

Jolt Atlas: Verifiable Inference via Lookup Arguments in Zero Knowledge

Wyatt Benno, Alberto Centelles, Antoine Douchet,
Khalil Gibran

ICME Labs

Abstract. We present Jolt Atlas, a zero-knowledge machine learning (zkML) framework that extends the Jolt proving system to model inference. Unlike zkVMs (zero-knowledge virtual machines), which emulate CPU instruction execution, Jolt Atlas adapts Jolt’s lookup-centric approach and applies it directly to ONNX tensor operations. The ONNX computational model eliminates the need for CPU registers and simplifies memory consistency verification. In addition, ONNX is an open-source, portable format, which makes it easy to share and deploy models across different frameworks, hardware platforms, and runtime environments without requiring framework-specific conversions. Our lookup arguments, which use sumcheck protocol, are well-suited for non-linear functions—key building blocks in modern ML. We apply optimisations such as neural teleportation to reduce the size of lookup tables while preserving model accuracy, as well as several tensor-level verification optimisations detailed in this paper. We demonstrate that Jolt Atlas can prove model inference in memory-constrained environments—a prover property commonly referred to as **streaming**. Furthermore, we discuss how Jolt Atlas achieves zero-knowledge through the BlindFold technique, as introduced in Vega. In contrast to existing zkML frameworks, we show practical proving times for classification, embedding, automated reasoning, and small language models. Jolt Atlas enables cryptographic verification that can be run on-device, without specialised hardware. The resulting proofs are succinctly verifiable. This makes Jolt Atlas well-suited for privacy-centric and adversarial environments. In a companion work, we outline various use cases of Jolt Atlas and how they apply as guardrails in agentic commerce and for trustless AI context (often referred to as *AI memory*).

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

Table of Contents

Jolt Atlas: Verifiable Inference via Lookup Arguments in Zero Knowledge <i>Wyatt Benno, Alberto Centelles, Antoine Douchet, Khalil Gibran</i>	1
1 Introduction	3
1.1 Jolt’s inheritance	4
1.2 Our Contributions	5
1.3 Related Work	5
2 Jolt Atlas Architecture	6
2.1 Execution Trace	6
2.2 Proof DAG Structure	6
3 Zero Knowledge via BlindFold	8
3.1 The BlindFold Protocol	9
3.2 R1CS Encoding of the Sumcheck Verifier	11
3.3 Integration with Jolt Atlas	12
4 Lookup Arguments in Small Space	14
4.1 Prefix-Suffix Decomposition of Large Lookup Tables	14
4.2 Neural Teleportation for Lookup Table Compression	15
5 ONNX: The Bridge to Machine Learning	16
5.1 Why ONNX	16
5.2 Operator Support	16
5.3 From ONNX Graph to Proof	17
5.4 CPU trace vs Tensor trace	17
6 Benchmarks	17
6.1 nanoGPT (~0.25M parameters, 4 transformer layers)	18
6.2 GPT-2 (125M parameters)	18
7 Future Work	18

1 Introduction

At the intersection of machine learning, cryptography, and computer science, an active and very lively field is emerging called zero-knowledge machine learning (zkML). It enables verifiable inference with potentially hidden model weights, hidden input data, and even hidden model architecture. zkML protocols build on the techniques from the more general and long-studied field of zero-knowledge succinct arguments of knowledge (zkSNARKs). These allow a prover to convince a verifier that a statement is true, without revealing any information beyond the statement’s validity. Such arguments are often succinctly verifiable, meaning that even though the underlying program and prover computation can take significant time and computational resources, the computational work of the verifier is cheap. In the case of zkML, the statements being proven are about machine learning models.

Historically, the task of achieving practical zkML systems has been considered highly impractical. This is due to a combination of factors. Modern neural networks comprise millions to billions of parameters. Even evaluating the model entails substantial arithmetic on high-dimensional tensors. General-purpose SNARK proving still imposes a substantial 100,000X to 1,000,000X overhead over native execution [5].

The situation is compounded by the fact that neural networks involve more than linear algebra; they rely heavily on non-linear operations such as softmax or ReLU whose faithful realisation in low-degree arithmetic constraint systems can blow up constraint counts and/or degrees, often making them the dominant cost unless specialized techniques such as lookup arguments are used. Meanwhile, even the ostensibly “friendly” parts of ML, namely matrix multiplication and tensor contractions, dominate FLOPs and arise at large scale; proving them efficiently requires protocols that exploit their algebraic structure, because treating them as generic circuits tends to introduce at minimum large constant-factor overheads.

Finally, many SNARK constructions require materialising large portions of the witness and intermediate polynomials in memory to produce commitments and openings. For ML inference, this can translate into multi-gigabyte peak memory even for modest models, pushing beyond the RAM budget of commodity or on-device provers.

This work addresses these challenges by adapting the lookup-based proving paradigm from Jolt [1] to the computational model of neural networks, moving from CPU instruction execution to tensor computation.

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. Lookups eliminate the need to arithmetise non-linear functions, while sumcheck verifies large batches of computations with logarithmic verifier work and enables streaming evaluation. Exploiting the structure of computation when applying the sumcheck protocol is central to Jolt’s efficiency [19]. Their techniques and implementation are thoroughly documented [17].

Below we summarise the core ideas behind Jolt that are adopted in this work.

1.1 Jolt’s inheritance

Lookup-centric proving. A central design choice in Jolt is to express instructions as *lookups* rather than as arithmetic constraints. Instead of proving complex relations directly as polynomials, the prover reduces to membership statements of the form “ (q, v) appears in a table \mathcal{T} ,” where \mathcal{T} captures an operation’s behavior. This is particularly effective for non-linear and piecewise primitives (comparisons, bit-level behavior, range checks) that are expensive in standard circuit encodings. This lookup-first approach is precisely the mechanism Jolt Atlas adopts for ML activations and other non-linear tensor operators (Section 4).

Proof as a DAG of sumchecks (staging and batching). Jolt organises the proof as a directed acyclic graph (DAG) of sumcheck instances: nodes are sumchecks, edges are evaluation claims that feed later checks. This dependency structure determines which sumchecks can be batched together and which must be staged sequentially. Jolt Atlas inherits this viewpoint almost verbatim.

