Jolt Atlas: Verifiable Machine Learning Inference via Lookup Arguments in Zero Knowledge

Wyatt Benno, Alberto Centelles, Antoine Douchet,
Khalil Gibran, Houman Shadab

ICME Labs

Abstract. We present Jolt Atlas, a zero-knowledge machine learning (zkML) framework that extends the Jolt proving system to verify model inference. Unlike zkVMs that emulate RISC-V instruction execution, we adapt the Jolt lookup-centric approach to ONNX operations on tensors. The ONNX computational model eliminates the need for CPU registers and simplifies memory consistency verification. Lookup arguments based on the sumcheck protocol, as used in Twist and Shout, prove to be suitable for non-linear functions, which are building blocks in ML models. We apply optimizations such as neural teleportation to reduce the size of lookup tables while preserving model accuracy, as well as other tensor-level verification optimisations explained in this paper, and show that we can prove ML models in memory-constrained environments (a property of a prover known streaming). Furthermore, Jolt Atlas achieves zero-knowledge through the BlindFold technique as described in Vega. In contrast to other zkML frameworks, we demonstrate practical proving times for article classification models and for transformer self-attention layers, and outline the many applications of this work.

Keywords: zkML, Jolt, sumcheck, ONNX, verifiable AI, zero-knowledge, lookup arguments, teleportation

1 Introduction

The intersection of machine learning and cryptography has given rise to zero-knowledge machine learning (zkML), enabling verifiable inference without revealing model weights or input data. Traditional approaches to zkML suffer from prohibitive computational costs when proving non-linear operations such as ReLU and softmax through arithmetic circuits [16]. Jolt Atlas addresses this challenge by adapting the lookup-based proving paradigm from Jolt [1] to the domain of ONNX model inference.

Jolt is a zkVM that achieves state-of-the-art proving performance for RISC-V programs through a novel combination of lookup arguments and the sumcheck protocol which, by treating interactivity as a computational resource, enables the verifier to delegate $O(n)$ computation to the prover while performing only $O(\log n)$ work itself. Exploiting the structure of computation when applying the sum-check protocol is central to Jolt's efficiency [2].

Jolt Atlas extends this paradigm to machine learning by replacing the RISC-V instruction set architecture with ONNX computational graphs. This substitution yields several architectural differences from zkVMs:

1. **DAG-structured computation**: ONNX models are directed acyclic graphs with deterministic data flow, enabling optimizations not available in general-purpose computation. Additionally, ONNX graphs don't use CPU registers or RAM.
2. **Tensor operations**: Jolt Atlas operates on tensors (i.e., multi-dimensional vectors of scalars), rather than scalars. Each ONNX operation reads from specific input tensors and writes to a designated output tensor in a pattern known at preprocessing time. Instead of naively decomposing tensor operations into scalar computations and verifying each element independently, we verify tensor relations directly at the multilinear polynomial level.

1.1 Space-Efficient Proving via Sumcheck

A key advantage of sumcheck-based proofs, as emphasized in [2], is their amenability to *streaming* and *space-efficient* implementations. This enables proving AI model inference on resource-constrained devices such as smartphones or edge hardware.

Most SNARK provers require storing the entire witness in memory—for a model with N parameters and activations, this means $O(N)$ space. For even small language models (SLM) or transformers, this can exceed available RAM on consumer devices.

The sumcheck protocol processes data in $O(\log N)$ passes rather than requiring simultaneous access to all values. In each round, the prover:

1. Streams through the relevant polynomial evaluations
2. Computes the round polynomial coefficients incrementally
3. Discards intermediate values no longer needed

This reduces peak memory from $O(N)$ to $O(N^{1/C})$ with appropriate parameterization, at the cost of additional passes over the data (which can be stored on disk or recomputed).

