are computed from the child public values.

For the single child proof π , write the inner public-value symbols with superscript ch . For example, $R_{\text{final}}^{\text{ch}}$ is the final **app** memory root exposed by the child proof, and $(C_{\text{app}}^{\text{vk}})^{\text{ch}}$ is the **app** verifying key commit carried as a child public value. Let H_{exe} and H_{vm} denote the hash functions used to compute C_{exe} and C_{vm} .

5.4.2. Child proof checks. The root verifies π as an internal-recursive proof and checks that the child public values claim successful App VM termination:

$$\text{exit}^{\text{ch}} = 0, \quad b_{\text{term}}^{\text{ch}} = 1.$$

It also checks that the child is internal-recursive:

$$f_{\text{int}}^{\text{ch}} = 2, \quad f_{\text{rec}}^{\text{ch}} \in \{1, 2\}.$$

The root circuit hardcodes the expected **VkCommit** for the internal-recursive layer:

$$C_{\text{irec}}^{\text{root}} = (C_{\text{irec,cached}}^{\text{root}}, h_{\text{irec,pre}}^{\text{root}}).$$

If $f_{\text{rec}}^{\text{ch}} = 1$, then the child is the first internal-recursive proof and

$$(C_{\text{irec}}^{\text{vk}})^{\text{ch}} = 0.$$

If $f_{\text{rec}}^{\text{ch}} = 2$, then the internal-recursive verifying key commit has already been set. The cached commitment relation is

$$\text{VkCheck}((C_{\text{irec,cached}}^{\text{vk}})^{\text{ch}}, \pi).$$

The root does not re-expose the internal-recursive verifying key commit. Instead, it equates both child public value components to the hardcoded expected **VkCommit**:

$$\begin{aligned} (C_{\text{irec,cached}}^{\text{vk}})^{\text{ch}} &= C_{\text{irec,cached}}^{\text{root}}, \\ (h_{\text{irec,pre}}^{\text{vk}})^{\text{ch}} &= h_{\text{irec,pre}}^{\text{root}}. \end{aligned}$$

5.4.3. *User public values and commitments.* The root circuit exposes the user public values stored by the guest program. Let

$$R_{\text{user}} = \text{MerkleRoot}(\text{PV}_{\text{user}})$$

be the Merkle root of the user public value vector. The root circuit checks a Merkle proof that R_{user} is contained in the public-values address space of the final **app** memory root:

$$\text{MerkleVerify}_{\text{user}}(R_{\text{user}}, R_{\text{final}}^{\text{ch}}) = 1.$$

Thus the user public outputs specified in the App VM program do not become public values of a proof until the root proof is generated.

The root also exposes C_{exe} and C_{vm} . It checks

$$\begin{aligned} C_{\text{exe}} &= H_{\text{exe}}(C_{\text{prog}}^{\text{ch}}, R_{\text{init}}^{\text{ch}}, \text{pc}_{\text{init}}^{\text{ch}}), \\ C_{\text{vm}} &= H_{\text{vm}}((C_{\text{app}}^{\text{vk}})^{\text{ch}}, (C_{\text{leaf}}^{\text{vk}})^{\text{ch}}, (C_{\text{if}}^{\text{vk}})^{\text{ch}}). \end{aligned}$$

5.5. **Deferral integration.** The proof architecture also supports *deferrals*: the App VM may call standalone circuits C_i that are not themselves VM circuits. Each C_i is modeled as a function

$$f_i : \text{Input}_i \rightarrow \text{Output}_i$$

with hash functions $g_i : \text{Input}_i \rightarrow \text{Digest}$ and $h_i : \text{Output}_i \rightarrow \text{Digest}$; for a raw input x and raw output y , we call $g_i(x)$ the *input commit* and $h_i(y)$ the *output commit*.

During guest execution, the VM records the input and output commits of each deferred call in accumulators stored in a dedicated deferral address space. These accumulators are hash onions, updated by incremental hashing as calls are made. Below, H_{cmp} denotes a two-to-one compression function on digests and H_{sp} a variable-length sponge hash, both instantiated with Poseidon2 over the native field. A digest occupies **DIGEST_SIZE** memory cells.

5.5.1. *Deferral extension.* Deferrals are exposed to guest programs as a VM extension, configured with the deferred functions f_i and, for each C_i , a *deferral circuit commit* c_i^{circ} (defined below) committing to the verifying keys of its aggregation path. The deferral index must satisfy $i < \text{MAX_DEF_CIRCUITS}$, currently 512.