Twist and Shout: fast memory checking. Jolt’s “Twist and Shout” optimisation accelerates memory consistency checking using one-hot addressing and structured increments that reduce expensive bookkeeping in RAM/register trace verification. Although ONNX computation removes general-purpose registers and arbitrary RAM access, we still need a notion of *consistency of reads and writes* over tensor buffers (Section 2); the lesson from Twist and Shout is that domain-specific access patterns admit much cheaper consistency proofs than prior offline memory checking arguments [2].

Virtual polynomials. Jolt avoids committing to large sparse polynomials by defining them *virtually* as algebraic combinations of a small set of low-degree polynomials. A virtual polynomial is a part of the witness that is never committed directly. The term “virtual polynomial” was originally introduced in the Binius paper [7].

Prefix-suffix decomposition and streaming (small-space proving). Jolt’s lookup arguments can be implemented in small space by algebraically decomposing large tables so that their multilinear extension factors into prefix and suffix components evaluated on smaller domains [14]. This enables streaming provers that trade time (typically $O(C)$ streaming passes) for space, reducing peak memory from $O(|\mathcal{T}|)$ to roughly $O(|\mathcal{T}|^{1/C})$ while incurring total prover time on the order of $O(C|\mathcal{T}|)$ field operations, where \mathcal{T} is the number of non-zero elements of a table, for a tunable parameter C . This directly motivates Jolt Atlas’s focus on “streaming” proving for large models on constrained devices (Section 4).

Zero knowledge via BlindFold : Jolt obtains zero knowledge by applying the BlindFold technique, with small overhead.

1.2 Our Contributions

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

1. **DAG-structured computation:** ONNX models are directed acyclic graphs with deterministic data flow, enabling optimisations not available in general-purpose computation. Additionally, ONNX graphs do not 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.
3. **Faster commitments:** Unlike Jolt, our computational model doesn't require us to commit to the full witness at once, but rather to the inner values of tensors. Dory, the polynomial commitment scheme used in Jolt, is only justified for witnesses over 2^{26} elements. We therefore replace it with HyperKZG [21], a KZG-based PCS for multilinear polynomials derived from the Gemini transformation [3], trading transparent setup for pairing-based succinct openings that are well suited to on-chain verification.

In our companion work “Agentic Trust: Solving the Principal-Agent Problem in Autonomous Commerce Through Cryptographic Verification”, we show how to use zkML with Automated Reasoning models for succinctly verifiable agentic guardrails with over 99% accuracy.

1.3 Related Work

EZKL [13] is a zkML framework built on the Halo2 proof system from ONNX that relies on both circuit-based arithmetic constraint systems. As already mentioned, this incurs high costs for non-linear operations. While they introduced lookup tables (meaning roughly two advice columns, and two fixed columns of the Halo2 “grid”) for certain non-linear operations such as ReLU, these tables cannot be materialised for large ranges, so they are restricted to small-value operations. Furthermore, the Halo2 proof system does not natively exploit sparsity in lookups or in the matrix-multiplication structure, so the prover pays a largely dense constraint cost.

Bionetta [20] is a Groth16-based [8] zkML framework focused on client-side proving and zero-knowledge for EVM on-chain verification. As a variant of Groth16, UltraGroth also requires a trusted setup. In Bionetta, weights are hardcoded into the circuit during compilation/setup. That lets them implement matrix-vector and matrix-matrix multiplications with a fixed matrix in 0 constraints (note that this doesn't mean that general MatMul is free). However,

like EZKL, it suffers from the fundamental limitation that arithmetic circuits are inefficient for non-linear operations. Furthermore, it doesn't support general ONNX models.

DeepProve [6] uses a GKR-style sumcheck-based proof system based on the Ceno zkVM [11]. As described in [18], 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.

2 Jolt Atlas Architecture

We now describe how Jolt Atlas organises its zero-knowledge proof of an ONNX model as a DAG of sumcheck instances that feed into a succinct R1CS verifier, using BlindFold to hide the witness.

2.1 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.

2.2 Proof DAG Structure

The proof is organised 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.

Following Jolt's architecture [17], 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.

The terminology in the coming sections follows the excellent survey “Sum-check is all you need” [19].

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

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

producing a random point $r = (r_{\text{cycle}}, r_{\text{var}})$ and committed claims $\tilde{A}z(r), \tilde{B}z(r), \tilde{C}z(r)$.

Stage 2: Inner Sumcheck and Virtualisation.

Inner sumcheck. The verifier samples a random linear combination coefficient ρ and the prover shows:

$$\tilde{A}z(r) + \rho \tilde{B}z(r) + \rho^2 \tilde{C}z(r) = \sum_y (\tilde{A}_{\text{small}}(r_{\text{var}}, y) + \rho \tilde{B}_{\text{small}}(r_{\text{var}}, y) + \rho^2 \tilde{C}_{\text{small}}(r_{\text{var}}, y)) \cdot z(y)$$

where $z(y)$ is the vector whose entries are the evaluations of the witness polynomial $\tilde{P}_i(r_{\text{cycle}})$.

Product virtualisation sumchecks. Witness variables defined as element-wise products of two committed polynomials:

$$\tilde{V}(r_{\text{cycle}}) = \sum_t \tilde{e}q(r_{\text{cycle}}, t) \cdot L(t) \cdot R(t).$$

The four products virtualised are:

- Combined instruction lookup input: $\text{LeftInput}(t) \times \text{RightInput}(t)$.
- Selective write to destination tensor: $\text{td_addr}(t) \times \text{write_flag}(t)$
- Conditional selection condition: $\text{ts1_val}(t) \times \text{select_flag}(t)$
- Conditional selection result: $\text{td_write_val}(t) \times \text{select_flag}(t)$

Stage 3: Lookups and Instruction Verification. Batches:

- **PCSumcheck:** Ensures each execution step (PC) transitions to the next instruction (NextPC):

$$\widetilde{\text{NextPC}}(r_{\text{cycle}}) = \sum_t \widetilde{\text{PC}}(t) \cdot \tilde{e}q_{+1}(r_{\text{cycle}}, t).$$

- **ReadRafSumcheck:** A sparse-dense sumcheck verifying lookup-table reads. Each 64-bit lookup index is decomposed into chunks, and a prefix-suffix protocol checks:

$$\widetilde{\text{LookupOutput}}(r_{\text{cycle}}) = \sum_{k, j} \tilde{e}q(r_{\text{addr}}, k) \cdot \tilde{e}q(r_{\text{cycle}}, j) \cdot \text{val}(k) \cdot \text{ra}(k),$$

where $\text{ra}(k)$ is the random-address indicator polynomial and $\text{val}(k)$ the table value polynomial. The prefix-suffix decomposition reduces the prover’s memory footprint from $O(|\mathcal{T}|)$ to $O(|\mathcal{T}|^{1/C})$.