As explained in section 2, the prefix-suffix decomposition directly enables space-efficient lookup proving:

- Instead of materializing a 2^N-entry lookup table in memory, the prover evaluates prefix and suffix tables of size $2^{N/C}$ each, where C is a free parameter
- The decomposition $\tilde{\mathcal{T}}(x) = \sum_i p_i(x_{\text{hi}}) \cdot s_i(x_{\text{lo}})$ allows streaming evaluation: process high-order bits, accumulate partial results, then combine with low-order contributions
- Memory usage scales with $O(|\mathcal{T}|^{1/C})$ rather than $O(|\mathcal{T}|)$

These techniques enable Jolt Atlas to prove inference for models with millions of parameters on devices with limited RAM. The prover can trade time for space: more passes over the data in exchange for dramatically reduced memory footprint. This is particularly relevant for on-device ML verification where privacy requires keeping data local rather than sending it to powerful cloud servers.

1.2 Related Work

EZKL [16] is a zkML framework built on the Halo2 proof system. It converts ONNX models to ZK-SNARK circuits, enabling privacy-preserving inference proofs. However, EZKL relies on circuit-based arithmetic constraint systems, which incur high costs for non-linear operations like ReLU.

DeepProve [3] uses a GKR-style sumcheck-based proof system. As described in [14], the GKR protocol is considered a general-purpose technique and "general-purpose techniques should sometimes be viewed as heavy hammers that are capable of pounding arbitrary nails, but are not necessarily the most efficient way of hammering any particular nail". In particular, for matrix multiplication, a primitive operation in ML models, "the GKR protocol introduces at least a constant factor overhead for the prover. In practice, this is the difference between a prover that runs many times slower than an (unverifiable) MATMULT algorithm, and a prover that runs a fraction of a percent slower". Furthermore, operations such as Gather require reading the tensor on which they operate, so we need to prove that those reads were correct. The most recent open-source implementation of DeepProve lacks these lookup arguments.

Other zkML approaches include zkCNN [4] for convolutional networks and various circuit-based systems. Most suffer from the fundamental limitation that arithmetic circuits are inefficient for non-linear operations, motivating our lookup-based approach.

2 Lookup Arguments for Non-Linear Operations in Small Space

Non-linear operations like Softmax cannot be efficiently expressed as low-degree polynomial relations. Jolt Atlas uses lookup arguments to verify these operations: the prover demonstrates that each input-output pair appears in a precomputed lookup table.

In a nutshell, a lookup table $\mathcal{T} : \{0,1\}^w \to \mathbb{F}$ maps w-bit inputs to field elements. Its multilinear extension $\tilde{\mathcal{T}}$ can be evaluated at any point $r \in \mathbb{F}^w$.

To verify a lookup (q, v) where $v = \mathcal{T}[q]$, the prover commits to the query vector and demonstrates that:

$$\tilde{\mathcal{T}}(r) = v \quad \text{at random } r$$

using the sumcheck protocol.

2.1 Prefix-Suffix Decomposition of Large Lookup Tables

A naive lookup table for operations of two 64-bit inputs would have 2^{128} entries—infeasible in practice. Following Jolt [1], we exploit the algebraic structure of lookup tables through *prefix-suffix decomposition*.

For many tables $\mathcal{T}$, the multilinear extension decomposes as:

$$\tilde{\mathcal{T}}(x) = \sum_i p_i(x_{\text{hi}}) \cdot s_i(x_{\text{lo}})$$

where $x_{\text{hi}}, x_{\text{lo}}$ partition the input bits into high-order (prefix) and low-order (suffix) components. The prefix functions p_i and suffix functions s_i operate on smaller inputs, reducing table sizes exponentially. The generalisation of the decomposition above is known as the prefix-suffix decomposition and allows the prover to minimise its memory usage. For a more comprehensive study of this protocol, see appendix A in [12].

