Zero-Knowledge Proofs of Training for Deep Neural Networks

Kasra Abbaszadeh
University of Maryland
kasraz@umd.edu

Christodoulos Pappas
Hong Kong University of Science and Technology
cpappas@connect.ust.hk

Dimitrios Papadopoulos
Hong Kong University of Science and Technology
dipapado@cse.ust.hk

Jonathan Katz
University of Maryland
jkatz2@gmail.com

ABSTRACT

A zero-knowledge proof of training (zkPoT) enables a party to prove that they have correctly trained a committed model based on a committed dataset without revealing any additional information about the model or the dataset. An ideal zkPoT should offer provable security and privacy guarantees, succinct proof size and verifier runtime, and practical prover efficiency. In this work, we present Kaizen, a zkPoT targeted for deep neural networks (DNNs) that achieves the above ideals all at once. In particular, our construction enables a prover to iteratively train their model by the (mini-batch) gradient-descent algorithm where the number of iterations need not be fixed in advance; at the end of each iteration, the prover generates a commitment to the trained model attached with a succinct zkPoT, attesting to the correctness of the entire training process. The proof size and verifier time are independent of the iteration number.

Kaizen relies on two essential building blocks to achieve both prover efficiency and verification succinctness. First, we construct an optimized GKR-style (sumcheck-based) proof system for the gradient-descent algorithm with concretely efficient prover cost; this scheme allows the prover to generate a proof for each iteration of the training process. Then, we recursively compose these proofs across multiple iterations to attain succinctness. As of independent interests, we propose a framework for recursive composition of GKR-style proofs and techniques such as aggregatable polynomial commitment schemes to minimize the recursion overhead.

Benchmarks indicate that Kaizen can handle a large model of VGG-11 with 10 million parameters and batch size 16. The prover runtime is 22 minutes (per iteration), which is 43× faster than generic recursive proofs, while we further achieve at least 224× less prover memory overhead. Independent of the number of iterations and, hence, the size of the dataset, the proof size is 1.36 megabytes, and the verifier runtime is only 103 milliseconds.

1 INTRODUCTION

Machine learning with deep neural networks (DNNs) has received unprecedented attention in recent years. At the same time, this widespread use of neural networks has raised concerns about the provenance and integrity of DNN models; see, e.g., a recent blog post [1] from Google about integrity issues in the context of general machine learning models. Such concerns can arise at any stage of the model development process. In this work, we focus on the problems of *proofs of training* (PoTs) for ensuring the integrity of the training stage. Here, roughly speaking, a model owner wants to convince a verifier that a model $\mathcal{M}$ was correctly trained—i.e., trained using a publicly known learning algorithm and public specifications of (e.g., the model architecture and batch size) —using a specific dataset $\mathcal{D}$.[1] When the model and/or dataset are committed (i.e., not publicly available), one can also consider *zero-knowledge* PoTs (zkPoTs) that reveal no additional information about $\mathcal{M}$ or $\mathcal{D}$ beyond the correctness of the training procedure. At the most basic level, a (zk)PoT can be used by a model owner to substantiate the provenance of a model they release; this can be useful even when the model or the dataset is public since the cost to train a model can be orders of magnitude more expensive than verifying that training was correct. A (zk)PoT can also be useful for other applications:

- ***Arbitrating copyright claims.*** There is growing concern about models being trained on copyrighted data [35], and a model owner may want to prove that their model was trained *without* using some particular copyrighted data item. To achieve this goal, the model owner can commit to the dataset $\mathcal{D}$ used to train the model, prove using known techniques [11, 47] that the data item in question is not present in $\mathcal{D}$, and then use a (zk)PoT to show that the model was trained using the committed dataset $\mathcal{D}$.
- ***Distributed training.*** A model owner might distribute the task of training a model across multiple untrusted workers [44]. In this setting, the model owner may send portions of the dataset to each worker and have them return partial results, which can then be aggregated into a final model. PoTs can allow workers to prove that the partial model they return was trained correctly.
- ***Proof of ownership.*** A deep learning model $\mathcal{M}$ might be stolen or extracted using model-inference attacks [54]. In that case, the real model owner can prove that they were the ones who trained $\mathcal{M}$ by committing to the dataset $\mathcal{D}$ they used to train the model and then using a PoT to prove that $\mathcal{M}$ was trained using $\mathcal{D}$. An adversary would not be able to generate such a proof without access to $\mathcal{D}$; even with such access, they would have to invest the resources necessary to train the model from scratch.

An ideal zkPoT should have strong security guarantees, be succinct[2], and not impose too much overhead on the prover. As we discuss next, however, known techniques fall short of achieving at least one of these properties when it comes to proofs of training for DNNs.

1.1 Prior Work and Limitations

Jia et al. [36] proposed a proof of training for DNNs. Unfortunately, their construction does not offer cryptographically strong security guarantees, and recent work [25, 26] has identified various concrete attacks on their scheme. Moreover, their construction is neither

[1] We note this is different from *proofs of inference* [27, 43, 45, 57] where, given a model $\mathcal{M}$ and an input x, the goal is to prove that the classification of x was done using $\mathcal{M}$.
[2] Succinctness means the proof size and the verifier time are sublinear the computation size. For some applications it might be critical; consider e.g., a prover posting a (zk)PoT on blockchain [3], and it is desirable to minimize the on-chain verification overhead.

zero-knowledge nor succinct; it reveals model weights and data items to the verifier who partially re-executes training. Recently, Garg et al. [29] constructed zkPoTs with provable security; however, their construction is not succinct and, as a result, only supports basic learning algorithms such as logistic regression but cannot support DNNs due to the significant verification overhead.

In principle, one can build a zkPoT using generic zero-knowledge proofs. Unfortunately, despite recent advances in such constructions [6, 20, 34, 59], they are still not sufficiently scalable to practically instantiate zkPoTs for complex models like DNNs. In particular, succinct zero-knowledge proofs (zkSNARKs) [13, 38, 48] incur prohibitive prover costs in terms of both the proof-generation time (at least 1000× slowdown compared to the training time) and the required memory overhead. On the other hand, constructions of proofs with efficient provers [12, 30, 61] are not succinct.

As already noted, existing works on zero-knowledge proofs of inference [43, 45, 57] solve a different (and somewhat easier) problem than the one we are considering here. While some of the optimizations they propose are also useful in our setting, the techniques used in those works are not sufficient to yield efficient zkPoTs for DNNs—note, in particular, that DNN training involves multiple iterations, where each iteration can be roughly 100× more expensive than DNN inference. Eisenhofer et al. [23] used generic SNARKs to address the problem of verifiable machine unlearning, where the goal is to retrain a model by removing particular items from the dataset. The computational complexity of unlearning is fairly low; thus, generic proofs can efficiently support this problem.

1.2 Our Contributions

In this work, we propose a technique for constructing efficient and succinct zkPoTs for DNN models that are trained using multiple iterations of the mini-batch gradient-descent algorithm. We also implement and evaluate Kaizen,[3] an instantiation of our approach. Conceptually, our work involves two high-level components:

Efficient proofs of gradient descent. First, we construct an optimized *proof of gradient descent* (PoGD), a scheme for proving the correctness of a single gradient-descent iteration. Our PoGD is constructed from GKR-style (i.e., sumcheck-based) proof systems [59, 62, 63] since they offer concretely efficient prover overhead and succinct verification. Moreover, such proofs can be optimized for matrix operations such as matrix multiplication or convolution used in gradient-descent for DNNs to obtain sublinear proof-generation time [45, 53]; this is in contrast to generic proofs, which have prover overhead at least linear in the complexity of the computation. More details about our PoGD are presented in Sections 3.1 and 4.

Recursive composition of sumcheck-based proofs. Although our PoGDs offer succinct proofs for each gradient-descent iteration, they do not suffice to obtain succinct proofs for DNN training overall because such training involves a number of iterations that are typically linear in the size of the dataset. Thus, having the PoT include a PoGD for each iteration would result in a linear-sized proof. Alternately, using a single GKR-style proof for all iterations would result in a linear prover memory and verification overheads; note such proofs have verification linear in the computation depth.

[3] Kaizen is a Japanese term translated as "change for the better".

Instead, we use recursive proof composition, also referred to as *incrementally verifiable computation* (IVC) [10, 14, 55], to achieve succinctness. Roughly, at each iteration we prove both that the iteration was performed correctly and (recursively) that there is a valid proof of correctness for all previous iterations. In this way, the proof generated at the final iteration demonstrates the correctness of the entire computation with a proof size and verification cost independent of the iteration number. Additionally, the incremental nature of an IVC allows for prover memory cost only proportional to the complexity of one iteration, not the entire computation.

The challenge in recursively composing our sumcheck-based PoGDs is the relatively large circuit size of the verifier algorithm, which imposes a significant recursion cost; this is in contrast to generic IVC schemes [14, 17, 39], which offer efficient recursion cost but incur substantial prover cost overall for the gradient-descent computation since they rely on generic zero-knowledge proofs. To overcome this, we propose a framework for minimizing the recursion overhead when building IVC from sumcheck-based proofs. In particular, inspired by Halo [17] and follow-up works [15], we use aggregation schemes to reduce the cost of verifying the polynomial commitments used in the PoGDs for all executed iterations.

A difficulty here is that sumcheck-based proofs use commitments to *multivariate* polynomials instead of the *univariate* polynomials involved in prior work [15, 17]; existing aggregation schemes for polynomial commitments only handle univariate polynomials, and naively extending them to support multivariate polynomials would result in quasilinear prover overhead. To mitigate this limitation, and as a key building block for our recursion framework, we propose an aggregation scheme for multivariate polynomial commitments with linear prover overhead and logarithmic verifier overhead. More details of our aggregation scheme are presented in Section 5.

We further improve the overhead of recursive composition of sumcheck-based proofs with sumcheck-specific optimizations for lightweight hash functions [7] and aggregation of the evaluations of circuit-wiring predicates; see Sections 3.2 and 6 for more details. To our knowledge, building IVC from sumcheck-based proofs has not been considered by prior work, and our framework and techniques might be of independent interest as stand-alone primitives.

Implementation and evaluations. We evaluate Kaizen on several well-known architectures [41, 42, 51], e.g., a convolutional model VGG-11 with 10 million parameters, 11 layers, and batch size 16. The model is trained on the CIFAR-10 dataset. Our prover time is 22 minutes per iteration, and independent of the number of iterations and the dataset size, the proof size is 1.36 megabytes, and the verifier time is only 103 milliseconds. Moreover, we compare Kaizen with generic IVCs [17, 21, 39] as the baseline, where we achieve **43**× faster prover time and at least **224**× less prover memory usage.

1.3 Organization of the Paper

In Section 2, we introduce notation and include some background information on zero-knowledge proofs and DNNs. In Section 3, we present an overview of our techniques. Section 4 describes our PoGD protocol. We introduce our aggregation scheme in Section 5, and use this as a key building block of our recursive sumcheck framework presented in Section 6. In Section 7, we present Kaizen, which we evaluate and compare to prior generic IVCs in Section 8.

2 PRELIMINARIES

We use $\mathbb{F}$ to denote a finite field. We let λ be the security parameter, and negl be a negligible function. We define $[n] := \{0, 1, \ldots, n-1\}$. We use bold lowercase letters $\boldsymbol{x}, \boldsymbol{y}$ for vectors, and bold uppercase letters $\boldsymbol{X}, \boldsymbol{Y}$ for matrices. For a vector $\boldsymbol{x}$, we use both $\boldsymbol{x}_i$ and $\boldsymbol{x}[i]$ to denote the ith element of $\boldsymbol{x}$, and for a matrix $\boldsymbol{X}$, both $\boldsymbol{X}_{i,j}$ and $\boldsymbol{X}[i, j]$ indicate the element in row i and column j. We use $\boldsymbol{X}[i, :]$ to denote the ith row and $\boldsymbol{X}[:, j]$ to denote the jth column of the matrix $\boldsymbol{X}$. We use $\circ$ for the Hadamard (element-wise) product.

Merkle tree. A Merkle tree [47] MT is a data structure that can be used to commit to a vector. It consists of the following algorithms:

- Commit: on input a vector $\boldsymbol{x}$ and randomness r, returns the root ρ of the Merkle tree as a commitment to the vector.
- Open: on input a vector $\boldsymbol{x}$, randomness r, and opening index i, returns an element $\boldsymbol{x}_i$ and proof p_i.
- Verify: on input a commitment ρ, index i, element $\boldsymbol{x}_i$, and proof p_i, returns accept or reject.

The commitment time is linear in the length of the vector; proofs and verifier times are logarithmic in the length of the vector.

Multilinear extensions. Let $V : \{0,1\}^\ell \to \mathbb{F}$ be a function. The *multilinear extension* of V is defined as the unique multilinear polynomial $\widetilde{V} : \mathbb{F}^\ell \to \mathbb{F}$ with $\widetilde{V}(\boldsymbol{x}) = V(\boldsymbol{x})$ for all $\boldsymbol{x} \in \{0,1\}^\ell$. For $\boldsymbol{b} \in \{0,1\}^\ell$, let $\beta_{\boldsymbol{b}} : \{0,1\}^\ell \to \{0,1\}$ denote the function with $\beta_{\boldsymbol{b}}(\boldsymbol{x}) = 1$ if $\boldsymbol{b} = \boldsymbol{x}$ and $\beta_{\boldsymbol{b}}(\boldsymbol{x}) = 0$ otherwise, and note that $\widetilde{\beta}_{\boldsymbol{b}}(\boldsymbol{x}) = \prod_{i=1}^{\ell}((1-\boldsymbol{x}_i)(1-\boldsymbol{b}_i) + \boldsymbol{x}_i\boldsymbol{b}_i)$. We can then compute $\widetilde{V}$ as

$$\widetilde{V}(\boldsymbol{x}) = \sum_{\boldsymbol{b}\in\{0,1\}^\ell} \widetilde{\beta}_{\boldsymbol{b}}(\boldsymbol{x}) \cdot V(\boldsymbol{b}).$$

We can define multilinear extensions of vectors by viewing a vector $\boldsymbol{v} = (v_0, v_1, \ldots, v_{n-1}) \in \mathbb{F}^n$ as a function $v : \{0,1\}^{\log n} \to \mathbb{F}$ such that $\forall i \in [n] : v(i) = \boldsymbol{v}_i$. A similar method is applied to the matrices.

2.1 Proofs, Arguments, and Commitments

An interactive proof for relation R with corresponding language L_R is an interactive protocol between a prover $\mathcal{P}$ and a verifier $\mathcal{V}$ on joint input x. The protocol satisfies *completeness* if, in an honest execution when the prover additionally holds a witness w with $(x, w) \in R$, the verifier always accepts. *Soundness* guarantees that a malicious $\mathcal{P}$ cannot convince $\mathcal{V}$ to accept when $x \notin L_R$, except with negligible probability. *Knowledge soundness* is a stronger property requiring that if a malicious prover can cause the verifier to accept with high probability on joint input x, then it is possible to extract a witness w from the prover such that $(x, w) \in R$. Such a protocol is called an *argument* if (knowledge) soundness holds against only computationally-bounded provers. Informally, a proof/argument is *zero-knowledge* if a malicious verifier learns no information from the protocol about w other than $(x, w) \in R$. We say a proof/argument is *succinct* if the runtime of $\mathcal{V}$ and the communication between $\mathcal{P}$ and $\mathcal{V}$ are $\mathrm{poly}(\lambda, |x|, \log|w|)$; see Appendix A.1 for formal definitions.

