Verifiable Computation with Distributed Proving and Verification

Ritual

April 20, 2026

Abstract

Replicated state machine execution fails for the workload class that motivates this paper. Correctness requires proving that a given output is consistent with the computation, not comparing outputs across replicas. This is the domain of verifiable computation [7, 16]. Delegated execution requires that the network can verify a result without re-executing the computation. Distributed proof generation exploits the algebraic structure of computational objects (compositional hierarchy, parameter symmetry, repeated sub-computation), while distributed verification assigns proof shards to validator committees via deterministic sampling. The prover-friendly design point — optimizing proving latency rather than proof size, since proofs are verified by committees off-chain rather than enshrined on-chain — inverts the standard ZKP tradeoff and shrinks the verification window between hardware attestation and cryptographic finality.

1 Introduction

Symphony [20] is an execution-aware SMR framework organized around a single principle: *protocol-enforced guarantees*.

1.1 The Execution-Cost Inversion

When SMR was the bottleneck, execution was an afterthought: all replicas could afford to execute all transactions because transactions were lightweight (token transfers, storage updates, simple contract calls), and the cost of reaching agreement dominated. For the workload class that motivates Symphony, this relationship inverts. Neural network inference with billions of parameters requires tens to hundreds of gigabytes of GPU memory and seconds to minutes of compute per forward pass; Monte Carlo simulations, probabilistic optimization, and other resource-intensive computations have similar profiles. Execution latency exceeds SMR latency by three to six orders of magnitude. Replicated execution of these workloads is not merely expensive: it requires every validator to possess specialized hardware, undermining the decentralization properties that replicated execution was designed to preserve.

1.2 Randomness as a Structural Barrier

The cost argument alone would motivate optimization but not architectural change. The structural barrier is randomness, which enters through two independent channels.

Hardware non-reproducibility. Replicated state machine execution [13] assumes that given the same input and pre-state, all correct replicas compute the same post-state. For GPU-accelerated workloads, however, this assumption fails at the hardware level. Floating-point accumulation in parallel GPU threads is non-associative: the order in which partial sums are reduced depends on thread scheduling, which varies across architectures and even across runs on the same device [9]. Two honest replicas executing the same neural network forward pass

with the same weights, the same input, and the same random seed on different GPU hardware will produce bit-different outputs. This is not a software bug or a randomness problem; it is a consequence of IEEE 754 arithmetic under parallel reduction. Deterministic GPU modes exist but impose severe performance penalties (10–30× slowdown) and are not universally available, defeating the purpose of hardware acceleration. No amount of seed-sharing or protocol-level coordination eliminates hardware-level divergence across heterogeneous validators.

Algorithmic randomness. Independently of hardware, many target workloads are intentionally randomized. Neural network inference with temperature sampling ($T > 0$), Monte Carlo simulation, and probabilistic optimization all map inputs to *distributions* over outputs. When replicas sample independently, they produce different outputs not because any replica is faulty but because the function is inherently randomized. A standard workaround—deriving shared randomness from a beacon or VRF seed—makes the computation deterministic given the seed, but re-exposes it to hardware non-reproducibility and makes the output predictable from the public seed.

Consequence. Either source of randomness, alone, breaks replication. Together they make it untenable for the target workload class. Correctness requires proving that a given output is consistent with the computation, not comparing outputs across replicas. This is the domain of verifiable computation [7, 16]. Section 2 formalizes the distinction and defines the randomness models Symphony supports.

Workloads that are randomized (Definition 2.1) or that exceed the per-block computational budget are executed by a single staked executor [18]. The remaining replicas verify proofs of the execution rather than re-executing.

This paper makes the following contributions:

- Proof generation exploits the algebraic structure of computational objects (compositional hierarchy, parameter symmetry, repeated sub-computation) for workload-level decomposition (Section 3.2).
- Verification is distributed across validator committees via deterministic sampling [8] (Section 3.3).
- Because proofs are not aggregated into a single on-chain artifact, the system avoids the most expensive step in typical ZKP pipelines: recursive compression (Section 3.3.5).
- We derive committee safety bounds via hypergeometric analysis (Section 4.1.2).

Section 3 presents verifiable computation with structure-exploiting distributed proving and committee-based verification. Section 4 establishes safety and liveness. Section 5 discusses limitations and future work. Section 7 concludes.

2 Randomized Computation and the Limits of Replication

A *workload* is a triple $W = (\Theta, F, D)$ where:

- $\Theta \subseteq \mathbb{R}^d$ is a parameter space (finite-dimensional, possibly very large),
- $\mathcal{X}$ is the input space and $\mathcal{Y}$ is the output space,
- $F : \Theta \times \mathcal{X} \to \mathcal{D}(\mathcal{Y})$ maps parameters and input to a probability distribution over outputs,
- $D = (c_{\text{compute}}, c_{\text{memory}}, c_{\text{time}})$ is a resource descriptor.

