

zkRNN: Zero-Knowledge Proofs for Recurrent Neural Network Inference

Fatemeh Zarinjoui
Sharif University of Technology
Tehran, Iran
ftm.zariin@gmail.com

Maryam Zarezadeh
Barkhausen Institut
Dresden, Germany
maryam.zarezadeh@barkhauseninstitut.org

Behzad Abdolmaleki
University of Sheffield
Sheffield, United Kingdom
behzad.abdolmaleki@sheffield.ac.uk

Bhavish Mohee
University of Sheffield
Sheffield, United Kingdom
bhavishmohee@gmail.com

Aysajan Abidin
COSIC, KU Leuven
Leuven, Belgium
aysajan@kuleuven.be

Stefan Köpsell
Barkhausen Institut
Dresden, Germany
stefan.koepsell@barkhauseninstitut.org

Abstract—Zero-knowledge proofs (ZKPs) enable verification of computations without revealing private information and have recently been applied to neural-network inference. Existing approaches, however, primarily target feedforward and convolutional architectures and do not naturally extend to recurrent neural networks (RNNs), whose sequential dependencies and shared parameters introduce new verification challenges. We present zkRNN, a zero-knowledge framework for verifiable RNN inference that allows a prover to demonstrate the correctness of model outputs without revealing model parameters, intermediate hidden states, or private inputs. Our approach extends the GKR protocol with a recursive sum-check framework tailored to recurrent computation. We introduce an arithmetic-circuit representation that captures hidden-state transitions and weight sharing across time steps and employ recursive proof composition to aggregate proofs generated at each recurrent step. As a result, verifier time and proof size remain polylogarithmic in the size of a single-step circuit and independent of sequence length. We further support streaming inference through incremental proof generation and aggregation, enabling low-latency verification for sequential workloads. Experimental results demonstrate efficient proof generation and sequence-length-independent verification across realistic RNN configurations. For sequence lengths up to $T = 256$, zkRNN maintains a proof size of roughly 18 KB, while prover time scales linearly with sequence length and verification completes in under 10 seconds. These results broaden the applicability of verifiable machine learning to sequential models and time-dependent data.

Index Terms—Zero-knowledge proof, Recurrent neural network, Verifiable inference, IVC, Machine learning, RNN, ML.

1. Introduction

As ML systems are increasingly deployed in privacy-sensitive domains such as healthcare, finance, and government, ensuring correct predictions is critical [1], [2].

Since prediction providers (e.g., cloud services) may be untrusted [3], [4], it is essential to verify prediction correctness without revealing model parameters and intermediate states. Zero-knowledge proofs (ZKPs) provide a cryptographic foundation for verifiable ML. A ZKP enables a prover to convince a verifier of a statement’s validity without revealing any additional information [5]. Recent work has applied ZK and succinct proof systems to verify ML inference and training without re-executing expensive computations, advancing the field of verifiable machine learning (VML) [6], [7], [8]. Efficient ZKP-based verification has been demonstrated for feedforward and convolutional neural networks (CNNs) [3], [9], [10], primarily using the GKR protocol and its variants [11]. These methods represent neural networks as layered arithmetic circuits over finite fields, enabling efficient proof generation and verification for large models. However, they assume acyclic computation graphs, where each layer depends only on preceding layers and no information flows backward.

Recurrent Neural Networks (RNNs) introduce feedback through hidden states, updating their state at each time step based on the current input and previous state. This sequential structure makes them effective for sequence modeling tasks such as natural language processing, time-series forecasting, and control systems [12], [13]. However, RNNs pose significant challenges for ZK frameworks because their computations involve recurrent dependencies, shared weights across time steps, and dynamically evolving hidden states. Existing approaches often require unrolling the network over time, leading to substantially larger circuits and reduced proof efficiency. This disconnect between cryptographic verifiability and sequential learning leads to the question: Can we create a ZKP system that verifies RNN inference while remaining asymptotically efficient and zero-knowledge? In particular, the verifier’s time and proof size should not grow linearly with the sequence length, and the proof should reveal no information about the model parameters and intermediate hidden states. Solving this problem requires rethinking how recurrent computations are represented within

arithmetic circuits and how proofs can be composed across dependent time steps. In this work, we present zkRNN, a ZKP framework for verifiable inference over RNNs. zkRNN bridges the gap between ZK theory and sequential ML by creating a circuit representation that captures hidden state transitions, unrolled computation across time steps, and shared weights in a way that works with sumcheck-based verification [11]. Our approach builds on recursive proof techniques and sumcheck-inspired constructions, but introduces key innovations that make ZK verification of RNNs both feasible and scalable. Specifically, zkRNN features a recursive architecture that treats each RNN time step as an independently provable component. This modular design allows verification of each step’s computation—made up of linear transformations and non-linear activations—using efficient sumcheck protocols, while keeping steps consistent through commitments to the changing hidden state. We also develop a generalized arithmetic circuit representation that captures shared weights and temporal dependencies, and adjust recursive proof composition techniques to combine per-step proofs logarithmically in relation to the sequence length. Beyond the main proof system, zkRNN includes a streaming verification mode designed to reduce proof-availability delay in sequence-based applications. Instead of a “compute-then-prove” approach, zkRNN provides inference outputs right as inputs arrive, while generating and recursively combining proofs in the background. This “prove-as-you-go” model significantly lowers latency and makes verifiable inference practical for online translation, event prediction, and secure time-series analysis. It should be mentioned that zkRNN provides privacy guarantees only for *inference*. Specifically, it hides the model weights and the hidden-state values, ensuring that none of these are revealed in the ZKP. However, zkRNN does not protect the training data or any information arising during training (e.g., gradients or optimization updates). Zero-knowledge protection for training data is the goal of ZK training protocols, whereas this work focuses solely on ZKPs of inference for recurrent neural networks.

1.1. Our Contributions

zkRNN introduces a novel, end-to-end framework for model-private verifiable inference over recurrent architectures. It preserves the succinctness and zero-knowledge properties of ZK systems while extending their applicability to models with temporal structure and stateful computation. Our main contributions are summarized as follows:

- *Zero-knowledge inference for RNNs.* zkRNN provides a ZKP framework specifically designed for recurrent neural network inference. Unlike prior zkML systems that primarily target feedforward architectures, zkRNN explicitly handles hidden-state transitions, temporal dependencies, and weight sharing across time steps within an arithmetic-circuit representation compatible with sumcheck-based verification.
- *Integrated recursive verification of sequential computations.* We introduce a recursive proof architecture for

recurrent computation that verifies hidden-state evolution across time while preserving consistency between consecutive states. By combining recursive proof composition with commitment aggregation, zkRNN achieves verifier complexity and proof size that remain independent of sequence length and scale only polylogarithmically with the size of the final-step computation.

- *Streaming, low-latency proof architecture.* zkRNN incorporates a streaming execution mode that outputs inference results as inputs arrive while proofs are generated and recursively aggregated in parallel. This “prove-as-you-go” design aligns proof generation with sequential inference workloads and reduces end-to-end verification latency.
- *Prototype implementation and empirical validation.* We implement zkRNN and demonstrate that this single framework achieves efficient proof generation and succinct verification across realistic RNN workloads. We evaluate quantized inference across RNN models with hidden size up to $h = 256$ and sequence length up to $T = 2^8 = 256$. The prover time scales linearly with T and with h , and verification ranges from tens of milliseconds to under 14 seconds over our tested configurations. The final recursive-root size is a few kilobytes and independent of T , empirically validating succinct and sequence-length-independent verification for sequential inference.

By enabling ZKPs for RNN inference, zkRNN extends VML to sequential tasks such as time-series analysis and NLP. It preserves privacy and soundness while remaining efficient for long input sequences, advancing practical VML for recurrent neural networks.

1.2. Technical Overview

In this section, we provide an overview of zkRNN, our ZKP framework for verifying RNN inference. Our design tackles the main challenges of applying ZKP methods, typically designed for feedforward and convolutional networks, to the sequential and stateful computations of RNNs. The framework combines optimized sumcheck protocols, GKR-style proofs [11], polynomial commitment methods [14], and recursive proof composition to allow for efficient and scalable zero-knowledge verification across long input sequences.

Proof Construction at Each Time Step. Each RNN time step consists of two main computation stages: linear transformations (matrix-vector multiplications) and non-linear activations. We combine the linear operations, including weight multiplications and bias additions, into a single matrix-vector multiplication using an enhanced input vector and weight matrix. This approach allows us to apply Thaler’s time-optimal sumcheck protocol [15], resulting in lower prover complexity and efficiency improvements compared to standard circuit representations.

GKR-Based Verification of Nonlinear Activations. For non-linear activations, such as ReLU and tanh, we use GKR-based proofs enhanced with the Dao–Thaler sumcheck technique [16]. Non-linear functions are represented as arithmetic circuits and verified using bit-decomposition and con-

sistency checks. This ensures correctness while minimizing overhead. We express binary constraints and sign checks as specific sumcheck instances, preventing circuit growth and aiding succinctness.

Recursive Proof Composition and Commitment Aggregation. To enable efficient verification over long sequences, we recursively combine per-step proofs. This method reduces proof size from linear to constant in relation to the number of time steps. However, recursion brings challenges, especially for polynomial commitment openings. We tackle these challenges with an aggregation scheme based on evaluation reduction and the Orion polynomial commitment method [14]. We merge several evaluation proofs into a single proof, verified only at the last recursion step, achieving an overhead complexity of $O(\log^2(h))$ time for the verifier, where h is the hidden state size. This approach maintains scalability without sacrificing zero-knowledge security. In particular, zkRNN maintains a prover time linear in the number of time steps, while the verifier and proof size remain polylogarithmic in the final iteration circuit size and independent of the sequence length.

Streaming Architecture for Incremental Proof Generation. A core design principle of zkRNN is its compatibility with the streaming nature of real-world RNN tasks. Rather than adopting a monolithic ‘compute-then-prove’ model, our framework returns inference outputs as inputs arrive while the corresponding zero-knowledge proofs are generated and recursively folded in the background. This ‘prove-as-you-go’ approach enables proof generation to proceed incrementally as inputs are processed, rather than starting only after the full sequence has been evaluated. The final proof may still become available after the corresponding inference outputs, but overlapping proof generation with inference can reduce this proof-availability delay compared with a monolithic compute-then-prove design.

Relationship to Trace-Alignment Formulations. A natural approach to proving RNN inference is to flatten the entire execution trace into a time-indexed matrix and enforce transition constraints between consecutive hidden states using row-alignment relations. While such a formulation correctly captures recurrence, it produces a monolithic computation whose proving and verification procedures remain tied to the full execution trace. In contrast, zkRNN treats each recurrent transition as an independently provable computation and composes these proofs recursively. This separation enables proof aggregation, streaming execution, and sequence-length-independent verification without requiring the verifier to reason about the entire unrolled computation at once. Consequently, the contribution of zkRNN is not the representation of recurrence itself, but the integration of recurrent state transitions with recursive proof composition and commitment aggregation in a manner tailored to verifiable inference.

Efficiency and Soundness Enhancements. We also boost performance and security with several techniques: *i) Evaluation reduction:* condenses multiple evaluation checks into a single sumcheck instance, thereby lowering verifier circuit size. *ii) Sumcheck-friendly Fiat–Shamir hashing:* uti-

lizes ZK-friendly hash functions with low multiplicative complexity, such as MiMC [17], to enable efficient non-interactive transformation. *iii) Tree-based recursion:* ensures soundness across numerous time steps by verifying proofs in a logarithmic-depth tree structure.

Overall, zkRNN advances ZKP methods into sequential neural inference by integrating optimized sumcheck protocols, generalized GKR circuits, and recursive composition with commitment aggregation. This cohesive framework enables succinct, scalable, and privacy-preserving verification of RNN computations over long input sequences, significantly expanding the range of VML. The formal recursive framework for RNN inference proofs is presented in Appendix D.

1.3. Related Work

SafetyNets [3] was the first framework for verifiable DNN inference using interactive proofs. By combining the GKR protocol with a matrix-multiplication proof system [18], it achieves efficient verification but supports only quadratic activations and sum pooling, limiting flexibility. Moreover, it does not provide zero-knowledge guarantees since both the model and inputs are known to the verifier. vCNN [19] accelerates CNN verification using zk-SNARKs, employing quadratic polynomial programs [20] for convolutional layers, quadratic arithmetic programs [21] for other components, and commit-and-prove SNARKs [22] to link them. While effective for small models, its verification and storage costs increase with model size. Mystique [23], built on VOLE-based proofs [24], introduces optimized ZKP building blocks, efficient protocols for arithmetic/Boolean conversion, commitment integration, and numeric representation reconciliation, as well as improved matrix-multiplication proofs. Le et al. [25] further combine VOLE [26] with range, exponent, and lookup proofs to reduce the cost of non-linear layers, while Hao et al. [27] improve ZK verification of non-linear functions using lookup-based techniques and digital decomposition. VerfCNN [28] proposes an optimized zkSNARK framework for CNN inference with efficient multi-channel convolution verification. However, its focus on feedforward convolutional architectures differs fundamentally from zkRNN, which targets recurrent computation with hidden-state evolution, temporal dependencies, and parameter sharing. Li and Fan’s SUMMER [29] introduces recursive ZK proofs for RNN training, proving forward passes, gradient computation, and parameter updates through recursive composition. In contrast, zkRNN focuses on ZK proofs of RNN inference, verifying sequential evaluation while hiding model parameters and intermediate states. Although both systems employ recursion and sumcheck-based techniques, they address distinct problems—training versus inference—and make different design choices for commitments, non-linearities, and proof generation. Modular Sumcheck Proofs [30] presented an initial construction for verifiable RNN inference within a general modular sumcheck/GKR framework. While demonstrating feasibility, the framework was not specifically opti-

TABLE 1. COMPARISON OF ZKRNN WITH REPRESENTATIVE RECURSIVE PROOF SYSTEMS.

Feature	HI [34]	Nova [31]	SN [32]	HN [33]	zkRNN
Recursive proof composition	✓	✓	✓	✓	✓
Incremental / iterative computation	Partial	✓	✓	✓	✓
Stateful computation support	Partial	✓	✓	✓	✓
Explicit RNN modeling	✗	✗	✗	✗	✓
Hidden-state transition representation	✗	✗	✗	✗	✓
Native weight-sharing treatment for RNNs	✗	✗	✗	✗	✓
Streaming / prove-as-you-go RNN inference	✗	✗	✗	✗	✓

HI=Halo Infinite, SN = SuperNova; HN = HyperNova.