- **BooleanitySumcheck:** Verifies that each ra_i chunk polynomial (the random-address accumulations) satisfies the booleanity constraint

$$\text{ra}_i(k)^2 - \text{ra}_i(k) = 0$$

- **HammingWeightSumcheck:** Verifies that each ra_i chunk polynomial is a one-hot vector. The claimed sum is

$$\sum_k \sum_{i=0}^{D-1} \gamma^i \text{ra}_i(k) = \sum_{i=0}^{D-1} \gamma^i,$$

where γ is a fresh Fiat-Shamir challenge. Since the booleanity sumcheck already ensures $\text{ra}_i(k) \in \{0, 1\}$, this equality forces $\sum_k \text{ra}_i(k) = 1$ for every chunk i .

- **InstructionInputSumcheck:** Verifies instruction inputs match claimed values

Stage 4: Random-Address Virtualisation and Memory Checking.

This stage batches two sumchecks:

- **RASumCheck:** Virtualises the $D = 8$ per-chunk ra_i polynomials into d groups, verifying that the full random-address product $\prod_i \text{ra}_i$ equals the claimed evaluation.
- **MemoryDag:** Verifies read-write consistency, ensuring tensor values read during execution match previously written values.

Stage 5: Memory Value Evaluation. Verifies the evaluation of the memory-value polynomial $\widetilde{\text{Val}}$ at the random point $(r_{\text{addr}}, r_{\text{cycle}})$ produced by Stage 4’s read-write checking. The claimed sum $\widetilde{\text{Val}}(r_{\text{addr}}, r_{\text{cycle}})$ is reduced via sumcheck over the cycle dimension:

$$\widetilde{\text{Val}}(r_{\text{addr}}, r_{\text{cycle}}) = \sum_j \text{Inc}(j) \cdot \text{Wa}(j) \cdot \text{Lt}(j, r_{\text{cycle}}),$$

here $\text{Inc}(j)$ is the committed increment polynomial, $\text{Wa}(j) = \widetilde{eq}(r_{\text{addr}}, \text{td}(j))$ is the write-allocation polynomial indicating whether cycle j writes to the queried address, and $\text{Lt}(j, r_{\text{cycle}})$ is the “less-than” multilinear extension that selects only writes occurring before the queried cycle. The output claims open Inc and Wa at the sumcheck’s final point.

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

Stage 7: BlindFold. Folds the existing instance with a Nova randomised instance.

3 Zero Knowledge via BlindFold

Jolt Atlas’ prover executes multiple sumcheck stages (e.g., bytecode, instruction lookups, etc.), each producing a transcript of polynomial coefficients and verifier challenges. These transcripts are not zero-knowledge, since they reveal the

witness. BlindFold [9] retrofits zero-knowledge onto these proofs by encoding the sumcheck verifier as an R1CS circuit and applying Nova-style folding with a random satisfying pair. Like Jolt, we achieve zero-knowledge via BlindFold by:

1. **Hiding sumcheck messages:** All prover messages (round polynomial coefficients, claimed evaluations) are sent as Pedersen commitments rather than plaintext values.
2. **Nova-style folding:** The existing 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. **Succinct verifier R1CS:** We construct a small R1CS circuit encoding only the verifier's algebraic checks.

3.1 The BlindFold Protocol

In this section we provide a description of BlindFold as described in [9].

Relaxed R1CS Standard R1CS $(AZ) \circ (BZ) = CZ$ does not fold cleanly due to cross-terms. Following Nova [10], we use relaxed R1CS:

$$(AZ) \circ (BZ) = u \cdot (CZ) + \mathbf{E}, \quad (1)$$

where $u \in \mathbb{F}$ is a scalar and $\mathbf{E} \in \mathbb{F}^m$ is an error vector. A standard R1CS instance is a special case with $u = 1$ and $\mathbf{E} = \mathbf{0}$.

Instance (public). $\mathcal{U} = (\bar{E}, u, \bar{W}, \mathbf{x}, \{\bar{R}_i\}, \{\bar{V}_i\})$

- $\bar{E} = \text{Com}(\mathbf{E}, r_E)$: Pedersen commitment to the error vector
- $\bar{W} = \text{Com}(\mathbf{W}, r_W)$: Pedersen commitment to the witness
- $\bar{R}_i = \text{Com}(\mathbf{c}^{(i)}, \rho_i)$: per-round commitments to polynomial coefficients
- \bar{V}_i : evaluation commitments for PCS binding constraints

Witness (private). $\mathcal{W} = (\mathbf{E}, r_E, \mathbf{W}, r_W, \{\mathbf{c}^{(i)}, \rho_i\})$.