Definition 2.1 (Algorithmically Randomized Workload)**.** *A workload W is algorithmically randomized if there exist $\theta \in \Theta$ and $x \in \mathcal{X}$ such that $F_\theta(x)$ has positive Shannon entropy: $H(F_\theta(x)) > 0$.*

Randomness model. Symphony's delegated execution path supports three randomness models, declared when registering a delegated computation [18]:

1. *Sealed-seed execution (primary).* The user (or the protocol) selects a random seed r and encrypts it to the executor's TEE attestation-derived public key: $e = \mathsf{Enc}(\mathsf{pk}_{\mathrm{TEE}}, r)$. The ciphertext e is committed on-chain when the delegated computation is registered. The TEE decrypts r inside the enclave, evaluates $y = F_\theta(x; r)$, and produces an attestation covering the tuple $(\mathsf{M}, \theta, x, e, y)$, where M is the workload identity (the TD measurement in Intel TDX, i.e., the hash of the trust domain's initial state). The executor—the machine operator outside the enclave—never observes r in plaintext. Grinding is prevented: the executor cannot enumerate seeds because the seed is encrypted to the TEE, and cannot selectively suppress outputs because the attestation is produced atomically with the computation. The workload identity M ensures the enclave binary is immutable post-attestation, so the executor cannot modify the program to leak r or branch on the output before reporting it.

   This model provides *distributional integrity under the TEE soundness assumption*: the output y is the result of faithfully executing $F_\theta(x; r)$ for a seed r that the executor could not observe, choose, or grind over. The soundness parameter s_1 of the hardware attestation [19] bounds the probability that this guarantee is violated. If the TEE is compromised (side-channel attack, firmware vulnerability), the attacker can decrypt r and grind; this failure mode is already captured by s_1.

2. *Executor-chosen randomness.* The executor selects r freely and proves $y = F_\theta(x; r)$ for the chosen r. The proof establishes *support membership*: y is in the range of $F_\theta(x)$. It does not establish distributional fairness—the executor can grind over r to bias the output. This model is appropriate when the application cares only that y is a feasible output, not that it was sampled fairly (e.g., selecting any valid completion of a prompt).

3. *Protocol-derived randomness.* The seed r is derived from a protocol-provided source (beacon, VRF) fixed before execution. The executor proves $y = F_\theta(x; r)$ for the specific r. This model provides deterministic replay and eliminates executor grinding, but the output is predictable given r and x.

Under model (1), the "valid sample" claim is backed by the TEE's execution integrity: the seed was user-chosen and executor-blind, and the program was measured and immutable. Under model (2), the claim is limited to support membership. Under model (3), it is deterministic correctness.

User-chosen vs. protocol-chosen seeds. In model (1), the seed originator matters. If the user provides r, they can choose it adversarially to manipulate their own output; this is acceptable when the output affects only the user's own state (e.g., a user requesting inference on their own prompt). When the output affects shared state (e.g., oracle prices, risk assessments consumed by other contracts), the seed should originate from a source the user does not control: a randomness beacon, a VRF, or a commit-reveal scheme where the seed is committed before the user's input is known. The choice of seed originator is declared at registration time alongside the randomness model.

3 Verifiable Computation with Distributed Proving and Verification

Delegated execution requires that the network can verify a result without re-executing the computation. This section describes how Symphony implements verifiable computation [7, 16] with two key extensions: (a) distributed proof generation that exploits the algebraic structure of the computational objects, and (b) distributed verification via committee sampling.

3.1 Verifiable Delegated Computation

A single staked executor evaluates $F_\theta(x)$, producing output y and proofs $\{\pi_1, \ldots, \pi_k\}$ for k registered non-interactive proof systems. This is standard verifiable computation [7] extended in two directions:

1. *Randomized computations.* For a randomized workload W (Definition 2.1), the proof establishes that y is consistent with $F_\theta(x)$ under the declared randomness model (Section 2). Under the primary sealed-seed model, the user encrypts a seed r to the TEE's attestation-derived key; the enclave decrypts r, evaluates $F_\theta(x; r)$, and attests the result alongside the workload identity M and the on-chain ciphertext e. The executor never observes r and cannot grind. Under executor-chosen randomness, the proof shows $y = F_\theta(x; r)$ for some executor-selected r, establishing support membership but not distributional fairness. Under protocol-derived randomness, the proof shows $y = F_\theta(x; r)$ for a beacon/VRF seed r committed on-chain, establishing deterministic correctness. The ZKP (when present) proves the computation trace including r; the TEE attestation covers the enclave environment, workload identity, and the ciphertext binding. Note that SNARK-friendly inference circuits operate over finite fields ($\mathbb{F}_p$), where arithmetic is exact and associative. Hardware non-reproducibility [18] does not arise in the ZKP verification path; the only source of non-determinism within the circuit is the randomness input r.

2. *Multiple proof systems simultaneously.* The executor produces proofs for multiple proof systems, each with different soundness-completeness-latency profiles [19]. The proofs are independent: each proof system's proof is generated and verified without reference to the others.

3.2 Structure-Exploiting Workload Decomposition

For large workloads, proof generation by a single machine is infeasible or impractically slow. Symphony distributes proof generation across multiple machines. The key insight is that the computational objects over which delegated STFs operate are not arbitrary circuits — they exhibit algebraic structure that makes decomposition natural and efficient.