A proof/argument is *public-coin* if the messages sent by the honest verifier consist simply of random challenges. Under certain conditions, a public-coin protocol can be made non-interactive using the Fiat-Shamir transform [28] in the random-oracle model by replacing the challenge for each round with the output of the random oracle evaluated on the transcript of prior rounds.

Incrementally verifiable computation (IVC). IVC schemes [55] enables succinct verification of iterative computations. Let $\mathcal{F}$ be a function, $\omega_0, \ldots, \omega_{i-1}$ be witness auxiliary inputs, and z_0 be an initial input; define $\forall k \in [i] : z_{k+1} = \mathcal{F}(z_k, \omega_k)$. IVC allows a prover to incrementally generate a proof π_i substantiating that there exist $\omega_0, \ldots, \omega_{i-1}$ such that z_i was correctly computed from z_0. More formally, an IVC protocol consists of the following algorithms:

- $\mathcal{G}$: on input security parameter, returns public parameters pp.
- $\mathcal{P}$: on input an iteration counter i, initial input z_0, last output z_{i-1}, auxiliary input ω_{i-1}, proof π_{i-1}, and public parameters pp, returns the next iteration output z_i and a proof π_i.
- $\mathcal{V}$: on input an iteration counter i, initial input z_0, last output z_i, proof π_i, and public parameters pp, returns accept or reject.

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

$$R_{\mathsf{IVC}_i} = \left\{ \begin{array}{c} (i, z_0, z_i), (\omega_0, \ldots, \omega_{i-1}) : \\ \forall k \in [i] : z_{k+1} = \mathcal{F}(z_k, \omega_k) \end{array} \right\}.$$

A canonical technique for constructing IVC is recursive composition of a baseline succinct non-interactive proof. In particular, let $\mathcal{P}_b$ and $\mathcal{V}_b$ be the prover and verifier algorithms of the baseline proof, respectively. Let $\mathcal{F}_A$ be the augmented function defined as:

$$(\mathcal{F}(z_i, \omega_i), \mathcal{V}_b(i, z_0, z_i, \pi_i)) \leftarrow \mathcal{F}_A(i+1, z_0, z_i, \omega_i, \pi_i).$$

$\mathcal{P}$, on input $i+1, z_i, \omega_i$, and π_i, outputs z_{i+1} and invokes $\mathcal{P}_b$ to generate a proof π_{i+1} that $\mathcal{F}_A(i+1, z_0, z_i, \omega_i, \pi_i) = (z_{i+1}, 1)$. The incremental design of IVC ensures that prover memory overhead, proof size, and verifier overhead are independent of the number of iterations and only proportional to the complexity of $\mathcal{F}_A$.

Polynomial commitments. A polynomial commitment scheme (PCS) enables a prover $\mathcal{P}$ to generate a commitment to a polynomial and later open the polynomial at some input evaluation point. More formally, a PCS consists of the following procedures:

- KeyGen: on input the security parameter, number of variables ℓ, and degree bound d, returns public parameters pp.
- Commit: on input a polynomial f, randomness r, and public parameters pp, returns a commitment σ to the polynomial.
- Open: on input a point x, a polynomial f, the randomness r, and public parameters pp, returns the evaluation $y = f(x)$ and an evaluation opening proof π.
- Verify: on input a commitment σ, point x, evaluation y, and evaluation opening proof π, returns accept or reject.

A PCS is *hiding* if the commitment reveals no information about the polynomial. A PCS satisfies *evaluation binding* if a prover cannot open a commitment σ to two distinct evaluations at any input point x. Knowledge soundness ensures that a prover generating a correct evaluation proof must know the polynomial underlying the commitment. A zero-knowledge PCS guarantees that evaluation proofs reveal no additional information about f other than the value $f(x)$. Formal definitions are provided in Appendix A.1.

2.2 GKR-Based Zero-Knowledge Arguments

Goldwasser et al. [31] proposed an interactive proof system for layered arithmetic circuits, referred to as the GKR protocol, based on the sumcheck protocol. We review the scheme here.

The sumcheck protocol. Fix an ℓ-variate polynomial $f : \mathbb{F}^\ell \to \mathbb{F}$ with variable-degree bound d. The sumcheck protocol [46] enables a verifier $\mathcal{V}$ to delegate computation of the sum of f over the binary hypercube, i.e., $H = \sum_{b\in\{0,1\}^\ell} f(b)$, to a prover $\mathcal{P}$. At the end of the protocol, $\mathcal{V}$ needs the ability to learn the evaluation of f at a single (random) point $r \in \mathbb{F}^\ell$. The prover time is $O(d^\ell)$, and the verification time and proof size are $O(d\ell)$. The soundness error is $O(d\ell/|\mathbb{F}|)$. We present the scheme in Protocol 1 of Appendix A.

The GKR protocol. Let C be a circuit of depth d over a finite field $\mathbb{F}$, with layer 0 the output layer and layer d the input layer, and where each gate in the ith layer takes two inputs from the $(i+1)$st layer. Let S_i be the number of gates in the ith layer, and $s_i := \lceil \log S_i \rceil$. Define *wiring predicates* $add_i, mult_i : \{0,1\}^{s_i+2s_{i+1}} \to \{0,1\}$, where add_i (resp., $mult_i$) takes as input a gate label z for layer i, and gate labels x, y for layer $i+1$, then returns 1 iff gate z is an addition (resp., multiplication) gate with inputs x and y. Fixing some input to the circuit, we define $v_i : \{0,1\}^{s_i} \to \mathbb{F}$ so that $v_i(b)$ is the output of the bth gate in the ith layer. Note that

$$\widetilde{v}_i(z) = \sum_{x,y\in\{0,1\}^{s_{i+1}}} \widetilde{add}_i(z,x,y)\cdot(\widetilde{v}_{i+1}(x)+\widetilde{v}_{i+1}(y)) + \widetilde{mult}_i(z,x,y)\cdot\widetilde{v}_{i+1}(x)\cdot\widetilde{v}_{i+1}(y). \quad (1)$$

$\mathcal{P}$ can prove the evaluation of C as follows. Assume $\mathcal{V}$ has oracle access to $\widetilde{v}_0$ and $\widetilde{v}_d$ (i.e., the input and output). $\mathcal{V}$ samples a random challenge r_0 and evaluates $\widetilde{v}_0(r_0)$. Then, $\mathcal{P}, \mathcal{V}$ run the sumcheck protocol on Equation (1) for $i = 0$, at the end of which $\mathcal{V}$ asks $\mathcal{P}$ for two evaluations of $\widetilde{v}_1$, say at points $r_{1,0}, r_{1,1}$. Say $\mathcal{P}$ claims $y_0 = \widetilde{v}_1(r_{1,0})$ and $y_1 = \widetilde{v}_1(r_{1,1})$. The verifier then chooses $\gamma_0, \gamma_1 \leftarrow \mathbb{F}$ and the parties run the sumcheck protocol to prove that

$$\gamma_0\cdot y_0 + \gamma_1\cdot y_1 = \sum_{x,y\in\{0,1\}^{s_{i+1}}} (\gamma_0\cdot\widetilde{add}_i(r_{1,0},x,y) + \gamma_1\cdot\widetilde{add}_i(r_{1,1},x,y))\cdot(\widetilde{v}_{i+1}(x)+\widetilde{v}_{i+1}(y)) + (\gamma_0\cdot\widetilde{mult}_i(r_{1,0},x,y) + \gamma_1\cdot\widetilde{mult}_i(r_{1,1},x,y))\cdot\widetilde{v}_{i+1}(x)\cdot\widetilde{v}_{i+1}(y).$$

This reduces the problem to a claim of two evaluations of $\widetilde{v}_1$. The parties then continue this procedure for d iterations; at the final iteration $\mathcal{V}$ can check the evaluations of $\widetilde{v}_d$ using its oracle access. The formal protocol is provided in Protocol 2 of Appendix A. The GKR protocol can be made zero-knowledge by using zero-knowledge sumchecks [16, 59] and zero-knowledge polynomial commitments to provide oracle access to $\widetilde{v}_0$ and $\widetilde{v}_d$. The protocol can be made non-interactive using the Fiat-Shamir transform.

2.3 Deep Neural Networks and Training

A DNN maps an input feature to a prediction by applying a sequence of transformations across L *layers*. The ℓth layer consists of a linear operation parameterized by weights $\boldsymbol{W}_\ell$ followed by application of a non-linear activation function. A layer is *dense* or *convolutional*. In dense layers, the linear operation returns $\boldsymbol{T} = \boldsymbol{W}\cdot\boldsymbol{U}$ given an input $\boldsymbol{U}$ and weights $\boldsymbol{W}$, and convolutional layers returns $\boldsymbol{T} = \boldsymbol{W} * \boldsymbol{U}$, where if $\boldsymbol{U}$ is of size $u\times u$ and $\boldsymbol{W}$ is of size $w\times w$, then $\boldsymbol{T}$ is a $(u-w+1)\times(u-w+1)$ matrix such that

$$\boldsymbol{T}[i,j] = \sum_{a,b=0}^{w-1} \boldsymbol{U}[i+a, j+b]\cdot\boldsymbol{W}[a,b].$$

Following the linear operation, an activation function is performed; this is $\mathrm{ReLU}(x) = \max(x,0)$, $\tanh(x) = (e^{2x}-1)/(e^{2x}+1)$, each applied coordinate-wise, or $\mathrm{Softmax}(x_i) = (e^{x_i})/(\sum_j e^{x_j})$. A layer may also include a pooling that slides a *window* over the input and extracts one element from each window deterministically, e.g., it returns the maximum or average value in each of the windows.

Training DNNs using mini-batch gradient-descent. When training a DNN, the architecture, i.e., the number of the layers, the type of each layer, and an initial set of weights, is fixed in advance, and the goal is to optimize the weights at each layer to minimize a specified loss function $\mathcal{L}$ over the training dataset. The training dataset is a set of data items, where each items consists of a feature and associated labels. The most popular optimization method for DNNs is mini-batch gradient-descent [50], which proceeds in a series of *epochs*. In each epoch, the dataset $\mathcal{D}$ is randomly partitioned into batches $\boldsymbol{B}_1, \ldots, \boldsymbol{B}_k$, each containing a specified number N of data items. An epoch consists of k iterations (one per batch). In the ith iteration, let $\boldsymbol{U}_{i,0}$ be the concatenation of the input features of the data items in $\boldsymbol{B}_i$ and let $\boldsymbol{U}_{i,\ell}$ (for $\ell > 0$) be the concatenation of outputs of the ℓth layer given the current weights. In the ith iteration the current weights $\{W_{i-1,\ell}\}_{\ell=1}^{L}$ are updated to obtain new weights $\{W_{i,\ell}\}_{\ell=1}^{L}$ via the following steps:

(1) *Forward pass:* We feed the inputs $\boldsymbol{U}_{i,0}$ to the model, apply layers sequentially, and for $\ell = 1$ to L, compute $\boldsymbol{U}_{i,\ell}$.

(2) *Backward pass:* Let H_i be the average loss over the batch, and define the gradients $\boldsymbol{G}_{i,\ell} := \partial H_i/\partial\boldsymbol{W}_{i-1,\ell}$ and $\boldsymbol{R}_{i,\ell} := \partial H_i/\partial\boldsymbol{U}_{i,\ell-1}$. These are computed for $\ell = L$ to 1 as:

$$\boldsymbol{G}_{i,\ell} = \boldsymbol{R}_{i,\ell+1}\cdot\frac{\partial\boldsymbol{U}_{i,\ell}}{\partial\boldsymbol{W}_{i-1,\ell}},\ \boldsymbol{R}_{i,\ell} = \frac{\partial\boldsymbol{U}_{i,\ell}}{\partial\boldsymbol{U}_{i,\ell-1}}\cdot\boldsymbol{R}_{i,\ell+1}. \quad (2)$$

$\boldsymbol{R}_{i,L+1}$ is evaluated using the labels of the data items. Moreover, the gradients $\partial\boldsymbol{U}_{i,\ell}/\partial\boldsymbol{W}_{i-1,\ell}$ and $\partial\boldsymbol{U}_{i,\ell}/\partial\boldsymbol{U}_{i,\ell-1}$ are evaluated by applying some linear operations and non-linear activations to the inputs $\boldsymbol{U}_{i,\ell-1}$ and weights $\boldsymbol{W}_{i-1,\ell}$, respectively.

(3) *Update:* Given a learning rate η_ℓ for each layer, the weights are updated as $\boldsymbol{W}_{i,\ell} = \boldsymbol{W}_{i-1,\ell} - \eta_\ell\cdot\overline{\boldsymbol{G}}_{i,\ell}$, where $\overline{\boldsymbol{G}}_{i,\ell}$ indicates the average of $\boldsymbol{G}_{i,\ell}$ over the data items in the batch.

A high-level illustration is presented in Figure 1, and more details are presented in Algorithm 1 of Appendix A.

3 TECHNICAL OVERVIEW

In this section, we provide an overview of our techniques; low-level technical details are still deferred to subsequent sections.

3.1 Sumcheck Proofs for Gradient Descent

Our PoGD is a GKR-style sumcheck-based proof for one iteration of the mini-batch gradient-descent algorithm (cf. Section 2.3). In particular, given a fixed batch of data points and some model weights, PoGD enables the prover to update the weights by applying a gradient-descent iteration, and then, convince the verifier that the update is executed correctly. For the linear portions of the algorithm, we use optimized sumcheck-based proofs with sublinear proof-generation time. For the non-linear operations, we use bit decompositions along with a generic version of the GKR protocol. We discuss each of these components in more detail next.

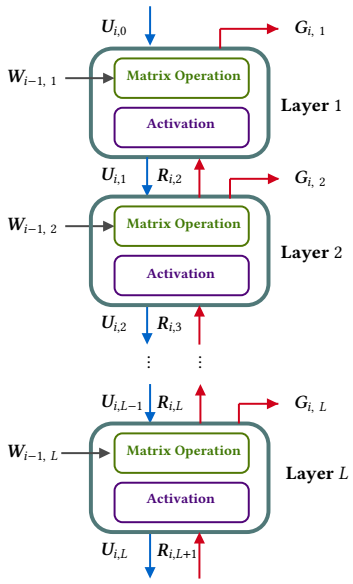

Figure 1: A high-level illustration of the ith gradient-descent iteration. $U_{i-1,0}$ denotes the input features, $U_{i,\ell}$ (for $\ell > 0$) denotes the output of layer ℓ, and $W_{i-1,\ell}$ denotes the current weights of layer ℓ. $R_{i,\ell}$ and $G_{i,\ell}$ are the gradients computed in the backward pass. Blue are red arrows indicate the flow of the forward and backward passes, respectively.

Linear operations. Linear operations, such as matrix multiplication or convolution, are the most computationally intensive parts of the gradient-descent algorithm. Fortunately, prior work [45, 53] has shown optimized sumcheck protocols for these operations that have proof-generation time sublinear in the runtime of the computation itself. In particular, Thaler [53] proposed a sumcheck proof for the multiplication of two $n \times n$ matrices with $O(n^2)$ prover time; note the underlying computation takes time $\Theta(n^3)$ if naive matrix multiplication is used, and even the best matrix multiplication algorithms require time $\omega(n^2)$. More recently, Liu et al. [45] gave a sumcheck protocol for convolution of an $n \times n$ input matrix with a $w \times w$ weights where the prover time is $O(n^2 + w^2)$, asymptotically faster than computing the convolution itself in $O(n^2 \cdot w^2)$ time.