Example: ReLU. The ReLU function $\text{ReLU}(x) = \max(0, x)$ for signed integers decomposes as:

$$\widetilde{\text{ReLU}}(r) = p_{\text{Relu}}(r_{\text{hi}}) \cdot s_{\text{One}}(r_{\text{lo}}) + p_{\text{NotMSB}}(r_{\text{hi}}) \cdot s_{\text{Relu}}(r_{\text{lo}})$$

where:

- p_{Relu} computes the high-order bits contribution when the input is non-negative
- p_{NotMSB} evaluates to 1 when the sign bit is 0 (non-negative input)
- s_{Relu} returns the low-order bits when the sign bit is 0
- s_{One} returns constant 1

The implementation defines these decompositions declaratively:

```
fn suffixes(&self) -> Vec<Suffixes> {
    vec![Suffixes::One, Suffixes::Relu]
}

fn combine<F>(&self, prefixes: &[PrefixEval<F>],
              suffixes: &[SuffixEval<F>]) -> F {
    prefixes[Prefixes::Relu] * suffixes[0] +
    prefixes[Prefixes::NotUnaryMsb] * suffixes[1]
}
```

2.2 Tensor Lookups Optimisation

Conceptually, an ONNX operation on an n-element tensor performs n lookups. However, instead of naively decomposing tensor operations into scalar computations and verifying each element independently, we verify tensor relations directly at the multilinear polynomial level. By applying the Schwartz-Zippel lemma, we can probabilistically check that the tensor operation is correct without inspecting every individual element.

2.3 Neural Teleportation for Lookup Table Compression

Activation functions like `erf` (used in GELU) and `tanh` present a challenge: they require lookup tables spanning the full input range, which increases the computational costs of both prover and verifier. To mitigate this, Jolt Atlas employs *neural teleportation* [15] to compress these tables while preserving model accuracy.

The key observation is that many activation functions *saturate*—for large inputs, the output approaches a constant (e.g., $\text{erf}(x) \to \pm 1$ as $x \to \pm\infty$). This saturation behavior enables a trade-off: dividing inputs by a teleportation factor $\tau > 1$ reduces the effective input range while preserving approximate output equivalence.

Teleportation Transform. Given an activation function σ and factor τ, the teleported computation replaces:

$$y = \sigma(x) \quad \text{with} \quad y' = \sigma(x/\tau)$$

For saturating functions, this approximation is accurate:

- **Small inputs** (linear region): $\sigma(x/\tau) \approx \sigma(x)/\tau$, introducing bounded error
- **Large inputs** (saturation region): $\sigma(x/\tau) = \sigma(x) = \pm 1$, exact equivalence

Implementation. Jolt Atlas transforms the ONNX graph by inserting a division node before selected activations:

```
model.apply_teleportation(tau=4.0, activations=[Erf, Tanh])
```

With $\tau = 4$, the lookup table range reduces by 75%, from $[-R, R]$ to $[-R/4, R/4]$. The maximum table size is bounded by 2^{16} entries, sufficient for 16-bit fixed-point activations.

Accuracy Trade-off. Empirical evaluation shows that $\tau = 4$ introduces output differences of less than 55 units on a 128-scale fixed-point representation (approximately 0.4% relative error), acceptable for most inference tasks. The trade-off between table size and accuracy can be tuned per-model.

3 Zero Knowledge via BlindFold

Jolt Atlas achieves zero-knowledge through the BlindFold technique [10], which converts sumcheck-based proofs into zero-knowledge proofs without significant overhead. The key insight, is to run the standard (non-ZK) prover with hiding commitments, then apply BlindFold to a *succinct verifier circuit* that encodes only the verifier's $O(\log n)$ checks.

3.1 The BlindFold Approach

Standard sumcheck proofs reveal intermediate values through the prover's round polynomial coefficients. BlindFold addresses this by:

1. **Hiding sumcheck messages**: All prover messages (round polynomial coefficients, claimed evaluations) are sent as Pedersen commitments rather than plaintext values.
2. **Succinct verifier R1CS**: Instead of applying ZK to the full computation, we construct a small R1CS circuit encoding only the verifier's algebraic checks. Since sumcheck verification requires only $O(\log n)$ operations, this circuit is exponentially smaller than the original computation.
3. **Nova-style folding**: The real instance is folded with a random satisfying instance, producing a folded witness $w_{\text{folded}} = w + r \cdot w_{\text{rand}}$ that reveals nothing about the original witness.