3.2.1 Algebraic Structure of Computational Objects

The workloads relevant to Symphony, including neural network inference, probabilistic simulation, and optimization, share structural properties that generic circuit-level decomposition [17] does not exploit:

Compositional hierarchy. A neural network is a composition of layers: $F = f_L \circ f_{L-1} \circ \cdots \circ f_1$, where each $f_i : \mathbb{R}^{d_i} \to \mathbb{R}^{d_{i+1}}$ is a layer function (linear transformation followed by nonlinearity). Layers are natural partition boundaries: the interface between consecutive layers is an activation tensor of fixed, architecture-determined dimension (e.g., shape (s, d_{model}) for a transformer), not an arbitrary set of wire values.

Parameter symmetry. In transformer architectures [15], attention heads within a layer are structurally identical sub-computations operating on different parameter slices. If a layer has h attention heads, each computing $\text{Attention}(Q_i, K_i, V_i)$ for $i \in [h]$, the heads are interchangeable up to their parameters. This symmetry enables balanced partitioning: each head can be assigned to a different prover with guaranteed uniform workload.

Repeated sub-computation. Residual blocks in ResNets, recurrent cells in RNNs, and timesteps in diffusion models are iterated applications of the same or similar structure. Regularity enables uniform partition sizing and predictable proving costs.

These structures are well-studied in the parallel and distributed computing literature [6, 10]. Applying them to the distributed *proving* problem (rather than the distributed *execution* problem) yields partition boundaries aligned with the object's algebraic structure.

3.2.2 Formal Decomposition

Graph partition model. Represent the computation as a directed acyclic graph $G = (V, E)$, where vertices V are primitive operations and edges E carry tensors between them. A *partition* of G into k shards is a division of V into disjoint subsets $V_1, \ldots, V_k$. Each shard V_i induces a sub-computation SG_i (the subgraph restricted to V_i).

Definition 3.1 (Boundary and Boundary Width)**.** *The boundary (cut-set) between shards V_i and V_j is $E^{i,j}_{cut} = \{(u, v) \in E \mid u \in V_i,\ v \in V_j\}$. The boundary data $T^{i,j}_{boundary}$ is the set of tensors on these edges. The boundary width is:*

$$W_B^{i,j} = \sum_{t \in T^{i,j}_{boundary}} size(t)$$

measured in scalar field elements. The total boundary width of a partition is $W_B = \sum_{i<j} W_B^{i,j}$.

A forward pass through G produces intermediate tensors $T_0, T_1, \ldots, T_k$, where T_0 is the input and $T_i = \mathsf{SG}_i(T_{i-1})$ for a sequential partition, or more generally, SG_i takes as input the tensors on all incoming cut edges.

Shard proofs and linking. The parallel proving phase generates a set of proofs $\{\pi_i\}_{i=1}^k$, where each π_i attests to the statement "T_i is the correct output of computing SG_i on input T_{i-1}." The choice of proof system per shard is independent: different shards may use different proof constructions.

Three security properties must hold for the composite proof to be sound:

1. *Shard validity*: each π_i is a valid proof for its sub-computation.
2. *Completeness*: the union of all shards spans the full computation graph. No shard is missing.
3. *Boundary consistency (linking)*: the public output committed to in π_i is identical to the public input committed to in π_{i+1}, for all adjacent shards.

Linking cost. The cost of enforcing boundary consistency is a direct function of the boundary width W_B. Whether linking is implemented via recursive proof composition [3], accumulation schemes, or permutation arguments enforcing copy constraints across shard boundaries, the cost scales linearly with W_B: $C_{\text{link}} = O(W_B)$. For recursive formulations, the prover cost of π_{i+1} includes the cost of verifying π_i, which itself scales with the number of public inputs at the boundary.

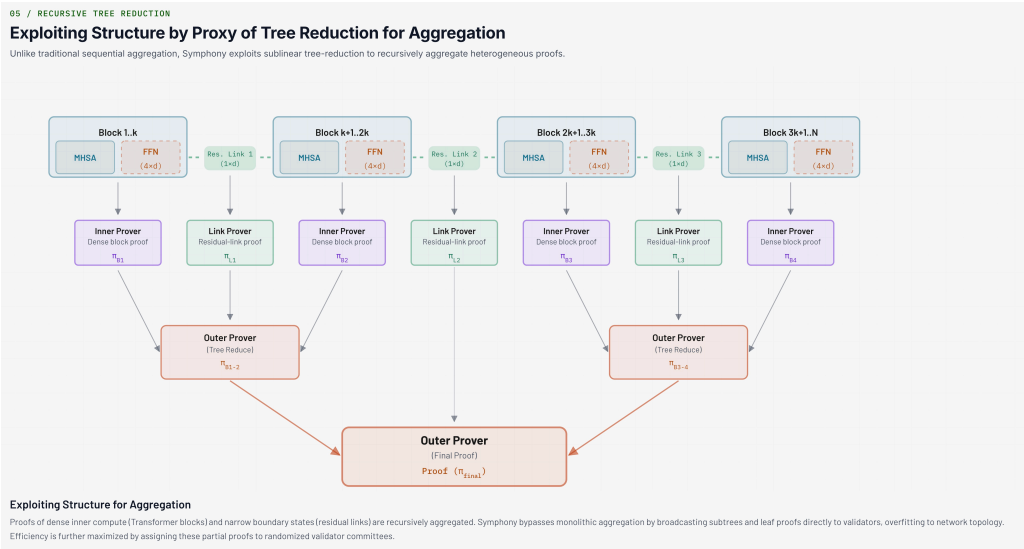

Figure 1: Structure-exploiting tree reduction for proof aggregation. Inner provers generate dense block proofs and residual-link proofs independently. Outer provers recursively aggregate subtrees, overfitting to network topology. Symphony bypasses monolithic aggregation by broadcasting subtrees and leaf proofs directly to randomized validator committees.