Handling non-linear operations. Sumcheck proofs only support arithmetic operations over finite fields. However, neural networks use non-linear operations that cannot be compactly encoded in an arithmetic circuit, even once the underlying values have been quantized (cf. Section 4.1). We use standard techniques to handle these. For example, proofs about a comparison such as $a \geq 0$ can be handled using bit decomposition: the prover provides the binary representation of a (including a sign bit) as an auxiliary input, and the verification circuit checks that the binary representation is consistent. The sign bit is then taken as the result of the comparison. To reduce the size of the circuit and wiring predicates, we express consistency checks directly as sumcheck instances rather than reducing them to an arithmetic circuit. As another example, we handle exponentiations using piece-wise linear approximations, which have been shown to provide sufficient accuracy [19, 24, 49].

3.2 Recursive Composition of Sumcheck Proofs

A challenge in applying recursive proof composition to our PoGD is the relatively large size of the baseline verification circuit $\mathcal{V}_b$ and hence $\mathcal{F}_A$ (cf. the discussion of IVC in Section 2.1). The circuit $\mathcal{V}_b$ mainly consists of three components: verifying opening proofs for polynomial commitments, verifying sumcheck messages, and verifying the hashes of messages used to generate the challenge for the Fiat-Shamir transform. We discuss how we improve the efficiency of each of these components.

Commitment openings. For a GKR-based zkSNARK, the verifier needs to verify openings of several polynomial commitments. It is thus desirable to have a polynomial commitment scheme with a small verification circuit. Known constructions fall short: schemes based on pairings [37] have $O(1)$ verification overhead, but are not compatible with recursive proof composition [10], schemes based on Merkle trees [8, 32, 63] require the verifier to compute $\Omega(\log^2 N)$ hash evaluations for a polynomial of size N, and constructions based on prime-order groups [18, 56] have verification time $\Omega(\sqrt{N})$ making them impractical for recursive proof composition.

Several prior IVC schemes [15, 17] resolved this problem by employing commitment aggregation. Roughly speaking, rather than verifying the opening of commitments in each recursive step, the commitments across all steps are aggregated and the aggregate is verified at the end. Unfortunately, the aggregation scheme used in prior work only applies to polynomial commitment schemes for *univariate* polynomials, whereas for applications to GKR-based proofs we need commitments to *multivariate* polynomials. Naively extending existing univariate schemes to the multivariate case would incur superlinear, i.e., $\Theta(N \log N)$, prover time.

In this work, we propose an aggregation scheme compatible with multivariate polynomial commitments. For polynomials with N coefficients, the scheme has $O(N)$ prover time and $O(\log N)$ proof size and verifier time. Our aggregation technique is inspired by the evaluation reduction technique [31], which is a specific case of aggregation where multiple evaluations of a polynomial are combined into a single evaluation. We generalize this idea to the case of having multiple committed polynomials with multiple evaluation points. Our techniques applies to various polynomial commitment schemes. We also make the proofs zero-knowledge.

Sumcheck messages. $\mathcal{V}_b$ must evaluate the multilinear extensions of the wiring predicates at several random points. Evaluating these polynomials requires time linear in the size of the circuit. Interactive implementations of the GKR protocol mitigate this overhead by preprocessing the evaluations; the verifier samples the random points and computes the evaluations in advance [62, 63]. However, for recursive proof composition the baseline proof system must be non-interactive and such preprocessing is not possible.

We avoid this overhead using an approach similar to aggregation. The various wiring-predicate evaluations are combined into a single evaluation by a sumcheck run. The combined evaluation is passed to the next iteration, and verification is deferred until the end. While a similar effect could also be achieved using the evaluation reduction technique, which is a specific case of our commitment aggregation scheme, using the sumcheck protocol offers concretely better prover performance in this setting.

Fiat-Shamir challenges. Due to the Fiat-Shamir transform, $\mathcal{V}_b$ must verify the evaluation of a hash function on many inputs. In particular, $O(d_A \cdot \log|\mathcal{F}_A|)$ hashes need to be verified, where d_A denotes the depth of the augmented function $\mathcal{F}_A$. We use MiMC [4], a sumcheck-friendly hash, to reduce this cost. MiMC applies a low-degree round function for multiple rounds. As observed in prior work [7], we can represent each round of MiMC directly as a sumcheck instance, rather than by explicitly writing it as an arithmetic circuit, and thus further reduce the size of $\mathcal{V}_b$.

3.3 Constructing a zkPoT

We construct a zkPoT for DNNs by recursively composing our sumcheck-based PoGDs across multiple iterations. In doing so, there is one additional complication: we need to ensure that the prover uses a (random) partition of the committed dataset in each epoch. We ensure this as follows. The prover commits to the dataset using a Merkle tree. In each epoch, the prover then partitions the dataset using a public "quasi-random" permutation, e.g., a cubic function, applied to the indices of the n items in the dataset; the key for the permutation is varied for each epoch. During each iteration of an epoch, the prover provides the data items contained in the current partition along with their Merkle proofs. As part of computing gradient-descent iterations, the we additionally check the correctness of the partition as well as the Merkle proofs.

4 PROOFS OF GRADIENT DESCENT

We present our *proof of gradient descent* (PoGD), which is a GKR-style sumcheck-based proof for one mini-batch gradient descent iteration.

4.1 Handling Fixed-Point Operations

Model weights and data items are often represented by real numbers, but we need to encode them as field elements. We fix a field size p large enough to ensure no overflow during the computation. We quantize a real number r as a q-bit integer ($q \ll p$) with f-bit precision as follows: when $0 \le r \le 2^{q-f}$, map r to $s = \lfloor r \cdot 2^f \rceil \in \mathbb{F}_p$, and when $-2^{q-f} \le r < 0$, map r to $s = p - \lfloor r \cdot 2^f \rceil \in \mathbb{F}_p$.

Integer arithmetic and comparison. Two numbers can be added by adding their quantizations. For the multiplication of numbers a and b with quantizations s_a and s_b, respectively, we need to scale $z = s_a s_b$ by a factor of 2^{-f}. For this purpose, we ask the prover to provide $z_0, \ldots, z_{q-1}$, the bit-decomposition of z, as an auxiliary input along with a sign bit z_q with $z_q = 1$ indicating negativity; The verifier checks whether the provided bit-decomposition consists of binary values consistent with z, and returns the truncated output; i.e., if $z = (-1)^{z_q} \sum_{i=0}^{q-1} z_i \cdot 2^i$ output $(-1)^{z_q} \cdot \sum_{i=0}^{q-f-1} z_{i+f} \cdot 2^i$.

A similar technique can be used for comparisons. Say we need to check whether $z \ge 0$. The prover can provide the bit-decomposition of z, including a sign bit; this can be easily checked by the verifier, and the result of the comparison is then the sign bit. We can also handle divisions. To compute the quotient a/b, where s_a and s_b are the quantizations of a and b, respectively, the prover provides a quotient z and a remainder r. The verifier then checks whether $s_a = z \cdot s_b + r$ and $r < s_b$ and returns $z \cdot 2^f + r \cdot 2^{-f}$ as the output. To further reduce the circuit size, we accumulate scaling factors for each layer and defer scalings of to the output of the layer.

Sumcheck for binary operations. Scalings and bit-decomposition consistency checks can be handled by a field arithmetic circuit and verified by applying a generic proof to the circuit. However, for complex models, such binary operations blows up the size of the circuit and wiring predicates. To overcome this, we express these operations directly as sumcheck instances. In particular, let z be an n-sized vector of values that the prover is required to provide their bit-decomposition. Let Z_{Bit} be a matrix of size $n \times q$ and z_{Sign} be an n-sized vector, where $Z_{\mathsf{Bit}}[i,:]$ is claimed to be the bit-decomposition of $z[i]$, and $z_{\mathsf{Sign}}[i]$ indicates the associated sign bit. To check whether Z_{Bit} and z_{Sign} are binary, the verifier sends challenges r, r_x, r_y, and parties run the sumcheck protocol on:

$$0 = \sum_{x \in \{0,1\}^{\log n}, y \in \{0,1\}^{\log q}} \widetilde{\beta}_x(r_x) \cdot \widetilde{\beta}_y(r_y) \cdot \left(r \cdot \widetilde{z}_{\mathsf{Sign}}(x) \cdot (1 - \widetilde{z}_{\mathsf{Sign}}(x)) + \widetilde{Z}_{\mathsf{Bit}}(x,y) \cdot (1 - \widetilde{Z}_{\mathsf{Bit}}(x,y))\right). \quad (3)$$

The verifier then sends another challenge r_z, and parties verify the consistency of the bit-decompositions by running a sumcheck on:

$$\widetilde{z}(r_z) = \sum_{x \in \{0,1\}^{\log n}, y \in \{0,1\}^{\log q}} \widetilde{\beta}_x(r_z) \cdot \widetilde{Z}_{\mathsf{Bit}}(x,y) \cdot (1 - 2 \cdot \widetilde{z}_{\mathsf{Sign}}(x)) \cdot 2^y. \quad (4)$$

We can handle scalings similarly. In particular, we only need to restrict the range of x and y to the proper intervals in Equation (4); the range of x selects which values are targeted and must be scaled, and the range of y determines what scaling factor we apply.

Exponentiation. Natural exponentiation is used in functions such as Softmax and tanh. In practice, weights and data items are typically normalized to the interval $[-1, 1]$. Similar to prior work [29, 49], we employ the following piece-wise approximation for e^x:

$$f(x) = \begin{cases} 0 & x < -1/2 \\ \frac{1}{2} + x & -1/2 \le x \le 1/2 \\ 1 & x > -1/2 \end{cases}$$

4.2 Our PoGD Design

The prover $\mathcal{P}$ updates weights $\boldsymbol{W}_{i-1}$ to $\boldsymbol{W}_i$ using a data batch $\boldsymbol{B}_{i-1}$ (cf. the discussion of gradient-descent in Section 2.3); the PoGD uses sumcheck proof messages to convince a verifier $\mathcal{V}$ that the update is done correctly. For this purpose, $\mathcal{P}$ uses a polynomial commitment scheme to give $\mathcal{V}$ oracle access to the inputs and outputs of the algorithm (i.e., $\widetilde{\boldsymbol{W}}_i$, $\widetilde{\boldsymbol{W}}_{i-1}$, $\widetilde{\boldsymbol{B}}_{i-1}$), and also auxiliary inputs denoted by $\widetilde{\mathbf{AUX}}_i$, including bit-decompositions, quotients, and remainders; note that $\boldsymbol{W}_i$, $\boldsymbol{W}_{i-1}$, and $\mathbf{AUX}_i$ are concatenations of $\boldsymbol{W}_{i,\ell}$, $\boldsymbol{W}_{i-1,\ell}$, and $\mathbf{AUX}_{i,\ell}$, respectively for all layers $\ell = 1$ to L.

PoGD then proceeds in several phases. In the first phase, parties verify the update step of the algorithm, the second phase verifies the backward pass, and the third phase verifies the forward pass. Once sumcheck messages are generated, parties combine random evaluations of inputs and outputs received from different phases together by running additional sumcheck instances. $\mathcal{V}$ verifies the final evaluations using its oracle access. For linear operations, parties run sublinear sumcheck protocols [45, 53], and for non-linear operation they run a generic GKR proof, e.g., Virgo++ [62] in our implementation. Below, we sketch the four phases of our PoGD.

Phase 1: For $\ell = 1$ to L, $\mathcal{V}$ asks for a random evaluation of the trained weights $\widetilde{\boldsymbol{W}}_{i,\ell}$. Having this evaluation, parties run GKR on $\boldsymbol{W}_{i,\ell} = \boldsymbol{W}_{i-1,\ell} - \eta_\ell \cdot \overline{\boldsymbol{G}}_{i,\ell}$, where the run ends with random evaluations of the inputs $\widetilde{\boldsymbol{W}}_{i-1,\ell}$ and $\widetilde{\boldsymbol{G}}_{i,\ell}$. The former proceeds to Phase 4, and the latter proceeds to Phase 1 to bootstrap the next sumchecks.

Phase 2: For $\ell = 1$ to L, having the evaluation of $\widetilde{\boldsymbol{G}}_{i,\ell}$ received from Phase 1 and also the evaluation of $\widetilde{\boldsymbol{R}}_{i,\ell}$ from the previous iteration (If $\ell \neq 1$) parties verify Equation (2). We use the sublinear sumcheck protocols for linear operations, GKR exploiting bit-decomposition of inputs for non-linear operations, and the sumcheck protocol on Equation (3) and Equation (4) for binary operations of scalings and consistency checks. The run ends with evaluations of $\widetilde{\boldsymbol{W}}_{i-1,\ell}$, $\widetilde{\boldsymbol{U}}_{i,\ell-1}$, $\widetilde{\boldsymbol{R}}_{i+1,\ell}$, and $\widetilde{\mathbf{AUX}}_{i,\ell}$, where $\widetilde{\boldsymbol{R}}_{i+1,\ell}$ proceeds to the next iteration, $\widetilde{\boldsymbol{U}}_{i,\ell-1}$ proceeds to Phase 3, and others proceed to Phase 4. For $\ell = L$, parties run GKR on the loss calculation from the actual labels of the data items included in the batch, yielding evaluations of $\widetilde{\boldsymbol{B}}_{i-1}$ and $\widetilde{\boldsymbol{U}}_{i,L}$ and proceeding to Phase 4 and Phase 3, respectively.

Phase 3: For $\ell = L$ to 1, having evaluations of $\widetilde{\boldsymbol{U}}_{i,\ell}$ received from Phase 2 and the previous iteration (if $\ell \neq L$), parties verify the forward pass by running the sublinear sumcheck protocols for the linear operations, GKR for non-linear operations, and the sumcheck protocol for scalings and consistency checks. The run ends with evaluations of inputs of the layer, i.e., $\widetilde{\boldsymbol{W}}_{i-1,\ell}$, $\widetilde{\boldsymbol{U}}_{i,\ell-1}$, and $\widetilde{\mathbf{AUX}}_{i,\ell}$, where $\widetilde{\boldsymbol{U}}_{i,\ell-1}$ proceeds to the next iteration, and $\widetilde{\boldsymbol{W}}_{i-1,\ell}$ and $\widetilde{\mathbf{AUX}}_{i,\ell}$ proceed to Phase 4; note that $\widetilde{\boldsymbol{U}}_{i,0}$ is viewed as an evaluation $\widetilde{\boldsymbol{B}}_{i-1}$.

Phase 4: Parties combine all evaluations of $\widetilde{\boldsymbol{W}}_{i-1}$, $\widetilde{\boldsymbol{B}}_{i-1}$, and $\widetilde{\mathbf{AUX}}_i$ received from the previous phases. In particular, consider a generic ℓ-variate multilinear polynomial f and multiple evaluations of form $\{(x_i, y_i)\}_{i=1}^k$, where $f(x_i) = y_i$. Given challenges $\{\gamma_i\}_{i=1}^k$, we reduce evaluations into a single one by running the sumcheck protocol on:

$$\sum_{i=1}^{k} \gamma_i \cdot y_i = \sum_{z \in \{0,1\}^\ell} f(z) \cdot \sum_{i=1}^{k} \gamma_i \cdot \widetilde{\beta}_z(x_i). \tag{5}$$

In Phase 4 of our PoGD, parties run this sumcheck by instantiating f with $\widetilde{\boldsymbol{W}}_{i-1}$, $\widetilde{\boldsymbol{B}}_{i-1}$, and $\widetilde{\mathbf{AUX}}_i$. At the end of the run, for each of the polynomials, the verifier receives a single evaluation. Along with the evaluation of $\widetilde{\boldsymbol{W}}_i$ from Phase 1, the verifier checks these using the oracle access, i.e., by asking for evaluation opening proofs.