For each configured C_i , the VM stores an input accumulator A_i^{in} and an output accumulator A_i^{out} . They are stored in order at the beginning of the deferral address space (address space 4, cf. §3), with the pair for C_i starting at $[2i \cdot \text{DIGEST_SIZE}]_4$. The accumulators are initialized in memory as

$$A_i^{\text{in}} = c_i^{\text{circ}}, \quad A_i^{\text{out}} = 0,$$

and each deferred call to C_i with commits $(c^{\text{in}}, c^{\text{out}})$ updates

$$A_i^{\text{in}} \leftarrow H_{\text{cmp}}(A_i^{\text{in}}, c^{\text{in}}), \quad A_i^{\text{out}} \leftarrow H_{\text{cmp}}(A_i^{\text{out}}, c^{\text{out}}).$$

Seeding A_i^{in} with c_i^{circ} places a commitment to the deferral verifying keys in the initial memory state, which the root layer uses to constrain that the correct deferral circuits were used.

The extension introduces two opcodes. **CALL** reads an input commit c^{in} from memory, computes $y = f_i(x)$ and $c^{\text{out}} = h_i(y)$ on the host, applies the accumulator updates above, and writes the *output key* $(c^{\text{out}}, \text{len}(y))$ to memory, where the byte length $\text{len}(y)$ is stored as an 8-byte little-endian integer currently constrained to 32 bits. **OUTPUT** reads an output key from memory and writes the corresponding raw output y , constraining that the written bytes hash to c^{out} . The raw input x is never materialized in the VM and is entirely unconstrained there; its validity must be constrained by the deferral circuit C_i . The opcodes guarantee the following:

- Reading an invalid input or output commit results in an invalid proof.
- **CALL** binds only the output commit into the accumulators; the returned length is an unauthenticated prover hint until an **OUTPUT** reads the full output key and authenticates it against the output bytes.
- **OUTPUT** does not check that its key was returned by a prior **CALL**, and does not ensure that the write region is allocated; manually constructed output keys are not trusted deferral results.
- Only outputs whose byte length is divisible by **DIGEST_SIZE** are supported.

Since all deferred calls share this opcode pair, in practice h_i is the Poseidon2 sponge hash of Output_i serialized as bytes.

5.5.2. Deferral circuits. Each deferral circuit C_i must constrain that $f_i(x) = y$ and expose an input commit bound to $g_i(x)$, the output commit $h_i(y)$, and its deferral index, constrained to the constant i , as public values. The commit recorded by the VM must additionally bind any cached trace commitments of the C_i proof: the leaf aggregation layer below absorbs these into the exposed input commit via the sponge hash, producing a *folded input commit* $\tilde{c}^{\text{in}}$, which is $H_{\text{cmp}}(g_i(x), 0)$ when there are no cached traces. The hash g_i must be collision-resistant; otherwise an adversary could substitute a different input during aggregation while the VM accepts the original commit.

5.5.3. Per-circuit deferral aggregation and the deferral hook. The VM-side deferral opcodes record commits but do not prove the deferred computation inline. Instead, the proofs for each C_i are aggregated through a separate deferral tree with the same leaf, internal-for-leaf, and internal-recursive shape, using a deferral aggregation subcircuit that verifies each child proof and Merklizes the input and output commits. Because this aggregation builds a binary Merkle tree, each node combines at most two children, rather than using the arities N_{leaf} and N_{internal} of the VM path. A verified C_i proof with folded input commit $\tilde{c}^{\text{in}}$ and output commit c^{out} contributes a Merkle leaf, and internal layers combine child commitments M_L, M_R , as

$$M = H_{\text{cmp}}(H_{\text{cmp}}(\tau_{\text{leaf}}, \tilde{c}^{\text{in}}), c^{\text{out}}), \quad M = H_{\text{cmp}}(H_{\text{cmp}}(\tau_{\text{int}}, M_L), M_R),$$

where the domain separation tags $\tau_{\text{leaf}} \neq \tau_{\text{int}}$ prevent a leaf hash from being reinterpreted as an internal hash. Each layer exposes the Merkle commitment M , the number of aggregated proofs, the Merkle depth, and the deferral index as public values, and propagates the VkCommits of the deferral circuit, the deferral leaf and internal-for-leaf circuits, and the internal-recursive circuit as in §5.3.1. As in the VM tree, all C_i share the same internal-recursive circuit, so its VkCommit is not folded into the deferral circuit commit c_i^{circ} below but is instead checked against a hardcoded constant in the deferral hook. All children in a subtree must have the same deferral index, and the depth is range-checked so that neither the proof count nor the depth can wrap in the field. Tail aggregates are combined with padding: a missing child subtree is represented by the canonical zero hash for its depth, and the parent's depth is constrained against the selected zero hash.