Residual connections. In architectures with skip connections (ResNets, Transformers with residual streams), shard SG_{i+1} may depend not only on the output of SG_i but also on the output of some earlier shard SG_{i-j}. The boundary data must include these non-contiguous dependencies, and the linking proof must verify consistency across all incoming edges, not just the immediately preceding shard. This means the partition DAG has skip edges — edges connecting non-adjacent shards, arising from residual connections in the underlying architecture. The boundary width W_B accounts for them: the boundary data at shard V_{i+1} includes tensors from both SG_i and SG_{i-j}.

Fat vs. narrow boundaries. Not all partition points are equal. A partition placed at a point where the data flow is narrow (few scalar elements crossing the boundary) incurs low linking cost. A partition placed inside an expansion layer (where the internal dimension is much larger than the input/output dimension) incurs high linking cost.

Optimal partitioning is a multi-objective problem: balance the computational complexity within each shard (to equalize prover workload) while minimizing W_B (to reduce linking overhead). For architectures with alternating expansion and contraction (the common pattern in neural networks), the narrow points between components are the natural partition boundaries.

No aggregation. Because shard proofs are verified by committees off-chain and are not stored on-chain (Section 3.3.5), the k shard proofs do not need to be recursively compressed into a single succinct proof. Each π_i is broadcast independently to its assigned committee. The only additional proof obligation beyond the individual shard proofs is boundary linking. This is a direct consequence of the prover-friendly design: aggregation is an optimization for on-chain verification cost, which does not apply here.

We refer to the graph-partition-based proving strategy described above — partitioning at narrow boundaries, exploiting compositional hierarchy and architectural symmetry, with independent shard proofs and boundary linking — as *structure-exploiting decomposition*.

Relationship to IVC. Incrementally verifiable computation (IVC) [14] decomposes by time step ($F = g^T$ for T iterations of step function g). Structure-exploiting decomposition decom-

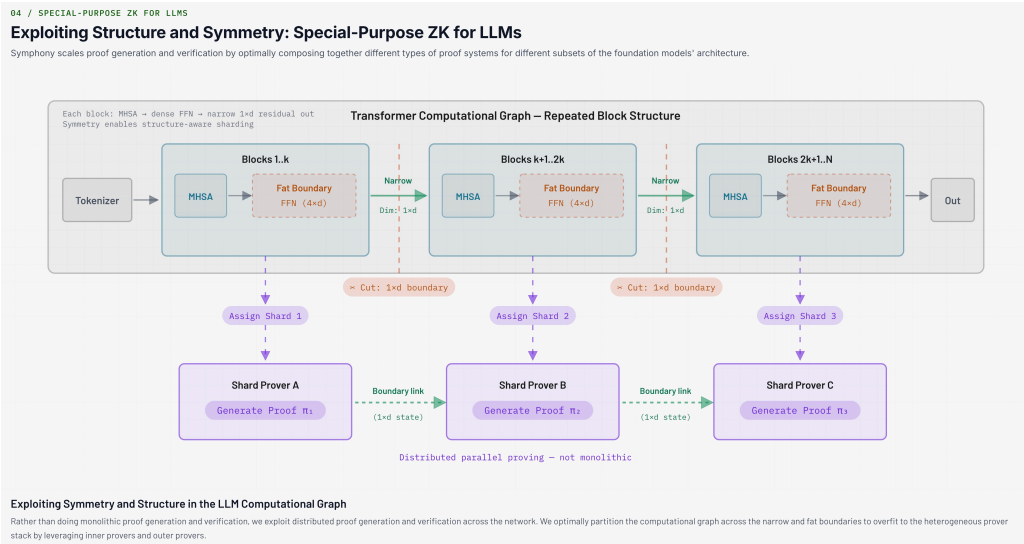

Figure 2: Structure-exploiting decomposition for transformer inference. Partition boundaries are placed at narrow inter-block interfaces (dimension d_{model}), avoiding fat boundaries inside FFN expansion layers (dimension $d_{\text{ffn}} \approx 4 \times d_{\text{model}}$). Each shard is proved independently; linking cost scales with boundary width.

poses by the structural units of the computational object (layers, heads, blocks), which may include both sequential components (depth) and parallel components (width). IVC requires a uniform step function; structure-exploiting decomposition accommodates heterogeneous sub-computations with varying resource profiles.