Note: ✗ indicates no explicit support for the feature, though it may be encoded indirectly via circuits or constraints.

mized for recurrent inference. In contrast, zkRNN develops a dedicated framework for scalable zero-knowledge RNN inference through recursive proof composition, commitment aggregation, and streaming verification.

Comparison with Existing Recursive Proof Systems. General recursive proof systems such as Nova [31], SuperNova [32], HyperNova [33], and Halo Infinite [34] provide iteration-succinct proofs for arbitrary computations and can, in principle, support RNN inference. As summarized in Table 1, these schemes support recursive proof composition and can be applied to iterative computations. However, unlike frameworks that treat RNN inference as a generic recursive computation, zkRNN is specifically designed for verifiable RNN inference, introducing explicit representations of recurrent hidden-state transitions, dedicated support for weight sharing across time steps, and a streaming proof-generation architecture tailored to sequential inference workloads. Table 1 provides a detailed comparison between zkRNN and Halo Infinite, Nova, SuperNova, and HyperNova [33].

2. Preliminaries

Notation. Let λ be the security parameter and $\text{negl}(\cdot)$ a negligible function. All arithmetic is over a finite field $\mathbb{F}$, and PPT denotes probabilistic polynomial time. For $m \in \mathbb{N}$, let $[m] = 0, \dots, m-1$. Bold lowercase (uppercase) letters denote vectors (matrices); e.g., $\mathbf{v} = (v_1, \dots, v_\ell) \in \mathbb{F}^\ell$. We use x_t (or x_i) for the RNN input at time step t .

Interactive Proofs. An interactive proof is a protocol between a computationally unbounded prover P and a probabilistic polynomial-time verifier V . The prover seeks to convince the verifier that a computation $C(x) = y$ was performed correctly.

Multilinear Extensions. Let $V : \{0, 1\}^\ell \rightarrow \mathbb{F}$ be a function defined over the Boolean hypercube. The *multilinear extension* $\tilde{V} : \mathbb{F}^\ell \rightarrow \mathbb{F}$ is the unique multilinear polynomial that agrees with V on all Boolean inputs; that is, for all $\mathbf{b} \in \{0, 1\}^\ell$, we have $\tilde{V}(\mathbf{b}) = V(\mathbf{b})$. This extension can be written explicitly as: $\tilde{V}(\mathbf{x}) = \sum_{\mathbf{b} \in \{0, 1\}^\ell} V(\mathbf{b}) \cdot \beta_{\mathbf{b}}(\mathbf{x})$,

where $\beta_{\mathbf{b}} : \mathbb{F}^\ell \rightarrow \mathbb{F}$ is the multilinear Lagrange basis polynomial associated with the point $\mathbf{b}$, given by: $\beta_{\mathbf{b}}(\mathbf{x}) = \prod_{i=1}^\ell ((1 - x_i)(1 - b_i) + x_i b_i)$. In practice, many objects such as vectors and matrices can be viewed as functions over the Boolean hypercube and naturally extended in this way. Specifically:

- A vector $\mathbf{v} = (v_0, v_1, \dots, v_{n-1}) \in \mathbb{F}^n$, where $n = 2^k$, can be interpreted as a function $v : \{0, 1\}^k \rightarrow \mathbb{F}$ mapping the binary representation of index i to v_i . Its multilinear extension $\tilde{v}$ is then defined over $\mathbb{F}^k$.
- A matrix $A \in \mathbb{F}^{n \times m}$, where $n = 2^k$ and $m = 2^l$, can be interpreted as a function $A : \{0, 1\}^{k+l} \rightarrow \mathbb{F}$, and its multilinear extension $\tilde{A}$ is defined over $\mathbb{F}^{k+l}$.

Incrementally Verifiable Computation (IVC). IVC, introduced by Valiant [35], enables the construction of succinct proofs for iterative computations, where each step builds upon the previous one. Let F be a deterministic function modeling one step of a computation, and let $\omega_0, \dots, \omega_{i-1}$ be auxiliary inputs (or witnesses) used at each step. Define a sequence of outputs $\{z_k\}_{k=0}^i$ via $\forall k \in [i] : z_{k+1} = F(z_k, \omega_k)$. An IVC scheme allows a prover to incrementally produce a proof π_i attesting that there exists a sequence of witnesses $\omega_0, \dots, \omega_{i-1}$ such that z_i was computed correctly from z_0 through i applications of F . Importantly, the cost of verification is independent of the number of iterations i , and the prover maintains only a small memory footprint. Formally, an IVC scheme consists of the following three algorithms:

- $G(1^\lambda) \rightarrow \text{pp}$: On input the security parameter λ , outputs public parameters pp .
- $P(i, z_0, z_{i-1}, \omega_{i-1}, \pi_{i-1}, \text{pp}) \rightarrow (z_i, \pi_i)$: On input the iteration index i , initial value z_0 , previous output z_{i-1} , auxiliary input ω_{i-1} , previous proof π_{i-1} , and public parameters pp , outputs the new output z_i and a new proof π_i .
- $V(i, z_0, z_i, \pi_i, \text{pp}) \rightarrow \{\text{accept}, \text{reject}\}$: On input the iteration index i , initial value z_0 , current output z_i , proof π_i , and public parameters pp , the verifier accepts if and only if the proof is valid.

Each proof π_i is a (zero-knowledge) argument for the relation:

$$\mathcal{R}_i^{\text{IVC}} = \{((i, z_0, z_i), (\omega_0, \dots, \omega_{i-1})) \mid \forall k \in [i] : z_{k+1} = F(z_k, \omega_k)\}$$

In this paper, we used the method for constructing IVC via recursive proof composition. Given a succinct non-interactive argument system (P_b, V_b) for a baseline relation, we define an augmented function F_A as:

$$F_A(i+1, z_0, z_i, \omega_i, \pi_i) = (F(z_i, \omega_i), V_b(i, z_0, z_i, \pi_i))$$

The prover computes $z_{i+1} = F(z_i, \omega_i)$ and invokes P_b to generate a new proof π_{i+1} certifying that F_A evaluated correctly. This construction ensures that the size of the proof and the cost of verification remain constant across iterations, while enabling verification of the full computation trace up to step $i+1$.

Sumcheck Protocol. The sumcheck protocol [36] is a fundamental interactive proof for verifying claims of the

form $H = \sum_{\mathbf{b} \in \{0,1\}^\ell} f(\mathbf{b})$, where $f : \mathbb{F}^\ell \rightarrow \mathbb{F}$ is a multivariate polynomial. Over ℓ rounds, the protocol reduces this high-dimensional sum to the evaluation of f at a random point $\mathbf{r} \in \mathbb{F}^\ell$. In each round, the prover sends a univariate polynomial encoding a partial sum, and the verifier checks consistency before sampling a random challenge. Further details are given in Fig. 10 (Appendix B).

Evaluation Reduction Sumcheck. To verify k evaluations $\{(x_i, y_i)\}_{i=1}^k$ of a multilinear polynomial f , we use the evaluation-reduction technique [11]. The verifier samples random coefficients $\alpha_1, \dots, \alpha_k$ and forms $T = \sum_{i=1}^k \alpha_i y_i$. Using the multilinear basis polynomial $\tilde{\beta}$, the evaluations can be aggregated as

$$\sum_{i=1}^k \alpha_i f(x_i) = \sum_{z \in \{0,1\}^\ell} f(z) \underbrace{\left(\sum_{i=1}^k \alpha_i \tilde{\beta}(z, x_i) \right)}_{L(z)}$$

Therefore, verifying all k evaluations reduces to a single sumcheck instance for $\sum_{z \in \{0,1\}^\ell} f(z) L(z) = T$, reducing verifier work to essentially that of one evaluation check.

The GKR protocol. The GKR protocol [11] is an interactive proof system for verifying computations represented as layered arithmetic circuits over a finite field $\mathbb{F}$. Let v_i denote the gate values at layer i , and let add_i and mult_i describe the circuit wiring. The layer values satisfy

$$v_i(z) = \sum_{x,y} \left[\text{add}_i(z, x, y) (v_{i+1}(x) + v_{i+1}(y)) + \text{mult}_i(z, x, y) v_{i+1}(x) v_{i+1}(y) \right]. \quad (1)$$

Given a claimed output, the verifier recursively checks (1) from the output layer to the input layer using the sumcheck protocol (Fig. 11 in Appendix B). At each layer, a random evaluation point reduces verification to a small number of claims about the next layer. After d iterations, the remaining claims are checked directly at the input layer. GKR achieves logarithmic verifier time and communication in the circuit size and forms a core building block of many modern ZK systems. Combined with ZK sumcheck protocols [37], [38], polynomial commitments, and the Fiat-Shamir transform [39], it yields efficient non-interactive zero-knowledge proofs.

Recurrent Neural Networks (RNNs). A RNN [40] is a neural architecture for sequential data that maintains a hidden state across time steps. Given an input sequence $x_1, \dots, x_T$, the hidden state is updated as

$$a_t = F_1(W_{aa}a_{t-1} + W_{ax}x_t + b_a)$$

and the output is computed as

$$y_t = F_2(W_{ya}a_t + b_y)$$

where W_{aa}, W_{ax} , and W_{ya} are weight matrices, b_a and b_y are bias vectors, and F_1, F_2 are activation functions. RNN computation is inherently sequential since a_t depends on a_{t-1} , enabling the model to capture temporal dependencies but limiting parallelization. Figure 1 illustrates this recurrent structure. Several variants have been proposed to

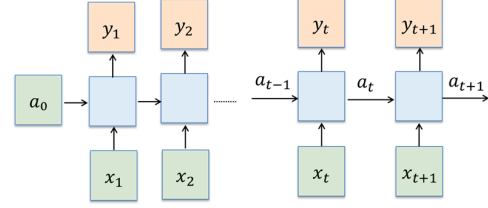


Figure 1. Simple RNN processing a sequential input.

address long-term dependency issues, notably Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRUs), which introduce gating mechanisms to improve information flow (Appendix C). Our framework supports vanilla RNNs, GRUs, and LSTMs (Appendix D); for clarity, we present the detailed ZKP construction for the vanilla RNN.

3. Zero-Knowledge Proofs for Recurrent Neural Network

3.1. Handling Fixed-Point Arithmetic

To support real-valued RNN inference over $\mathbb{F}_p$, we use the fixed-point encoding of [10], representing a real value r as a q -bit signed integer with f fractional bits. Specifically, r is encoded as $s = \lfloor r \cdot 2^f \rfloor$, with negative values mapped to their field representation in $\mathbb{F}_p$. The prime p is chosen large enough to avoid overflow during inference.

Arithmetic operations. Addition and subtraction are performed directly on quantized values. For multiplication, if s_a and s_b are the fixed-point encodings of a and b , then $z = s_a s_b$ corresponds to $ab \cdot 2^{2f}$ and is rescaled by 2^f . For verifiability, the prover provides a bit-decomposition $z_0, \dots, z_{q-1}$ and sign bit z_q , and the verifier checks $z_i \in \{0, 1\}$ and $z = (-1)^{z_q} \sum_{i=0}^{q-1} z_i 2^i$. The fixed-point product is recovered as $(-1)^{z_q} \sum_{i=0}^{q-f-1} z_{i+f} 2^i$.

Comparisons. Comparisons are implemented using the bit-decomposition of z , where the sign bit z_q determines whether $z \geq 0$.

Division. Given quantized values $s_a, s_b \in \mathbb{F}_p$, the prover supplies a quotient z and remainder r satisfying $s_a = z \cdot s_b + r$, $0 \leq r < s_b$. The verifier checks this relation and outputs $z \cdot 2^f$ as the fixed-point quotient.

Scaling optimization. To reduce circuit complexity, scaling is deferred to the output of each RNN layer. Intermediate products remain in scaled form, and a single rounding/truncation is applied at the layer output (e.g., after computing z^t and y^t).

Sumcheck for Binary Operations. To support non-linear functions such as ReLU and tanh, we represent values via bit-decompositions. Following [10], Binary constraints are verified through sumcheck instances rather than explicit circuit gates, reducing circuit size. Let $\mathbf{Z}_{\text{bit}} \in \mathbb{F}^{n \times q}$ denote

the claimed bit-decompositions of $\mathbf{z} \in \mathbb{F}^n$, and let $\mathbf{z}_{\text{sign}} \in \mathbb{F}^n$ be the associated sign bits. Binary validity is checked using

$$0 = \sum_{x \in \{0,1\}^{\log n}} \sum_{y \in \{0,1\}^{\log q}} \beta_x(r_x) \beta_y(r_y) \left(\alpha \mathbf{z}_{\text{sign}}(x) (1 - \mathbf{z}_{\text{sign}}(x)) + \mathbf{Z}_{\text{bit}}(x, y) (1 - \mathbf{Z}_{\text{bit}}(x, y)) \right), \quad (2)$$

where β_x and β_y are multilinear basis polynomials. Consistency between $\mathbf{z}$ and its bit-decomposition is verified by

$$\mathbf{z}(r_z) = \sum_{x \in \{0,1\}^{\log n}} \sum_{y \in \{0,1\}^{\log q}} \beta_x(r_x) \mathbf{Z}_{\text{bit}}(x, y) (1 - 2\mathbf{z}_{\text{sign}}(x)) 2^y \quad (3)$$

These checks ensure binary validity and correct reconstruction without increasing circuit size, and naturally support non-linear activations in the GKR framework.

Handling Exponentiation. Direct evaluation of e^x in arithmetic circuits is costly. Following [10], [41], [42], we use a piecewise-linear approximation. Assuming inputs are normalized to $[-1, 1]$,

$$e^x \approx \begin{cases} 0 & \text{if } x < -\frac{1}{2}, \\ \frac{1}{2} + x & \text{if } -\frac{1}{2} \leq x \leq \frac{1}{2}, \\ 1 & \text{if } x > \frac{1}{2}. \end{cases}$$

This approximation significantly reduces proving cost while maintaining adequate inference accuracy; its impact is evaluated in Section 4.

3.2. Each Step Proof

At each time step t of the vanilla RNN, the hidden state a_t and output y_t are computed as $a_t = F_1(W_{aa} \cdot a_{t-1} + W_{ax} \cdot x_t + b_a)$, $y_t = F_2(W_{ya} \cdot a_t + b_y)$, where F_1 and F_2 are non-linear activation functions (e.g., ReLU or tanh). We aim to construct a succinct ZKP for each time step to ensure that the above computations are performed correctly and consistently with committed values. Our proof for a single time step, formalized in Fig. 2, adopts a modular strategy that decomposes the computation and leverages specialized sumcheck-based protocols for each component. This hybrid approach is designed to maximize prover efficiency by utilizing the most suitable cryptographic tool for each distinct task: a time-optimal sumcheck protocol for linear algebra, and a GKR-style proof for non-linear logic.