Folding (Nova-style) Given two satisfying relaxed R1CS pairs $(\mathcal{U}_1, \mathcal{W}_1)$ and $(\mathcal{U}_2, \mathcal{W}_2)$:

Cross-term computation. Compute $\mathbf{T} \in \mathbb{F}^m$:

$$T_i = (AZ_1)_i \cdot (BZ_2)_i + (AZ_2)_i \cdot (BZ_1)_i - u_1(CZ_2)_i - u_2(CZ_1)_i. \quad (2)$$

This arises from expanding $(AZ') \circ (BZ')$ where $\mathbf{Z}' = \mathbf{Z}_1 + r\mathbf{Z}_2$:

$$(AZ') \circ (BZ') = \underbrace{(AZ_1) \circ (BZ_1)}_{\text{from } (\mathcal{U}_1, \mathcal{W}_1)} + r \cdot \underbrace{\mathbf{T}}_{\text{cross-term}} + r^2 \cdot \underbrace{(AZ_2) \circ (BZ_2)}_{\text{from } (\mathcal{U}_2, \mathcal{W}_2)}. \quad (3)$$

Folded instance (verifier-computable).

$$\bar{E}' = \bar{E}_1 + r \cdot \bar{T} + r^2 \cdot \bar{E}_2, \quad u' = u_1 + r \cdot u_2, \quad (4)$$

$$\bar{W}' = \bar{W}_1 + r \cdot \bar{W}_2, \quad \mathbf{x}' = \mathbf{x}_1 + r \cdot \mathbf{x}_2, \quad (5)$$

$$\bar{R}'_i = \bar{R}_{1,i} + r \cdot \bar{R}_{2,i}, \quad \bar{V}'_i = \bar{V}_{1,i} + r \cdot \bar{V}_{2,i}. \quad (6)$$

Folded witness (prover-only).

$$\mathbf{E}' = \mathbf{E}_1 + r \cdot \mathbf{T} + r^2 \cdot \mathbf{E}_2, \quad r'_E = r_{E_1} + r \cdot r_T + r^2 \cdot r_{E_2}, \quad (7)$$

$$\mathbf{W}' = \mathbf{W}_1 + r \cdot \mathbf{W}_2, \quad r'_W = r_{W_1} + r \cdot r_{W_2}. \quad (8)$$

Correctness follows from Pedersen's additive homomorphism:

$$\text{Com}(\mathbf{W}', r'_W) = \bar{W}_1 + r \cdot \bar{W}_2 = \bar{W}'. \quad (9)$$

Protocol (Prover)

Input. Real instance-witness pair $(\mathcal{U}_1, \mathcal{W}_1)$ with $u_1 = 1$ and $\mathbf{E}_1 = \mathbf{0}$ (non-relaxed, from actual sumcheck execution).

1. Sample random satisfying pair $(\mathcal{U}_2, \mathcal{W}_2, \mathbf{Z}_2)$:
 - Sample $\mathbf{W}_2 \xleftarrow{\$} \mathbb{F}_p^n$ with the same structural layout (coefficients, intermediates, next-claims per round).
 - Sample $u_2 \xleftarrow{\$} \mathbb{F}_p \setminus \{0\}$ and $\mathbf{x}_2 \xleftarrow{\$} \mathbb{F}_p$.
 - Compute $\mathbf{E}_2 = (A\mathbf{Z}_2) \circ (B\mathbf{Z}_2) - u_2 \cdot (C\mathbf{Z}_2)$ to force satisfaction.
 - Commit: $\bar{E}_2 = \text{Com}(\mathbf{E}_2, r_{E_2})$, $\bar{W}_2 = \text{Com}(\mathbf{W}_2, r_{W_2})$.
 - Extract round coefficients from \mathbf{W}_2 and commit: $\bar{R}_{2,i} = \text{Com}(\mathbf{c}_2^{(i)}, \rho_{2,i})$.
2. Compute cross-term \mathbf{T} and commit: $\bar{T} = \text{Com}(\mathbf{T}, r_T)$.
3. Fiat–Shamir challenge: append $(\mathcal{U}_1, \mathcal{U}_2, \bar{T})$ to the transcript; derive $r \leftarrow \mathcal{H}(\text{transcript})$.
4. Fold: compute $(\mathcal{U}', \mathcal{W}')$ as explained above.
5. Output proof: $\pi = (\mathcal{U}_1, \mathcal{U}_2, \bar{T}, \mathcal{W}')$.

Protocol (Verifier)

Input. Proof $\pi = (\mathcal{U}_1, \mathcal{U}_2, \bar{T}, \mathcal{W}')$ and the Fiat–Shamir transcript.

1. Check non-relaxed: verify $u_1 = 1$ and $\bar{E}_1 = \mathcal{O}$ (identity element).
2. Replay Fiat–Shamir: recompute r from $(\mathcal{U}_1, \mathcal{U}_2, \bar{T})$.
3. Recompute folded instance \mathcal{U}' from public data.
4. Commitment openings: verify
 - $\bar{W}' \stackrel{?}{=} \text{Com}(\mathbf{W}', r'_W)$,
 - $\bar{E}' \stackrel{?}{=} \text{Com}(\mathbf{E}', r'_E)$,
 - $\bar{R}'_i \stackrel{?}{=} \text{Com}(\mathbf{c}'^{(i)}, \rho'_i)$ for each round.

5. Coefficient consistency: verify the coefficients embedded in \mathbf{W}' match $\{\mathbf{c}'^{(i)}\}$ (prevents using different coefficient values in the R1CS witness vs. the committed round polynomials).
6. Evaluation commitment check (PCS binding): for each extra constraint i , verify

$$\bar{V}'_i = y'_i \cdot G + b'_i \cdot H, \quad (10)$$

where y'_i, b'_i are extracted from \mathbf{W}' .

7. R1CS satisfaction: verify

$$(A\mathbf{Z}') \circ (B\mathbf{Z}') = u' \cdot (C\mathbf{Z}') + \mathbf{E}'. \quad (11)$$

3.2 R1CS Encoding of the Sumcheck Verifier

To apply BlindFold, we need to encode all of the verifier's algebraic checks — across all stages and rounds — into a single R1CS instance $(A\mathbf{Z}) \circ (B\mathbf{Z}) = C\mathbf{Z}$, so that the BlindFold folding protocol can prove the sumcheck transcripts are valid without revealing them.