3.2.3 Reference Decompositions: Transformers

The partition design principles above apply concretely to autoregressive transformer models. The key dimensions are: d_{model} (hidden dimension, e.g., 4096), d_{ffn} (FFN intermediate dimension, typically $4 \times d_{\text{model}}$), and s (sequence length). Tensors flowing between transformer blocks have shape (s, d_{model}).

Narrow boundaries. The natural partition points are between transformer blocks, or between the multi-head self-attention (MHSA) sub-layer and the feed-forward network (FFN) sub-layer. At these points, the boundary tensor has dimension d_{model}, giving $W_B = s \cdot d_{\text{model}}$.

Fat boundaries. The FFN sub-layer projects from d_{model} to $d_{\text{ffn}} \approx 4 \cdot d_{\text{model}}$, applies a nonlinearity, then projects back to d_{model}. Partitioning *inside* the FFN expansion creates a boundary tensor of dimension d_{ffn}, increasing W_B by a factor of $\sim 4\times$ and correspondingly increasing the linking cost.

Partition strategy. Group consecutive transformer blocks into M partitions, cutting at the narrow inter-block boundaries. Within each partition, attention heads provide further parallelism (heads are independent up to the output projection). Residual connections within a partition are internal and do not affect W_B; residual connections *across* partition boundaries add skip-edge tensors to the boundary data.

Other architectures. Convolutional networks decompose by stage (convolution-pooling groups), with feature maps as boundary data. Diffusion models decompose by timestep group, with the

noisy image tensor at each boundary. Models exhibiting embarrassingly parallel structure (embedding lookups, random forests, mixture-of-experts routing) can be partitioned ahead of time without a forward pass. For arbitrary workloads, the developer provides a decomposition manifest specifying the sub-computation DAG, interface types, and partition assignment.

3.2.4 Verification Units

Each shard proof π_i (plus its boundary linking data) constitutes one *verification unit*. The total number of verification units for a computation is k (the number of shards). Each verification unit is assigned to a validator committee independently.

3.3 Distributed Verification via Committee Sampling

Full-network verification (every validator checks every proof) provides the strongest safety guarantee but is redundant in the sense that proof validity is a deterministic property. Committee-based verification introduces a different trust model: the SMR protocol no longer relies on every validator independently verifying every proof, but instead trusts that sampled committees with honest majorities will correctly report verification results. This reduces per-validator load but changes the safety guarantee from deterministic to statistical, parameterized by the committee compromise probability (Section 4.1.2).

3.3.1 Committee Assignment

Proof shards are assigned to validator subsets (committees) via deterministic sampling. The mechanism draws from two traditions:

- Algorand's cryptographic sortition [8]: committee members self-select using a VRF, preventing an adaptive adversary from targeting committee members before they reveal themselves.
- Ethereum's beacon chain committees [5]: deterministic assignment of validators to committees for attestation duties, where the assignment is public but the large validator set limits adaptive targeting.

Symphony's committee assignment uses a verifiable random function seeded by the epoch's randomness beacon: $(\text{epoch}, \text{shard_id}) \rightarrow \text{committee}$. **Note:** unlike Algorand's sortition, Symphony's deterministic assignment reveals committee membership before verification begins, creating an adaptive-corruption surface. If an adversary can corrupt or DoS specific validators after learning their committee assignment, the hypergeometric safety bounds (Section 4.1.2) do not apply. Mitigations include short epochs (limiting the corruption window) and requiring committees to be large enough that targeted corruption of a majority is impractical. A full treatment of adaptive adversaries is left to future work.

3.3.2 Specialization

Validators opt into verification duties by workload type and proof system type. ZKP verification does not require GPU hardware because proof verification is a CPU-bound operation involving field arithmetic and hash evaluations that runs on commodity hardware. Validators without any specialized hardware can verify both ZKPs and hardware attestations.

A general-purpose fallback ensures that every proof shard has a committee: validators that do not opt into any specialization are assigned to the fallback pool. Specialization enables verification key caching (a validator repeatedly verifying the same proof system type can cache verification parameters) and hardware-specific optimizations.

3.3.3 Quorum and Quorum-Skip

Each committee uses a simple majority quorum. Once a quorum of committee members has verified and attested to a proof shard, the shard is considered verified. The corresponding lattice dimension advances.

Validators that join the committee after quorum has been reached can *skip* verification of that shard—an optimization that reduces work for late-joining validators. This is analogous to Tendermint's fast-sync [4]: a syncing validator can accept the quorum's attestation rather than re-verifying from scratch. The security argument is identical: the quorum attestation is a certificate of the shard's validity, backed by the committee's stake. No safety is weakened; liveness improves for validators that are syncing or joining mid-epoch.

3.3.4 Security-Efficiency Tradeoff

The committee size k parameterizes the tradeoff between verification efficiency and safety. Smaller committees reduce the aggregate verification work (each proof shard is checked by k validators instead of n) but increase the committee compromise probability (Section 4.1.2).

The total verification work scales as:

$$\text{Total work} = M \times S \times k \times c_{\text{verify}}$$

where M is the number of workload partitions, S is the number of proof shards per partition, k is the committee size, and c_{verify} is the per-shard verification cost. For full-network verification, $k = n$; for committee-based verification, $k \ll n$, yielding a factor of n/k reduction in per-validator work on average.