3.2.1. Unified Proofs for Linear Operations. A standard practice in modern ML systems is to reformulate computational steps as matrix-matrix or matrix-vector multiplications, motivated by the hardware efficiency of GPUs and other accelerators. We adopt this strategy to create a more efficient and simpler proof system. Consider the standard recurrent update equation $z_i = W_{ax}x_i + W_{aa}a_{i-1} + b_a$. Instead of proving three separate operations (two matrix-vector products and one vector addition), we express this as a single, larger matrix-vector product. We first construct an *augmented input vector* that concatenates the current input,

the previous hidden state, and a constant “1” for the bias term:

$$\tilde{x}'_i = \begin{bmatrix} x_i \\ a_{i-1} \\ 1 \end{bmatrix} \quad (4)$$

Correspondingly, we define an *augmented weight matrix* that combines the respective weight matrices and the bias vector:

$$\tilde{W}_a = [W_{ax} \quad W_{aa} \quad b_a] \quad (5)$$

The entire linear transformation for the hidden state is then compactly written as $z_i = \tilde{W}_a \tilde{x}'_i$. This reformulation is applied to all linear steps in the RNN, including the output computation. By unifying the linear components this way, we can prove all of them using a single cryptographic primitive: Thaler’s time-optimal sumcheck protocol for matrix multiplication (SumMatMul) [15]. As detailed in Steps 2 and 4 of Protocol (cf. Fig. 2), this protocol is invoked to prove the correctness of these unified matrix products. This approach is faster than arithmetizing the operations in a generic circuit and allows us to prove the most computationally intensive part of the RNN step efficiently.

3.2.2. Proofs for Non-Linear Activations. While the unified matrix multiplication handles the linear layers, the non-linear activation functions, e.g., $a_i = F_1(z_i)$ and $y_i = F_2(s_i)$, which typically represent functions like ReLU or tanh, require a different approach. For these, we rely on a generic GKR-style proof system operating on a layered arithmetic circuit. Following the methodology of prior work [10], [42], [43], [44], the GKR circuit itself manages these non-arithmetic operations using the bit-decomposition and piecewise linear approximations detailed in Section 3.1. The prover provides the binary representation of the inputs to the non-linear functions (e.g., the bits of z_i) as part of a committed auxiliary witness, AUX_i . The GKR proof, which we accelerate with the optimized sumcheck from Dao and Thaler [16], then operates on these bits to verify the function’s logic. For example, a ReLU function ($\max(z, 0)$) is verified by checking the sign bit provided in the decomposition. A critical component is proving that this auxiliary data is correct. To ensure the prover cannot cheat by providing invalid bits, we include dedicated sumcheck proofs. These consistency checks are expressed *directly as sumcheck instances* rather than being arithmetized, which avoids a significant blow-up in the circuit size and formally attests to the consistency between the original values and their committed bit-decompositions.

3.3. Recursive Proof Composition

Verifying an inference sequence of length T by simply concatenating T instances of the step-proof described in Section 3.2 would require the verifier to verify each time step individually. This linear scaling in verification time and proof size is neither practical nor efficient for long sequences. Instead, zkRNN employs a recursive composition strategy based on IVC [35].

Protocol A: Proof of RNN Step

Parameters. Let SumMatMul be the time-optimal sumcheck protocol for matrix multiplication [15]. All linear operations in the RNN step are reduced to a single matrix multiplication instance and proven by invoking SumMatMul. Let GKR be a generic GKR-style ZKP system which uses the optimized sumcheck protocol [16] as a subroutine. Let PCS = (KeyGen, Commit, Open, Verify) be the Orion polynomial commitment scheme [14].

Initialization: On inputs x_i and previous state a_{i-1} , $\mathcal{P}$ computes the RNN step: linear state $z_i = W_{ax}x_i + W_{aa}a_{i-1} + b_a$, state $a_i = F_1(z_i)$, linear output $s_i = W_{ya}a_i + b_y$, output $y_i = F_2(s_i)$, and auxiliary data AUX_i (bit-decompositions). $\mathcal{P}$ sends $\mathcal{V}$ commitments $\sigma_{a_i} \leftarrow \text{PCS.Commit}(a_i, r_{a_i})$ and $\sigma_{AUX_i} \leftarrow \text{PCS.Commit}(AUX_i, r_{AUX_i})$. (Commitments to weights σ_W and previous state $\sigma_{a_{i-1}}$ are known).

Phase 1: Proof of Output Layer.

- 1) $\mathcal{V}$ samples r_y and requests evaluation $\tilde{y}_i(r_y)$.
- 2) Having $\tilde{y}_i(r_y)$, parties run GKR on $y_i = F_2(s_i)$. The circuit uses bit-decompositions from AUX_i . This ends with evaluations $\tilde{s}_i(r_{s_i})$ and $\widetilde{AUX}_i(r_{AUX,1})$.
- 3) Having $\tilde{s}_i(r_{s_i})$, parties run SumMatMul on $s_i = W_{ya}a_i + b_y$. At the end, parties receive evaluations $\tilde{a}_i(r_{a,\text{out}})$, $\tilde{W}_{ya}(r_{W_{ya}})$, and $\tilde{b}_y(r_{b_y})$.

Phase 2: Proof of State Layer.

- 1) Parties run GKR on $a_i = F_1(z_i)$ using $\tilde{a}_i(r_{a,\text{out}})$ as the claim. This ends with evaluations $\tilde{z}_i(r_{z_i})$ and $\widetilde{AUX}_i(r_{AUX,2})$.
- 2) Having $\tilde{z}_i(r_{z_i})$, parties run SumMatMul on $z_i = W_{ax}x_i + W_{aa}a_{i-1} + b_a$. At the end, parties receive evaluations $\tilde{x}_i(r_{x_i})$, $\tilde{a}_{i-1}(r_{a_{i-1}})$, $\tilde{W}_{ax}(r_{W_{ax}})$, $\tilde{W}_{aa}(r_{W_{aa}})$, and $\tilde{b}_a(r_{b_a})$.

Phase 3: Consistency Checks.

- 1) Parties run sumchecks (analogous to Eq 2, 3) to prove bit-decomposition correctness for AUX_i . This ends with evaluations combined into a single $\widetilde{AUX}_i(r_{AUX,3})$.

Phase 4: Evaluation Reduction.

- 1) Parties combine received evaluations of $\widetilde{AUX}_i$ from Phases 1, 2, and 3 into a single evaluation $\widetilde{AUX}_i(r_{\text{in},AUX})$.
- 2) The final set of evaluations $\mathcal{E}^{\text{raw}} = \{\tilde{a}_{i-1}, \tilde{x}_i, \tilde{a}_i, \tilde{W}_{ax}, \widetilde{AUX}_i\}$ and wiring predicates $\mathcal{W}^{\text{raw}}$ (and $\pi_{i,G}$ if non-interactive) are returned. These are verified by invoking PCS.Open or passed to the aggregation scheme.

Figure 2. Proof of RNN Step

Incrementally Verifiable Inference. We construct our recursive proof system by defining an augmented transition function, denoted as F_A . The augmented function F_A accepts cryptographic proofs from its children, denoted as $\{\pi_{\text{child}}\}_{j=1}^k$. The execution of F_A at a *node* checks that the child proof packages are valid (when present). In particular, at a leaf node, simply the RNN step for that time index will be computed, and a leaf proof will be produced; at an internal node, first F_A verifies its k children if verification passes, they will be aggregated to the parent proof. This structure ensures that a single proof at the root inductively implies the correctness of the entire inference history covered by the leaves beneath it [35]. A critical challenge in realizing F_A is the cost of verifying child ZK arguments inside the arithmetic circuit. Each π_{child} consists of transcripts from the GKR and matrix multiplication sumcheck protocols used to validate the linear and non-linear operations of the RNN layer. To verify π_{child} , the augmented circuit must perform *In-Circuit Sumcheck Verification*. This involves simulating the honest verifier's protocol: deriving random challenges via the Fiat-Shamir transformation and checking the arithmetic identities of the sumcheck protocol. However, the standard verifier also requires checking polynomial commitment openings, which relies on heavy cryptographic hashing. To maintain efficiency, our in-circuit verifier performs only the arithmetic checks and challenge generation, deferring the expensive polynomial opening checks via the

aggregation techniques described next.

Evaluation reduction and aggregated openings. zkRNN employs a unified mechanism for compressing many commitment-evaluation claims into a single final opening. The approach follows the polynomial-commitment aggregation method introduced in [10], whose full structure is recalled in Fig. 12 in Appendix B. Here we present a concise description of the underlying ideas before explaining how Protocol B, presented in Figs. 4, and 5, incorporates them into the zkRNN proof system. The aggregation method begins by reducing multiple evaluation checks to one. Suppose a collection of polynomials $f_1, \dots, f_k$ is committed, and for each we must verify an evaluation $y_i = f_i(x_i)$. The verifier and prover construct a univariate interpolation map L satisfying $L(i) = x_i$ and require the prover to supply g_i , a univariate polynomial meant to represent $f_i \circ L$. After verifying that $g_i(i) = y_i$ for all i , the verifier samples random coefficients and a random point, forming a single pair (x^*, y^*) defined by $x^* = L(r)$ and $y^* = \sum_i \alpha_i g_i(r)$. The prover forms an aggregate polynomial $f^* = \sum_i \alpha_i f_i$, whose correctness at the point x^* is sufficient to ensure correctness of all original evaluations except with negligible probability. To avoid revealing structural information about each f_i , random masking polynomials are added to the compositions prior to aggregation, ensuring the reduction step remains zero-knowledge.

The second phase aggregates the commitments them-

selves. The prover computes a commitment σ^* to f^* and must now prove that this commitment is consistent with the intended linear combination of the original commitments. As shown in Fig. 12 (cf. Appendix B), this is achieved by opening a small, verifier-chosen set of positions in the encoded representations underlying the commitments. Because the encoding used in the commitment scheme is linear, the corresponding positions in the encoding of f^* must match the same linear combination of the sampled positions in the encodings of the input polynomials (including the masks). Coding-theoretic properties guarantee that any deviation from the prescribed relation affects a substantial fraction of positions, and therefore a logarithmic number of random checks suffices for soundness. Protocol B (see Figs. 4, and 5) applies this aggregation procedure incrementally across the recurrent computation. At each time step, the RNN transition proof yields several committed polynomials and associated evaluation claims. These are merged with the previously maintained aggregate triple by invoking the same evaluation-reduction and commitment-aggregation steps described above. The outcome is a new tuple $(\sigma_t^*, x_t^*, y_t^*)$ that summarizes all commitment–evaluation relations up to that point, allowing the verifier to discard the individual claims. After the final step, the verifier performs a single opening of the final aggregate commitment at the associated point (x_T^*, y_T^*) . By the security of the aggregation method [10], this opening certifies the correctness of every commitment–evaluation pair encountered throughout the entire execution, yielding a succinct and sound verification procedure based on aggregated openings.

Since the input sequence is public, committing to the input vectors is not necessary. However, the sumcheck protocols in zkRNN may still produce evaluation claims of the input MLEs, of the form $v_i = \tilde{x}_i(r_i)$. To avoid carrying all such claims through the recursion, we aggregate them using a random linear combination. Namely, Fiat–Shamir is used to derive coefficients α_i , and the claims are compressed into a single value $v^* = \sum_i \alpha_i v_i$. The verifier can then recompute $\sum_i \alpha_i \tilde{x}_i(r_i)$ directly from the public inputs and verify it.

Wiring Aggregation. To avoid inflating the augmented circuit F_A with the arithmetic logic required to evaluate wiring predicates (e.g., $\widetilde{add}_i, \widetilde{mult}_i$), we delegate these evaluations to the prover. Instead of computing them natively, the circuit treats the results as claims and employs the *Evaluation Reduction Sumcheck* (2) to compress them into a single aggregate claim wp_{agg} . This aggregate is propagated via the accumulator and verified solely at the recursion root, effectively decoupling the verification complexity from the circuit topology [45].

Tree-based organization. The multivariate commitment aggregation scheme guarantees knowledge soundness primarily for a constant number of recursive folds; deep sequential recursion can degrade these guarantees and potentially expose the system to attacks [46]. To mitigate this, we adopt a tree-based recursion strategy [47] described in Fig. 3, where leaves represent per-iteration step proofs.

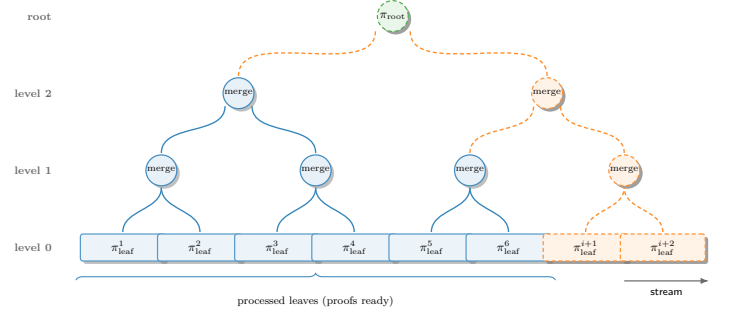


Figure 3. Streaming proof generation using a k -ary tree (shown for $k = 2$). Leaf proofs (π_{leaf}) are incrementally merged as the stream continues (dashed lines), culminating in a single root proof (π_{root}).

Each internal node verifies its k child proofs in-circuit and reapplies the reduction and aggregation pipeline to emit a single parent proof. This topology reduces the recursion depth to $O(\log_k T)$, requiring the external verifier to check only the final proof at the root. Compared with generic recursive SNARK constructions, the recursion performed by zkRNN is specialized to the proof objects generated by recurrent inference. In particular, recursion operates directly on aggregated sumcheck evaluations, wiring claims, and commitment accumulators rather than on arbitrary circuits. This specialization substantially reduces the amount of information propagated between recursive levels and enables compact final proofs.