We provide more formal details in Protocol 3 and present the security proof for our construction in Appendix B.

Theorem 1. *The PoGD construction presented in Protocol 3 satisfies completeness, knowledge soundness, and zero knowledge.*

Let $s_{\mathsf{in},\ell}$ be the input size and $s_{\mathsf{out},\ell}$ be the output size of the lth layer, N be the batch size, q be the quantization bit-length. We use Orion [60] as our polynomial commitment scheme due to its linear-time and concretely efficient prover and succinct verification. The prover time is $O(\sum_{i=0}^{L-1} N\, s_{\mathsf{in},\ell}\, s_{\mathsf{out},\ell} + Nqs_{\mathsf{out},\ell})$. The proof size and verifier time are $O(\sum_{i=0}^{L-1} \log^2(N\, s_{\mathsf{in},\ell}\, s_{\mathsf{out},\ell}) + \log^2(Nqs_{\mathsf{out},\ell}))$, which is dominated by evaluation opening proofs. In the next sections, we discuss how to reduce the verification cost using our aggregation scheme to $O(\sum_{i=0}^{L-1} \log(N\, s_{\mathsf{in},\ell}\, s_{\mathsf{out},\ell}) + \log(Nqs_{\mathsf{out},\ell}))$ when we compose PoGDs recursively; see Sections 5 and 6. The protocol can be made non-interactive by the Fiat-Shamir heuristic.

5 AGGREGATABLE COMMITMENT SCHEMES

We present an aggregatable polynomial commitment scheme for *multivariate* polynomials, a key building block for our recursive proof composition framework. In our setting, a prover has generated commitments $\sigma_1, \ldots, \sigma_k$ to polynomials $f_1, \ldots, f_k : \mathbb{F}^\ell \to \mathbb{F}$ each is ℓ-variate with variable-degree at most d. The prover then wants to convince a verifier that $y_i = f_i(x_i)$ for some points $x_1, \ldots, x_k$ more efficiently than giving k independent evaluation opening proofs. We propose an scheme that allow the prover and verifier to map $\{(\sigma_i, x_i, y_i)\}_{i=1}^k$ to a tuple (σ^*, x^*, y^*) such that σ^* is a commitment to a polynomial f^* of size at most as each input polynomial f_i and such that (with high probability) $f^*(x^*) = y^*$ iff $\forall i : f(x_i) = y_i$; thus a proof that $f^*(x^*) = y^*$ can substantiate the original claim to the verifier. The stronger property of knowledge soundness ensures that a prover who can generate valid aggregation messages and an evaluation opening proof for (σ^*, x^*, y^*) must know f^* and the inputs $\{f_i\}_{i=1}^k$. The scheme is zero-knowledge if the process for generating (σ^*, x^*, y^*) and the eventual opening proof for f^* leak no additional information about the $\{f_i\}_{i=1}^k$. We present more formal definitions of these properties in Appendix C.

5.1 Aggregating Evaluations

We begin by reviewing a technique [31] for aggregating multiple evaluations of a single polynomial. We then show how to generalize that to support distinct polynomials, and to add zero-knowledge.

Single polynomial, multiple points. Consider the case where $f_1 = \cdots = f_k = f$ and $\sigma_1 = \cdots = \sigma_k = \sigma$, but $\{x_i\}$ are distinct. The prover and verifier interpolate a degree-$(k-1)$ univariate polynomial $L : \mathbb{F} \to \mathbb{F}^\ell$ with $L(i) = x_i$ for $i \in \{1, \ldots, k\} \subseteq \mathbb{F}$. The prover then sends the verifier a polynomial $g = f \circ L : \mathbb{F} \to \mathbb{F}$ of degree $O(dk\ell)$. The verifier checks that $\forall i : g(i) = y_i$, then, for a random $r \in \mathbb{F}$, the parties set $x^* = L(r)$ and $y^* = g(r)$; in this case, the output polynomial and commitments are $f^* = f$ and $\sigma^* = \sigma$.

We claim that $f(x^*) = y^*$ iff $\forall i : f(x_i) = y_i$ with high probability. Completeness is immediate. Soundness can be argued as follows. Without loss of generality, say $f(x_1) \neq y_1$. If $g = f \circ L$, the verifier rejects since $g(1) \neq y_1$. If $g \neq f \circ L$ then g and $f \circ L$ agree on at most $\deg(g) = O(dk\ell)$ points, and we have $f(x^*) = y^*$ with a negligible probability at most $O(dk\ell/|\mathbb{F}|)$.

Multiple polynomials, multiple points. We now extend the above idea to the general case of (possibly) distinct polynomials $\{f_i\}$ and points $\{x_i\}$. As before, the parties define $L : \mathbb{F} \to \mathbb{F}^\ell$ such that $\forall i : L(i) = x_i$. The prover then sends the k univariate polynomials $g_i = f_i \circ L$, and the verifier checks whether $\forall i : g_i(i) = y_i$. Then, for random challenges $\{\alpha_i\}_{i=1}^k$ and r, the parties set $x^* = L(r)$, $g = \sum_i \alpha_i g_i$, and $y^* = g(r)$. The output polynomial is $f^* = \sum_i \alpha_i f_i$; we defer to Section 5.2 the details of how the parties can generate the commitment σ^*; we refer to that as *commitment aggregation*.

Again, we claim that $f(x^*) = y^*$ iff $\forall i : f(x_i) = y_i$ with high probability. Completeness holds because

$$f^*(x^*) = (f^* \circ L)(r) = \textstyle\sum_i \alpha_i (f_i \circ L)(r) = \sum_i \alpha_i g_i(r) = g(r).$$

For soundness, if there exists some $f_i(x_i) \neq y_i$, the prover must send $g_i \neq f_i \circ L$ to pass verification. But then with high probability over choice of $\{\alpha_i\}, r$, we have $y = g(r) \neq (f^* \circ L)(r) = f^*(x^*)$.

Zero knowledge. Before discussing commitment aggregation, we briefly discuss how to make the above idea zero-knowledge using masking polynomials. For all i, the prover randomly samples an ℓ-variate polynomial h_i and sends a commitment σ'_i to h_i along with an evaluation $v_i = h_i(x_i)$. Given random challenges $\{\beta_i\}$, for all i, the prover then sends $g_i = (f_i + \beta_i h_i) \circ L$, and the verifier checks that $g(i) = y_i + \beta_i v_i$. The rest of the protocol is similar to above; here the output polynomial is $f^* = \sum_i \alpha_i f_i + \alpha_i \beta_i h_i$.

Naively, one could choose each h_i, above, as a random ℓ-variate polynomial of variable-degree at most d. The number of coefficients in each h_i would then be $O(d^\ell)$ and large, making it inefficient in practice to commit to the $\{h_i\}$. We show (cf. Appendix C) it suffices to sample each h_i as $h_i = \sum_{j=1}^{\ell} h_{i,j}$, where each $h_{i,j}$ is a random *univariate* polynomial (in the jth variable) of degree $d\ell$. The size of each h_i is then $O(d\ell^2)$, and the cost of zero knowledge is $O(kd\ell^2)$.

5.2 Aggregating Commitments

What remains is to show how parties can compute σ^*, a commitment to $f^* = \sum_i \alpha_i f_i$ given $\{\sigma_i\}$, commitments to the $\{f_i\}$; the procedure naturally extends to the case of the zero-knowledge scheme, where $f^* = \sum_i \alpha_i f_i + \alpha_i \beta_i h_i$, and parties also hold $\{\sigma'_i\}$. Of course, if the underlying commitment scheme is homomorpic then commitment aggregation is trivial. However, practical homomorphic commitment schemes [18, 37, 56] are known only from group assumptions, and cannot be used efficiently for recursive proof composition. Indeed, in our application we use Orion [60], a commitment scheme based on Merkle trees that is not homomorphic. We describe how it is possible to commit to the linear combination of input polynomials for that scheme. Although the discussion is specific to Orion, the technique can be extended to other hash-based schemes [8, 63].

Orion. We briefly review Orion. In that scheme, the coefficients of a polynomial are mapped to the coefficients of a square matrix $\boldsymbol{M}$, say of size $n \times n$. Then, a linear error-correcting code $E : \mathbb{F}^n \to \mathbb{F}^p$ is applied to $\boldsymbol{M}$ twice. First, we apply the code to each row of $\boldsymbol{M}$, yielding an $n \times p$ matrix $\boldsymbol{C}_1$. Then, we apply the code to each column of $\boldsymbol{C}_1$, yielding a $p \times p$ matrix $\boldsymbol{C}_2$. The prover computes a Merkle root for each column of $\boldsymbol{C}_2$, and then a Merkle root computed over those p roots is returned as a commitment to the polynomial.

The verifier checks the well-formedness of a commitment by sending a random vector $\boldsymbol{r} \in \mathbb{F}^n$ along with $t = \Theta(\lambda)$ random indices in $[p]$. The prover responds with a vector $\boldsymbol{v} = \boldsymbol{r}^T \cdot \boldsymbol{M}$ along with the corresponding t columns of $\boldsymbol{C}_1$ and the associated Merkle proofs for the encodings of those columns. The verifier (1) evaluates the codeword $\boldsymbol{w} = E(\boldsymbol{v})$; (2) for each challenge index i, the returned column $\boldsymbol{C}_1[:, i]$ satisfies $\boldsymbol{w}_i = \boldsymbol{r}^T \cdot \boldsymbol{C}_1[:, i]$; and (3) the Merkle proofs are all correct. Evaluation proofs can be generated similarly, but we omit details. Orion achieves zero knowledge and succinctness by randomized encoding and composing the verification procedure described above with a generic succinct zero-knowledge proof [63].

Aggregation for Orion. Assume the verifier holds commitments $\{\sigma_i\}$ to polynomials $\{f_i\}$ known to the prover, and let $\boldsymbol{M}_i, \boldsymbol{C}_{i,1}, \boldsymbol{C}_{i,2}$ be the corresponding matrices that were generated by the prover when committing to f_i. The prover generates a commitment σ^* to $f^* = \sum \alpha_i f_i$ and sends it to the verifier; let $\boldsymbol{M}^*, \boldsymbol{C}_1^*, \boldsymbol{C}_2^*$ be the corresponding matrices generated during this process. For an honest prover $\boldsymbol{M}^* = \sum_i \alpha_i \boldsymbol{M}_i$, $\boldsymbol{C}_1^* = \sum_i \alpha_i \boldsymbol{C}_{i,1}$, and $\boldsymbol{C}_2^* = \sum_i \alpha_i \boldsymbol{C}_{i,2}$.

On the other hand, if $\boldsymbol{M}^* \neq \sum_i \alpha_i \boldsymbol{M}_i$ then, letting δ be the minimum distance of E, the matrices $\boldsymbol{C}_2^*$ and $\sum_i \alpha_i \boldsymbol{C}_{i,2}^*$ must differ in at least δ^2 entries.[4] The key observation is that, in addition to checking the well-formedness of σ^*, the verifier can check that the correct linear relationship holds between random entries of $\boldsymbol{C}_2^*$ and the corresponding entries of $\{\boldsymbol{C}_{i,2}\}$; if E has constant relative distance, it suffices to check that the linear combination relationship holds for $\Theta(\lambda)$ randomly opened entries of each $\boldsymbol{C}_2^*$ and $\{\boldsymbol{C}_{i,2}\}$.

We provide more formal details in Protocol 4 and prove the security of our aggregation scheme in Appendix C.

Theorem 2. *The aggregation scheme presented in Protocol 4 satisfies completeness, knowledge soundness, and zero knowledge.*

The prover cost of our scheme is dominated by generating the commitment σ^*, which can be done in $O(d^\ell)$, and evaluating the $\{g_i\}$, which can be done in $O(k \cdot d^\ell)$ time. The proof size is dominated by the Merkle proofs, which have size $O(kd\ell)$; this is also the verifier complexity. Moreover, the scheme can be made non-interactive via the Fiat-Shamir transform.

6 RECURSIVE SUMCHECK PROOFS

We present a generic framework for building IVC from GKR-style sumcheck-based proofs. In particular, our framework relies on the recursive proof composition technique when the baseline proof system $(\mathcal{P}_b, \mathcal{V}_b)$ is instantiated by sumcheck-based non-interactive succinct proofs. When the baseline proof is recursively composed, the prover must generate a proof for the executions of the verifier algorithm $\mathcal{V}_b$. We discuss different components of this algorithm and our techniques to improve the efficiency of each components.

Aggregation instead of opening. $\mathcal{V}_b$ verifies some sumcheck messages for the previous iteration of the augmented function $\mathcal{F}_A$. For this purpose, $\mathcal{V}_b$ receives several random evaluations of the inputs and outputs of $\mathcal{F}_A$; more precisely, evaluations of the multilinear extensions of the inputs and outputs. The inputs are committed by a succinct polynomial commitment scheme; such commitments prevent a blow-up in the input size across iterations. The prover then must provide opening proofs for the evaluations required by $\mathcal{V}_b$ to verify sumcheck messages. Outputs of $\mathcal{F}_A$ might be given to $\mathcal{V}_b$ in plain or to be committed. Here, for simplicity, we assume they are given in plain, and they can be evaluated directly.

To reduce the complexity of $\mathcal{V}_b$, we replace opening proofs for committed polynomials with aggregation messages. In particular, for each ith iteration, $\mathcal{V}_b$ receives two commitment/evaluation instances as follows. First, $\mathcal{V}_b$ receives $\mathrm{agg}_{i-1} = (\sigma^*_{i-1}, x^*_{i-1}, y^*_{i-1})$, which is the aggregate instance returned the previous iteration; this includes a commitment σ^*_{i-1} and evaluation point (x^*_{i-1}, y^*_{i-1}), and it is the aggregation of all commitments and their evaluations up to the iteration $i-2$. Second, $\mathcal{V}_b$ receives $(\sigma_{i-1}, x_{i-1}, y_{i-1})$, which is the instance for the inputs of the previous execution of $\mathcal{F}_A$. Here, again σ_{i-1} is a commitment to the inputs of $\mathcal{F}_A$, and (x_{i-1}, y_{i-1}) is a evaluation point. These two instances are aggregated using our polynomial commitment aggregation scheme (cf. Section 5) and the aggregate output $\mathrm{agg}_i = (\sigma^*_i, x^*_i, y^*_i)$ is passed to the next iteration. In this way, we defer the opening to the final iteration.

[4] This assumes $\boldsymbol{C}_2^*, \{\boldsymbol{C}_{i,2}\}$ were computed correctly, but in fact, it suffices to check that σ^* is well-formed to ensure that each σ_i were well-formed.