The protocol selects k to achieve a target committee compromise probability (e.g., 10^{-9}) given the validator set size n and adversary bound f, using the hypergeometric bounds from Lemma 4.3.

3.3.5 Prover-Friendly Design Point

Committee-based off-chain verification changes which axis of the prover/verifier tradeoff is binding, relative to the on-chain verification model used by ZK rollups.

In ZK rollups, the proof is verified by an L1 smart contract. Every byte of proof costs gas; every verification cycle costs gas. This creates strong pressure to minimize proof size (constant-size SNARKs, recursive compression) and verification time (constant or polylog verification), even at the cost of slower proving.

Symphony's verification model inverts this. Proofs are gossiped across the peer-to-peer layer to committee members who verify off-chain. Proof size is bounded by network bandwidth (hundreds of megabytes per second in practice), not by on-chain gas costs. Proofs are not stored on-chain. Verification cost is distributed across committee members on commodity hardware, not concentrated in a single EVM transaction. The binding constraint is *prover latency*: the interval between hardware attestation (milliseconds) and ZKP verification (minutes to hours) is the verification window during which safety depends on the TEE and the economic model. Every reduction in proving time shrinks this window and strengthens the system's cryptographic safety posture.

Because proofs are not aggregated into a single on-chain artifact (Section 3.2.2), the system avoids the most expensive step in typical ZKP pipelines: recursive compression. Each shard proof is verified independently by its committee, so the prover never pays the cost of folding k proofs into one. This alone saves the majority of proving time. Additionally, because proof compactness is not a constraint (proofs are bounded by P2P bandwidth, not on-chain gas), the system is free to use proof constructions optimized for prover speed rather than succinctness. This includes GKR-based systems with $O(N)$ prover time (vs. $O(N \log N)$ for PLONK-style

systems), constructions that trade proof size for fewer prover rounds, and schemes whose parallelization properties align with the shard decomposition. Verification cost per shard (c_{verify}) remains CPU-bound field arithmetic on commodity hardware, distributed across committee members.

4 Safety and Liveness

4.1 Safety

Safety means that no correct replica accepts an invalid state transition.

4.1.1 Delegated Execution: Single Proof System

Lemma 4.1 (Single-System Delegated Safety under Full-Network Verification)**.** *For a delegated computation verified by a single non-interactive proof system V_i with soundness parameter s_i, where every correct validator independently runs* $\mathsf{Verify}(x, \pi_i)$*, the probability that an incorrect result is accepted is at most s_i.*

Proof. An incorrect result y^* is accepted only if Verify accepts an invalid proof. By the soundness definition [19], this occurs with probability $\leq s_i$. Since every correct validator independently verifies, no trust delegation is involved: the safety guarantee is purely the proof system's soundness parameter. □

For the proof systems relevant to Symphony, full-network verification is practical. TEE attestation verification is a signature check (milliseconds). SNARK verification is $O(1)$ in proof size. Neither requires specialized hardware. Every validator can and should verify every top-level proof.

Committee-based shard verification. When proof generation uses the shard decomposition of Section 3.2.2, a single computation may produce k shard proofs plus boundary-linking data. Full-network verification of every shard is redundant: each shard proof is valid or invalid regardless of how many nodes check it. For shard-level verification, committees reduce per-validator work at the cost of an additional failure mode:

$$\Pr[\text{shard incorrectly accepted}] \leq s_i + \Pr[\text{committee compromised}]$$

where the committee compromise probability is bounded by the hypergeometric analysis in Section 4.1.2. Validators not on a shard's committee accept the committee's attestation without independently verifying that shard. This is a trust-model change from full-network verification and applies only at the shard level, not at the top-level proof-system level.

4.1.2 Distributed Verification: Hypergeometric Committee Bounds

We derive the probability that a randomly sampled committee contains a Byzantine majority.

Setup. The validator set has n members, of which at most f are Byzantine. A committee of size k is sampled uniformly at random (without replacement) from the validator set. The committee uses a simple majority quorum: a decision is made when $\lceil k/2 \rceil + 1$ members agree.

The number of Byzantine members in a committee of size k follows a hypergeometric distribution:

$$X \sim \text{Hypergeometric}(N = n, K = f, n = k)$$

where N is the population size, K is the number of "defective" items (Byzantine validators), and n is the sample size.

Definition 4.2 (Committee Compromise)**.** *A committee is compromised if it contains a Byzantine majority: $X > k/2$.*

Lemma 4.3 (Committee Safety Bound)**.** *The probability that a committee of size k sampled from n validators with $f < n/3$ Byzantine members contains a Byzantine majority is:*

$$\Pr[X > k/2] = \sum_{x=\lfloor k/2 \rfloor+1}^{\min(k,f)} \frac{\binom{f}{x}\binom{n-f}{k-x}}{\binom{n}{k}}$$

Proof. Direct application of the hypergeometric probability mass function. The sum ranges over the possible numbers of Byzantine committee members that constitute a majority. □

Observations.

1. Committee safety improves with committee size k, at a rate that depends on the adversary fraction f/n. The qualitative trend (larger k yields lower compromise probability) is robust; the quantitative values depend on the recomputed table.