Let the Jolt Atlas proof consist of S stages, where stage s has n_s sumcheck rounds with degree- d_s univariate polynomials, and let $N = \sum_s n_s$ denote the total number of rounds. We define the witness vector \mathbf{Z} as:

$$\mathbf{Z} = \left[\underbrace{u}_{\text{scalar}}, \underbrace{r_1, \dots, r_N, \sigma_0^{(1)}, \dots, \sigma_0^{(k)}, \alpha_1, \dots, \gamma_1, \dots}_{\text{public inputs } \mathbf{x}}, \underbrace{c_0^{(1)}, c_1^{(1)}, \dots, t_1^{(1)}, \dots, \sigma_1^{(1)}, \dots}_{\text{private witness } \mathbf{W}} \right],$$

The public input region contains values known to both prover and verifier: the N Fiat-Shamir challenges r_j , the K initial claimed sums $\sigma_0^{(k)}$ (one per independent chain), batching coefficients α_j , and constraint challenge values γ_i . The private witness region contains, for each round j , the polynomial coefficients $(c_0^{(j)}, \dots, c_{d_s}^{(j)})$, the $d_s - 1$ Horner intermediates $(t_0^{(j)}, \dots, t_{d_s-2}^{(j)})$, and the next claimed sum σ_{j+1} . Additional witness slots hold polynomial evaluation values y_{ω_k} and auxiliary variables for output/input binding constraints (described below).

For each round j of stage s , the sumcheck prover sends coefficients $(c_0, c_1, \dots, c_{d_s})$ and receives a challenge $r_j \xleftarrow{\$} \mathbb{F}$. The verifier checks two relations:

1. *Sumcheck identity*:

$$2c_0 + c_1 + c_2 + \dots + c_{d_s} = \sigma_j,$$

where σ_j is the claimed sum (output of the previous round, or the initial claim σ_0 for the first round).

2. Horner evaluation (next claimed sum). Let $g_j(X) = \sum_{\ell=0}^{d_s} c_\ell X^\ell$ be the prover's degree- d_s univariate polynomial in round j . The verifier checks that the next claimed sum satisfies

$$\sigma_{j+1} = g_j(r_j) = c_0 + r_j(c_1 + r_j(c_2 + \dots + r_j c_{d_s})). \quad (12)$$

To encode this efficiently in R1CS, we introduce auxiliary variables t_0, \dots, t_{d_s-2} implementing Horner's rule:

$$t_{d_s-2} := c_{d_s-1} + r_j \cdot c_{d_s}, \quad (13)$$

$$t_{i-1} := c_i + r_j \cdot t_i \quad \text{for } i = d_s - 2, d_s - 3, \dots, 1, \quad (14)$$

$$\sigma_{j+1} := c_0 + r_j \cdot t_0. \quad (15)$$

Each multiplication in (13)–(15) is enforced by one multiplicative R1CS constraint of the form $\langle A, \mathbf{Z} \rangle \cdot \langle B, \mathbf{Z} \rangle = \langle C, \mathbf{Z} \rangle$:

$$\text{(innermost)} \quad \langle A, \mathbf{Z} \rangle = c_{d_s}, \quad \langle B, \mathbf{Z} \rangle = r_j, \quad \langle C, \mathbf{Z} \rangle = t_{d_s-2} - c_{d_s-1}, \quad (16)$$

$$\text{(middle)} \quad \langle A, \mathbf{Z} \rangle = t_i, \quad \langle B, \mathbf{Z} \rangle = r_j, \quad \langle C, \mathbf{Z} \rangle = t_{i-1} - c_i, \quad i = d_s - 2, \dots, 1, \quad (17)$$

$$\text{(final)} \quad \langle A, \mathbf{Z} \rangle = t_0, \quad \langle B, \mathbf{Z} \rangle = r_j, \quad \langle C, \mathbf{Z} \rangle = \sigma_{j+1} - c_0. \quad (18)$$

Equivalently, (16)–(18) enforce exactly the Horner recurrences (13)–(15).

Circuit size. Each sumcheck round contributes one sumcheck-identity constraint and d_s Horner-evaluation constraints, for a total of $d_s + 1$ constraints per round. Output and input-binding constraints add a number of gates proportional to the number of terms in the sum-of-products expression, which is bounded by a constant per stage. The total R1CS size is therefore

$$O\left(\sum_s n_s \cdot d_s\right) = O(N \cdot d_{\max}), \quad (19)$$

where N is the total number of sumcheck rounds across all stages and d_{\max} is the maximum polynomial degree. Since the number of sumcheck rounds is logarithmic in the original computation size (each round halves the domain), the BlindFold circuit is logarithmic in the size of the computation being proven.

3.3 Integration with Jolt Atlas

During stages prior to BlindFold, each sumcheck instance records the following data into an *opening accumulator* (a running collection of proof artifacts):

- Polynomial coefficients $\{c_0^{(j)}, c_1^{(j)}, \dots, c_d^{(j)}\}$ for each round j , representing the prover's univariate message $g_j(X)$.
- Round commitments $\bar{R}_j = \text{Com}(\mathbf{c}^{(j)}, \rho_j)$: Pedersen commitments to each round's coefficients, sent to the verifier during the sumcheck.
- Blinding factors ρ_j : the randomness used in each round commitment.
- Verifier challenges r_j : the Fiat–Shamir challenges derived after each round.
- Initial claimed sum $\sigma_0^{(s)}$ for each stage s .
- Input/output claim constraints: the algebraic relations that link each stage's initial claim (resp. final output) to polynomial evaluations from other stages (as described in the input/output binding sections).

- Polynomial evaluation values y_{ω_k} : retrieved from the accumulator by opening identifier ω_k .

From the extracted data, the prover constructs the BlindFold R1CS as follows:

1. **Stage configuration.** Each sumcheck round becomes one R1CS “stage” with its polynomial degree, chain linkage information, and (for the first and last rounds of each Jolt Atlas stage) input/output constraints.
2. **Constraint assembly.** The builder processes all stage configurations in sequence. For each stage it allocates private witness variables (coefficients, Horner intermediates, next-claim), emits the sum-check identity constraint and the Horner evaluation gates, and chains the round’s output variable to the next round’s input variable. Input and output constraints are attached to the first and last rounds of each Jolt Atlas stage respectively, with polynomial evaluations wired through the global opening map Φ .
3. **PCS binding constraint.** After all sumcheck stages, one additional constraint ties the sumcheck reductions to the polynomial commitment scheme. During the joint opening phase, all polynomial evaluations y_{ω_i} accumulated across stages are batched into a single claim:

$$y_{\text{joint}} = \sum_i \beta_i \cdot y_{\omega_i}$$

where β_i are batching coefficients derived from the Fiat-Shamir transcript. The PCS proof produces an evaluation commitment $\bar{V} = y_{\text{joint}} \cdot G + b \cdot H$, where b is a blinding factor and G, H are public generators.