3.4. Putting Everything Together

To prevent verification cost from scaling linearly with the number of steps, zkRNN defers commitment openings by folding these per-step claims into accumulators. Concretely, after Protocol A (cf. Fig. 2), the prover runs the aggregation prover AGG.P on $\mathcal{E}^{\text{raw}}$ to obtain a single data accumulator agg_i together with a short aggregation proof and an auxiliary witness that can be carried into the next recursion layer. Intuitively, agg_i summarizes all commitment–evaluation relations generated at step i in a form that can be merged with other summaries, enabling the protocol to postpone the expensive polynomial-opening verification to a single check at the end. In the same spirit, wiring predicate verification is also deferred: the wiring claims $\mathcal{W}^{\text{raw}}$ are compressed into a single wiring accumulator wp_i using evaluation reduction, so that wiring consistency can be checked once at the root. The resulting leaf proof package for step i therefore contains (i) the sumcheck/GKR transcript component for the step, (ii) the aggregated data accumulator and its proof/witness, and (iii) the aggregated wiring claim. Succinctness over long streams is obtained by recursively composing these leaf packages in a k -ary tree as specified in Protocol B (cf. Figs. 4, and 5). The system maintains a small buffer of pending proof packages at each recursion level; each new leaf is placed into the level-0 buffer, and whenever any level accumulates k children it is immediately merged

into a parent package, continuing eagerly as the stream progresses. A parent package is defined by an augmented function F_A that verifies and combines its k children. Inside F_A , the required Fiat–Shamir challenges and hash computations are recomputed, and each child’s arithmetic transcript is verified in-circuit. Crucially, polynomial openings are not verified in-circuit; instead, successful in-circuit verification produces the new evaluation claims implied by the child computation, and the parent additionally verifies that the child’s aggregation proof correctly folded its input accumulators and these new claims into the child’s output accumulator. The parent then produces its own transcript proof $\pi_{\text{parent},G}$ attesting to correct execution of F_A , merges the children’s data accumulators into $\text{agg}_{\text{parent}}$ via AGG.P, and folds the children wiring accumulators into $\text{wp}_{\text{parent}}$. By using a tree of arity $k \geq 2$, the recursion depth is $O(\log_k T)$.

Because the inputs x_i form a potentially unbounded public stream, zkRNN does not commit to the input vectors, but it still aggregates the inputs evaluation claims produced during the protocol. Using random linear combination, the protocol compresses these claims into a single public-input aggregate that is merged up the recursion tree alongside the data accumulators. At verification time, the verifier recomputes the corresponding aggregate locally from the stream prefix, ensuring that the root proof is tied to the same public inputs that produced the online outputs. When the user requests a proof for the processed prefix, the system merges all buffered nodes into a single root package (padding if necessary). The verifier recomputes all Fiat–Shamir challenges and hashes, and verifies the root transcript component, which certifies that the root executed F_A correctly and thereby that all child transcript verifications and accumulator-validations were carried out correctly throughout the tree. The verifier then checks the final wiring accumulator against the known wiring predicates, verifies the aggregation proof showing that agg_{root} correctly summarizes all evaluation claims induced across the entire computation, and verifies the aggregated public-input claim against its locally maintained stream aggregate. Finally, the verifier performs a single polynomial-opening verification using PCS.Verify for agg_{root} . This last check binds all deferred commitment–evaluation relations to their underlying commitments, and together with the recursive transcript verification implies that every reported output y_i is consistent with correct execution of the RNN recurrence under the committed model and initial state, while intermediate states and auxiliary witnesses remain hidden.

Theorem 3.1. *The Proof of RNN Step protocol (Protocol A described in Fig. 2) satisfies completeness, knowledge soundness, and zero-knowledge.*

Theorem 3.2. *Protocol B, presented in Figs. 4, and 5 is a Zero-knowledge proof of RNN and satisfies completeness, knowledge soundness, and zero-knowledge.*

Due to space limitations, we present the proofs in Appendix E.

3.5. Complexity Analysis

We analyze zkRNN with sequence length T , hidden-state size h , input dimension n , and quantization precision q . Each prover step costs $O(h^2 + hn)$ for the matrix–vector product and $O(hq)$ for non-linear checks, yielding total prover time $O(T(h^2 + hn + hq))$, which is linear in T . The verifier reads $(x_1, \dots, x_T)$ in $O(Tn)$ time and verifies a constant number of aggregated polynomial-commitment openings, for total verification time $O(Tn + \log^2(h^2 + hn) + \log^2(hq))$. Thus, verification is succinct in the circuit size aside from input reading. Proof size is independent of T . The root proof π_{root} contains a constant number of sumcheck transcripts and commitment openings, dominated by Merkle authentication paths [48]. Hence, communication complexity is $O(\log^2(h^2 + hn) + \log^2(hq))$, remaining succinct in the circuit size and independent of T .

4. Evaluation

4.1. System Implementation

Software. We implemented the scheme in C++ as a streaming recursive proof architecture built on top of KAIZEN’s cryptographic primitives [10]. The implementation is available at [49]. The system manages the recursive proof generation via a `StreamOrchestrator` that utilises multi-level buffers to process timesteps incrementally. For proof generation, Orion [14] is utilised for polynomial commitments and generic GKR circuits to prove the correctness of non-linear activations (`tanh`, `softmax`). All computations are bridged to field arithmetic using fixed-point quantisation with Q fractional bits to ensure compatibility between the prover’s tensor computations and the verifier’s checks. The implementation optimises for memory by performing a single forward pass per timestep, reusing results for both execution and proof generation.

Models. We evaluate inference proving on square RNN configurations ($n = m = k$) at three distinct hidden sizes and a common sequence-length grid $T \in \{8, 32, 64, 128, 256\}$ to assess performance across different scales. Parameter count is $nm + m^2 + km + m + k$ and is independent of T :

- *Small:* A lightweight configuration with dimensions $n = m = k = 32$, resulting in $\approx 3\text{K}$ parameters.
- *Medium:* A standard configuration with dimensions $n = m = k = 128$, totaling $\approx 49\text{K}$ parameters.
- *Large:* A complex configuration with dimensions $n = m = k = 256$, giving $\approx 197\text{K}$ parameters.

Unless stated otherwise, aggregation arity is $k' = 2$.

Hardware. We run all experiments on a machine equipped with a 12th Generation Intel(R) Core i7-12650H processor (2.30 GHz) and 64GB of RAM under WSL2/Linux. Our current implementation is not parallelised and utilises only a single CPU core. We report the average running time across multiple executions (ranging from 3 to 20 runs depending on the configuration) to ensure statistical reliability.

Protocol B (Part I): Streaming and Recursive ZK Proof of RNN Inference

Parameters. Let SumMatMul be the zero-knowledge time-optimal sumcheck protocol for matrix multiplication [15]. Let GKR be a generic GKR-style zero-knowledge proof system which uses the optimized sumcheck protocol [16] as a subroutine. Let PCS = (KeyGen, Commit, Open, Verify) be the Orion polynomial commitment scheme [14]. Let AGG = (Prove, Verify) be the non-interactive zero-knowledge aggregation scheme [10], which also supports public-input aggregation, and let MiMC [17] be the hash function used throughout the protocol. Let ProofOfRNNStep be the step-proof protocol described in Fig. 2. Let $k \geq 2$ be the recursion arity (the number of children per internal node). Let π_0 be a trivially satisfying proof package.

Model and state. Model commitments $\sigma_W = \{\sigma_{W_{ax}}, \sigma_{W_{aa}}, \sigma_{W_{ya}}, \sigma_{b_a}, \sigma_{b_y}\}$. Initial state a_0, σ_{a_0} .

Goal. The system processes a stream of inputs $\{x_i\}_{i=1}^T$, providing immediate outputs $\{y_i\}_{i=1}^T$. At any point, the user can request a single, succinct root proof that validates the entire sequence of computations up to that point.

Streaming Orchestration The system maintains the current state (a, σ_a) and a buffer of pending proofs for each recursion level.

OnInput(x_i): For each input, the system generates a leaf proof π_{leaf} and output y_i via **ProveLeaf**, then immediately returns y_i . The new state (a_i, σ_{a_i}) is stored. The new π_{leaf} is buffered, triggering eager recursive merging via **ProveInternalNode** whenever any level accumulates k proofs.

FinalizeProof(): All buffered proofs are recursively merged up to the root via **ProveInternalNode** (padding if necessary), computing the final, single proof π_{root} .

ProveLeaf($i, x_i, a_{i-1}, \sigma_W, \sigma_{a_{i-1}}, pp$): Generates the leaf proof for a single RNN step.

- 1) **Compute Witness:** Compute the RNN step $(z_i, a_i, s_i, y_i) \leftarrow \text{RNN}(x_i, a_{i-1}, W)$ and the auxiliary data AUX_i .
- 2) **Commit State:** Compute the new state commitments $\sigma_{a_i} \leftarrow \text{PCS.Commit}(a_i, r_{a_i}, pp)$ and $\sigma_{\text{AUX}_i} \leftarrow \text{PCS.Commit}(\text{AUX}_i, r_{\text{AUX}_i}, pp)$.
- 3) **Generate Step Proof:** Call Protocol A to get the raw proof and claims:

$$(\pi_{i,G}, \mathcal{E}^{\text{raw}}, \mathcal{W}^{\text{raw}}) \leftarrow \text{ProofOfRNNStep.Prove}(x_i, y_i, \sigma_W, \sigma_{a_{i-1}}, \sigma_{a_i}, \sigma_{\text{AUX}_i}).$$

- 4) **Generate Leaf Aggregates:** If needed, Run AGG.P on the evaluation instances $\mathcal{E}^{\text{raw}}$ per polynomial family, producing the tuple $(\text{agg}_i, \pi_{i,A}, \pi_{i,E})$ of per-family. Concurrently, apply evaluation reduction to $\mathcal{W}^{\text{raw}}$ to compute the wiring aggregate wp_i .
- 5) **Assemble and Return:** Define the proof package $\pi_{\text{leaf}} := (\pi_{i,G}, \pi_{i,A}, \pi_{i,E}, \emptyset, \text{agg}_i, \emptyset, \text{wp}_i)$ and Return $(a_i, \sigma_{a_{i-1}}, y_i, \pi_{\text{leaf}})$.

Figure 4. Streaming and leaf proof generation for Protocol B

4.2. Performance

We evaluate zkRNN on several dimensions: (1) prover and verifier efficiency across varying arithmetization granularities, (2) scalability with respect to sequence length T at fixed aggregation arity $k' = 2$, and (3) memory and proof size efficiency.

Performance Analysis. We first compare the performance of zkRNN across the Small, Medium, and Large configurations. As shown in Table 2, prover generation time varies significantly based on hidden layer state dimension and the total recurrence depth T . For the Small baseline configuration, proof generation demands 0.61s to 17.97s as T climbs from 8 to 256. In Medium circuit scale, prover latency expands from 16.58s ($T = 8$) to 475.62s ($T = 256$). For Large configuration, the unrolled state transition path induces significant arithmetic burdens with prover times ranging from 167.72s at short depth up to ≈ 1.4 hours at $T = 256$. This reveals a clear shift in what bottlenecks the prover. For short sequences, performance is driven by the model’s width whereas for longer ones, the sequence length itself takes over as the primary bottleneck, adding a predictable, linear amount of work for each additional step. While verifier execution time remains orders of magnitude lower than the prover runtime, its cost is not entirely invariant to circuit dimensions. Across the global evaluation

matrix, verifier latency spans from 0.036s to 13.30s. This upward trend occurs as end-to-end verification requires validating the core GKR protocol layer while also ingesting and verifying the Orion PCS opening data, which grows with both sequence length T and model size. The memory allocation profile also mirrors this unrolled structural behavior, scaling with hidden dimension and T . Notably, doubling the sequence length from $T = 32$ to $T = 64$, increases memory peak by approximately 68% for Small, 91% for Medium and 90% for Large configuration. Verifier-facing proof material is split into two components, namely the recursive GKR accumulator root which stays constant at 17.66KB across configurations at $k' = 2$ and the Orion PCS opening bundle which grows with T and model width. The verifier must ingest and check this material during the verification process, which partly explains the increase in verifier time from 0.036s to 13.30s. Hence, the major takeaway is that recursion gives a constant-size IVC root for on-chain anchoring, while the bulk of per-timestep leaf and aggregation proving remains on the prover.

To isolate the empirical effects of the aggregation arity k' , we fix the sequence length $T = 32$ and sweep model size using square configurations $n = m = k \in \{32, 128, 256\}$. Figure 6 plots prover time and verifier GKR proof size against $k' \in \{2, 4, 8, 16, 32\}$. The choice of k' acts as a

Protocol B (Part II): Streaming and Recursive ZK Proof of RNN Inference

ProveInternalNode($\{\pi_{\text{child}_j}\}_{j=1}^k, pp$): This function generates a parent proof by verifying and aggregating k child proofs.

1) **Define the Augmented Function F_A (Verification).**

- a) **Evaluate Hashes:** Evaluate Fiat-Shamir challenges and Merkle hashes required for verifying the proof components (e.g., $\pi_{j,G}, \pi_{j,A}$) within each child proof $\{\pi_{\text{child}_j}\}_{j=1}^k$.
- b) **In-Circuit Verification.**
 - i) Parse the k child proofs $\{\pi_{\text{child}_j}\}_{j=1}^k$. Each π_j is parsed as $(\pi_{j,G}, \pi_{j,A}, \pi_{j,E}, \{\text{agg}_{j,\text{in},m}\}_{m=1}^k, \text{agg}_j, \{\text{wp}_{j,\text{in},m}\}_{m=1}^k, \text{wp}_j)$, where $\{\text{agg}_{j,\text{in},m}\}_{m=1}^k$ are the input aggregates to child j , and agg_j is its output aggregate (same for $\{\text{wp}_{j,\text{in},m}\}_{m=1}^k$ and wp_j).
 - ii) For each child $j \in [1, k]$:
 - A) **Verify Sumchecks & Wiring Predicate:** Run the verifier for $\pi_{j,G}$ (e.g., GKR.V, SumMatMul.V). This verification takes the input aggregates $\{\text{agg}_{j,\text{in},m}\}_{m=1}^k$ and $\{\text{wp}_{j,\text{in},m}\}_{m=1}^k$ along with other public data. Successful verification yields the set of data evaluation instances $\mathcal{E}_j^{\text{new}}$ generated within the child's step and the updated predicate aggregation wp_j .
 - B) **Verify Data Aggregation:** Run AGG.V on $\pi_{j,A}$ to confirm agg_j correctly aggregates its inputs, $\{\text{agg}_{j,\text{in},m}\}_{m=1}^k$ and $\mathcal{E}_j^{\text{new}}$.
 - iii) If all sumchecks (a) and aggregations (b) pass, the F_A computation is valid.

2) **Generate and Assemble Parent Proof.** The prover now generates the components for the parent proof package.