2. The hypergeometric distribution (sampling without replacement) gives tighter bounds than the binomial approximation (sampling with replacement), especially when k is a non-negligible fraction of n.

3. These bounds assume uniform staking ($w_i = W/n$ for all i). Non-uniform staking requires weighted sampling analysis, e.g., via Wallenius' noncentral hypergeometric distribution. The safety guarantee does not "degrade gracefully" in general: heavy-tailed stake distributions can create non-linear vulnerability to committee capture, and the uniform-staking bounds should not be extrapolated to non-uniform settings without explicit analysis.

4. These bounds assume a non-adaptive adversary: committee membership is determined before the adversary decides whom to corrupt. If committee assignments are public and the adversary is adaptive (can target specific committee members after learning the assignment), the hypergeometric bounds do not apply. Cryptographic sortition (as in Algorand [8]) mitigates adaptive corruption; Symphony's deterministic assignment does not provide this property (see Section 3.3.1).

Comparison to binomial approximation. The binomial approximation treats each committee member as an independent Bernoulli trial with Byzantine probability f/n. This overestimates the committee compromise probability (sampling without replacement reduces variance). For $k \ll n$, the difference is small. For $k/n > 0.1$, the hypergeometric distribution provides meaningfully tighter bounds.

Per-epoch stability. Committee assignments are fixed for an epoch (a configurable number of blocks). Within an epoch, the same committee verifies all proofs for its assigned shard. This amortizes the cost of committee selection and enables caching of verification keys. Between epochs, committees are re-sampled.

Adaptive corruption. The hypergeometric bounds assume a static adversary: the set of Byzantine validators is fixed before committee assignment. An adaptive adversary who can corrupt validators after learning their committee assignment circumvents these bounds entirely. Symphony's deterministic committee assignment (seeded by epoch randomness) reveals committee membership publicly, creating a window for adaptive targeting. Mitigations: (a) short epochs reduce the corruption window, (b) late-reveal assignment (validators learn their assignment only when they need to verify) reduces the targeting window, (c) cryptographic sortition à

la Algorand [8] (validators self-select and prove membership via VRF) eliminates the adaptive-corruption surface but adds protocol complexity. The current design does not implement (c); the hypergeometric bounds should be interpreted as applying under the non-adaptive adversary model only.

4.2 Liveness

Liveness means that every correct transaction eventually makes progress.

4.2.1 Prover Liveness

After the executor submits a result, proof generation for each proof system proceeds independently. A proof system may fail to produce a proof within the expected latency (prover crash, resource exhaustion, proof system bug).

The result remains at its current lattice position. Applications that require a higher lattice position than currently achieved simply wait. If the proving deadline expires:

1. The proof escrow for that proof system is returned to the payer (no penalty for the prover, as proof failure is not necessarily Byzantine behavior).
2. The result's lattice position is frozen at its current state.
3. Applications requiring the unachieved proof system dimension must either (a) wait for a re-proving attempt, (b) fall back to a lower safety requirement, or (c) trigger a compensating action via the saga mechanism [18].

5 Discussion

5.1 Limitations

Verification window for high-latency proof systems. For large workloads, SNARK proving may take hours. During this verification window, safety depends on the hardware attestation and the economic safety model. This is a constraint of current proof technology. As proof systems improve, the window narrows; the architecture accommodates this progression without structural change.

Hypergeometric bounds require large validator sets and uniform staking. The committee safety bounds (Lemma 4.3) assume uniform staking and a non-adaptive adversary. They are meaningful only when $n \gg 100$. For small validator sets, committees must be large relative to n, reducing the efficiency benefit of distributed verification. For non-uniform staking (the norm in production PoS systems), the bounds require extension to weighted sampling (e.g., Wallenius' noncentral hypergeometric distribution), and the safety guarantees may degrade non-linearly under heavy-tailed stake distributions.

Workload decomposition requires structure. Structure-exploiting decomposition (Section 3.2) relies on the algebraic structure of the computational object. Unstructured computations (e.g., a large arbitrary boolean circuit) do not benefit from this approach and must use generic decomposition, which is less efficient.

Adaptive adversaries and committee sampling. The hypergeometric committee safety bounds (Section 4.1.2) assume a non-adaptive adversary whose Byzantine validators are fixed before committee assignment. If the adversary is adaptive and can corrupt or deny service to specific validators after learning their committee assignment, the bounds do not apply. Symphony's deterministic committee assignment reveals membership publicly, creating a window for targeted corruption. Cryptographic sortition (as in Algorand [8]) mitigates this but is not implemented in the current design. The committee safety guarantees should be interpreted as holding under the non-adaptive adversary model; extending them to adaptive adversaries requires protocol changes.

6 Related Work

The theoretical foundations of verifiable computation were laid by Gennaro, Gentry, and Parno [7]: a client delegates computation to a server and verifies the result without re-executing. Walfish and Blumberg [16] survey the subsequent development. Pinocchio [12] made verifiable computation practical for arithmetic circuits.