The extra R1CS constraint asserts the linear relation above, using the same witness variables y_{ω_i} that appear in the stage output/input constraints, while \bar{V} is included in the relaxed R1CS instance so that the BlindFold verifier can check that the committed value matches the witness. This is the constraint that binds the sumcheck reductions to the polynomial commitment proof: without it, a prover could satisfy all sumcheck constraints with evaluation values that are inconsistent with the committed polynomials.

4. **Witness assignment.** The prover populates the full witness vector \mathbf{Z} by placing the initial claims and all Fiat-Shamir challenges into the public-input slots, and filling the private-witness slots with round coefficients, Horner intermediates, next-claims (computed via Horner evaluation), opening values, and PCS blinding factors.
5. **Instance construction.** A non-relaxed relaxed-R1CS instance $(\mathcal{U}_1, \mathcal{W}_1)$ (i.e., with $u = 1$ and $E = 0$) is formed with $u = 1$, $\mathbf{E} = \mathbf{0}$, the witness commitment $\bar{W} = \text{Com}(\mathbf{W}, r_W)$, all round commitments $\{\bar{R}_j\}$, the evaluation commitment \bar{V} from HyperKZG, and the public inputs. Pedersen generators are derived deterministically so that both prover and verifier use the same basis.
6. **BlindFold execution.** The prover runs the BlindFold protocol (Section 5) on $(\mathcal{U}_1, \mathcal{W}_1)$, producing a proof π that is appended to the Jolt Atlas proof.

Verification. The verifier reconstructs the same R1CS from the stage configurations (which are deterministic from the circuit structure and the Fiat–Shamir transcript). It then:

1. Recovers the public inputs—the sumcheck challenges $\{r_j\}$ and initial claims $\{\sigma_0^{(s)}\}$ —by replaying the Fiat–Shamir transcript from the main Jolt Atlas proof.
2. Reconstructs the real instance \mathcal{U}_1 by collecting round commitments $\{\bar{R}_j\}$ from the sumcheck proofs (which are part of the Jolt Atlas proof) and the evaluation commitment \bar{V} from the HyperKZG proof.
3. Runs the BlindFold verifier on π , which checks commitment openings, coefficient consistency, and relaxed-R1CS satisfaction on the folded instance.

The verifier never sees the polynomial coefficients, Horner intermediates, or evaluation values—only their commitments. BlindFold guarantees that these hidden values satisfy the sumcheck verification equations, completing the zero-knowledge property.

4 Lookup Arguments in Small Space

Non-linear operations like Softmax cannot be expressed as 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 essence, a lookup table $\mathcal{T} : \{0, 1\}^w \rightarrow \mathbb{F}$ 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.

4.1 Prefix-Suffix Decomposition of Large Lookup Tables

A naive lookup table for operations on two 64-bit inputs would require 2^{128} entries, which is infeasible. Jolt [1] exploits the observation that many operations—including addition, comparison, and bitwise logic—can be evaluated *chunk-by-chunk*: the result on the full input can be expressed as a function of results on small input segments.

Concretely, the input bits are partitioned into C segments $x = (x^{(1)}, \dots, x^{(C)})$, each of width b (e.g., $C = 8$ segments of $b = 8$ bits). For a table \mathcal{T} admitting this decomposition, the multilinear extension factors as

$$\tilde{\mathcal{T}}(x) = \sum_i \prod_{j=1}^C s_{i,j}(x^{(j)}), \quad (20)$$

where each subtable function $s_{i,j}$ operates on only b bits, reducing the total table size from 2^{Cb} to $O(C \cdot 2^b)$.

For a detailed treatment of which operations admit this decomposition and the resulting prover complexity, see Appendix A of [14].

Example: ReLU. For a w -bit signed integer with most significant bit b_0 , the ReLU function $\text{ReLU}(x) = \max(0, x)$ is entirely determined by the sign bit:

$$\text{ReLU}(x) = (1 - b_0) \cdot x.$$

Split the input into high bits x_{hi} (containing b_0) and low bits x_{lo} , each of width $w/2$. Since $x = \text{val}(x_{\text{hi}}) \cdot 2^{w/2} + \text{val}(x_{\text{lo}})$, we can write:

$$\begin{aligned} \text{ReLU}(x_{\text{hi}}, x_{\text{lo}}) &= (1 - b_0)(\text{val}(x_{\text{hi}}) \cdot 2^{w/2} + \text{val}(x_{\text{lo}})) \\ &= \underbrace{(1 - b_0) \cdot \text{val}(x_{\text{hi}}) \cdot 2^{w/2}}_{p_{\text{ReLU}}(x_{\text{hi}})} \cdot \underbrace{1}_{s_{\text{One}}(x_{\text{lo}})} + \underbrace{(1 - b_0)}_{p_{\text{NotMSB}}(x_{\text{hi}})} \cdot \underbrace{\text{val}(x_{\text{lo}})}_{s_{\text{ReLU}}(x_{\text{lo}})} \end{aligned}$$

The first term captures the high-order contribution (which vanishes for negative inputs), and the second passes through the low-order value gated by the sign indicator. Since this identity holds on all Boolean inputs, it extends to the multilinear extensions, giving the decomposition $\widetilde{\text{ReLU}}(r) = p_{\text{ReLU}}(r_{\text{hi}}) \cdot s_{\text{One}}(r_{\text{lo}}) + p_{\text{NotMSB}}(r_{\text{hi}}) \cdot s_{\text{ReLU}}(r_{\text{lo}})$.

4.2 Neural Teleportation for Lookup Table Compression

Activation functions like **erf** (used in GELU) and **tanh** require lookup tables spanning the full input range, which increases the computational costs of both prover and verifier. To mitigate this, Jolt Atlas draws on the idea of *neural teleportation* introduced by TeleSparse [12], adapting it into a simplified form suited to the proving setting.