3.2 Succinct Verifier Circuit

The verifier R1CS takes as *witness* (not public inputs) the round polynomial coefficients and claimed evaluations. The circuit encodes:

- **Round consistency**: For each sumcheck round i, verify $g_i(0) + g_i(1) = \text{claimed\_sum}_{i-1}$
- **Challenge application**: Verify $\text{claimed\_sum}_i = g_i(c_i)$ where c_i is the Fiat-Shamir challenge
- **Final identity**: The algebraic relation combining all evaluation claims

Commitment consistency is handled by the split-committed R1CS framework: the instance contains commitments $\bar{W}_i = \text{Com}(W_i, \rho_i)$, and satisfaction requires these commitments to open correctly.

3.3 The Complete ZK Protocol

1. **Prover runs sumchecks with hiding commitments**: For each round, compute the round polynomial and send its commitment (not the coefficients themselves).
2. **Prover commits to evaluations**: For each polynomial evaluation claim $y = P(r)$, send $C_y = \text{Com}(y, \rho)$ rather than y directly.
3. **Prover constructs split-committed R1CS**: The instance contains all commitments and Fiat-Shamir challenges; the witness contains the actual values and randomness.
4. **Prover runs BlindFold**: Sample a random satisfying pair $(u_{\text{rand}}, w_{\text{rand}})$, fold with the real instance, and send the folded witness.
5. **Prover sends ZK-Dory evaluation proofs**: For each committed evaluation C_y, prove that it commits to the correct evaluation of the corresponding polynomial.
6. **Verifier checks**: Verify the folded witness satisfies the folded instance, and verify all ZK-Dory proofs.

The verifier learns *nothing* about actual values—only that the committed values satisfy the required algebraic relations.

4 Jolt Atlas Architecture

Having established the BlindFold approach to zero-knowledge, we now describe how Jolt Atlas organizes its proof as a DAG of sumcheck instances that feed into the succinct verifier R1CS.

4.1 Execution Trace

An ONNX model execution produces a *trace*: a sequence of computational steps where each step corresponds to one ONNX operation applied to specific tensor elements. Each trace entry contains:

- The operation type (Add, Mul, ReLU, etc.)
- Source tensor addresses and values
- Destination tensor address and value
- For non-deterministic operations (e.g., division), an advice value

The trace length T determines the constraint system size. Traces are padded to the next power of two for efficient polynomial representation.

4.2 Proof DAG Structure

The proof is organized as a DAG where nodes represent sumcheck instances and edges represent polynomial evaluation claims passed between stages. Two sumchecks cannot be batched together if one depends on the other's output—this dependency structure defines the staging.

The prover maintains a `StateManager` coordinating:

- Witness polynomials and their (hiding) commitments
- An opening accumulator collecting evaluation claims for batch verification
- The Fiat-Shamir transcript binding all messages
- The split-committed R1CS witness accumulating round coefficients

4.3 Proof Stages and BlindFold Integration

Following Jolt's architecture [7], proving proceeds through multiple stages. Each stage produces sumcheck proofs with hiding commitments, and the round polynomial coefficients become witness elements in the succinct verifier R1CS.

Stage 1: Outer Sumcheck. The `SpartanDag` proves the outer R1CS constraint:

$$\sum_x \widetilde{eq}(\tau, x) \cdot (\tilde{A}z(x) \cdot \tilde{B}z(x) - \tilde{C}z(x)) = 0$$

The round polynomial coefficients are committed (not revealed), and the claims $\tilde{A}z(r), \tilde{B}z(r), \tilde{C}z(r)$ are also committed.

Stage 2: Inner Sumcheck and Virtualization. Batches the inner sumcheck (binding witness polynomials to the random point from Stage 1) with product virtualization sumchecks. All round coefficients feed into the verifier R1CS witness.

Stage 3: Lookups and Instruction Verification. Batches:

- `ReadRafSumcheck`: Verifies read-only access for lookup tables
- `BooleanitySumcheck`: Ensures lookup indices are in valid ranges
- `InstructionInputSumcheck`: Verifies instruction inputs match claimed values

Stage 4: Memory Checking. The `MemoryDag` verifies read-write consistency, ensuring tensor values read during execution match previously written values.