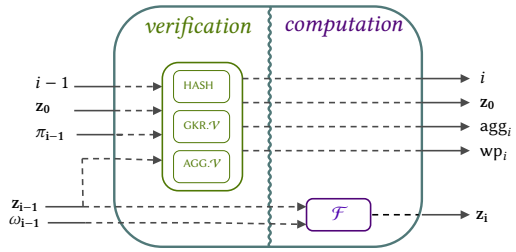

Figure 2: High-level illustration of the augmented function $\mathcal{F}_A$. $\mathcal{F}$ denotes the iteration function. HASH, GKR.$\mathcal{V}$, **and** AGG.$\mathcal{V}$ **denote hash evaluation, sumcheck verifier, and aggregation verifier components of $\mathcal{V}_b$, respectively.** agg_i **denotes the ith aggregate commitment/evaluation instance and** wp_i **to denote the ith aggregate predicate evaluation.**

Handling wiring predicates. In addition to the input and output evaluations, $\mathcal{V}_b$ also requires several evaluations of the wiring predicates of $\mathcal{F}_A$ at random points to validate sumcheck messages. Hard-coding all these predicates in $\mathcal{V}_b$ blows up its circuit size and, hence, the recursion cost. Instead, we delegate these evaluations to the prover. We then apply an evaluation reduction sumcheck, i.e., the sumcheck protocol on Equation (5) to combine all wiring predicate evaluations into a single one, which is then passed to the next iteration. The final verifier receives the aggregation of wiring predicate evaluation for all iterations and validates its correctness.

Sumcheck-friendly hashing. $\mathcal{V}_b$ verifies challenges for sumcheck proofs and aggregation messages as generated by the Fiat-Shamir transform. Furthermore, $\mathcal{V}_b$ need to evaluate multiple hashes to verify Merkle proofs included in aggregation messages. To reduce the circuit size of this component, we instantiate the hash function with MiMC-p/p, which is a sponge hash function, and each round function is $F_i(x) = (x+k_i)^3 \mod p$ for some round key k_i. Suppose that we have n number of hash evaluations and let d to denote the number of rounds. We can prove each hash round by running a single sumcheck protocol. Let $\boldsymbol{v}_0$ be the vector of hash outputs, $\boldsymbol{v}_d$ be the vector of inputs, and $\boldsymbol{v}_i := F_i(\boldsymbol{v}_{i-1})$. Given an initial random challenge r_0, the prover applies the sumcheck protocol on:

$$\widetilde{v}_0(r_0) = \sum_{z \in \{0,1\}^{\log n}} \widetilde{\beta}_z(r_0) \cdot (\widetilde{v}_1(z) + k_1)^3, \tag{6}$$

The run ends with an evaluation $\widetilde{v}_1(r_1)$, which proceeds the round. The prover continues this process to generate proof for all rounds.

Putting everything together. In the ith iteration, $\mathcal{F}_A$ takes as input the iteration counter i, initial inputs z_0, last output z_{i-1}, an auxiliary input ω_{i-1}, and proof π_{i-1}. The proof includes sumcheck proofs, aggregation messages, aggregate commitment/evaluation instance agg_{i-1}, aggregate wiring predicate evaluation wp_{i-1}, and a commitment and evaluation $(\sigma_{i-1}, x_{i-1}, y_{i-1})$ from iteration $i-1$. $\mathcal{F}_A$ invokes $\mathcal{V}_b(i-1, z_0, z_{i-1}, \pi_{i-1})$, which evaluates required hashes, verifies proof messages, aggregates agg_{i-1} with $(\sigma_{i-1}, x_{i-1}, y_{i-1})$, aggregates wiring predicate evaluations received at the end of the sumcheck verification with wp_{i-1}, and outputs the updated aggregates wp_i and agg_i. Once $\mathcal{V}_b$ is completed successfully, $\mathcal{F}_A$ then evaluates $z_i \leftarrow \mathcal{F}(z_{i-1}, \omega_{i-1})$ and returns $(i, z_0, \mathrm{agg}_i, \mathrm{wp}_i, z_i)$.

We illustrate the augmented function in Figure 2. When execution of $\mathcal{F}_A$ is completed, the IVC prover applies hash sumchecks, i.e., sumchecks on Equation (6), to MiMC-p/p hashes and the baseline prover $\mathcal{P}_b$ for other components. Furthermore, the prover needs to generate aggregation messages for the next iteration by running the prover of our aggregation scheme. In the final iteration, the prover additionally requires to generate an evaluation opening proof for the final aggregate commitment/evaluation instance.

In Protocol 5 of Appendix D, we provide more formal details of our recursive sumcheck framework and prove its security.

Theorem 3. *Protocol 4 presents an IVC construction that satisfies completeness, knowledge soundness, and zero knowledge.*

The complexity of the scheme depends on the baseline proof system and the polynomial commitment instantiation. For instance, consider an instantiation, where the GKR-style baseline proof is implemented by Virgo++ [62] using Orion [60] as the commitment; the prover then is $O(|\mathcal{F}| + |z| + |\omega|)$, and proof size and verifier are $O(d \cdot \log|\mathcal{F}| + \log^2(|z| + |\omega|))$, where d denotes the depth of $\mathcal{F}$.

7 ZERO-KNOWLEDGE PROOF OF TRAINING

We first define the notion of zkPoT as a cryptographic primitive and then propose Kaizen a zkPoT for DNNs.

7.1 Definition

The prior work [29] proposed a monolithic definition of zkPoT, i.e., the prover executes a universal circuit, representing the entire training computation being verified, and generates a proof of the correct execution. Unfortunately, this approach is unsuited for iterative training algorithms as it requires fixing the number of training iterations in advance; the prover should complete the all iterations to generate a verifiable proof. For applications where a model owner incrementally updates their model, it is desirable to have a succinct zkPoT at any stage of the training process. To achieve this goal, we present an incremental definition of zkPoT, which generates a succinct zkPoT at the end of each iteration, attesting to the integrity of the entire training computation.

A consideration of such incremental definitions is ensuring data consistency; each iteration is only given a batch of data items claimed to be randomly sampled from the dataset. To resolve this, assume that there exists some fixed permutation of the dataset for each epoch and the dataset is committed to by a *position-binding* commitment, e.g., its Merkle root. At each iteration, the prover uses the batch consistent with the permutation and provides opening proofs to each data item in the batch. Before running the training iterations, Merkle openings are first validated. More precisely, let C be the iteration circuit in which the permutations are hard-coded. Let $\boldsymbol{W}_0$ be some initial model and $\mathcal{D}$ be the dataset committed to by ρ. In each ith iteration, C takes as input the iteration counter i, dataset commitment ρ, the last model weights $\boldsymbol{W}_{i-1}$, a batch of data samples $\boldsymbol{B}_{i-1} \subseteq \mathcal{D}$, and the batch opening proof $\boldsymbol{p}_{B_{i-1}}$. C first validates the opening proof and the consistency of data items in the batch with the hard-coded permutations. If verification passes, it applies the training algorithm and returns weights $\boldsymbol{W}_i$. In practice, we can compactly hard-code permutations by fixing a key for an efficient pseudorandom permutation for each epoch, which enables C to evaluate the permuted positions required at each iteration.

A zkPoT enables the prover to first commit to the dataset and initial model weights, then sample a batch for each iteration and execute C. Once an iteration is completed, the prover can generate a commitment to the updated weights and a succinct proof that all iterations are correctly executed. More formally, a zkPoT consists:

- KeyGen: on input the security parameter, the setup algorithm generates the public parameters pp.
- DataCom: on input the training dataset $\mathcal{D}$, randomness $r_{\rho_{\mathcal{D}}}$, and public parameters pp, returns a commitment $\rho_{\mathcal{D}}$.
- WeiCom: on input some weights $\boldsymbol{W}_i$, a randomness $r_{\sigma_{\boldsymbol{W}_i}}$, and public parameters pp, returns a commitment $\sigma_{\boldsymbol{W}_i}$.
- BatchOpen: on input the dataset $\mathcal{D}$, randomness $r_{\rho_{\mathcal{D}}}$, the iteration counter i, and public parameters pp, returns the batch $\boldsymbol{B}_{i-1}$ for the ith training iteration with an opening proof $\boldsymbol{p}_{B_{i-1}}$.
- Prove: on input an iteration counter i, dataset commitment $\rho_{\mathcal{D}}$, initial weights commitment σ_{W_0}, current weights $\boldsymbol{W}_{i-1}$ with a commitment $\sigma_{W_{i-1}}$, a batch $\boldsymbol{B}_{i-1}$ with opening $\boldsymbol{p}_{\boldsymbol{B}_{i-1}}$, a proof π_{i-1}, and public parameters pp, returns weights $\boldsymbol{W}_i$ and a proof π_i.
- Verify: on input an iteration counter i, the dataset commitment $\rho_{\mathcal{D}}$, the initial weights commitment σ_{W_0}, the trained weights commitment σ_{W_i}, and a proof π_i, returns accept or reject.

A proof-of-training satisfies soundness if the probability of passing verification for an adversary prover, returning inconsistent iteration counter and commitments is negligible. Knowledge soundness ensures that a prover accepted by the verifier must know satisfying weights and the dataset underlying the returned commitments. Zero knowledge guarantees that no information about the models and datasets is revealed; see Appendix E for formal definitions. For practicality, we require the proof size and the verifier runtime to be $O(|C|)$ and independent of the number of iterations and the prover cost to be $O(|C| \log |C|)$ per iteration.

7.2 Implementing a zkPoT for DNNs

Kaizen commits to the dataset by its Merkle root, and the opening of a particular data batch includes Merkle proofs for each of the data items in the batch. Model weights are committed by generating a polynomial commitment to their multilinear extension.

Let PRP be a pseudorandom permutation, e.g., a cubic function, with an initial permutation key s_0. The key can be updated for each eth epoch to $s_e = \mathcal{H}(e + s_0)$, given a cryptographic hash function $\mathcal{H}$; this ensures that each epoch trains the model on a randomly sampled partition. Let N be the batch size. For each ith iteration of eth epoch, the training iteration circuit C is executed as follows:

(1) $s_e \leftarrow \mathcal{H}(e + s_0)$, $\forall k \in [N] : v_k \leftarrow \mathsf{PRP}(s_e, (i-1)n + k)$.
(2) Validate $\forall k \in [N] : \mathsf{MT.Verify}(v_k, \rho_{\mathcal{D}}, \boldsymbol{B}_{i-1}[k], \boldsymbol{p}_{B_i}[k]) = 1$.
(3) If the verification passes, return $\boldsymbol{W}_i = \mathsf{GD}(\boldsymbol{W}_{i-1}, \boldsymbol{B}_{i-1})$, where GD denotes the the gradient descent algorithm (cf. Algorithm 1).

Consider an instantiation of our recursive sumcheck framework (i.e., Protocol 5), where the function $\mathcal{F}$ is instantiated by C defined above. The baseline proof system is instantiated by the construction that verifies the execution of C by generating sumcheck messages for step 1 by a generic GKR-style proof [62], step 2 by running sumcheck for MiMC-p/p hash (cf. Section 6), and step 3 via running our PoGD scheme as presented in Protocol 3.

A technical consideration is that in protocol 5, $\mathcal{V}_b$ receives the initial inputs and the last iteration output in plain. In Kaizen, these include model weights and can be significantly large. To prevent a blow-up in the size of $\mathcal{V}_b$, Kaizen instead passes commitments to weights. The prover provides the evaluations of weights required for verifying sumcheck proofs, and these commitments and their evaluations are aggregated across the training iterations.

We present more formal details of the Kaizen construction in Protocol 6 and a proof of its security in Appendix E.

Theorem 4. Kaizen *is a zero-knowledge proof of training and satisfies completeness, knowledge soundness, and zero knowledge.*

Followed by the discussion in Section 4.2, let $s_{\mathsf{in},l}$ be the input and $s_{\mathsf{out},l}$ the output sizes of the lth layer, N the batch size, and q the bit-length of quantization. The Kaizen prover cost per iteration is $O(\sum_{i=0}^{L-1} N s_{\mathsf{in},l} s_{\mathsf{out},l} + N q s_{\mathsf{out},l})$ both for generating PoGD messages and the recursion overhead. The final proof size and verifier cost are $O(\sum_{i=0}^{L-1} \log^2(N\, s_{\mathsf{in},\ell}\, s_{\mathsf{out},\ell}) + \log^2(N q s_{\mathsf{out},\ell}))$, independent of the number of iterations and the size of the dataset.

8 IMPLEMENTATION AND EVALUATION

We discuss the implementation of Kaizen and its evaluation results. The sumcheck and GKR protocols are implemented based on several open-source libraries [45, 62]. We run all experiments on a Linux VM with 8 physical CPUs, 2.80 GHz Intel(R) Xeon(R) Platinum 8370, and 256GB of RAM. Our current implementation is not parallelized, and we report the average running time of 10 executions for each of the benchmarks. Our code is publicly available [2].

Protocol instantiation. Our construction works on any finite field. We choose the extension field $\mathbb{F}_{p^2}$ for the prime $p = 2^{61} - 1$. We use MiMC-p^2/p^2 [4] with 80 rounds and SHA-256 as our hash. The pseudorandom permutation PRP for shuffling the dataset is replaced by a cubic function over a field of the size as the dataset. We set $q = 64$ and $F = 32$ for the quantization scheme to ensure no overflow. The generic GKR-style proof used in our PoGD and Kaizen protocols is instantiated by the Virgo++ [62]. Our parameters are chosen to achieve a $\lambda = 100$-bit security level.

Moreover, we instantiated the polynomial commitment scheme in our construction with Orion [60]. A difficulty with the Orion protocol is that its linear error-correcting code, i.e., Spielman [52], has a small relative distance, and this yields a significant number of Merkle tree openings for the aggregation scheme. In particular, given a relative distance $\gamma = 0.15$, as we use in our implementation, λ-bit security level requires 3045 openings. To reduce this number, instead of parsing the coefficients of polynomials as square matrices, we can parse them as matrices with a small constant number of columns and linear-sized rows. Although this yields a quasi-linear evaluation opening time, the commitment and aggregation times remain linear. As in Kaizen, the evaluation opening is invoked only once at the final iteration, hence, the effect of this modification to the prover per iteration cost is negligible; here, we set the column-size to one to minimize the aggregation cost, yielding 426 openings.

Furthermore, we replace MiMC-p^2/p^2 with SHA-256 for up to 4 bottom layers of Merkle trees used in the Orion implementation to reduce the commitment-generation time for large polynomials.

Gradient descent API. We implemented an interference for the DNN gradient descent computation. A user can select specifications such as the model architectures, the batch size, and learning rate. The interference outputs sumcheck instances and GKR circuits, required for the mini-batch gradient-descent computation. The user can add any number of convolutional or dense layers to the architecture and specify input, output, and weight sizes. Kaizen can support a wide range of activations such as ReLU, tanh, Sigmoid, and Softmax. For pooling methods, we can support both average and max pooling with arbitrary window sizes and strides.

DNN Models and training datasets. We measure the performance of our constructions on three well-known model architectures, LeNet [42], AlexNet [41], and VGG-11 [51]. For AlexNet and VGG-11, we truncated the number of parameters by reducing the number of channels and the size of the output size of the layers. LeNet has 3 convolutional layers, 2 average poolings, 2 dense layers, and a total number of 61706 trainable parameters. AlexNet has 5 convolutional layers, 3 average poolings, 2 dense layers, and a total number of 4.2 million parameters. Finally, the VGG-11 has 8 convolutional layers, 5 average pooling, 3 dense layers, and a total number of 10.1 million parameters. We use Softmax as the activation for the output layers and ReLU for others. We use the MNIST dataset [22], including $60,000$ gray-scale 32×32 images for LeNet, and CIFAR-10 [40], including $60,000$ RGB 64×64 images for AlexNet and VGG-11.