- a) set $u_{\text{parent}} = \sigma_{a_0} || \sigma_{a_{\text{child}_1}} || \dots || \sigma_{a_{\text{child}_k}} || \pi_{\text{child}_1} || \dots || \pi_{\text{child}_k}$ and commit to it $\sigma_{u_{\text{parent}}} \leftarrow \text{PCS.Commit}(u_{\text{parent}}, r_{u_{\text{parent}}}, pp)$.
- b) **Prove Execution & Aggregate Wiring:** Generate a GKR/sumcheck proof $\pi_{\text{parent},G}$. This single proof attests to the correct execution of the entire F_A (from Step 1) and combine received wiring predicate evaluations into $\text{wp}_{\text{parent}}$ by applying evaluation reduction sumcheck. Similarly, combine all evaluations of $\tilde{u}_{\text{parent}}$. This proof includes the necessary sumchecks for all hash evaluations performed within F_A .
- c) **Aggregate Data:** Generate the parent data aggregate $(\text{agg}_{\text{parent}}, \pi_{\text{parent},A})$ by running the aggregation prover, AGG.P on $\{\text{agg}_j, \pi_{j,E}\}_{j=1}^k$ and remaining commitments and their evaluations (such as $\tilde{u}_{\text{parent}}$'s). Also aggregate the public inputs evaluations of $\{\tilde{x}_i\}_{i=1}^k$.
- d) **Prepare Final Opening:** Generate the final proof component, $\pi_{\text{parent},E}$. If this is the root node, this component is the evaluation opening proof from PCS.Open($\text{agg}_{\text{parent}}$). Otherwise, it is the secret witness for $\text{agg}_{\text{parent}}$ required by the next level of recursion.

3) **Return Parent Proof Package.**

Assemble and return the complete proof $\pi_{\text{parent}} := (\pi_{\text{parent},G}, \pi_{\text{parent},A}, \pi_{\text{parent},E}, \{\text{agg}_j\}_{j=1}^k, \text{agg}_{\text{parent}}, \{\text{wp}_j\}_{j=1}^k, \text{wp}_{\text{parent}})$.

Verify($i, \sigma_W, \sigma_{a_0}, \sigma_{a_i}, \pi_{\text{root}}, pp$):

- a) Parse π_{root} as $(\pi_G, \pi_A, \pi_E, \{\text{agg}_{\text{in},m}\}_{m=1}^k, \text{agg}_{\text{root}}, \{\text{wp}_{\text{in},m}\}_{m=1}^k, \text{wp}_{\text{root}})$.
- b) Evaluate all Fiat-Shamir and Merkle hashes for π_G and π_A .
- c) Run GKR.V/SumMatMul.V on π_G (using public inputs $i, \sigma_W, \sigma_{a_0}, \sigma_{a_i}, x_i$ and aggregates $\{\text{agg}_{\text{in},j}\}_{m=1}^k, \{\text{wp}_{\text{in},m}\}_{m=1}^k$) to verify the root node's computation. This check yields the new data claims $\mathcal{E}^{\text{root}}$ and the final wiring aggregate wp_{root} .
- d) Directly verify wp_{root} against the known wiring predicates.
- e) Run AGG.V on π_A to verify that agg_{root} correctly aggregates the child inputs $\{\text{agg}_{\text{in},j}\}_{m=1}^k$ and the new claims $\mathcal{E}^{\text{root}}$.
- f) Directly verify $\times_{\text{root}}$ against the known aggregate of $\{x_i\}_{i=1}^t$ which has been computed locally.
- g) If this is the final iteration Run PCS.Verify($\text{agg}_{\text{root}}, \pi_E$). Otherwise Check that π_E is a valid opening witness for agg_{root} .

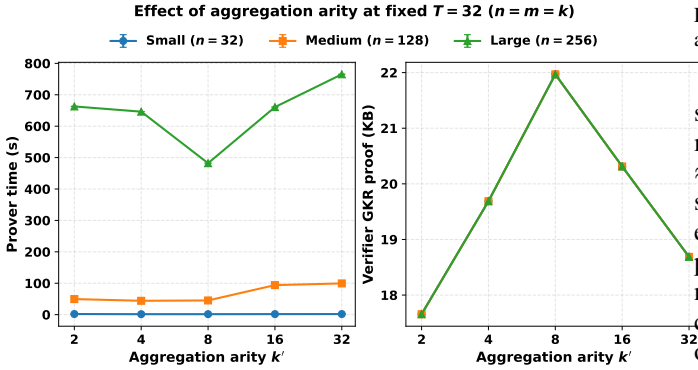
Figure 5. Recursive aggregation and verification for Protocol B

vital architectural lever within the streaming IVC tree framework, simultaneously balancing the topological tree depth and localized node verification complexity. Topologically, increasing k' reduces the height of the aggregation hierarchy to roughly $O(\log_{k'} T)$ levels from leaves to root, hence minimizing the absolute number of recursive aggregation rounds executed over a fixed T . Mathematically, however, each k' -ary node must absorb, verify and compress k' child proofs within a single GKR multi-layered folding step. The evaluation across the model configurations reveals that the extent to which prover execution latency is impacted by variations in k' , is strongly governed by the circuit scale. For Small configuration, the prover time remains structurally insulated from k' modifications and is stable flatly between 1.5s and 2.0s across the sweep. As dimensions grow, a

defined parabolic optimization curve emerges. For Medium configuration, the peak performance occurs at $k' = 4$, with a proof generation time of in 45.2s, and closely trailing behind by $k' = 8$, completing proof generation in 49.6s. This pattern becomes clearer as the dimensions are pushed to Large configuration, where $k' = 8$ emerges as the global peak performance with 482s proof time execution, yielding at least 34% speedups compared to other k' values. The mechanics behind this acceleration are tied to the topological collapse of the tree. Under $k' = 2$ and $T = 32$, the prover is forced to navigate a deep tree of $\lceil \log_2 32 \rceil = 5$ distinct levels. Transitioning to $k' = 8$ flatten this architecture into only two effective structural levels, hence truncating recursive rounds while keeping the internal node arithmetic highly manageable. On the other hand, higher $k' = 16$ or

TABLE 2. PERFORMANCE RESULTS

Config.	n	T	Prover (s)	Verifier (s)	PCS (MB)	IVC Root (KB)	Memory (GB)
S	32	8	0.61	0.036	18.3	17.66	0.056
	32	32	2.02	0.122	73.3	17.66	0.179
	32	64	3.75	0.304	142.4	17.66	0.301
	32	128	7.03	1.001	284.9	17.66	0.586
	32	256	17.97	6.872	586.3	17.66	1.305
M	128	8	16.58	0.078	25.5	17.66	0.401
	128	32	54.41	0.314	102.4	17.66	1.252
	128	64	118.08	0.780	204.5	17.66	2.391
	128	128	256.78	2.723	408.9	17.66	4.661
	128	256	475.62	9.531	818.0	17.66	9.202
L	256	8	167.72	0.186	29.1	17.66	1.542
	256	32	662.85	0.744	116.7	17.66	4.599
	256	64	1304.90	1.742	233.6	17.66	8.750
	256	128	2516.80	3.980	467.0	17.66	17.054
	256	256	4951.48	13.297	933.1	17.66	33.664

Figure 6. Prover Time and Final Verifier GKR Proof Size vs. Aggregation arity for $T = 32$

$k' = 32$ causes the aggregation costs to expand aggressively and revert to degraded runtimes.

Effect of aggregation arity. Furthermore, analyzing the verifier-facing GKR accumulator root size also reveals an intriguing curve that counterbalances the prover optimization trajectory. Unlike the prover-side PCS material, this specific metric represents the final compact IVC certificate delivered to the end verifier and remains isolated from scaling T or model configurations at fixed k' . Instead, its shifts are subtly based on the geometric structure of k' . The initial upward trajectory by about 24% observed from $k' = 2$ (≈ 17.66 KB) through $k' = 8$ (≈ 21.97 KB) is driven by the internal information density of the verification bundles. In other words, wider k' -ary internal nodes must verify more child claims at the same time, causing a more complex parallel structure in the final proof. However, at higher $k' = 16$ and $k' = 32$, a drop in size is observed as 20.31KB and 18.69KB respectively, because the underlying tree layout completely collapses and effectively eliminates the intermediate multi-level tree hierarchies. Consequently, these dual performance characteristics establish a vital Pareto optimization rule for deployment engineering. While $k' = 8$ yields the lowest prover latency for large-scale implementations, it also generates the largest verifier GKR proof within the experimental

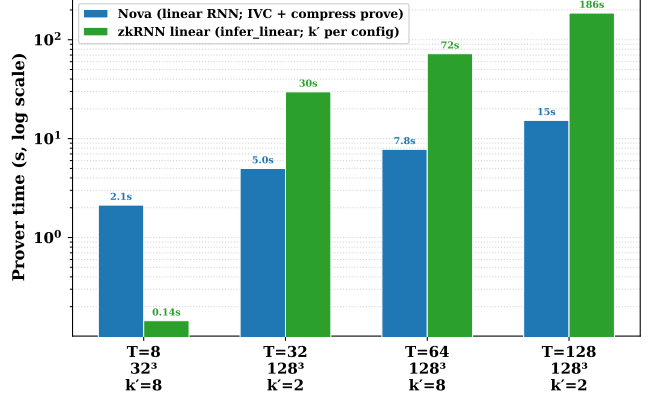


Figure 7. Comparison of Prover Times (s, log scale) between Nova [31] and zkRNN-linear models

sweep. However, the absolute divergence between the maximum and minimum proof sizes represents a minor penalty of ≈ 4.3 KB. Trading this minor increase in verifier GKR proof size for a substantial 27% reduction in raw prover latency establishes $k' = 8$ as the most effective default deployment parameter for high dimensions. Crucially, this threshold is not a universal constant but, as a general rule, the optimal choice for k' scales dynamically with the underlying density of the model and the sequence length, up to the point where wide folding gates saturate the local hardware capacity.

Benchmarking.

We compare zkRNN against Nova [31] configured as an IVC over a linear RNN step circuit matching the sequence length T and layer dimensions (n, m, k) . This implemented baseline of Nova folds identical RICS [50] (Rank-1 Constraint System) steps into a recursive SNARK and compresses the result. zkRNN-linear proves augmented sum-matmul relations per timestep and streams them through the arity aggregation tree but omits tanh/softmax GKR and Orion PCS. We use the optimal k' in each scenario for zkRNN-linear to avoid penalizing the model for a poor aggregation setting. According to the reported median prover times in Figure 7, at small scale ($T = 8, n = 32, k' = 8$), zkRNN-linear is faster by almost 2s as Nova pays a fixed compression overhead on short runs. At higher scale, Nova is found to be substantially faster than zkRNN-linear. However, this gap is not from activations or PCS but rather from how recursion is realized. Nova uses a highly-optimized incremental folding pipeline over a compact RICS step whereas zkRNN-linear emits a full GKR-style SumMatMul proof at each timestep and perform explicit aggregation, which both have significant costs.

Internal Architecture Baselines. A natural alternative to zkRNN is to flatten the recurrent trace into one arithmetic circuit and prove it in a monolithic manner. This avoids recursive composition but then, requires utilising the full-length- T execution before proving. Therefore, proof generation cannot overlap cleanly with inference and consequently, memory size grows with T . To separate the cost of recursive

TABLE 3. SUMMARY OF INTERNAL PROVING MODES

Mode	Structure	Recursion	PCS	Recursive root size
Streaming	Folded leaves	✓	✓	$O(1)$
Monolithic	One bundle, batched GKR	✗	✗	$O(T)$
Flat	T leaves, per-step GKR	✗	✗	$O(T)$

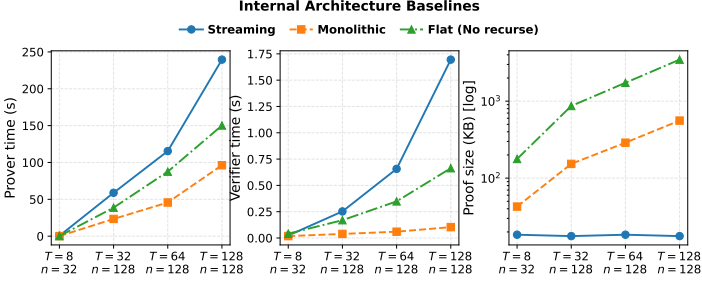


Figure 8. Comparison of Streaming with PCS verification against Monolithic and Flat non-recursive GKR-only proving modes.

aggregation from the per-timestep statement, three internal modes on the same T and model dimensions, are compared and their features are presented in Table 3. Note that in this comparison, Streaming refers to the complete implementation and it verifies both the recursive GKR proof and the Orion PCS binding. In contrast, Monolithic and Flat are internal GKR-only baselines and do not include Orion PCS verification.

Figure 8 plots prover time, verifier time, and GKR proof/root size across four matched configurations. Only Streaming keeps a sequence-independent recursive GKR root, while in our implementation it also preserves PCS-backed evaluation binding. Monolithic mode improves prover time by avoiding recursive folds, but its GKR proof material grows with sequence length (43KB at $T = 8$ to 559KB at $T = 128$). Flat mode is similarly competitive in prover time, but ships every per-step GKR transcript and therefore produces even larger GKR proof material. Since Monolithic and Flat mode do not include Orion PCS verification, their verifier times should be interpreted as GKR-only ablation costs rather than full end-to-end verifier costs; as a result, their verifier time is lower than the Streaming mode verifier time. Overall, recursive aggregation introduces prover overhead, but it enables T -independent recursive GKR root size while preserving PCS-backed evaluation binding in the full Streaming system. A fully unrolled single-circuit monolithic baseline with PCS-backed binding remains future work.

Approximation Accuracy. To quantify the effect of the piecewise approximation used for exponentiation-related nonlinearities, we compare the outputs produced by the approximated activation functions using each configuration profiles (Small, Medium, Large) against the original floating-point implementation over 1500 synthetic sequences. Input values are drawn uniformly from $[-0.25, 0.25]$ and quantized to Q32, while the target labels are assigned by a double-precision floating-point oracle to

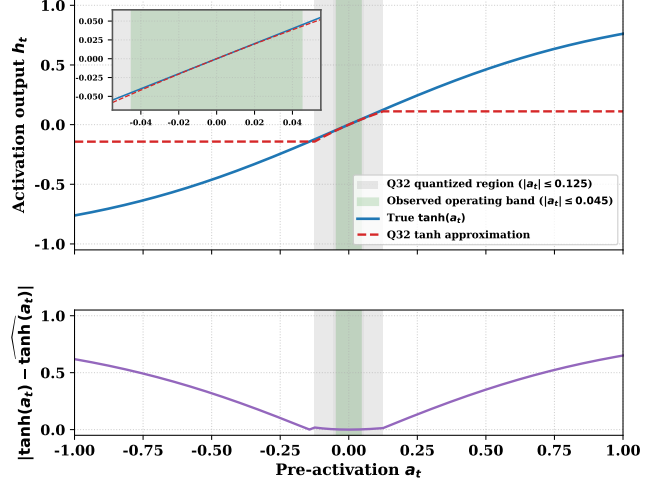


Figure 9. Validation of the Q32 piecewise tanh under in-network pre-activation magnitudes

TABLE 4. IMPACT OF ACTIVATION APPROXIMATION ON TASK ACCURACY AND SCALAR ERROR PROFILES.