In the blockchain context, verifiable computation appears as ZK rollups (zkSync, StarkNet, Polygon zkEVM), where a sequencer executes transactions and produces a validity proof checked by the L1. Optimistic rollups (Optimism [11], Arbitrum [2]) use interactive fraud proofs instead.

Symphony differs from rollups in three ways. First, verification is multi-dimensional (product lattice over heterogeneous proof systems [19]) rather than a single proof system. Second, execution is within the L1 SMR, not on a separate L2 chain. Third, the non-interactivity constraint excludes fraud proofs [19]. The computational objects also differ: rollups verify general EVM execution, while Symphony's primary targets are resource-intensive workloads (neural network inference) with exploitable algebraic structure.

Incrementally verifiable computation (IVC) [14] enables step-by-step proof accumulation. Recursive proof composition [3] allows proofs to verify other proofs. These techniques are complementary to Symphony's workload-level decomposition: IVC decomposes by time step, while structure-exploiting decomposition decomposes by the structural units of the computational object (Section 3.2).

Algorand's sortition [8] uses cryptographic self-selection for committee membership, providing security against adaptive adversaries who can target committee members after learning their identity. Ethereum's beacon chain [5] assigns validators to committees for attestation duties. Celestia's Data Availability Sampling [1] distributes data availability checks across random subsets.

Symphony applies committee sampling to proof verification but uses deterministic (non-self-selecting) assignment, which does not provide Algorand's adaptive-corruption resistance. The hypergeometric safety analysis (Section 4.1.2) follows the statistical methodology of Gilad et al. but under a weaker adversary model (non-adaptive).

EigenLayer's Actively Validated Services (AVS) restake Ethereum validators for additional verification duties. Symphony's committee-based verification can be viewed as a specific AVS design with the additional structure of the product lattice and workload decomposition.

7 Conclusion

The sealed-seed randomness model binds the executor to a user-provided seed encrypted to the TEE, preventing grinding without making the output predictable. Proof generation exploits the algebraic structure of the computational objects: the boundary width at partition points determines the linking cost, and narrow boundaries between architectural components (transformer blocks, convolutional stages) yield efficient decompositions. The prover-friendly design point — optimizing proving latency rather than proof size, since proofs are verified by committees

off-chain rather than enshrined on-chain — inverts the standard ZKP tradeoff and shrinks the verification window between hardware attestation and cryptographic finality.

References

[1] M. Al-Bassam. LazyLedger: A distributed data availability ledger with client-side smart contracts. arXiv:1905.09274, 2019.

[2] Offchain Labs. Arbitrum. 2021.

[3] S. Bowe, A. Gabizon, M. Mihaescu. Recursive proof composition from accumulation schemes. IACR ePrint 2019/1021, 2019.

[4] E. Buchman, J. Kwon, Z. Milosevic. The latest gossip on BFT consensus. arXiv:1807.04938, 2018.

[5] V. Buterin, D. Hernandez, T. Kamphefner, K. Pham, Z. Qiao, D. Ryan, J. Sin, Y. Wang, Y. X. Zhang. Combining GHOST and Casper. arXiv:2003.03052, 2020.

[6] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, A. Y. Ng. Large scale distributed deep networks. In *Proc. NeurIPS*, pp. 1232–1240, 2012.

[7] R. Gennaro, C. Gentry, B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Proc. CRYPTO*, pp. 465–482, 2010.

[8] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, N. Zeldovich. Algorand: Scaling Byzantine agreements for cryptocurrencies. In *Proc. 26th ACM Symp. Operating Systems Principles (SOSP)*, pp. 51–68, 2017.

[9] H. Jooybar, W. W. L. Fung, M. O'Connor, J. Devietti, T. M. Aamodt. GPUDet: A deterministic GPU architecture. In *Proc. ASPLOS*, pp. 1–12, 2013.

[10] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borch, E. Boyle, P. Boyle, J. P. Cantin, C. Chao, C. Clark, J. Corriel, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khatamian, J. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, D. H. Woo. In-datacenter performance analysis of a tensor processing unit. In *Proc. 44th Int. Symp. Computer Architecture (ISCA)*, pp. 1–12, 2017.

[11] Optimism Foundation. 2021.

[12] B. Parno, J. Howell, C. Gentry, M. Raykova. Pinocchio: Nearly practical verifiable computation. In *Proc. IEEE Symp. Security and Privacy*, pp. 238–252, 2013.

[13] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.* 22(4):299–319, 1990.

[14] P. Valiant. Incrementally verifiable computation. In *TCC*, 2008.

[15] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, I. Polosukhin. Attention is all you need. In *Proc. NeurIPS*, pp. 5998–6008, 2017.

[16] M. Walfish, A. J. Blumberg. Verifying computations without reexecuting them. *CACM*, 58(2), 2015.

[17] H. Wu, W. Zheng, A. Chiesa, R. A. Popa, I. Stoica. DIZK: A distributed zero knowledge proof system. In *Proc. USENIX Security*, pp. 675–692, 2018.

[18] Ritual. Superposition of Execution Models. 2026.

[19] Ritual. Multi-System Verification Model. 2026.

[20] Ritual. Symphony: Execution-Aware Consensus and Proposer Disaggregation. 2026.