8.1 Performance of Our Constructions

We first measure the time the prover requires to generate sumcheck proofs of our PoGD at each iteration. Then, we report the overall prover and verifier overheads and the evaluation results of Kaizen.

Prover cost of PoGD sumchecks. We report the prover time for sumchecks used in PoGD with a breakdown into the four phases of update step, backward pass, forward pass, and evaluation reduction. The batch size is ranged from $N = 4$ to 16. The results are reported and summarized in Table 1; note that the prover cost of polynomial commitments are excluded, and we discuss them in the next reports. As shown in the table, it takes only 12.68 seconds to generate sumcheck messages for the gradient-descent of LeNet with $N = 16$. For AlexNet and VGG-11 it is 35.88 and 157.6 seconds, respectively.

Kaizen Performance. Before starting the training procedure, the prover generates the Merkle root of the dataset. This takes only 18 seconds for MNIST and 218 seconds for CIFAR-10, where the Merkle tree is built using MiMC hash. Then, the prover executes the training iterations and generates proofs and commitments once an iteration is completed. We report the prover time per iteration with a breakdown into generating polynomial commitments, aggregation overhead[5], generating sumcheck messages, and the overhead of proving the verifier function $\mathcal{V}_b$. Moreover, we report the proof size and verifier time with breakdown into sumcheck proofs and polynomial commitment evaluation openings proofs generated in the final iteration. We range $N = 4$ to 16, and summarize in Table 2. As shown in the table, Kaizen offers a prover runtime of 380 seconds per gradient-descent iteration for LeNet, 687 seconds for AlexNet, and 1357 seconds for VGG-11 given a batch size $N = 16$.

[5]The cost of generating commitments to aggregate polynomials returned by the aggregation scheme is considered part of the aggregation cost.

The verification cost is independent of the number of iterations and, hence, the size of the dataset. For batch size $N = 16$, The proof size is 900KB for LeNet, 1092KB for AlexNet, and 1360KB for VGG-11. The verifier time is only 66ms for LeNet, 81.84ms for AlexNet, and 103ms for VGG-11. This extremely fast verifier runtime is achieved by applying GKR-style proofs to the iteration circuit C and using Orion as our polynomial commitment scheme. As we observe, 67%-78% of the prover overhead is expended by the commitment and aggregation overheads. For some applications, it might be desirable to reduce the prover time at the cost of more verification time using different polynomial commitment schemes. For instance, one can instantiate Kaizen with Bulletproofs [18] as the commitment scheme, which can achieve 2× faster prover at the cost of having verifier time in the order of hundreds of seconds.

8.2 Comparison to Generic IVCs

We compare the performance of Kaizen with prior art generic IVC constructions, including Fractal [21], Halo [17], and Nova [39]. The generic constructions require the computation to be modeled as an arithmetic circuit. In particular, the schemes we discuss here rely on the *rank-1 constraint system* (R1CS) circuit representation. We carefully extracted the size of R1CS for the iteration circuit C to measure the performance of prior IVCs. For each operation of the gradient-descent algorithm, we consider the same computation procedure as our implementation, e.g., we use the standard matrix multiplication method, convolutions are turned into a few FFTs and evaluated by the Cooley–Tukey FFT algorithm, and non-linear operations are handled with bit-decompositions as auxiliary inputs. In addition to the circuit size, we also consider the output size of C, which includes model weights. Generic IVCs require the iteration outputs to be hashed by the verifier circuit multiple times. However, thanks to the use of GKR-style proofs, our construction can give the verifier only a commitment to the multilinear extension of outputs. We consider the hash function as Poseidon [33] for Nova and Fractal and Rescue [5] for Halo, as discussed in their protocol.

Given $N = 16$, the size of C for LetNet is about 250 million, for AlexNet is 618 billion, and for VGG-11 is 4.63 billion of gates; the weight hash constraint sizes are excluded. For this batch size, comparisonsare reported in Table 3. As shown, for VGG-11, our prover is 43× faster than state-of-art generic IVC, which is Nova. Our verifier time is in the order of tens of milliseconds, comparable with Fractal and significantly faster than Halo and Nova. Our proof size is relatively large, however, for practical applications it might be considered negligible. Unfortunately, we are unable to run any of the mentioned IVC schemes even for LeNet with $N = 1$, as their memory usage exceeds our limitation, i.e., 256GB. In particular, we could only run Nova on circuit sizes up to 100 million gates, whereas LetNet for $N = 1$ has 123 million gates, including 16 million gates for C and 107 million gates for weight hash. We extrapolated their performance based on the asymptotic complexities and their reported per-gate performance for small circuit sizes. Due to the substantial constants, reports are expected to be underestimated. Moreover, we measured the memory usage of the prior art (i.e., Nova) on circuit sizes up to 100 million gates. We observed that the memory usage of Nova grows linearly in the circuit size. Based on this, we expect to achieve 224× less memory overhead for VGG-11.

	LeNet			AlexNet			VGG-11		
	$N = 4$	$N = 8$	$N = 16$	$N = 4$	$N = 8$	$N = 16$	$N = 4$	$N = 8$	$N = 16$
Phase 1: Update	0.045	0.082	0.155	3.104	5.587	10.55	7.464	13.43	25.38
Phase 2: Backward	2.232	3.336	5.589	2.288	4.109	8.11	18.28	25.35	42.72
Phase 3: Forward	1.300	2.150	3.816	5.371	8.210	14.03	17.39	30.53	57.78
Phase 4: Evaluation	0.795	1.573	3.127	1.702	2.163	3.084	10.07	17.17	31.35
Total Prover (s)	**4.372**	**7.141**	**12.68**	**16.23**	**20.07**	**35.88**	**53.204**	**86.48**	**157.6**

Table 1: Prover runtime as a function of the batch size N when generating sumcheck messages in our PoGD.

	LeNet			AlexNet			VGG-11		
	$N = 4$	$N = 8$	$N = 16$	$N = 4$	$N = 8$	$N = 16$	$N = 4$	$N = 8$	$N = 16$
Commitments	73.37	97.87	146.7	84.5	140.7	253.1	141.4	254.4	480.3
Proof of Agg.	76.69	100.2	152.2	97.50	168.1	305.4	168.2	307.2	585.3
Proof of $\mathcal{V}_b$	68.47	68.75	68.75	89.05	90.01	90.22	90.60	90.87	131.7
Proof of C	4.543	7.480	13.35	16.91	21.42	38.58	53.88	87.83	160.32
Total Prover (s)	**223.0**	**274.9**	**380.2**	**287.9**	**420.23**	**687.3**	**454.0**	**740.3**	**1357**
Sumcheck Proofs	457.7	472.7	487.7	619.1	637.1	656.0	837.6	861.7	886.8
Comm. Openings	366.8	434.2	412.2	388.9	395.9	436.3	424.4	448.3	473.3
Total Proof (KB)	**824.5**	**861.9**	**899.9**	**1008**	**1033**	**1092**	**1262**	**1310**	**1360**
Sumcheck Verifier	40.58	42.02	43.12	54.78	55.58	58.53	74.33	76.75	78.58
Comm. Verifier	19.92	20.81	22.85	21.00	22.14	23.31	22.13	23.31	24.84
Total Verifier (ms)	**60.50**	**63.01**	**65.97**	**75.78**	**77.72**	**81.84**	**96.46**	**100.06**	**103.12**

Table 2: The prover runtime per iteration, proof size, and verifier runtime of Kaizen with batch size N.

	Prover (s)			**Proof Size (KB)**			**Verifier (s)**		
	LeNet	AlexNet	VGG-11	LeNet	AlexNet	VGG-11	LeNet	AlexNet	VGG-11
Fractal [21]	326,568	5,225,712	20,902,976	234	267	298	0.020	0.025	0.030
Halo [17]	50,850	813,595	3,254,382	4.86	5.09	5.45	3,970	63,528	254,114
Nova [39]	958	20,940	59,160	9.80	10.1	10.3	546	8,752	35,760
Kaizen	**380**	**687**	**1,357**	**900**	**1092**	**1360**	**0.066**	**0.081**	**0.103**

Table 3: The prover runtime per iteration, proof size, and verifier runtime for Kaizen and generic IVCs given batch size $N = 16$.

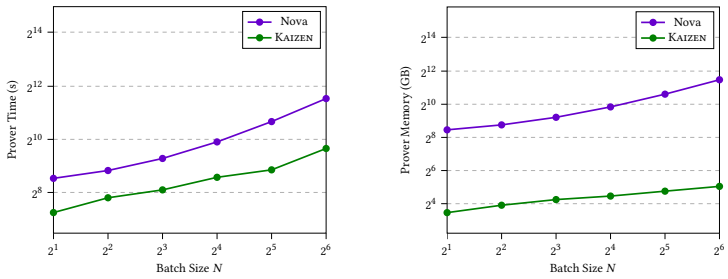

Figure 3: The prover time (per iteration) and memory overhead of Kaizen and Nova for LeNet.

More on Kaizen vs. Nova. We give a more detailed comparison between Kaizen and Nova in terms of the prover runtime and memory usage. Nova offers better prover overhead compared to other generic IVCs. Specifically, we provide a comparison for the case of LeNet with batch sizes ranging from $N = 2$ to 64. Results are reported in Figure 3. As shown, our prover is 2.4-3.7× faster, and we require 31.9-86.42× less memory overhead. As the batch size grows, the advantage of Kaizen over Nova becomes more evident. We note that a larger batch size can also enable the prover to apply further optimizations by parallelizing both the training and proof generation computations in practice; e.g., prior work [58] showed that GKR-style proofs are amenable to high degrees of parallelism.

ACKNOWLEDGMENTS

The work of Kasra Abbaszadeh and Jonathan Katz was supported by DARPA under Contract No. HR00112020025. The work Dimitrios Papadopoulos was supported Hong Kong RGC under grant 16200721. The views, opinions, and/or findings expressed are those of the authors and should not be interpreted as reflecting the position or policy of the Department of Defense or the U.S. Government, and no official endorsement should be inferred.

REFERENCES

[1] 2023. Increasing transparency in AI security. https://security.googleblog.com/2023/10/increasing-transparency-in-ai-security.html.