Model Profile	Accuracy (%)	Avg. Error	Max Error
Floating-point tanh	100.0	—	—
<i>Q32 Piecewise tanh Approximation</i>			
Small ($T=8, h=32$)	98.8	7.7×10^{-5}	1.5×10^{-3}
Medium ($T=32, h=128$)	97.8	8.0×10^{-5}	1.8×10^{-3}
Large ($T=64, h=128$)	97.2	7.8×10^{-5}	1.9×10^{-3}

establish a 100% baseline. Task accuracy is measured as the fraction of test sequences where the network’s final timestep prediction matches the oracle label over 500 evaluations per profile. Table 4 shows that the approximation introduces only limited numerical error while substantially simplifying proof generation. The minor performance degradation of only 1.2% to 2.8% confirms that the introduction of fixed-point quantization and piecewise linear segmentation does not cause catastrophic error compounding through the timesteps. Figure 9 provides a dual-panel visualization that explains this accuracy retention by mapping the true $\tanh(a_t)$ function against zkRNN’s Q32 piecewise approximation across the active pre-activation space. The upper panel shows the two curves are visually indistinguishable within the green-highlighted operating band ($|a_t| \leq 0.045$). The lower panel tracks the residual scalar error, confirming that it remains to almost zero inside the active operating band and only begins to expand noticeably outside of this range. By optimizing the cryptographic constraints strictly within the narrow empirical operating band when the RNN actually runs, rather than enforcing a heavy approximation across the whole line $(-\infty, \infty)$, the framework drastically cuts prover complexity while maintaining robust task accuracy.

Discussion of Recursive Aggregation. A natural baseline is a non-recursive design where each RNN timestep

TABLE 5. CONCEPTUAL COMPARISON BETWEEN RECURSIVE AND NON-RECURSIVE VERIFICATION.

Configuration	Prover Time	Verifier Time	Proof Size
Non-recursive	Linear in T	Linear in T	Linear in T
zkRNN (recursive)	Linear in T + aggregation overhead	Poly-logarithmic in T	Compact

proof is verified independently. In this setting, verifier work and proof size grow linearly with the sequence length T . Table 5 highlights the contrast with zkRNN’s recursive aggregation, which compresses all timestep proofs into a single succinct proof. Although recursion adds prover-side overhead, it significantly reduces verification and communication costs. Thus, the main advantage of recursion in zkRNN is not lower proving effort, but the compression of verification work and proof size into a compact certificate for the entire recurrent computation.

References

- [1] S. K. Mudgal, R. Agarwal, J. Chaturvedi, R. Gaur, and N. Ranjan, “Real-world application, challenges and implication of artificial intelligence in healthcare: an essay,” *Pan African Medical Journal*, vol. 43, no. 1, 2022.
- [2] I. Mathauer and M. Oranje, “Machine learning in health financing: benefits, risks and regulatory needs,” *Bulletin of the World Health Organization*, vol. 102, no. 3, p. 216, 2023.
- [3] Z. Ghodsi, T. Gu, and S. Garg, “Safetytnets: Verifiable execution of deep neural networks on an untrusted cloud,” *Advances in Neural Information Processing Systems*, vol. 30, 2017.
- [4] L. Zhao, Q. Wang, C. Wang, Q. Li, C. Shen, and B. Feng, “Veriml: Enabling integrity assurances and fair payments for machine learning as a service,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 10, pp. 2524–2540, 2021.
- [5] S. Goldwasser, S. Micali, and C. Rackoff, “The knowledge complexity of interactive proof systems,” in *Proceedings of the seventeenth annual ACM symposium on Theory of computing*. ACM, 1985, pp. 291–304.
- [6] D. Kang, T. Hashimoto, I. Stoica, and Y. Sun, “Scaling up trustless dnn inference with zero-knowledge proofs,” *arXiv preprint arXiv:2210.08674*, 2022.
- [7] Z. Xing, Z. Zhang, J. Liu, Z. Zhang, M. Li, L. Zhu, and G. Russello, “Zero-knowledge proof meets machine learning in verifiability: A survey,” *arXiv preprint arXiv:2310.14848*, 2023.
- [8] Z. Peng, T. Wang, C. Zhao, G. Liao, Z. Lin, Y. Liu, B. Cao, L. Shi, Q. Yang, and S. Zhang, “A survey of zero-knowledge proof based verifiable machine learning,” *arXiv preprint arXiv:2502.18535*, 2025.
- [9] T. Liu, X. Xie, and Y. Zhang, “Zkcnn: Zero knowledge proofs for convolutional neural network predictions and accuracy,” in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2968–2985.
- [10] K. Abbaszadeh, C. Pappas, J. Katz, and D. Papadopoulos, “Zero-knowledge proofs of training for deep neural networks,” in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 4316–4330.
- [11] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum, “Delegating computation: interactive proofs for muggles,” *Journal of the ACM (JACM)*, vol. 62, no. 4, pp. 1–64, 2015.
- [12] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT press, 2016.
- [13] Z. C. Lipton, J. Berkowitz, and C. Elkan, “A critical review of recurrent neural networks for sequence learning,” *arXiv preprint arXiv:1506.00019*, 2015.
- [14] T. Xie, Y. Zhang, and D. Song, “Orion: Zero knowledge proof with linear prover time,” in *Annual International Cryptology Conference*. Springer, 2022, pp. 299–328.
- [15] J. Thaler, “Time-optimal interactive proofs for circuit evaluation,” in *Annual Cryptology Conference*. Springer, 2013, pp. 71–89.
- [16] Q. Dao and J. Thaler, “More optimizations to sum-check proving,” *Cryptology ePrint Archive*, 2024.
- [17] M. Albrecht *et al.*, “Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity,” in *ASIACRYPT*, 2016.
- [18] G. Cormode, J. Thaler, and K. Yi, “Verifying computations with streaming interactive proofs,” *arXiv preprint arXiv:1109.6882*, 2011.
- [19] S. Lee, H. Ko, J. Kim, and H. Oh, “vcnn: Verifiable convolutional neural network based on zk-snarks,” *IEEE Transactions on Dependable and Secure Computing*, vol. 21, no. 4, pp. 4254–4270, 2024.
- [20] A. E. Kosba, D. Papadopoulos, C. Papamanthou, M. F. Sayed, E. Shi, and N. Triandopoulos, “{TRUESET}: Faster {Verifiable} set computations,” in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 765–780.
- [21] R. E. Burkard, “Quadratic assignment problems,” *European Journal of Operational Research*, vol. 15, no. 3, pp. 283–289, 1984.
- [22] M. Campanelli, D. Fiore, and A. Querol, “Legosnark: Modular design and composition of succinct zero-knowledge proofs,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 2075–2092.
- [23] C. Weng, K. Yang, X. Xie, J. Katz, and X. Wang, “Mystique: Efficient conversions for {Zero-Knowledge} proofs with applications to machine learning,” in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 501–518.
- [24] K. Yang, P. Sarkar, C. Weng, and X. Wang, “Quicksilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field,” in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2986–3001.
- [25] T. Lu, H. Wang, W. Qu, Z. Wang, J. He, T. Tao, W. Chen, and J. Zhang, “An efficient and extensible zero-knowledge proof framework for neural networks,” *Cryptology ePrint Archive*, 2024.
- [26] P. Schoppmann, A. Gascón, L. Reichert, and M. Raykova, “Distributed vector-ole: Improved constructions and implementation,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1055–1072.
- [27] M. Hao, H. Chen, H. Li, C. Weng, Y. Zhang, H. Yang, and T. Zhang, “Scalable zero-knowledge proofs for non-linear functions in machine learning,” in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 3819–3836.
- [28] W. Qu, Y. Guo, Y. Ying, and J. Zhang, “Verfcnn, optimal complexity zksnark for convolutional neural networks,” *Cryptology ePrint Archive*, 2025.
- [29] Y. Li and X. Fan, “Summer: Recursive zero-knowledge proofs for scalable rnn training,” *Cryptology ePrint Archive*, 2025.
- [30] D. Balbás, D. Fiore, M. I. González Vasco, D. Robissout, and C. Soriente, “Modular sumcheck proofs with applications to machine learning and image processing,” in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 1437–1451.
- [31] A. Kothapalli, S. Setty, and I. Tzialla, “Nova: Recursive zero-knowledge arguments from folding schemes,” in *Annual International Cryptology Conference*. Springer, 2022, pp. 359–388.
- [32] A. Kothapalli and S. Setty, “Supernova: Proving universal machine executions without universal circuits,” *Cryptology ePrint Archive*, 2022.

- [33] —, “Hypernova: Recursive arguments for customizable constraint systems,” in *Annual International Cryptology Conference*. Springer, 2024, pp. 345–379.
- [34] D. Boneh, J. Drake, B. Fisch, and A. Gabizon, “Halo infinite: Recursive zk-snarks from any additive polynomial commitment scheme,” *Cryptology ePrint Archive*, 2020.
- [35] P. Valiant, “Incrementally verifiable computation or proofs of knowledge imply time/space efficiency,” in *Theory of Cryptography: Fifth Theory of Cryptography Conference, TCC 2008, New York, USA, March 19-21, 2008. Proceedings 5*. Springer, 2008, pp. 1–18.
- [36] C. Lund, L. Fortnow, H. Karloff, and N. Nisan, “Algebraic methods for interactive proof systems,” *Journal of the ACM (JACM)*, vol. 39, no. 4, pp. 859–868, 1992.
- [37] A. Chiesa, M. A. Forbes, and N. Spooner, “A zero knowledge sumcheck and its applications,” *arXiv preprint arXiv:1704.02086*, 2017.
- [38] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. Ward, “Marlin: Preprocessing zksnarks with universal and updatable srs,” in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2020, pp. 738–768.
- [39] A. Fiat and A. Shamir, “How to prove yourself: Practical solutions to identification and signature problems,” in *Conference on the theory and application of cryptographic techniques*. Springer, 1986, pp. 186–194.
- [40] J. L. Elman, “Finding structure in time,” *Cognitive science*, vol. 14, no. 2, pp. 179–211, 1990.
- [41] S. Garg, A. Goel, S. Jha, S. Mahlouljifar, M. Mahmoody, G.-V. Policharla, and M. Wang, “Experimenting with zero-knowledge proofs of training,” in *Proceedings of the 2023 ACM SIGSAC conference on computer and communications security*, 2023, pp. 1880–1894.
- [42] P. Mohassel and P. Rindal, “Aby3: A mixed protocol framework for machine learning,” in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 35–52.
- [43] G. C. Cardarilli, L. Di Nunzio, R. Fazzolari, D. Giardino, A. Nannarelli, M. Re, and S. Spanò, “A pseudo-softmax function for hardware-based high speed image classification,” *Scientific reports*, vol. 11, no. 1, p. 15307, 2021.
- [44] A. Epasto, M. Mahdian, V. Mirokni, and E. Zampetakis, “Optimal approximation-smoothness tradeoffs for soft-max functions,” *Advances in Neural Information Processing Systems*, vol. 33, pp. 2651–2660, 2020.
- [45] J. Zhang, T. Liu, W. Wang, Y. Zhang, D. Song, X. Xie, and Y. Zhang, “Doubly efficient interactive proofs for general arithmetic circuits with linear prover time,” in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 159–177.
- [46] B. Bünz, P. Mishra, W. Nguyen, and W. Wang, “Accumulation without homomorphism,” *Cryptology ePrint Archive*, 2024.
- [47] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer, “Recursive composition and bootstrapping for snarks and proof-carrying data,” in *STOC*, 2013.
- [48] R. C. Merkle, “A digital signature based on a conventional encryption function,” in *Conference on the theory and application of cryptographic techniques*. Springer, 1987, pp. 369–378.
- [49] “zkRNN: Zero-knowledge proofs for recurrent neural network inference (anonymous repository),” <https://anonymous.4open.science/r/zkRNN-5CA8/>, 2026.
- [50] S. Setty, “Spartan: Efficient and general-purpose zkSNARKs without trusted setup,” *Cryptology ePrint Archive*, Paper 2019/550, 2019. [Online]. Available: <https://eprint.iacr.org/2019/550>
- [51] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
- [52] F. A. Gers, J. Schmidhuber, and F. Cummins, “Learning to forget: Continual prediction with lstm,” *Neural computation*, vol. 12, no. 10, pp. 2451–2471, 2000.
- [53] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, “Learning phrase representations using rnn encoder-decoder for statistical machine translation,” *arXiv preprint arXiv:1406.1078*, 2014.

Appendix A. Additional Preliminaries

Polynomial Commitment Scheme (PCS). A PCS lets a prover commit to a polynomial and later prove its evaluation at a point without revealing the polynomial. It consists of $\text{KeyGen}(1^\lambda, \ell, d) \rightarrow \text{pp}$, $\text{Commit}(f, r, \text{pp}) \rightarrow \sigma$, $\text{Open}(f, r, x, \text{pp}) \rightarrow (y, \pi)$ with $y = f(x)$, and $\text{Verify}(\sigma, x, y, \pi, \text{pp})$. A secure PCS satisfies *completeness*, *evaluation binding* (knowledge soundness), *hiding*, and *zero-knowledge*.