The last internal-recursive proof for each deferral circuit is passed to a *deferral hook* layer. The hook verifies the deferral proof and converts the Merkle commitment into the hash-onion form used by the VM: given the underlying commit sequence as witness, it recomputes both the Merkle commitment and the onion hashes O^{in} and O^{out} , the folds of H_{cmp} over the folded input commits starting from c_i^{circ} and over the output commits starting from 0 – that is, the expected final values of A_i^{in} and A_i^{out} . Here the deferral circuit commit c_i^{circ} is computed as the

sponge hash under H_{sp} of both **VkCommit** components of the deferral circuit and the deferral leaf and internal-for-leaf circuits, as propagated through the tree. The hook exposes

$$\text{DeferralPVs} = (A_{\text{init}}, A_{\text{final}}, d, \text{idx}),$$

where

$$A_{\text{init}} = H_{\text{cmp}}(H_{\text{cmp}}(c_i^{\text{circ}}, 0), H_{\text{cmp}}(0, 0)), \quad A_{\text{final}} = H_{\text{cmp}}(H_{\text{cmp}}(O^{\text{in}}, 0), H_{\text{cmp}}(O^{\text{out}}, 0)),$$

$d = 1$, and $\text{idx} = i$. These are roots of subtrees of the memory Merkle trie of §5.2 covering the accumulator pair of C_i : each accumulator digest is zero-padded to the width of a trie leaf (the inner hashes with 0) and the two leaves are Merklized together, with A_{init} reflecting the required initial accumulator values and A_{final} the claimed final ones. The value d is the depth of the accumulator subtree aggregated so far, and idx is its node index at that depth.

5.5.4. Combined VM-deferral aggregation. Deferral hook proofs enter the combined VM-deferral tree at the app layer: they are base proofs consumed by the leaf aggregation layer, alongside App segment proofs. When deferrals are enabled, the inner aggregation public values PVs_{out} from §5.3.1 are modified to additionally include

$$(f_{\text{def}}^{\text{out}}, C_{\text{def}}^{\text{out}}) \text{ and } \text{DeferralPVs}^{\text{out}},$$

where $f_{\text{def}}^{\text{out}}$ is a flag indicating which parts of the public value layout are meaningfully set. When deferrals are enabled, every aggregation proof carries both the VM public value slots from §5.3.1 and the additional deferral public value slots above. The unused side is left unset until VM and deferral public values are combined. The value $C_{\text{def}}^{\text{out}}$ is the deferral hook commit: it is calculated analogously to C_{vm} with the hash H_{vm} but using the verifying key commits from the deferral-only path, computed at the first internal-recursive layer aggregating deferral-only children and propagated unchanged thereafter.

The additional aggregation relations are:

$$\begin{aligned} f_{\text{def}}^{\text{out}} &= 0 && \text{for an aggregate of VM-only children,} \\ f_{\text{def}}^{\text{out}} &= 1 && \text{for an aggregate of deferral-only children,} \\ f_{\text{def}}^{\text{out}} &= 2 && \text{once VM and deferral public values have been combined.} \end{aligned}$$

VM-only and deferral-only proofs are aggregated separately at the early layers. In particular, the VM-only leaf and internal-for-leaf circuits aggregate App VM public values and leave the deferral slots unset, while the deferral-only leaf and internal-for-leaf circuits aggregate deferral hook public values and leave the VM slots unset. These circuits have the same position in the aggregation shape, but need not be identical circuits. From the internal-recursive layer onward, both paths use the same internal-recursive aggregation circuit.