Stage 5: Final Validation. Verifies the final tensor states are correctly computed from the execution trace.

Stage 6: Bytecode Verification. Verifies that executed operations match the committed ONNX graph structure.

5 Polynomial Commitment Scheme

All cryptographic assumptions in sumcheck-based SNARKs derive entirely from the polynomial commitment scheme (PCS). The sumcheck-based PIOP (Polynomial Interactive Oracle Proof) is *information-theoretically secure*—it requires no cryptographic assumptions. This clean separation means that replacing the PCS immediately yields different security properties: pairing-based schemes like Dory provide security under discrete logarithm assumptions, while lattice-based schemes like Greyhound or Hachi would provide post-quantum security.

The choice of PCS significantly impacts prover efficiency, proof size, and verification cost. This section explains the constraints, our current choice and where Jolt Atlas is going.

5.1 PCS Requirements

A PCS for Jolt Atlas must support:

1. **Multilinear polynomials**: ML workloads involve high-dimensional tensors represented as multilinear polynomials with many variables.
2. **Batch opening**: Verifying a DAG of sumchecks produces many evaluation claims that should be batched efficiently.
3. **Transparent or universal setup**: Avoiding trusted setup simplifies deployment.
4. **Small proofs and succint verifier**: On-chain verification requires compact proofs and an efficient verifier.

5.2 Dory

Jolt Atlas uses Dory [11], which provides:

- $O(\sqrt{N})$ prover time for N-coefficient polynomials
- $O(\log N)$ proof size
- $O(\log N)$ verifier time (in $\mathbb{G}_T$ operations)

– Transparent setup from random beacons

Dory commitments use inner-pairing-products over structured reference strings in $\mathbb{G}_1 \times \mathbb{G}_2$, enabling efficient batch verification.

The main limitation is verifier cost: each opening requires $O(\log N)$ operations in the pairing target group $\mathbb{G}_T$, which is a degree-12 field extension. This affects both recursive verification and on-chain deployment.

5.3 Tradeoffs

Alternative PCS choices present different tradeoffs:

– **KZG**: Faster verification but quotient polynomials destroy sparsity, losing Jolt's main efficiency advantage.
– **Lattice-based (Labrador, Greyhound, Hachi)**: Post-quantum secure with faster provers (no pairings), but larger proofs. Recent work on commitment size reduction (ABBA[5]) improves this trade-off. See Section 9 for details.

6 ONNX: The Bridge to Machine Learning

ONNX (Open Neural Network Exchange) serves as the standardized interface between machine learning frameworks and Jolt Atlas's verification system. This section describes ONNX's role and how Jolt Atlas handles its computational model.

6.1 Why ONNX

ONNX provides a framework-agnostic representation of neural network computations. Models trained in PyTorch, TensorFlow, or JAX can be exported to ONNX format and verified by Jolt Atlas without modification to the training workflow. This separation of concerns—train anywhere, verify universally—is essential for practical adoption.

The ONNX format represents computations as directed acyclic graphs where:

– **Nodes** represent operations (Add, MatMul, ReLU, etc.)
– **Edges** represent tensors flowing between operations
– **Attributes** parameterize operations (kernel sizes, axis specifications)
– **Initializers** store model weights and biases

This graph structure aligns naturally with Jolt Atlas's DAG-based proof architecture, where each ONNX node becomes a verification target.