[2] 2023. Kaizen. https://github.com/zkPoTs/kaizen.
[3] 2023. Modulus Labs. https://www.modulus.xyz.
[4] Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. 2016. MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity. In *Advances in Cryptology—Asiacrypt 2016 (LNCS)*. Springer, 191–219.
[5] Abdelrahaman Aly, Tomer Ashur, Eli Ben-Sasson, Siemen Dhooghe, and Alan Szepieniec. 2020. Design of Symmetric-Key Primitives for Advanced Cryptographic Protocols. *IACR Trans. Symmetric Cryptol.* 2020 (2020), 1–45.
[6] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. 2017. Ligero: Lightweight Sublinear Arguments Without a Trusted Setup. In *ACM Conf. Computer and Communications Security 2017*. ACM, 2087–2104.
[7] Alexandre Belling, Azam Soleimanian, and Olivier Bégassat. 2023. Recursion over Public-Coin Interactive Proof Systems; Faster Hash Verification. In *ACM Conf. Computer and Communications Security*. ACM.
[8] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. 2018. Fast Reed-Solomon Interactive Oracle Proofs of Proximity. In *Intl. Colloquium on Automata, Languages, and Programming (ICALP) (LIPIcs, Vol. 107)*. Schloss Dagstuhl—Leibniz-Zentrum für Informatik, 14:1–14:17.
[9] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. 2016. Interactive Oracle Proofs. In *Theory of Cryptography (TCC)*. Springer, 31–60.
[10] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014. Scalable Zero Knowledge via Cycles of Elliptic Curves. In *Advances in Cryptology—Crypto 2014*. Springer, 276–294.
[11] Josh Benaloh and Michael de Mare. 1994. One-Way Accumulators: A Decentralized Alternative to Digital Signatures. In *Advances in Cryptology—Eurocrypt '93*, Tor Helleseth (Ed.). Springer, 274–285.
[12] Rishabh Bhadauria, Zhiyong Fang, Carmit Hazay, Muthuramakrishnan Venkitasubramaniam, Tiancheng Xie, and Yupeng Zhang. 2020. Ligero++: A New Optimized Sublinear IOP. In *ACM Conf. Computer and Communications Security*. ACM, 2025–2038. https://doi.org/10.1145/3372297.3417893
[13] Nir Bitansky, Ran Canetti, Alessandro Chiesa, Shafi Goldwasser, Huijia Lin, Aviad Rubinstein, and Eran Tromer. 2017. The Hunting of the SNARK. *J. Cryptol.* 30, 4 (oct 2017), 989–1066. https://doi.org/10.1007/s00145-016-9241-9
[14] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. 2013. Recursive Composition and Bootstrapping for SNARKs and Proof-Carrying Data. In *Symposium on Theory of Computing (STOC)*. ACM, 111–120.
[15] Dan Boneh, Justin Drake, Ben Fisch, and Ariel Gabizon. 2021. Halo Infinite: Proof-Carrying Data from Additive Polynomial Commitments. In *Advances in Cryptology—Crypto 2021, Part I*. Springer, 649–680.
[16] Jonathan Bootle, Alessandro Chiesa, and Katerina Sotiraki. 2021. Sumcheck Arguments and Their Applications. In *Advances in Cryptology—Crypto 2021, Part I*. Springer, 742–773. https://doi.org/10.1007/978-3-030-84242-0_26
[17] Sean Bowe, Jack Grigg, and Daira Hopwood. 2019. Recursive Proof Composition without a Trusted Setup. Cryptology ePrint Archive, Paper 2019/1021.
[18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. 2018. Bulletproofs: Short Proofs for Confidential Transactions and More. In *IEEE Symposium on Security and Privacy*. IEEE, 315–334.
[19] Gian Carlo Cardarilli, Marco Re, and Luca Di Nunzio. 2021. A pseudo-softmax function for hardware-based high speed image classification. *Sci. Reports* (2021).
[20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. 2020. Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS. In *Advances in Cryptology—Eurocrypt 2020*. Springer, 738–768.
[21] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. 2020. Fractal: Post-quantum and Transparent Recursive Proofs from Holography. In *Advances in Cryptology—Eurocrypt 2020*. Springer, 769–793.
[22] Li Deng. 2012. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine* 29, 6 (2012), 141–142.
[23] Thorsten Eisenhofer, Doreen Riepel, Varun Chandrasekaran, Esha Ghosh, Olga Ohrimenko, and Nicolas Papernot. 2023. Verifiable and Provably Secure Machine Unlearning. arXiv:2210.09126
[24] Alessandro Epasto, Mohammad Mahdian, Vahab Mirrokni, and Manolis Zampetakis. 2020. Optimal Approximation-Smoothness Tradeoffs for Soft-Max Functions. In *Neural Information Proc. Systems (NeurIPS)*. Curran Associates Inc.
[25] Congyu Fang, Hengrui Jia, Anvith Thudi, Mohammad Yaghini, Christopher A. Choquette-Choo, Natalie Dullerud, Varun Chandrasekaran, and Nicolas Papernot. 2022. On the Fundamental Limits of Formally (Dis)Proving Robustness in Proof-of-Learning. https://doi.org/10.48550/ARXIV.2208.03567
[26] Congyu Fang, Hengrui Jia, Anvith Thudi, Mohammad Yaghini, Christopher A. Choquette-Choo, Natalie Dullerud, Varun Chandrasekaran, and Nicolas Papernot. 2023. Proof-of-Learning is Currently More Broken Than You Think. In *European Symposium on Security and Privacy*. IEEE, 797–816.
[27] Boyuan Feng, Lianke Qin, Zhenfei Zhang, Yufei Ding, and Shumo Chu. 2021. ZEN: An Optimizing Compiler for Verifiable, Zero-Knowledge Neural Network Inferences. Cryptology ePrint Archive, Paper 2021/087.
[28] Amos Fiat and Adi Shamir. 1987. How To Prove Yourself: Practical Solutions to Identification and Signature Problems. In *Advances in Cryptology—Crypto' 86*, Andrew M. Odlyzko (Ed.). Springer, 186–194.
[29] Sanjam Garg, Aarushi Goel, Somesh Jha, Saeed Mahloujifar, Mohammad Mahmoody, Guru-Vamsi Policharla, and Mingyuan Wang. 2023. Experimenting with Zero-Knowledge Proofs of Training. http://ia.cr/2023/1345
[30] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. 2016. ZKBoo: Faster Zero-Knowledge for Boolean Circuits. In *Proceedings of the 25th USENIX Conference on Security Symposium* (Austin, TX, USA) *(SEC'16)*. USENIX Association, USA, 1069–1083.
[31] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. 2008. Delegating Computation: Interactive Proofs for Muggles. In *ACM Symposium on Theory of Computing*. ACM, 113–122.
[32] Alexander Golovnev, Jonathan Lee, Srinath Setty, Justin Thaler, and Riad S. Wahby. 2021. Brakedown: Linear-Time and Post-Quantum SNARKs for R1CS. Cryptology ePrint Archive, Paper 2021/1043.
[33] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. 2021. Poseidon: A New Hash Function for Zero-Knowledge Proof Systems. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 519–535.
[34] Jens Groth. 2016. On the Size of Pairing-Based Non-interactive Arguments. In *Advances in Cryptology—Eurocrypt 2016*. Springer, 305–326.
[35] Melissa Heikkilä. 2022. This Artist is Dominating AI-Generated Art. And He's not Happy About It. MIT Technology Review.
[36] Hengrui Jia, Mohammad Yaghini, Christopher A. Choquette-Choo, Natalie Dullerud, Anvith Thudi, Varun Chandrasekaran, and Nicolas Papernot. 2021. Proof-of-Learning: Definitions and Practice. In *IEEE Symposium on Security and Privacy*. IEEE, 1039–1056.
[37] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. 2010. Constant-Size Commitments to Polynomials and Their Applications. In *Advances in Cryptology—Asiacrypt 2010*, Masayuki Abe (Ed.). Springer, 177–194.
[38] Joe Kilian. 1992. A Note on Efficient Zero-Knowledge Proofs and Arguments. In *Symposium on Theory of Computing (STOC)*. ACM, 723–732.
[39] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. 2022. Nova: Recursive Zero-Knowledge Arguments from Folding Schemes. In *Advances in Cryptology—Crypto 2022*. Springer, 359–388.
[40] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. [n. d.]. CIFAR-10 (Canadian Institute for Advanced Research). ([n. d.]). http://www.cs.toronto.edu/~kriz/cifar.html
[41] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Adv. in Neural Information Processing Systems*, Vol. 25. Curran Associates, Inc.
[42] Yan LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324. https://doi.org/10.1109/5.726791
[43] Seunghwan Lee, Hankyung Ko, Jihye Kim, and Hyunok Oh. 2020. vCNN: Verifiable Convolutional Neural Network based on zk-SNARKs.
[44] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *Conference on Operating Systems Design and Implementation*. USENIX Association, 583–598.
[45] Tianyi Liu, Xiang Xie, and Yupeng Zhang. 2021. zkCNN: Zero Knowledge Proofs for Convolutional Neural Network Predictions and Accuracy. In *Conf. on Computer and Communications Security*. ACM, 2968–2985.
[46] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. 1992. Algebraic Methods for Interactive Proof Systems. *J. ACM* 39, 4 (oct 1992), 859–868.
[47] Ralph C. Merkle. 1988. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology—Crypto '87*. Springer, 369–378.
[48] Silvio Micali. 2000. Computationally Sound Proofs. *SIAM J. Computing* 30, 4 (2000), 1253–1298.
[49] Payman Mohassel and Peter Rindal. 2018. ABY3: A Mixed Protocol Framework for Machine Learning. In *Conf. on Computer and Comm. Security*. ACM, 35–52.
[50] Herbert Robbins and Sutton Monro. 1951. A Stochastic Approximation Method. *The Annals of Mathematical Statistics* 22, 3 (1951), 400 – 407.
[51] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *Intl. Conf. on Learning Representations, ICLR*.
[52] D.A. Spielman. 1996. Linear-time Encodable and Decodable Error-Correcting Codes. *IEEE Trans. Information Theory* 42, 6 (1996), 1723–1731.
[53] Justin Thaler. 2013. Time-Optimal Interactive Proofs for Circuit Evaluation. In *Advances in Cryptology—Crypto 2013*. Springer, 71–89.
[54] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2016. Stealing Machine Learning Models via Prediction APIs. In *USENIX Security Symposium*. USENIX Association, USA, 601–618.
[55] Paul Valiant. 2008. Incrementally Verifiable Computation or Proofs of Knowledge Imply Time/Space Efficiency. In *Theory of Cryptography (TCC)*. Springer, 1–18.
[56] R. S. Wahby, I. Tzialla, A. Shelat, J. Thaler, and M. Walfish. 2018. Doubly-Efficient zkSNARKs Without Trusted Setup. In *IEEE Symposium on Security and Privacy*. IEEE, 926–943. https://doi.org/10.1109/SP.2018.00060

[57] Chenkai Weng, Kang Yang, Xiang Xie, Jonathan Katz, and Xiao Wang. 2021. Mystique: Efficient Conversions for Zero-Knowledge Proofs with Applications to Machine Learning. In *USENIX Security Symposium*. USENIX Association, 501–518.

[58] Tiancheng Xie, Jiaheng Zhang, Zerui Cheng, Fan Zhang, Yupeng Zhang, Yongzheng Jia, Dan Boneh, and Dawn Song. 2022. zkBridge: Trustless Cross-Chain Bridges Made Practical. In *Conf. on Computer and Communications Security*. ACM, 3003–3017.

[59] Tiacheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. 2019. Libra: Succinct Zero-Knowledge Proofs with Optimal Prover Computation. In *Advances in Cryptology—Crypto 2019*. Springer, 733–764.

[60] Tiancheng Xie, Yupeng Zhang, and Dawn Song. 2022. Orion: Zero Knowledge Proof with Linear Prover Time. In *Adv. in Cryptology—Crypto 2022*. Springer, 299–328.

[61] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. 2021. QuickSilver: Efficient and Affordable Zero-Knowledge Proofs for Circuits and Polynomials over Any Field. In *Conf. on Computer and Comm. Security*. ACM, 2986–3001.

[62] Jiaheng Zhang, Tianyi Liu, Weijie Wang, Yinuo Zhang, Dawn Song, Xiang Xie, and Yupeng Zhang. 2021. Doubly Efficient Interactive Proofs for General Arithmetic Circuits with Linear Prover Time. In *Conf. on Computer and Communications Security*. ACM, 159–177.

[63] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. 2020. Transparent Polynomial Delegation and Its Applications to Zero Knowledge Proof. In *IEEE Symposium on Security and Privacy*. 859–876.

A ADDITIONAL PRELIMINARIES

In this section, we provide additional preliminaries, including formal definitions of arguments, polynomial commitment, protocols of sumcheck and GKR, and more of an in-detail presentation of the gradient descent algorithm for neural networks.

A.1 Arguments and Commitments

Argument systems. An argument satisfies completeness if an honest prover always gets accepted. The scheme satisfies knowledge soundness if, for any accepted prover, there exists an extractor that returns the underlying witness having oracle access to the prover. Zero-knowledge ensures that a simulator can generate a transcript indistinguishable from a real protocol execution without having any knowledge of the witness. We use $\mathcal{G}$ to represent the setup that generates public parameters pp.

Definition 1. (Zero-knowledge arguments) Given an NP relation R with language L_R, a zero-knowledge argument of knowledge for R is tuple of algorithms $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ if the following holds.

- ***Completeness.*** For every $(x, w) \in R$ and $\mathsf{pp} \leftarrow \mathcal{G}(1^\lambda)$,

$$\Pr[\langle \mathcal{P}(\mathsf{pp}, w), \mathcal{V}(\mathsf{pp})\rangle(x) = 1] = 1$$

- ***Knowledge soundness.*** For any PPT adversary $\mathcal{A}$, there exist a PPT extractor algorithm $\xi^{\mathcal{A}}$ such that for any x and $\mathsf{pp} \leftarrow \mathcal{G}(1^\lambda)$,

$$\Pr\left[w \leftarrow \xi^{\mathcal{A}}(\mathsf{pp}, x) : (x, w) \in R\right] \geq \Pr\left[\langle \mathcal{A}(\mathsf{pp}), \mathcal{V}(\mathsf{pp})\rangle(x) = 1\right] - \mathsf{negl}(\lambda)$$

- ***Zero knowledge.*** There exists a PPT simulator $\mathcal{S}$ such that for every adversary $\mathcal{A}$, $(x, w) \in R$, and $\mathsf{pp} \leftarrow \mathcal{G}(1^\lambda)$,

$$\mathsf{View}(\langle \mathcal{P}(\mathsf{pp}, w), \mathcal{A}(\mathsf{pp}, z)\rangle(x)) \approx \mathcal{S}^{\mathcal{A}}(x, z)$$

An argument system is said to be transparent if the public parameter pp is simply a uniform random string, i.e., there is no secret state to generate pp.

Polynomial commitment schemes. The conventional definition of polynomial commitment promises binding and hiding security notions. Binding can be considered commitment binding, where a commitment cannot be opened at two polynomials, or evaluation binding, where a tuple of a commitment and an evaluation cannot be opened at two input points. Hiding ensures that the commitment reveals no information about the polynomial. However, we require stronger guarantees in this work. We rely on commitments that satisfy knowledge soundness, a stronger notion than binding, and zero-knowledge, which implies the hiding property.

Knowledge soundness ensures that an accepted prover must know the polynomial underlying the commitment and evaluation proofs. More precisely, for any accepted prover, there is an extractor that has oracle access to the prover and can extract the underlying polynomial. This notion implies both the commitment and the evaluation binding. Zero-knowledge guarantees that commitment and evaluation proofs reveal no additional information about the polynomial. Similar to arguments, we provide a simulation-based definition for zero-knowledge. Trivially, zero-knowledge implies hiding when the prover honestly generates the commitment.

Definition 2. (Zero-knowledge polynomial commitment) Given a finite field $\mathbb{F}$ and an ℓ-variate polynomial f with degree d, a zero-knowledge polynomial commitment scheme PCS is tuple of algorithms (KeyGen, Commit, Open, Verify) if the following holds.

- ***Completeness.*** For input $x \in \mathbb{F}$ and public parameters pp,

$$\Pr\left[b = 1 \;\middle|\; \begin{array}{c} \sigma \leftarrow \mathsf{Commit}(f, r, \mathsf{pp}); \\ (y, \pi) \leftarrow \mathsf{Open}(f, r, x, \mathsf{pp}); \\ b \leftarrow \mathsf{Verify}(\sigma, x, y, \pi, \mathsf{pp}) \end{array}\right]$$

- ***Knowledge soundness.*** For any PPT adversary $\mathcal{A}$, there exist a PPT extractor algorithm $\xi^{\mathcal{A}}$ such that given public parameters pp,

$$\Pr\left[\; f \leftarrow \xi^{\mathcal{A}}(\mathsf{pp}) : f(x) = y \;\right] \geq \Pr\left[\begin{array}{c} (\sigma, x, y, \pi) \leftarrow \mathcal{A}(\mathsf{pp}); \\ \mathsf{Verify}(\sigma, x, y, \pi, \mathsf{pp}) = 1 \end{array}\right] - \mathsf{negl}(\lambda)$$

- ***Zero knowledge.*** There exists a simulator $\mathcal{S}$ such that for every adversary $\mathcal{A}$ the two following experiments are indistinguishable.

$\underline{\mathsf{Real}_{\mathcal{A},f}(1^\lambda)}$
(1) $\mathsf{pp} \leftarrow \mathsf{KeyGen}(1^\lambda, \ell, d)$
(2) $\sigma \leftarrow \mathsf{Commit}(f, r, \mathsf{pp})$
(3) $x \leftarrow \mathcal{A}(\sigma, \mathsf{pp})$
(4) $(y, \pi) \leftarrow \mathsf{Open}(f, r, x, \mathsf{pp})$
(5) $b \leftarrow \mathcal{A}(\sigma, x, y, \pi, \mathsf{pp})$
(6) Return b

$\underline{\mathsf{Ideal}_{\mathcal{A},\mathcal{S}}(1^\lambda)}$
(1) $(\sigma, \mathsf{pp}) \leftarrow \mathcal{S}(1^\lambda, \ell, d)$
(2) $x \leftarrow \mathcal{A}(\sigma, \mathsf{pp})$
(3) $\pi \leftarrow \mathcal{S}(\sigma, x, f(x), \mathsf{pp})$
(4) $b \leftarrow \mathcal{A}(\sigma, x, f(x), \pi, \mathsf{pp})$
(5) Return b

In particular, for any $\mathcal{A}$, we have

$$|\Pr[\mathsf{Real}_{\mathcal{A},f}(1^\lambda) = 1] - \Pr[\mathsf{Ideal}_{\mathcal{A},\mathcal{S}}(1^\lambda) = 1]| \leq \mathsf{negl}(\lambda)$$

A.2 The Sumcheck and GKR Protocols

We present the sumcheck scheme in Protocol 1. Moreover, we provide details of the GKR interactive proof for layered circuits in Protocol 2. In our implementation, we use a variant of the GKR sumcheck-based proof that can support non-layered circuits with gates having arbitrary fan-in [62].

Theorem 5. *[46] Given a finite field $\mathbb{F}$ and an ℓ-variate polynomial $f : \mathbb{F}^\ell \to \mathbb{F}$ with degree bound d, Protocol 1 is an interactive proof for the statement $H = \sum_{b \in \{0,1\}^\ell} f(b)$ with soundness error $O(\frac{d\ell}{|\mathbb{F}|})$. It uses $O(\ell)$ of interaction, running time of $\mathcal{P}$ is $O((d+1)^\ell)$, and the proof size and the running time of $\mathcal{V}$ is $O(d\ell)$.*

Theorem 6. *[59] Let $C : \mathbb{F}^{s_{in}} \to \mathbb{F}^{s_{out}}$ be a depth-d layered arithmetic circuit. Protocol 2 is an interactive proof for C with soundness error $O(d \cdot \frac{\log |C|}{|\mathbb{F}|})$. It uses $O(d \cdot \log |C|)$ of interaction and running time of $\mathcal{P}$ is $O(|C|)$. Let T be the optimal computation time for all $\widetilde{add}_i$ and $\widetilde{mult}_i$. The running time of $\mathcal{V}$ is $O(s_{in} + s_{out} + d \cdot \log |C| + T)$. For log-space uniform circuits, it is $T = \mathsf{polylog}|C|$.*

Protocol 1. Sumcheck

The protocol proceeds in ℓ rounds.