As in the per-circuit deferral tree, deferral public values are aggregated in binary: a node combines at most two deferral-carrying children, even where the aggregation circuit verifies up to N_{internal} children. When an aggregation layer combines two children carrying deferral public values, it Merklizes their accumulator roots,

$$A_{\text{init}}^{\text{out}} = H_{\text{cmp}}(A_{\text{init}}^{(0)}, A_{\text{init}}^{(1)}), \quad A_{\text{final}}^{\text{out}} = H_{\text{cmp}}(A_{\text{final}}^{(0)}, A_{\text{final}}^{(1)}),$$

constrains the children to equal depth, and sets $d^{\text{out}} = d^{(0)} + 1$. The node index stays canonical: hook leaves enter with idx equal to their deferral index, siblings must satisfy $\text{idx}^{(1)} = \text{idx}^{(0)} + 1$ with $\text{idx}^{(0)}$ even, and the parent carries $\text{idx}^{\text{out}} = \text{idx}^{(0)}/2$, so the aggregated deferral proofs assemble into exactly the accumulator Merkle tree laid out in the deferral address space.

Configured deferral circuits that are never called still contribute leaves, constrained to satisfy $A_{\text{init}} = A_{\text{final}}$, $d = 1$, and $\text{idx} = i$, i.e., that their accumulators were left untouched; if no configured circuit is used at all, the final aggregation proof instead carries $f_{\text{def}}^{\text{out}} = 0$.

5.5.5. Root-layer deferral checks. When deferrals are enabled, the root aggregation circuit checks that $f_{\text{def}}^{\text{ch}} \in \{0, 2\}$. If $f_{\text{def}}^{\text{ch}} = 0$, no configured deferral circuit was used, and the root checks that the deferral public value slots and $C_{\text{def}}^{\text{ch}}$ are unset and that the deferral address space is unchanged between the initial and final memory states. If $f_{\text{def}}^{\text{ch}} = 2$, the root checks that $C_{\text{def}}^{\text{ch}}$ equals a constant expected deferral hook commit and that $\text{idx}^{\text{ch}} = 0$. It also checks that the initial and final deferral accumulator hashes are present in the corresponding **app** memory states:

$$\text{MerkleVerify}_{\text{def,init}}(A_{\text{init}}^{\text{ch}}, R_{\text{init}}^{\text{ch}}) = 1,$$

$$\text{MerkleVerify}_{\text{def,final}}(A_{\text{final}}^{\text{ch}}, R_{\text{final}}^{\text{ch}}) = 1.$$

These constraints bind the unconstrained VM-side deferral calls to the independently proven deferral computations; since the initial accumulators are seeded with the commits c_i^{circ} , they also constrain that the correct deferral circuits were used. If no deferral circuits are configured, this part of the aggregation pipeline is disabled and the continuation pipeline behaves as described above.

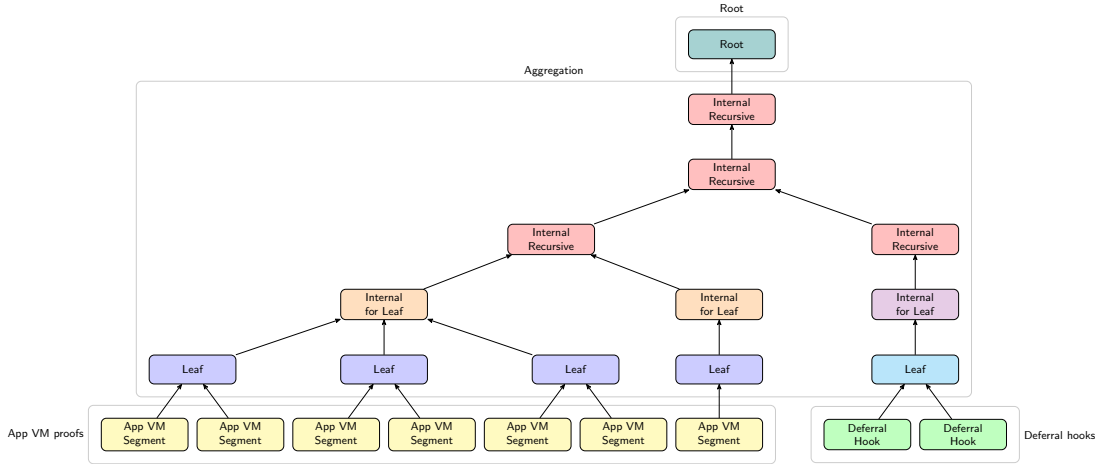


FIGURE 2. Deferral integration topology. Deferral hook proofs enter the combined VM-deferral tree at the app layer, alongside App VM segment proofs. The VM and deferral paths may use different leaf and internal-for-leaf circuits, shown with different colors, and then converge at the shared internal-recursive circuit.