6.2 Operator Support

ONNX defines over 180 operators across 19 domains. Jolt Atlas currently supports a subset focused on inference workloads:

Elementwise Operations:

- Arithmetic: `Add`, `Sub`, `Mul`, `Div`
- Activations: `Relu`, `Sigmoid` (via lookup), `Tanh` (via lookup)
- Comparisons: `Greater`, `Less`, `Equal`

Tensor Operations:

- Shape manipulation: `Reshape`, `Transpose`, `Squeeze`, `Unsqueeze`
- Data movement: `Gather`, `Scatter`, `Concat`, `Split`
- Broadcasting: Automatic broadcasting following NumPy semantics

Reduction Operations:

- `ReduceSum`, `ReduceMean`, `ReduceMax`
- Supports arbitrary axis specifications

Matrix Operations:

- `MatMul`: Standard matrix multiplication
- `Einsum`: Generalized tensor contraction supporting patterns like ‘‘`mk,kn->mn`’’ (matrix multiply), ‘‘`bmk,bkn->bmn`’’ (batched multiply), and attention patterns

6.3 From ONNX Graph to Proof

The verification pipeline proceeds as follows:

1. **Graph parsing**: The ONNX protobuf is parsed into an internal representation, resolving tensor shapes and operator attributes.
2. **Preprocessing**: Static analysis determines memory layout, identifies which operations require lookups versus polynomial verification, and generates the bytecode specification.
3. **Trace generation**: The model executes on the input, recording all intermediate tensor values. Each scalar operation becomes a trace entry.
4. **Proof generation**: The trace feeds into the Jolt Atlas prover, which generates sumcheck proofs for each operation according to its type.

6.4 Handling Unsupported Operators

When an ONNX model contains unsupported operators, several strategies apply:

- **Decomposition**: Complex operators are decomposed into supported primitives (e.g., `LayerNorm` becomes `ReduceMean` + `Sub` + `Mul`)

- **Custom precompiles**: Frequently-used patterns can be added as optimized precompiles
- **Model modification**: In some cases, model architectures can be adjusted to use supported operators without accuracy loss

Our initial focus is to support the operators required for common inference workloads—classification, embedding generation, and transformer attention—rather than the full ONNX specification.

7 Applications

Zero-knowledge machine learning enables new applications where computational integrity and privacy are essential. This section describes key use cases, with particular focus on autonomous AI agents.

7.1 Trustless AI Agents

The emergence of autonomous AI agents—software systems that perceive, decide, and act without human intervention—creates new requirements for computational verification [9]. When agents make consequential decisions (financial trades, resource allocation, access control), stakeholders need assurance that the agent executed its claimed model correctly.

The Verification Gap. Current agent architectures rely on trust assumptions:

- *Reputation-based*: Trust agents based on past behavior; vulnerable to gaming and provides no guarantees for novel situations
- *Crypto-economic*: Validators stake capital and re-execute computations; prohibitively expensive for complex ML inference
- *Hardware attestation*: TEEs (Trusted Execution Environments) provide isolation but depend on hardware manufacturers and have documented vulnerabilities (Spectre, Meltdown)

zkML Solution. Jolt Atlas enables a fourth approach: *cryptographic verification*. The agent produces a proof alongside its output, demonstrating that the output resulted from executing a specific model on specific inputs. Verification requires only the proof and public commitments—no re-execution, no hardware trust assumptions, no economic bonds.

This proof can be verified by:

- Other agents in a multi-agent system
- Smart contracts for on-chain settlement
- Auditors reviewing agent decisions post-hoc
- Users seeking assurance about AI-driven recommendations

7.2 Trustless Agentic Memory

A critical but often overlooked component of AI agents is memory: the stored context, embeddings, and retrieved information that inform decisions [8]. Current memory systems present a verification gap—even if inference is verified, corrupted or manipulated memories can compromise agent behavior.

The Memory Problem. Agent memories typically reside in vector databases (Pinecone, Weaviate, Chroma) controlled by third parties. This creates vulnerabilities:

- No proof that retrieved memories are authentic
- No verification that embeddings were computed correctly
- Memories are not portable across infrastructure providers
- Changing embedding models invalidates the entire memory store

Verifiable Embeddings. Jolt Atlas enables *trustless agentic memory* by proving correct embedding computation. When an agent stores a memory:

1. The text (or other data) is processed by an embedding model
2. Jolt Atlas generates a proof that the embedding was computed correctly
3. The embedding and proof are stored together
4. Retrieval includes verification that the embedding matches the claimed computation

This ensures that memories cannot be silently corrupted or substituted. Combined with decentralized storage, agents gain portable, verifiable memory that is not dependent on any single provider.