Definition 1 (Zero Knowledge Recurrent Neural Network (zkRNN)). *We say that a scheme $\text{zkRNN} = (\text{KeyGen}, \text{Com}, \text{Prove}, \text{Verify})$ is a zero-knowledge proof system for recurrent neural network inference if the following holds. Let λ be the security parameter and $\mathcal{C}$ the public RNN-step function $(a_k, y_k) = \mathcal{C}(W, a_{k-1}, x_k)$.*

Completeness. *Given an RNN model W with commitment $\sigma_W \leftarrow \text{Com}(W, \text{pp})$, an initial state commitment σ_{a_0} , a valid witness a_{i-1} and proof π_{i-1} for step $i - 1$ (with commitment $\sigma_{a_{i-1}}$), and public input x_i , the following equals 1:*

$$\Pr \left[\begin{array}{l} b = 1 \\ \left(\begin{array}{l} (a_i, y_i) \leftarrow \mathcal{C}(W, a_{i-1}, x_i); \\ (\sigma_{a_i}, \pi_i) \leftarrow \text{Prove}(i, \sigma_W, \sigma_{a_0}, a_{i-1}, \sigma_{a_{i-1}}, \pi_{i-1}, \\ x_i, y_i, \text{pp}); \\ b \leftarrow \text{Verify}(i, \sigma_W, \sigma_{a_0}, \sigma_{a_i}, y_i, \pi_i, \text{pp}) \end{array} \right) \end{array} \right]$$

Knowledge Soundness. *For any PPT adversary $\mathcal{A}$, there exists a PPT extractor $\mathcal{E}^{\mathcal{A}}$ such that*

$$\Pr \left[\begin{array}{l} (W, \{a_k\}_{k=0}^i) \leftarrow \mathcal{E}^{\mathcal{A}}(\text{pp}) : \\ \forall k \in \{1, \dots, i\} : (a_k, y_k) = \\ \mathcal{C}(W, a_{k-1}, x_k); \\ \sigma_W^* = \text{Com}(W, \text{pp}); \\ \sigma_{a_0}^* = \text{Com}(a_0, \text{pp}); \\ \sigma_{a_i}^* = \text{Com}(a_i, \text{pp}) \end{array} \right] \geq \Pr \left[\begin{array}{l} (i, \sigma_W^*, \sigma_{a_0}^*, \{x_k\}_{k=1}^i, \{y_k\}_{k=1}^i, \pi_i^*) \leftarrow \mathcal{A}(\text{pp}) : \\ \text{Verify}(i, \sigma_W^*, \sigma_{a_0}^*, \sigma_{a_i}^*, \{x_k\}_{k=1}^i, \{y_k\}_{k=1}^i, \pi_i^*, \text{pp}) = 1 \end{array} \right] - \text{negl}(\lambda).$$

Zero Knowledge. *For any PPT adversary $\mathcal{A}$, RNN model W , initial state a_0 , and input sequence $\{x_k\}_{k=1}^i$, there exists a simulator $\mathcal{S}$ (with oracle access to $\mathcal{C}$) such that the following experiments are computationally indistinguishable:*

Real _{$\mathcal{A}, W$} (i, pp):

- 1) $\sigma_W \leftarrow \text{Com}(W)$, $\sigma_{a_0} \leftarrow \text{Com}(a_0)$.
- 2) For $k = 1 \dots i$:
 $(a_k, y_k) = \mathcal{C}(W, a_{k-1}, x_k)$
 $(\sigma_{a_k}, \pi_k) \leftarrow \text{Prove}(k, \sigma_W, \sigma_{a_0}, a_{k-1}, \sigma_{a_{k-1}}, \pi_{k-1}, x_k, \text{pp})$
- 3) $b \leftarrow \mathcal{A}(\text{pp}, \sigma_W, \sigma_{a_0}, \sigma_{a_i}, \{x_k\}_{k=1}^i, \{y_k\}_{k=1}^i, \pi_i)$
- 4) Output b

Sumcheck Protocol

- 1) In the first round, P sends the polynomial

$$f_1(x_1) := \sum_{b_2, \dots, b_\ell \in \{0,1\}} f(x_1, b_2, b_3, \dots, b_\ell)$$

V checks if $H = f_1(0) + f_1(1)$ and sends $r_1 \leftarrow \mathbb{F}$.

- 2) In the i th round ($2 \leq i \leq \ell - 1$), P sends the polynomial

$$f_i(x_i) := \sum_{b_{i+1}, \dots, b_\ell \in \{0,1\}} f(r_1, \dots, r_{i-1}, x_i, b_{i+1}, \dots, b_\ell)$$

V checks if $f_{i-1}(r_{i-1}) = f_i(0) + f_i(1)$ and sends $r_i \leftarrow \mathbb{F}$.

- 3) In the ℓ th round, P sends the polynomial

$$f_\ell(x_\ell) = f(r_1, r_2, \dots, r_{\ell-1}, x_\ell)$$

V checks if $f_{\ell-1}(r_{\ell-1}) = f_\ell(0) + f_\ell(1)$. The verifier samples the final random challenge $r_\ell \in \mathbb{F}$. Given oracle access to f , V accepts if and only if $f_\ell(r_\ell) = f(r_1, r_2, \dots, r_\ell)$.

Figure 10. Sumcheck Protocol

Ideal_{A,S}(i, pp):

- 1) $(\sigma_W^*, \sigma_{a_0}^*, \sigma_{a_i}^*, \pi_i^*) \leftarrow \mathcal{S}(i, \text{pp}, \{x_k\}_{k=1}^i)$, given oracle access to $\mathcal{C}$
- 2) $b \leftarrow \mathcal{A}(\text{pp}, \sigma_W^*, \sigma_{a_0}^*, \sigma_{a_i}^*, \{x_k\}_{k=1}^i, \{y_k\}_{k=1}^i, \pi_i^*)$
- 3) Output b

We require $|\Pr[\text{Real}_{A,W} = 1] - \Pr[\text{Ideal}_{A,S} = 1]| \leq \text{negl}(\lambda)$.

Appendix B. Protocols

Theorem B.1. [36] Let $\mathbb{F}$ be a finite field and $f : \mathbb{F}^\ell \rightarrow \mathbb{F}$ an ℓ -variate polynomial of individual degree at most d . Sumcheck Protocol (Fig. 10) proves $H = \sum_{x \in \{0,1\}^\ell} f(x)$ with soundness error $O(d\ell/|\mathbb{F}|)$, using ℓ interaction rounds, verifier time $O(d\ell)$ plus one oracle query to f , and prover time $O(2^\ell)$.

Theorem B.2. [11] Let $\mathcal{C} : \mathbb{F}^{s_{in}} \rightarrow \mathbb{F}^{s_{out}}$ be a layered arithmetic circuit of depth d and size $|\mathcal{C}|$. The GKR protocol (Fig. 11) proves the evaluation of $\mathcal{C}$ with soundness error $O(d \log |\mathcal{C}|/|\mathbb{F}|)$, using $O(d \log |\mathcal{C}|)$ rounds, prover time $O(|\mathcal{C}|)$, and verifier time $O(s_{in} + s_{out} + d \log |\mathcal{C}| + T)$, where T is the cost of evaluating the wiring predicates.

Theorem B.3. [10] The aggregation scheme in Protocol 12 satisfies completeness, knowledge soundness, and zero knowledge.

Appendix C. RNN different variants

Long Short-Term Memory (LSTM). LSTMs address the vanishing-gradient problem using a memory cell with a Constant Error Carousel (CEC) that preserves error flow over long time lags [51]. A *forget gate* selectively removes obsolete information to prevent unbounded state growth [52]. The updates are

GKR Protocol

Let $\mathbb{F}$ be a finite field and $\mathcal{C} : \mathbb{F}^{s_{in}} \rightarrow \mathbb{F}^{s_{out}}$ be a layered arithmetic circuit of depth d . P wants to convince V that $\mathcal{C}(\text{in}) = \text{out}$, where in is the input given by V and out is the output sent to V. Without loss of generality, we pad s_{in} and s_{out} to powers of 2. The protocol proceeds as follows.

- 1) V picks a random $z \in \mathbb{F}$ and sends it to P. Both parties P and V compute $\tilde{V}_0(z)$, where $\tilde{V}_0$ is the multilinear extension of array out.
- 2) P and V invoke a sumcheck protocol [16] on

$$\tilde{V}_0(z) = \sum_{x, y \in \{0,1\}^{s_1}} \text{add}_0(z, x, y) (\tilde{V}_1(x) + \tilde{V}_1(y)) + \text{mult}_0(z, x, y) \tilde{V}_1(x) \tilde{V}_1(y).$$

At the end of the protocol, V receives two evaluations $\tilde{V}_1(u^{(1)})$, $\tilde{V}_1(v^{(1)})$. V computes $\text{add}_0(z, u^{(1)}, v^{(1)})$ and $\text{mult}_0(z, u^{(1)}, v^{(1)})$ and checks that $\text{add}_0(z, u^{(1)}, v^{(1)}) \cdot (\tilde{V}_1(u^{(1)}) + \tilde{V}_1(v^{(1)})) + \text{mult}_0(z, u^{(1)}, v^{(1)}) \cdot \tilde{V}_1(u^{(1)}) \cdot \tilde{V}_1(v^{(1)})$ equals to the last message of sumcheck.

- 3) For $i = 1, 2, \dots, d$:

- V picks two random $\alpha^{(i)}$ and $\beta^{(i)}$ and send them to P.
- P and V invoke a sumcheck protocol on

$$\alpha^{(i)} \tilde{V}_i(u^{(i)}) + \beta^{(i)} \tilde{V}_i(v^{(i)}) = \sum_{x, y \in \{0,1\}^{s_{i+1}}} \left(\alpha^{(i)} \text{add}_i(u^{(i)}, x, y) + \beta^{(i)} \text{add}_i(v^{(i)}, x, y) \right) \cdot (\tilde{V}_{i+1}(x) + \tilde{V}_{i+1}(y)) + \left(\alpha^{(i)} \text{mult}_i(u^{(i)}, x, y) + \beta^{(i)} \text{mult}_i(v^{(i)}, x, y) \right) \cdot \tilde{V}_{i+1}(x) \tilde{V}_{i+1}(y).$$

- 4) At the input layer d , V has two claims $\tilde{V}_d(u^{(d)})$ and $\tilde{V}_d(v^{(d)})$. As V has access to the input in, it is possible to directly compute $\tilde{V}_d$ and evaluate it at two points u^d and v^d . If claims are correct, V returns 1, otherwise returns 0.

Figure 11. GKR Protocol

Polynomial Commitment Aggregation Protocol

Parameters: Let PCS = (KeyGen, Commit, Open, Verify) be the Orion [14] polynomial commitment scheme with public parameters pp. Let $f_1, f_2, \dots, f_k$ be ℓ -variate polynomials with variable-degree d , each has an evaluation $y_i = f_i(x_i)$ and committed to by σ_i with randomness r_i . M_i denotes coefficients of f_i parsed as a square matrix and $C_{i,1}, C_{i,2}$ denote the associated encoded matrices generated during the Orion commitment procedure (cf. Section 5.2).

Phase 1: Evaluation Aggregation

- 1) P and V interpolate a polynomial $L : \mathbb{F} \rightarrow \mathbb{F}^\ell$ such that $L(i) = x_i$.
- 2) $\forall i = 1$ to k , P samples random univariate polynomials $h_{i,1}, \dots, h_{i,\ell}$, each of degree $d\ell$, sets $h_i(z_1, \dots, z_\ell) \leftarrow h_{i,1}(z_1) + \dots + h_{i,\ell}(z_\ell)$, and sends $v_i \leftarrow h_i(x_i)$ and $\sigma'_i \leftarrow \text{PCS.Commit}(h_i, r'_i, \text{pp})$. M'_i denotes the coefficient matrix of h_i and $C'_{i,1}, C'_{i,2}$ denote the associated encoded matrices.
- 3) $\forall i = 1$ to k , V responds with a random challenge β_i .
- 4) $\forall i = 1$ to k , P sends the polynomial $g_i \leftarrow (f_i + \beta_i h_i) \circ L$.
- 5) $\forall i = 1$ to k , V checks whether $g_i(i) = y_i + \beta_i v_i$ and responds with random challenges α_i and r .
- 6) Parties compute $g \leftarrow \sum_{i=1}^k \alpha_i g_i$ and return $x^* = L(r)$, and $y^* = g(r)$ as output evaluation points.
- 7) P computes the output witness polynomial $f^* = \sum_{i=1}^k \alpha_i f_i + \alpha_i \beta_i h_i$.

Phase 2: Commitment Aggregation

- 1) P sends $\sigma^* \leftarrow \text{PCS.Commit}(f^*, r^*, \text{pp})$. M^* denotes the coefficient matrix of f^* and $C^*_{1,1}, C^*_{2,2}$ denote the associated encoded matrices.
- 2) V randomly samples and sends a subset of indices I of size $t \in \Theta(\lambda)$.
- 3) $\forall i = 1$ to k , $\forall \text{idx} \in I$, P sends $s^* = C^*_{2,2}[\text{idx}]$, $s_i = C_{i,2}[\text{idx}]$, and $s'_i = C'_{i,2}[\text{idx}]$ with their corresponding Merkle proofs.
- 4) $\forall i = 1$ to k , $\forall \text{idx} \in I$, V checks whether $s^* = \sum_{i=1}^k \alpha_i s_i + \alpha_i \beta_i s'_i$ and the received Merkle proofs are valid.
- 5) The verifier returns (σ^*, x^*, y^*) as the aggregate instances, the prover returns $(\sigma^*, x^*, y^*), (f^*, r^*)$ as the aggregated instance/witness tuple.

Figure 12. Polynomial Commitment Aggregation

$$\begin{aligned}
f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_f), \\
i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i), \\
o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o), \\
\tilde{c}_t &= \tanh(W_c x_t + U_c h_{t-1} + b_c), \\
c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t, \\
h_t &= o_t \odot \tanh(c_t),
\end{aligned}$$

where σ is the sigmoid function, $\odot$ denotes element-wise multiplication, and W, U are input and recurrent weight matrices. The forget gate $f_t \in [0, 1]$ controls memory retention and adaptive resets [52].

Gated Recurrent Unit (GRU). The GRU simplifies the LSTM by merging the cell and hidden states into a single activation while retaining gating-based long-term dependency modeling [53]. It uses a *reset gate* r_t and an *update gate* z_t :

$$\begin{aligned}
r_t &= \sigma(W_r x_t + U_r h_{t-1}), \\
z_t &= \sigma(W_z x_t + U_z h_{t-1}), \\
\tilde{h}_t &= \tanh(W x_t + U(r_t \odot h_{t-1})), \\
h_t &= z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t,
\end{aligned}$$

where σ is the sigmoid function and $\odot$ denotes element-wise multiplication. The gates r_t and z_t regulate forgetting and retention without a separate memory cell [53].

Appendix D.