5.5.6. Guest STARK verification. The deferral framework is used for guest STARK verification. In that case the deferred input is a proof; the deferred output is the tuple consisting of the App execution commitment, the App VM commitment, and the user public values, returned to the guest as bytes; and each guest verifying key corresponds to a different deferral circuit – a recursion circuit that verifies the proof, constrains the output tuple against the proof’s

claimed public values, and exposes the compressed final Fiat–Shamir transcript state as its input commit. Any change to the proof produces a different transcript, so committing to the final transcript state is equivalent to committing to the proof itself.

5.6. On-chain verification. The root proof is a single proof of successful App VM execution, but it is still produced by the STARK backend. Its size and verification cost are too large for data- and compute-constrained environments such as blockchains. Therefore the EVM proof path adds a static layer that verifies the root proof inside a proof system with fixed proof size and fixed verification cost.

At the time of writing, we generate a static verifier circuit that constrains proof verification for the fixed circuit associated to the root verifier. The static verifier circuit is proven² using the Halo2 [EP25] proof system. The resulting SNARK proof proves successful App VM execution and exposes C_{exe} , C_{vm} , and all user-defined public outputs.

Finally, we generate an EVM smart contract that verifies proofs of the static verifier circuit. The smart contract depends on the aggregation circuits and the number of user public outputs. It does not depend on the App VM program and does not need to know the intermediate segment boundary states.

5.7. Concrete aggregation topology. The current SDK constants set

$$N_{\text{leaf}} = 4, \quad N_{\text{internal}} = 3.$$

Thus each leaf verifier aggregates up to four adjacent App VM segment proofs, and each internal verifier aggregates up to three child proofs. The same internal arity is used for the internal-for-leaf layer and all internal-recursive layers.

For an execution split into m App VM segments, the current topology produces

$$L_1 = \left\lceil \frac{m}{4} \right\rceil$$

leaf proofs. These are aggregated into

$$L_2 = \left\lceil \frac{L_1}{3} \right\rceil$$

internal-for-leaf proofs, and then internal-recursive aggregation continues with

$$L_{j+1} = \left\lceil \frac{L_j}{3} \right\rceil$$

until a single internal-recursive proof remains. In some cases, the prover may recursively wrap this single remaining internal-recursive proof in one or more additional internal-recursive layers in order to reduce its size. This ensures that the final internal-recursive proof has a bounded size, which is necessary to make the root aggregation circuit have fixed size trace matrices. For the EVM proof path, the resulting proof is then wrapped by the root layer and re-encoded by the static layer. The SDK treats the leaf and internal arities above as both the default and maximum arities, because the maximum number of children is a const generic in the recursion circuits. Using fewer children in a tail aggregate is supported, but choosing a circuit maximum larger than the number of children actually used can reduce proving efficiency.

²To further reduce the proof size, we verify the Halo2 proof inside another Halo2 circuit.

6. RISC-V SUPPORT AND RUST TOOLCHAIN

We enable developers to write guest programs targeting the OpenVM ISA in a familiar environment by leveraging the Rust programming language’s use of LLVM as a compiler backend. We specifically use LLVM’s support for 32-bit RISC-V as a compilation target, together with Rust macro support for injecting inline RISC-V assembly into the LLVM assembler.

6.1. Transpilation of RISC-V ELF to OpenVM executable. We take advantage of the RISC-V design’s inherent support for custom ISA extensions to make it easy for OpenVM VM extensions to integrate with the RISC-V ISA. We do this by introducing a transpiler framework that converts a RISC-V Executable and Linkable Format (ELF) file into an *OpenVM executable*, which comprises the following components:

- Program ROM
- Initial program counter
- Initial data memory

This executable specifies the initial guest VM state, as discussed in §3.2.

We define a *RISC-V machine code block* as a contiguous, 32-bit aligned bit sequence in RISC-V program memory, whose length is a multiple of 32 bits. Such a block may include instructions from standard and non-standard RISC-V ISA extensions or arbitrary bit sequences.

The transpiler can be instantiated with any supported set of VM extensions. Each VM extension that wishes to integrate with RISC-V defines a set of RISC-V machine code blocks alongside rules mapping each block to sequences of potentially multiple OpenVM instructions. Custom RISC-V machine code for VM extensions is classified into two distinct categories:

Intrinsic Instruction: A single custom instruction conforming to the RISC-V ISA specification.