TeleSparse defines a two-sided, per-neuron teleportation: for each neuron i with activation σ , it applies a scalar $\lambda_i > 0$, dividing the pre-activation input by λ_i and multiplying the output by a compensating factor, then rescaling the downstream weights accordingly. Because both the activation input and the subsequent weights are adjusted, the transform is lossless—the network computes the same function.

In the proving setting of Jolt Atlas’ implementation, per-neuron weight rescaling would require re-committing to modified weight tensors and complicates the circuit. Jolt Atlas therefore uses a *global, one-sided* approximation: every pre-activation input to a selected activation is divided by a single factor $\tau > 1$, with *no* compensating output multiplication and *no* weight rescaling. Concretely, the original computation $y = \sigma(x)$ is replaced by $y' = \sigma(x/\tau)$, which is a lossy approximation (i.e., $y' \neq y$ in general) but it is negligible in practice.

The key observation is that saturating activations like erf and tanh spend most of their operating range in the flat regions where $\sigma(x) \approx \pm 1$. For any input already in the saturation region, $\sigma(x/\tau) = \sigma(x)$, so the approximation is exact.

Error is concentrated in the narrow linear region near the origin, where it is bounded and small relative to the output scale. In typical trained networks, the vast majority of pre-activation values fall in the saturation region, so the global one-sided transform introduces only minor distortion.

With $\tau = 4$, the effective input range shrinks by $4\times$, from $[-R, R]$ to $[-R/4, R/4]$, and the lookup table is bounded by 2^{16} entries (sufficient for 16-bit fixed-point activations). Empirical evaluation shows that $\tau = 4$ introduces output differences of less than 55 units on a 128-scale fixed-point representation, acceptable for most inference tasks. The factor τ can be tuned per-model to balance table size against accuracy.

5 ONNX: The Bridge to Machine Learning

ONNX (Open Neural Network Exchange) serves as the standardised 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.

5.1 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.

The ONNX format represents computations as directed acyclic graphs where:

- **Nodes** represent operations (Add, MatMul, ReLU, etc.)
- **Edges** represent tensors flowing between operations
- **Attributes** parameterise operations (axis specifications, einsum patterns)
- **Initialisers** store model weights and biases

5.2 Operator Support

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

- **Elementwise Arithmetic:** Add, Sub, Mul, Div
- **Activations (via lookup tables):** Relu, Sigmoid, Softmax, Tanh, Erf
- **Comparisons:** Gte, Eq
- **Math Functions:** Rsqrt (reciprocal square root)
- **Tensor Operations:**
 - Shape manipulation: Reshape, Broadcast (automatic, NumPy semantics)
 - Data movement: Gather
 - Control flow: Select (conditional element selection)
- **Reduction Operations:**

- **ReduceSum** along arbitrary axes
- **ReduceMean** (decomposed internally into **ReduceSum** + **Div**)
- **Tensor Contraction:** **Einsum** (supports patterns like ‘**mk,kn->mn**’ (matrix multiply), ‘**bmk,bkn->bmn**’ (batched multiply), and attention patterns)

5.3 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.
4. **Proof generation:** The trace feeds into the Jolt Atlas prover, which generates a zero-knowledge proof.

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.

5.4 CPU trace vs Tensor trace

Currently, to verify a tensor operation such as add, mul, or ReLU, we take the ONNX trace, decompose it into a CPU trace, and feed that into the Jolt proof system. Multilinear polynomials are a natural representation for tensors; however, when we decompose the trace, we lose the expressiveness and structure that polynomial algebra provides.

Since multilinear polynomials naturally represent tensors, we instead verify tensor relations directly at the polynomial level. By applying the Schwartz–Zippel lemma, we can probabilistically check that a tensor operation is correct without inspecting every individual element.

The core ethos is very much Jolt-like (arguably even more so than Jolt itself): reducing commitment cost via virtual polynomials and reducing R1CS constraints via lookups. Almost all of our checks are performed using virtual polynomials, and we do not use or require R1CS.

6 Benchmarks

System. All benchmarks were run on a MacBook Pro (Apple M3) with 16 GB RAM.

Table 1. JOLT Atlas end-to-end proving breakdown for nanoGPT.

Stage	Wall clock
Verifying key generation	0.246 s
Proving key generation	0.246 s
Proof time	14 s
Verify time	0.517 s

Table 2. ezkl proving breakdown for nanoGPT (reported).

Stage	Wall clock
Verifying key generation	192 s
Proving key generation	212 s
Proof time	237 s
Verify time	0.34 s

6.1 nanoGPT (~ 0.25 M parameters, 4 transformer layers)

nanoGPT is the standard workload we use for cross-project comparison. It is a ~ 250 k-parameter GPT model with 4 transformer layers.

JOLT Atlas (end-to-end).

ezkl (same model). We report ezkl timings for the same model from their published benchmark.¹

Comparison. JOLT Atlas produces a proof for nanoGPT in ~ 14 s versus ezkl’s ~ 237 s proof time (not counting their >400 s of key generation), corresponding to roughly a $17\times$ speed-up on proof generation alone.

6.2 GPT-2 (125M parameters)

GPT-2 is a 125-million-parameter transformer model from OpenAI.

JOLT Atlas (end-to-end).

7 Future Work

Directions for future research and development include:

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

¹ <https://blog.ezkl.xyz/post/nanogpt/>

Table 3. JOLT Atlas end-to-end proving breakdown for GPT-2 (125M).

Stage	Wall clock
Proving/verifying key generation	0.872 s
Witness generation	~7.5 s
Commitment time	~3.5 s
Sum-check proving	~16 s
Reduction opening proof	~7 s
HyperKZG prove	~3 s
End-to-end total	~38 s

Hachi [15] is a lattice-based multilinear PCS. Compared to HyperKZG, 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 $12.5\times$ faster than Greyhound [16]. 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, *ABBA* [4] reduces commitment sizes through lattice-based commitment constructions built from commutators of quaternions. Smaller commitments directly benefit both proof size and verifier efficiency.