A Recursive Framework for RNN Inference

Modern recurrent models share a recurrent state updated over time with fixed parameters. Protocol C (Fig. 13) provides a unified streaming and recursive framework by replacing only the step circuit and its proof, while keeping the recursion mechanism unchanged. A generic step circuit C takes public input x_i , previous state bundle $s_{i-1} = (s_{i-1}^{(1)}, \dots, s_{i-1}^{(r)})$, and parameters W , and outputs the next state s_i , public output y_i , and auxiliary witness aux_i . The procedure **ProveLeaf** evaluates C , commits to the new state, and produces a step proof with commitment-evaluation claims $\mathcal{E}_i^{\text{raw}}$ and wiring claims $\mathcal{W}_i^{\text{raw}}$. As in Protocol A, linear computations are proven via SumMatMul, while non-linear operations use GKR and sumcheck protocols for bit-decomposition and consistency constraints. Hence, C is split into linear and non-linear components without enlarging the main circuit. To avoid per-step opening verification, Protocol C aggregates $\mathcal{E}_i^{\text{raw}}$ into a data accumulator agg_i and compresses $\mathcal{W}_i^{\text{raw}}$ into a wiring accumulator wp_i . Each leaf package contains $\pi_{i,G}$, the aggregated component $(\text{agg}_i, \pi_{i,A}, \pi_{i,E})$, and wp_i . Recursive composition is performed over a k -ary tree. An augmented function F_A verifies child proofs and aggregation instances, derives implied claims, and propagates accumulators upward without checking polynomial openings. The parent proves correct execution of F_A and folds child accumulators into $(\text{agg}_{\text{parent}}, \text{wp}_{\text{parent}})$, yielding recursion depth $O(\log_k T)$ while preserving streaming semantics. The verifier checks only the root package: the root transcript, wiring accumulator, aggregation proof, public-input aggregate, and a single

PCS.Verify call on the root accumulator. Thus, Protocol C supports vanilla RNNs, GRUs, LSTMs, and other recurrent architectures without modifying the recursion framework.

Appendix E. Security Proofs

E.1. Proof of Theorem 3.1

Completeness follows directly from the completeness of GKR, SumMatMul, and PCS. For soundness, suppose a dishonest prover $\mathcal{P}^*$ outputs $y_i^* \neq y_i$.

Phase 1: By the soundness of GKR and Schwartz-Zippel, an incorrect output claim implies, except with negligible probability, an incorrect evaluation of either a_i or AUX_i . If the error is in a_i , SumMatMul on $s_i = W_{ya} a_i + b_y$ reduces it to a claim on a_i , W_{ya} , or b_y ; otherwise it is already a claim on AUX_i .

Phase 2: An inconsistency on a_i is reduced by the GKR proof for $a_i = F_1(z_i)$ to a claim on z_i or AUX_i . Applying SumMatMul to $z_i = W_{ax} x_i + W_{aa} a_{i-1} + b_a$ further reduces it to a claim on a_{i-1} , W_{ax} , W_{aa} , b_a , or the public input x_i .

Phase 3: The bit-decomposition sumchecks (Equations (3) and (4)) ensure that any non-binary assignment or inconsistent reconstruction yields an invalid evaluation claim on AUX_i , except with negligible probability.

Phase 4: All remaining inconsistencies are evaluation claims on committed polynomials a_i , a_{i-1} , W_{ax} , W_{aa} , W_{ya} , b_a , b_y , or AUX_i . Claims on AUX_i are compressed into a single claim $\widetilde{\text{AUX}}_i(r_{\text{in}, \text{AUX}})$ via evaluation-reduction sumcheck. By its soundness and Schwartz-Zippel, invalid claims remain invalid after aggregation except with negligible probability.

Finally, all evaluations are checked through PCS.Open. By the binding and soundness of PCS, incorrect openings are rejected except with negligible probability. Therefore, any invalid execution is rejected with overwhelming probability.

Knowledge soundness follows from the extractability of PCS, which recovers witnesses committed in σ_W , $\sigma_{a_{i-1}}$, σ_{a_i} , and σ_{AUX_i} . By the soundness of Protocol A, these witnesses satisfy the RNN-step relation. Zero-knowledge follows from the zero-knowledge properties of GKR, SumMatMul, and PCS, whose simulators produce transcripts indistinguishable from real executions.

E.2. Proof of Theorem 3.2

Correctness. Each RNN-step proof is valid by the completeness of GKR, SumMatMul, and PCS. Since F_A aggregates only valid child proofs, every internal package is valid. By induction, the root package is valid. The verifier recomputes Fiat-Shamir challenges and verifies the root proof, aggregation instance, and final PCS opening, all of which succeed for an honest prover. Hence every honestly generated proof is accepted.

Knowledge Soundness. In the Fiat-Shamir random oracle model, let ξ_{AGG} and ξ_{step} be extractors for

Protocol C (Framework): Streaming and Recursive ZK Proof of Recurrent Inference for a Generic Step Circuit C

Parameters. Let SumMatMul be the time-optimal sumcheck protocol for matrix multiplication. Let GKR be a generic GKR-style zero-knowledge proof system. Let PCS = (KeyGen, Commit, Open, Verify) be a polynomial commitment scheme. Let AGG = (Prove, Verify) be a non-interactive aggregation scheme that supports (i) aggregation of evaluation claims per polynomial family, and (ii) aggregation of public input streams. Let $k \geq 2$ be the recursion arity. Let π_0 be a trivially satisfying proof package.

Model and recurrent state. The commitment σ_W binds all parameters of C (weights, biases, etc.). The recurrent state at time t is a bundle $\mathbf{s}_t = (s_t^{(1)}, \dots, s_t^{(r)})$ of persistent hidden values, with σ_{s_0} denoting the commitment(s) to the initial state bundle.

Goal. Given an input stream $\{x_i\}_{i=1}^T$, output $\{y_i\}_{i=1}^T$ online while maintaining proof packages that can be recursively merged into a single succinct root proof.

Streaming Orchestration Maintain the current committed state bundle $(\mathbf{s}, \sigma_{\mathbf{s}})$ and buffers of pending proof packages per recursion level.

OnInput(x_i): Generate a leaf proof π_{leaf} and output y_i via ProveLeaf, return y_i immediately, store the updated state $(\mathbf{s}_i, \sigma_{\mathbf{s}_i})$, and buffer π_{leaf} (eagerly merging via ProveInternalNode whenever a level accumulates k proofs).

FinalizeProof(): Recursively merge all buffered proofs up to the root via ProveInternalNode (padding if needed), obtain π_{root} , and output (π_{root}) .

ProveLeaf($i, x_i, \sigma_W, \mathbf{s}_{i-1}, \sigma_{\mathbf{s}_{i-1}}, pp$): Generates a leaf proof for one step of the chosen recurrent architecture.

- 1) **Run the step circuit C :** Compute $(\mathbf{s}_i, y_i, \text{aux}_i) \leftarrow C(x_i, \mathbf{s}_{i-1}; W)$.
- 2) **Commit updated state:** Compute commitment(s) $\sigma_{\mathbf{s}_i} \leftarrow \text{PCS.Commit}(\mathbf{s}_i, r_{\mathbf{s}_i}, pp)$. If the step-proof requires auxiliary witness polynomials, commit to them as $\sigma_{\text{aux}_i} \leftarrow \text{PCS.Commit}(\text{aux}_i, r_{\text{aux}_i}, pp)$.
- 3) **Generate step proof:** Produce the transcript proof and the raw claims $(\pi_{i,G}, \mathcal{E}_i^{\text{raw}}, \mathcal{W}_i^{\text{raw}}) \leftarrow \text{ProofOfStep}_C.\text{Prove}(x_i, y_i, \sigma_W, \sigma_{\mathbf{s}_{i-1}}, \sigma_{\mathbf{s}_i}, \sigma_{\text{aux}_i})$. Here, ProofOfStep_C is instantiated as follows: (i) handle *linear* components of C via reductions to canonical matrix multiplication and prove them with SumMatMul; (ii) handle *non-linearities* via fixed-point/quantization plus a GKR circuit (with the necessary consistency checks proved via sumcheck sub-protocols).
- 4) **Aggregate leaf claims:** If needed, run AGG.Prove on $\mathcal{E}_i^{\text{raw}}$ *per polynomial family* to obtain the leaf data accumulator(s) and their proof/witness. Compress wiring claims $\mathcal{W}_i^{\text{raw}}$ into a wiring accumulator wp_i via evaluation reduction.
- 5) **Assemble:** Output the leaf proof package $\pi_{\text{leaf}} := (\pi_{i,G}, \pi_{i,A}, \pi_{i,E}, \emptyset, \text{agg}_i, \emptyset, \text{wp}_i)$, together with $(\sigma_{\mathbf{s}_i}, y_i)$.

ProveInternalNode($\{\pi_{\text{child}_j}\}_{j=1}^k, pp$):

- 1) **Define F_A .** On input $\{\pi_{\text{child}_j}\}_{j=1}^k$ and required public data:
 - a) Recompute Fiat–Shamir challenges / hashes needed to verify each child.
 - b) For each $j \in [1, k]$:
 - i) Parse $\pi_{\text{child}_j} = (\pi_{j,G}, \pi_{j,A}, \pi_{j,E}, \{\text{agg}_{j,\text{in},m}\}_{m=1}^k, \text{agg}_j, \{\text{wp}_{j,\text{in},m}\}_{m=1}^k, \text{wp}_j)$.
 - ii) Run the verifier for $\pi_{j,G}$ (e.g., GKR.V/SumMatMul.V) using $\{\text{agg}_{j,\text{in},m}\}_{m=1}^k, \{\text{wp}_{j,\text{in},m}\}_{m=1}^k$ and other public data. If it accepts, obtain $\mathcal{E}_j^{\text{new}}$ and wp_j .
 - iii) Run AGG.V on $\pi_{j,A}$ to check that agg_j aggregates $\{\text{agg}_{j,\text{in},m}\}_{m=1}^k$ and $\mathcal{E}_j^{\text{new}}$.
 - c) If all sumchecks (a) and aggregations (b) pass, the F_A computation is valid.
- 2) Set $u_{\text{parent}} := \sigma_{a_{s_0}} \parallel \sigma_{s_{\text{child}_1}} \parallel \dots \parallel \sigma_{s_{\text{child}_k}} \parallel \pi_{\text{child}_1} \parallel \dots \parallel \pi_{\text{child}_k}$ and commit $\sigma_{u_{\text{parent}}} \leftarrow \text{PCS.Commit}(u_{\text{parent}}, r_{u_{\text{parent}}}, pp)$.
- 3) Generate $\pi_{\text{parent},G}$ proving correct execution of F_A , including the hash/challenge computations. In the same proof, fold $\{\text{wp}_j\}_{j=1}^k$ into $\text{wp}_{\text{parent}}$ via evaluation-reduction sumcheck, and similarly fold the evaluations of $\tilde{u}_{\text{parent}}$ as required.
- 4) Compute the parent data aggregate: $(\text{agg}_{\text{parent}}, \pi_{\text{parent},A}) \leftarrow \text{AGG.P}(\{(\text{agg}_j, \pi_{j,E})\}_{j=1}^k, (\text{remaining commitment-evaluation claims}))$.
- 5) Set $\pi_{\text{parent},E}$ to $\text{PCS.Open}(\text{agg}_{\text{parent}})$ if this is the root; otherwise set it to the deferred opening witness for $\text{agg}_{\text{parent}}$.
- 6) Return $\pi_{\text{parent}} := (\pi_{\text{parent},G}, \pi_{\text{parent},A}, \pi_{\text{parent},E}, \{\text{agg}_j\}_{j=1}^k, \text{agg}_{\text{parent}}, \{\text{wp}_j\}_{j=1}^k, \text{wp}_{\text{parent}})$.

Verify($t, \sigma_W, \sigma_{s_0}, \sigma_{\mathbf{s}_t}, \pi_{\text{root}}, pp$):

- 1) parse $\pi_{\text{root}} = (\pi_G, \pi_A, \pi_E, \{\text{agg}_{\text{in},m}\}_{m=1}^k, \text{agg}_{\text{root}}, \{\text{wp}_{\text{in},m}\}_{m=1}^k, \text{wp}_{\text{root}})$.
- 2) Recompute all Fiat–Shamir challenges / hashes needed for π_G, π_A .
- 3) Run GKR.V/SumMatMul.V on π_G using public inputs $(t, \sigma_W, \sigma_{s_0}, \sigma_{\mathbf{s}_t}, \{x_i\}_{i=1}^n)$ and child inputs $\{\text{agg}_{\text{in},m}\}_{m=1}^k, \{\text{wp}_{\text{in},m}\}_{m=1}^k$ and obtain $(\mathcal{E}^{\text{root}}, \text{wp}_{\text{root}})$.
- 4) Check wp_{root} against the known wiring predicates.
- 5) Verify π_A to check agg_{root} aggregates $\{\text{agg}_{\text{in},m}\}_{m=1}^k$ and $\mathcal{E}^{\text{root}}$ (and any additional evaluation claims output by the root verification), and verify the aggregated public-input claim for the covered segment against the locally computed one.
- 6) If final, run $\text{PCS.Verify}(\text{agg}_{\text{root}}, \pi_E)$; else check π_E is a valid deferred opening witness for agg_{root} .

Figure 13. A generic framework parameterized by a per-step circuit C ; correspond to different choices of C (vanilla RNN / GRU / LSTM) and ProofOfStep_C .

AGG and ProofOfRNNStep . Given an accepting proof $\text{Verify}(i, \sigma_W, \sigma_{a_0}, \sigma_{a_i}, \pi_{\text{root}}, pp) = 1$, soundness of GKR, sumcheck, and AGG.V implies that agg_{root} cor-

rectly aggregates all accepted child instances. Applying ξ_{AGG} to $(\text{agg}_{\text{root}}, \pi_A, \pi_E)$ extracts $u_{\text{root}} = \sigma_{a_0} \parallel \sigma_{a_{\text{child}_1}} \parallel \dots \parallel \sigma_{a_{\text{child}_k}} \parallel \pi_{\text{child}_1} \parallel \dots \parallel \pi_{\text{child}_k}$, recov-

ering child commitments and proofs. Recursively applying this procedure reaches all leaf proofs. For each leaf, ξ_{step} extracts $(a_{i-1}, a_i, W_{ax}, W_{aa}, W_{ya}, b_a, b_y)$ such that $(a_i, y_i) = \text{RNN}(x_i, a_{i-1}, W_{ax}, W_{aa}, W_{ya}, b_a, b_y)$. By the binding and extractability of PCS, these witnesses define a consistent model $W = (W_{ax}, W_{aa}, W_{ya}, b_a, b_y)$ and state sequence $\{a_0, a_1, \dots, a_i\}$. Therefore, every accepting proof yields an efficiently extractable valid RNN inference trace.

Zero-Knowledge. The simulator commits to random values W^* and a_0^* and generates simulated proofs, GKR/sum-check transcripts, and the final PCS opening. Since PCS is hiding and ProofOfRNNStep, AGG, GKR, and sumcheck are zero-knowledge, the simulated transcript is computationally indistinguishable from a real execution. Thus, π_{root}^* is indistinguishable from an honestly generated proof.