7.3 Agent-to-Agent Transactions

As AI agents increasingly interact with each other—negotiating, trading, collaborating—verification becomes essential for establishing trust between autonomous systems [9].

Example: Market-Making Agents. Consider two agents negotiating a trade:

- Agent A claims its pricing model values an asset at $X
- Agent B needs assurance this valuation is legitimate, not manipulated
- With zkML, Agent A provides a proof that $X resulted from executing its committed model on current market data
- Agent B verifies the proof (milliseconds) without accessing A's proprietary model

Example: Collaborative Inference. Multiple agents can collaborate on inference tasks:

- Agent A computes embeddings for a query
- Agent B performs retrieval using those embeddings
- Agent C generates a response based on retrieved context
- Each step produces a proof; the chain of proofs establishes end-to-end correctness

7.4 Decentralized Finance Applications

DeFi protocols can leverage zkML for:

Verifiable Oracles. Price feeds derived from ML models (sentiment analysis, volatility prediction) can include proofs of correct computation, eliminating trust in oracle operators.

Credit Scoring. Lending protocols can verify that credit decisions followed the stated model without accessing sensitive borrower data.

Trading Bot Verification. Automated trading strategies can prove they executed as specified, providing accountability for delegated capital management.

7.5 Privacy-Preserving Inference

The zero-knowledge property enables inference where sensitive data remains hidden:

Medical Diagnosis. A diagnostic model proves its output was computed correctly without revealing patient data to the verifier.

Document Classification. PII detection can classify documents (e.g., "contains SSN") without exposing document contents.

Model Protection. Inference can be verified without revealing model weights, protecting proprietary models while proving they were used correctly.

7.6 Compliance and Auditability

Regulated industries require audit trails for AI-driven decisions:

Financial Services. Regulators can verify that ML-based decisions (loan approvals, fraud detection) followed approved models without accessing customer data.

Healthcare. Proofs can demonstrate that diagnostic assistance used FDA-approved model versions.

Content Moderation. Platforms can prove content decisions followed stated policies, providing accountability without exposing proprietary moderation systems.

8 Benchmarks

Jolt Atlas demonstrates practical proving times across a range of ML workloads. All benchmarks were collected on a workstation with a multi-core CPU; expect $\pm 10\%$ variance depending on hardware.

8.1 Transformer Self-Attention

The self-attention mechanism is a core component of transformer models:

Stage	Time
Prove	20.8 s
Verify	143 ms
End-to-end CLI	25.8 s

Peak memory usage reached approximately 5.6 GB during sumcheck round 10.

8.2 Article Classification

A text classification model categorizing articles into business, tech, sport, entertainment, and politics:

Project	Latency	Notes
Jolt Atlas	∼0.7 s	
mina-zkml	∼2.0 s	
ezkl	4–5 s	
deep-prove	N/A	Missing gather primitive
zk-torch	N/A	Missing reduceSum primitive

8.3 Perceptron MLP

A basic multi-layer perceptron for sanity testing:

Project	Latency	Notes
Jolt Atlas	∼800 ms	Full memory consistency
deep-prove	∼200 ms	Lacks memory consistency checking

9 Future Work

Several directions for future development include:

Lattice-Based Polynomial Commitment Schemes: A promising direction is replacing Dory with lattice-based PCS constructions to achieve post-quantum security. Several recent schemes offer compelling trade-offs:

Hachi [6] is a lattice-based multilinear PCS currently in development. Compared to Dory, Hachi offers several advantages: (1) post-quantum security under Module-SIS assumptions, (2) faster prover time due to the absence of expensive pairing operations, and (3) $O(\sqrt{2^\ell} \cdot \lambda)$ verifier time for ℓ-variate polynomials, approximately 35% faster than Greyhound [13]. The main trade-off is larger proof sizes compared to pairing-based schemes.