- **On-Chain Verification:** Deploying proofs on Ethereum is challenging due to HyperKZG’s \mathbb{G}_T operations. Replacing HyperKZG with a lattice-based PCS may also mitigate this limitation.
- **Extended ONNX Support:** Adding support for additional operators.
- **Adaptive teleportation for prefix-suffix efficiency.** In the current design, TeleSparse employs a fixed global teleportation factor τ to shrink activation ranges and thereby bound lookup-table size. A natural direction for future work is to *optimize* τ jointly with the prefix-suffix parameters, rather than fixing it a priori. In particular, the prefix-suffix decomposition in Eq. 20 exposes an explicit trade-off between the number of chunks C , the per-chunk bit-width, and the resulting prover time and memory footprint. Since the effective activation range after teleportation directly determines the required bit-width (and hence the minimal feasible C), the choice of τ implicitly controls the cost profile of the lookup argument.

Future work could therefore aim to select τ so as to minimize overall proving time by balancing: (i) the streaming overhead induced by larger C in Eq. 20, (ii) the size of the activation lookup domain after range reduction, and (iii) the induced quantization and approximation error in the model. Beyond reducing the raw range, an optimized τ may also reshape the activation input distribution (e.g., by increasing mass in saturated regions for functions such as \tanh), which can further improve the effectiveness of the prefix-suffix factorization by reducing the “active” portion of the lookup ta-

ble. Developing principled methods to tune τ under explicit proof-cost and accuracy constraints remains an open and promising direction.

The rapid development of research in both cryptography and artificial intelligence continues to provide new techniques and optimisations to integrate into Jolt Atlas. Advances in polynomial commitment schemes, folding techniques, and hardware acceleration on the cryptographic side, combined with new model architectures, quantisation methods, and inference optimisations 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. 2023. URL: <https://eprint.iacr.org/2023/1217>.
- [2] Manuel Blum, Will Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. “Checking the Correctness of Memories”. In: *Algorithmica* 12.2/3 (1994), pp. 225–244.
- [3] Jonathan Bootle, Alessandro Chiesa, Yuncong Hu, and Michele Orrù. *Gemini: Elastic SNARKs for Diverse Environments*. Cryptology ePrint Archive, Paper 2022/420. 2022. URL: <https://eprint.iacr.org/2022/420>.
- [4] Alberto Centelles and Andrew Mendelsohn. *ABBA: Lattice-based Commitments from Commutators*. Cryptology ePrint Archive, Paper 2026/148. <https://eprint.iacr.org/2026/148>. 2026.
- [5] a16z crypto. *Big Ideas in Tech 2026*. <https://a16zcrypto.com/posts/article/big-ideas-things-excited-about-crypto-2026/>. 2026.
- [6] DeepProve. URL: <https://github.com/Lagrange-Labs/deep-prove>.
- [7] Benjamin E. Diamond and Jim Posen. *Succinct Arguments over Towers of Binary Fields*. Cryptology ePrint Archive, Paper 2023/1784. <https://eprint.iacr.org/2023/1784>. 2023.
- [8] Jens Groth. *On the Size of Pairing-based Non-interactive Arguments*. Cryptology ePrint Archive, Paper 2016/260. 2016. URL: <https://eprint.iacr.org/2016/260>.
- [9] Darya Kaviani and Srinath Setty. *Vega: Low-Latency Zero-Knowledge Proofs over Existing Credentials*. Cryptology ePrint Archive, Paper 2025/2094. 2025. URL: <https://eprint.iacr.org/2025/2094>.
- [10] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. *Nova: Recursive Zero-Knowledge Arguments from Folding Schemes*. Cryptology ePrint Archive, Paper 2021/370. 2021. URL: <https://eprint.iacr.org/2021/370>.
- [11] Tianyi Liu, Zhenfei Zhang, Yuncong Zhang, Wenqing Hu, and Ye Zhang. *Ceno: Non-uniform, Segment and Parallel Zero-knowledge Virtual Machine*. Cryptology ePrint Archive, Paper 2024/387. <https://eprint.iacr.org/2024/387>. 2024.

- [12] Mohammad M. Maheri, Hamed Haddadi, and Alex Davidson. *TeleSparse: Practical Privacy-Preserving Verification of Deep Neural Networks*. arXiv preprint arXiv:2504.19274. <https://arxiv.org/abs/2504.19274>. 2025.
- [13] Jason Morton et al. “EZKL: Easy Zero-Knowledge Machine Learning”. In: *GitHub Repository*. URL: <https://github.com/zkonduit/ezkl>.
- [14] Vineet Nair, Justin Thaler, and Michael Zhu. *Proving CPU Executions in Small Space*. Cryptology ePrint Archive, Paper 2025/611. 2025. URL: <https://eprint.iacr.org/2025/611>.
- [15] Ngoc Khanh Nguyen, George O’Rourke, and Jiapeng Zhang. *Hachi: Efficient Lattice-Based Multilinear Polynomial Commitments over Extension Fields*. Cryptology ePrint Archive, Paper 2026/156. <https://eprint.iacr.org/2026/156>. 2026.
- [16] Ngoc Khanh Nguyen and Gregor Seiler. *Greyhound: Fast Polynomial Commitments from Lattices*. Cryptology ePrint Archive, Paper 2024/1293. 2024. DOI: 10.1007/978-3-031-68403-6_8. URL: <https://eprint.iacr.org/2024/1293>.
- [17] a16z Research. *Jolt Documentation*. URL: <https://jolt.a16zcrypto.com>.
- [18] Justin Thaler. *Proofs, Arguments, and Zero-Knowledge*. URL: <https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK>.
- [19] Justin Thaler. *Sum-check Is All You Need: An Opinionated Survey on Fast Provers in SNARK Design*. Cryptology ePrint Archive, Paper 2025/2041. 2025. URL: <https://eprint.iacr.org/2025/2041>.
- [20] Dmytro Zakharov, Oleksandr Kurbatov, Artem Sdobnov, Lev Soukhanov, et al. *Bionetta: Efficient Client-Side Zero-Knowledge Machine Learning Proving*. arXiv preprint arXiv:2510.06784. 2025.
- [21] Jiaying Zhao, Srinath Setty, Weidong Cui, and Greg Zaverucha. *MicroNova: Folding-based Arguments with Efficient (On-chain) Verification*. Cryptology ePrint Archive, Paper 2024/2099. <https://eprint.iacr.org/2024/2099>. 2024.