Kernel Code: A 32-bit aligned arbitrary binary sequence whose length is a multiple of 32 bits. Kernel code need not comply with any RISC-V specification and is used primarily to embed foreign OpenVM assembly statically within ELF binaries, bypassing custom linking processes. Kernel code execution on standard RISC-V hardware requires specialized transpiler support.

We envision these two categories of code to be used as follows: VM extensions designed for compatibility with the RISC-V ISA, and in particular its register and memory architecture, can define custom RISC-V instructions for a tighter and more direct integration. Extensions that require specialized use of the OpenVM ISA, often for performance reasons, which are not compatible with the RISC-V architecture, may generate kernel code through external means (e.g., via a separate compilation toolchain) and statically link the kernel code into the ELF binary. Our design is inspired by existing GPU toolchains, where GPU kernels are written and compiled for a distinct machine architecture and then linked into the host application binary.

6.2. RV32IM extension. We provide zkVM support for the RV32I base and RV32M multiplication instruction sets by defining a VM extension for RV32IM in OpenVM. This is a set of instructions in the OpenVM ISA designed to support transpilation from the RV32IM instruction set. The combination of the transpiler and the VM extension provides a zkVM that fully supports execution of RISC-V ELF binaries using the RV32IM instruction set. The transpiler and zkVM constrain program execution to fully conform to the RISC-V ISA specification.

6.2.1. System call support. While the overall transpiler and ISA framework does support the RISC-V `ecall` instruction, we do not use `ecall` in any existing VM extensions. This is an intentional design choice: the `ecall` instruction allows the guest program to delegate execution logic to the host operating system, which is intended for environments where the ISA cannot fully specify or control the implementation. However, in the zkVM setting where the instruction set and underlying circuit implementation can be co-designed, it is more modular and efficient to extend the instruction set and zkVM circuit directly.

As an example, guest program inputs and outputs are supported via the RV32Io VM extension, where custom intrinsic instructions move data from the host hint stream to guest memory and write public outputs.

6.3. Rust toolchain. The Rust compiler supports 32-bit RISC-V as a target architecture. A Rust `no_std` program written without system calls will compile to an ELF binary that can be transpiled into an OpenVM executable. This executable is supported by any zkVM with the RV32IM extension.

Additional VM extensions that specify RISC-V transpilation rules can call their instructions from Rust by making use of the Rust inline assembly `asm!` macro. The macro allows the programmer to provide custom RISC-V assembly to the LLVM assembler to be injected into the ELF. Using this feature, VM extension developers can write *guest libraries* which provide Rust function bindings for their instructions.

6.4. Rust std library support. Rust programs written with the Rust standard library are supported with some limitations. The standard library provides functionalities which are traditionally implemented via system calls to the operating system. We externally link these functions to our implementations which use custom intrinsic functions when applicable.

REFERENCES

- [BBHR18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Paper 2018/046, 2018.
- [BEG⁺94] Manuel Blum, W. Evans, Peter Gemmell, Sampath Kannan, and M. Naor. Checking the correctness of memories. *Algorithmica*, 12:225–244, 09 1994.
- [Con23] ConsenSys. gnark. <https://github.com/ConsenSys/gnark>, 2023. Version 0.12.0.
- [EP25] Electric Coin Co. and Privacy Scaling Explorations. halo2. <https://crates.io/crates/halo2-axiom>, 2025.
- [GPR21] Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo – a turing-complete STARK-friendly CPU architecture. Cryptology ePrint Archive, Paper 2021/1063, 2021.
- [Hab22] Ulrich Haböck. Multivariate lookups based on logarithmic derivatives. Cryptology ePrint Archive, Paper 2022/1530, 2022.
- [ide25] iden3. circom. <https://github.com/iden3/circom>, 2025.
- [PH23] Shahar Papini and Ulrich Haböck. Improving logarithmic derivative lookups using GKR. Cryptology ePrint Archive, Paper 2023/1284, 2023.
- [Pol25] Polygon Zero. Plonky3. <https://github.com/Plonky3/Plonky3>, 2025.
- [RIS25] RISC Zero Team. RISC Zero. <https://github.com/risc0/risc0>, 2025.
- [Sta21] StarkWare. ethSTARK documentation. Cryptology ePrint Archive, Paper 2021/582, 2021.
- [Suc25] Succinct Labs. SP1. <https://github.com/succinctlabs/sp1>, 2025.
- [Val24] Valida Team. Valida. <https://github.com/valida-xyz/valida>, 2024.