A key consideration for lattice-based schemes is commitment size. While proof sizes are larger than pairing-based alternatives, recent work on *ABBA* [5] (also in preparation) explores techniques for reducing commitment sizes through

improved lattice-based commitment constructions. Smaller commitments directly benefit both proof size and verifier efficiency.

Neither Hachi nor ABBA are currently published; integration would proceed as these schemes mature and undergo community review.

On-Chain Verification: Deploying proofs on Ethereum is challenging due to Dory's $\mathbb{G}_T$ operations, which lack precompiles. Replacing Dory with a lattice-based PCS may also mitigate this limitation.

Extended ONNX Support: Adding support for additional operators including softmax, layer normalization, and convolutions.

The rapid development of research in both cryptography and artificial intelligence continues to provide new techniques and optimizations to integrate into Jolt Atlas. Advances in polynomial commitment schemes, folding techniques, and hardware acceleration on the cryptographic side, combined with new model architectures, quantization methods, and inference optimizations on the AI side, ensure that verifiable machine learning will remain an active and evolving field.

References

[1] Arasu Arun, Srinath Setty, and Justin Thaler. *Jolt: SNARKs for Virtual Machines via Lookups*. Cryptology ePrint Archive, Paper 2023/1217. `https://eprint.iacr.org/2023/1217`. 2023.

[2] Various Authors. *Sumcheck is All You Need*. Cryptology ePrint Archive, Paper 2025/2041. `https://eprint.iacr.org/2025/2041`. 2025.

[3] Various Authors. *Zero-Knowledge Proofs of Training for Deep Neural Networks*. Cryptology ePrint Archive, Paper 2024/162. `https://eprint.iacr.org/2024/162`. 2024.

[4] Various Authors. *zkCNN: Zero Knowledge Proofs for Convolutional Neural Network Predictions and Accuracy*. Cryptology ePrint Archive. 2021.

[5] a16z crypto. *ABBA: Lattice-Based Commitments with Small Openings*. Unpublished manuscript. In preparation. 2025.

[6] a16z crypto. *Hachi: A Lattice-Based Polynomial Commitment Scheme*. Unpublished manuscript. In preparation. 2025.

[7] a16z crypto. *Jolt Documentation: Architecture*. Online. `https://jolt.a16zcrypto.com/how/architecture/architecture.html`. 2024.

[8] ICME. *Trustless Agents Can't Work Without Trustless Agentic Memory*. ICME Blog. `https://blog.icme.io/trustless-agents-cant-work-without-trustless-agentic-memory/`. 2024.

[9] ICME. *Trustless Agents: Cryptographic Verification for AI Decision-Making*. ICME Blog. `https://blog.icme.io/trustless-agents/`. 2024.

[10] Darya Kaviani and Srinath Setty. *Vega: Low-Latency Zero-Knowledge Proofs over Existing Credentials*. Cryptology ePrint Archive, Paper 2025/2094. `https://eprint.iacr.org/2025/2094`. 2025.

[11] Jonathan Lee. *Dory: Efficient, Transparent arguments for Generalised Inner Products and Polynomial Commitments*. Cryptology ePrint Archive, Paper 2020/1274. `https://eprint.iacr.org/2020/1274`. 2020.

[12] Vineet Nair, Justin Thaler, and Michael Zhu. *Proving CPU Executions in Small Space*. Cryptology ePrint Archive, Paper 2025/611. https://eprint.iacr.org/2025/611. 2025.

[13] Ngoc Khanh Nguyen and Gregor Seiler. *Greyhound: Fast Polynomial Commitments from Lattices*. Cryptology ePrint Archive, Paper 2024/1293. https://eprint.iacr.org/2024/1293. 2024.

[14] Justin Thaler. *Proofs, Arguments, and Zero-Knowledge*. https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.pdf. Now Publishers, 2022.

[15] Yongqi Wang et al. *TeleSparse: Neural Teleportation for Efficient Model Training and Inference*. arXiv preprint. https://arxiv.org/abs/2305.13110. 2024.

[16] zkonduit. *EZKL: Easy Zero-Knowledge Machine Learning*. GitHub. https://github.com/zkonduit/ezkl. 2024.