(1) In the first round, $\mathcal{P}$ sends the polynomial

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

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

(2) In the ith round ($2 \le i \le \ell - 1$), $\mathcal{P}$ sends the polynomial

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

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

(3) In the ℓth round, $\mathcal{P}$ sends the polynomial

$$f_\ell(x_\ell) = f(r_1, r_2, \ldots, x_\ell)$$

$\mathcal{V}$ 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, $\mathcal{V}$ accepts if and only if $f_\ell(r_\ell) = f(r_1, r_2, \ldots, r_\ell)$.

A.3 DNNs and Gradient Descent

We consider two possible layers for the underlying neural network: (i) dense layers, where the linear operation is matrix multiplication; (ii) convolutional layers, where the linear operation is a convolution. A linear operation is followed by an activation, which can be ReLU and tanh for non-output and Softmax for the output layers. In convolutional layers, the activation is additionally followed by a average or max pooling. Moreover, a convolutional layer might include several input and output channels. For instance, if the input to the layer is an RGB image, it has three channels. In such cases, weights are convoluted with each input layer and added together. Each output channel has its own set of weights. We only consider one input/output channel per layer to simplify the algorithm description. The computation is presented in Algorithm 1. In line 20, $\mathsf{pad}(\boldsymbol{W}_{i,\ell}^T)$ zero-pads $\boldsymbol{W}_{i,\ell}^T$ to an specific dimension and then rotate it for π. More precisely, for $\boldsymbol{W}_{i,\ell}$ of size $w \times w$ and $\boldsymbol{U}_{i,\ell-1}$ of size $n \times n$, $\mathsf{pad}(\boldsymbol{W}_{i,\ell}^T)$ is of size $(2w - n) \times (2w - n)$. In line 24, $\overline{G}_{i,l}$ is the average gradient over the data points in the batch.

Algorithm 1 Mini-Batch Gradient Descent

Input: Weights $\{\boldsymbol{W}_{i-1,\ell}\}_{j=1}^{L}$ and Batch $\boldsymbol{B}_{i-1}$

Output: Updated weights $\{\boldsymbol{W}_{i,\ell}\}_{j=1}^{L}$

Forward pass:

Let $\boldsymbol{U}_{i,0}$ be the concatenation of input features included in $\boldsymbol{B}_{i-1}$

1: **for** $\ell = 1$ to L **do**
2: **if** the ℓ-th layer is dense **then**
3: $\boldsymbol{T}_{i,\ell} \leftarrow \boldsymbol{W}_{i-1,\ell} \cdot \boldsymbol{U}_{i,\ell-1}$
4: $\boldsymbol{U}_{i,\ell} \leftarrow \mathsf{act}_\ell(\boldsymbol{T}_{i,\ell})$ ▷ $\mathsf{act}_\ell \in \{\mathsf{ReLU}, \tanh, \mathsf{Softmax}\}$
5: **else if** the ℓ-th layer is convolution **then**
6: $\boldsymbol{T}_{i,\ell} \leftarrow \boldsymbol{W}_{i-1,\ell} * \boldsymbol{U}_{i,\ell-1}$
7: $\boldsymbol{Q}_{i,\ell} \leftarrow \mathsf{act}_\ell(\boldsymbol{T}_{i,\ell})$ ▷ $\mathsf{act}_\ell \in \{\mathsf{ReLU}, \tanh, \mathsf{Softmax}\}$
8: $\boldsymbol{U}_{i,\ell} \leftarrow \mathsf{pool}_\ell(\boldsymbol{Q}_{i,\ell})$ ▷ $\mathsf{pool}_\ell \in \{\mathsf{MaxPool}, \mathsf{AvgPool}\}$
9: **end if**
10: **end for**

Backward pass:

Let $\boldsymbol{R}_{i,L+1} = \partial \mathcal{L}(\boldsymbol{U}_{i,L}, Y_i)/\partial \boldsymbol{U}_{i,L}$, s.t. Y_i includes labels in $\boldsymbol{B}_{i-1}$

11: **for** $\ell = L$ to 1 **do**
12: **if** the ℓ-th layer is dense **then**
13: $\boldsymbol{T}'_{i,\ell} = \partial \boldsymbol{U}_{i,\ell}/\partial \boldsymbol{T}_{i,\ell} = \partial \mathsf{act}_\ell(\boldsymbol{T}_{i,\ell})/\partial \boldsymbol{T}_{i,\ell}$
14: $\boldsymbol{G}_{i,\ell} = (\boldsymbol{R}_{i,\ell+1} \circ \boldsymbol{T}'_{i,\ell}) \cdot \boldsymbol{U}_{i,\ell-1}^T$
15: $\boldsymbol{R}_{i,\ell} = \boldsymbol{W}_{i-1,\ell}^T \cdot (\boldsymbol{R}_{i,\ell+1} \circ \boldsymbol{T}'_{i,\ell})$
16: **else if** the ℓ-th layer is convolution **then**
17: $\boldsymbol{Q}'_{i,\ell} = \partial \boldsymbol{U}_{i,\ell}/\partial \boldsymbol{Q}_{i,\ell} = \partial \mathsf{pool}_\ell(\boldsymbol{Q}_{i,\ell})/\partial \boldsymbol{Q}_{i,\ell}$
18: $\boldsymbol{T}'_{i,\ell} = \partial \boldsymbol{U}_{i,\ell}/\partial \boldsymbol{T}_{i,\ell} = \boldsymbol{Q}'_{i,\ell} \circ (\partial \mathsf{act}_\ell(\boldsymbol{T}_{i,\ell})/\partial \boldsymbol{T}_{i,\ell})$
19: $\boldsymbol{G}_{i,\ell} = (\boldsymbol{R}_{i,\ell+1} \circ \boldsymbol{T}'_{i,\ell}) * \boldsymbol{U}_{i,\ell-1}$
20: $\boldsymbol{R}_{i,\ell} = \mathsf{pad}(\boldsymbol{W}_{i-1,\ell}^T) * (\boldsymbol{R}_{i,\ell+1} \circ \boldsymbol{T}'_{i,\ell})$ ▷ If $\ell \neq 1$
21: **end if**
22: **end for**

Update weights:

23: **for** $\ell = 1$ to L **do**
24: $\boldsymbol{W}_{i,l} = \boldsymbol{W}_{i-1,l} - \eta \cdot \overline{G}_{i,l}$ ▷ η is the learning rate
25: **end for**

B OUR POGD PROTOCOL

In this section, we present the complete PoGD scheme in Protocol 3. Moreover, we formally prove the security of the protocol. We note that the sumcheck runs on the linear operations in step (4) of phase 2 and step (2) of phase 3 and can be merged together. Parties can combine evaluations $\widetilde{T}_{i,\ell}(r^{(3)}_{2,\boldsymbol{T}_{i,\ell}})$ and $\widetilde{T}_{i,\ell}(r^{(1)}_{3,\boldsymbol{T}_{i,\ell}})$ and then invoke the sumcheck protocol on the combined evaluation. Similarly, scalings in step (6) of phase 2 and step (2) of phase 3 can be merged together.

Proof of Theorem 1. Following the completeness of GKR, $\mathsf{Sum}_{\mathsf{Mat}}$, and $\mathsf{Sum}_{\mathsf{Conv}}$, and PCS completeness of PoGD is immediate. We first show a negligible bound soundness error with a phase-by-phase analyze. Suppose that the prover sends some $\boldsymbol{W}^*_{i,\ell} \neq \boldsymbol{W}_{i,\ell}$.

P1: For $i = 1$ to L, following the Schwartz-Zipple lemma and the soundness of GKR, we have $\widetilde{\boldsymbol{W}}^*_{i,\ell}(r_{\mathsf{out},\boldsymbol{W}_{i,\ell}}) \neq \widetilde{\boldsymbol{W}}_{i,\ell}(r_{\mathsf{out},\boldsymbol{W}_{i,\ell}})$ with high probability unless: (i) $\widetilde{\boldsymbol{W}}^*_{i-1,\ell}(r_{1,\boldsymbol{W}_{i-1,\ell}}) \neq \widetilde{\boldsymbol{W}}_{i-1,\ell}(r_{1,\boldsymbol{W}_{i-1,\ell}})$, which proceeds to phase 4; (ii) $\widetilde{\boldsymbol{G}}^*_{i,\ell}(r_{1,\boldsymbol{G}_{i,\ell}}) \neq \widetilde{\boldsymbol{G}}_{i,\ell}(r_{1,\boldsymbol{G}_{i,\ell}})$, which proceeds to phase 2.

P2: Suppose that case (ii) is occurred in phase 1 and there exists at least one ℓ such that $\widetilde{\boldsymbol{G}}^*_{i,\ell}(r_{1,\boldsymbol{G}_{i,\ell}}) \neq \widetilde{\boldsymbol{G}}_{i,\ell}(r_{1,\boldsymbol{G}_{i,\ell}})$. Following the soundness of GKR, $\mathsf{Sum}_{\mathsf{Mat}}$, and $\mathsf{Sum}_{\mathsf{Conv}}$, there are three possible cases can occur: (i) $\widetilde{\boldsymbol{W}}^*_{i-1,\ell}(r_{2,\boldsymbol{W}_{i-1,\ell}}) \neq \widetilde{\boldsymbol{W}}_{i-1,\ell}(r_{2,\boldsymbol{W}_{i-1,\ell}})$, $\widetilde{\mathbf{AUX}}^*_{i,\ell}(r_{2,\mathbf{AUX}_{i,\ell}}) \neq \widetilde{\mathbf{AUX}}_{i,\ell}(r_{2,\mathbf{AUX}_{i,\ell}})$, or for the data batch $\widetilde{\boldsymbol{B}}^*_{i,\ell}(r_{2,\boldsymbol{B}_{i,\ell}}) \neq \widetilde{\boldsymbol{B}}_{i,\ell}(r_{2,\boldsymbol{B}_{i,\ell}})$, which proceeds to phase 4; The next case is (ii) $\widetilde{\boldsymbol{R}}^*_{i,\ell+1}(r_{2,\boldsymbol{R}_{i,\ell+1}}) \neq \widetilde{\boldsymbol{R}}_{i,\ell+1}(r_{2,\boldsymbol{R}_{i,\ell+1}})$, proceeding to the next layer; (iii) $\widetilde{\boldsymbol{U}}^*_{i,\ell}(r_{2,\boldsymbol{U}_{i,\ell}}) \neq \widetilde{\boldsymbol{U}}_{i,\ell}(r_{2,\boldsymbol{U}_{i,\ell}})$, proceeding to phase 3.

P3: Suppose (iii) occurs in phase 2 and there exists at least one ℓ such that $\widetilde{\boldsymbol{U}}^*_{i,\ell}(r_{2,\boldsymbol{U}_{i,\ell}}) \neq \widetilde{\boldsymbol{U}}_{i,\ell}(r_{2,\boldsymbol{U}_{i,\ell}})$. Following the soundness of GKR and $\mathsf{Sum}_{\mathsf{Mat}}$, and $\mathsf{Sum}_{\mathsf{Conv}}$, there are two possible cases: (i) $\widetilde{\boldsymbol{W}}^*_{i-1,\ell}(r_{3,\boldsymbol{W}_{i,\ell}}) \neq \widetilde{\boldsymbol{W}}_{i-1,\ell}(r_{3,\boldsymbol{W}_{i,\ell}})$, $\widetilde{\boldsymbol{B}}^*_{i,\ell}(r_{3,\boldsymbol{B}_{i,\ell}}) \neq \widetilde{\boldsymbol{B}}_{i,\ell}(r_{3,\boldsymbol{B}_{i,\ell}})$, or $\widetilde{\mathbf{AUX}}^*_{i,\ell}(r_{3,\mathbf{AUX}_{i,\ell}}) \neq \widetilde{\mathbf{AUX}}_{i,\ell}(r_{3,\mathbf{AUX}_{i,\ell}})$, which proceeds phase 4; (ii) $\widetilde{\boldsymbol{U}}^*_{i,\ell-1}(r_{3,\boldsymbol{U}_{i,\ell-1}}) \neq \widetilde{\boldsymbol{U}}_{i,\ell-1}(r_{3,\boldsymbol{U}_{i,\ell-1}})$, which proceeds to the next iteration, i.e., $\ell - 1$-th layer.

Protocol 2. GKR

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

(1) $\mathcal{V}$ picks a random $z \in \mathbb{F}$ and sends it to $\mathcal{P}$. Both parties $\mathcal{P}$ and $\mathcal{V}$ compute $\tilde{V}_0(z)$, where $\tilde{V}_0$ is the multilinear extension of array $\mathbf{out}$.

(2) $\mathcal{P}$ and $\mathcal{V}$ invoke a sumcheck protocol on

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

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

(3) For $i = 1, 2, \ldots, d$:

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

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

At the end of the protocol, $\mathcal{V}$ receives two evaluations $\tilde{V}_{i+1}(u^{(i+1)})$, $\tilde{V}_{i+1}(v^{(i+1)})$. $\mathcal{V}$ computes the following and if it does not equal to the last message of the sumcheck, returns 0.

$$\begin{aligned}&\left(\alpha^{(i)}\tilde{add}_i(u^{(i)},u^{(i+1)},v^{(i+1)}) + \beta^{(i)}\tilde{add}_i(v^{(i)},u^{(i+1)},v^{(i+1)})\right).\left(\tilde{V}_{i+1}(u^{(i+1)}) + \tilde{V}_{i+1}(v^{(i+1)})\right)\\ &+\left(\alpha^{(i)}\tilde{mult}_i(u^{(i)},u^{(i+1)},v^{(i+1)}) + \beta^{(i)}\tilde{mult}_i(v^{(i)},u^{(i+1)},v^{(i+1)})\right).\tilde{V}_{i+1}(u^{(i+1)}).\tilde{V}_{i+1}(v^{(i+1)})\end{aligned}$$

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

P4: Suppose that the case (i) in phases one, two, or three has caused an error, proceeding to this phase. Then, following the soundness of the evaluation reduction sumcheck and also the Schwartz-Zipple lemma, either $\widetilde{W}^*_{i-1}(r_{\mathsf{in},W_{i-1}}) \neq \widetilde{W}_{i-1}(r_{\mathsf{in},W_{i-1}})$, $\widetilde{B}^*_i(r_{\mathsf{in},B_i}) \neq \widetilde{B}_i(r_{\mathsf{in},B_i})$, or $\widetilde{\mathbf{AUX}}^*_i(r_{\mathsf{in},\mathbf{AUX}_i}) \neq \widetilde{\mathbf{AUX}}_i(r_{\mathsf{in},\mathbf{AUX}_i})$. Any of these cases cause $\mathcal{V}$ to return reject based the soundness of the underlying polynomial commitment scheme PCS.

Knowledge soundness follows immediately from the above. As PCS is knowledge sound, the extractor of the PoGD scheme can invoke the extractor of PCS to get witnesses committed to by the prover $\widetilde{W}^*_i$, $\widetilde{W}^*_{i-1}$, $\widetilde{B}^*_{i-1}$, and $\widetilde{\mathbf{AUX}}^*_i$. As PoGD is sound, they are satisfying. Moreover, as GKR, $\mathsf{Sum}_{\mathsf{Mat}}$, $\mathsf{Sum}_{\mathsf{Conv}}$, and PCS are zero-knowledge, the simulator of PoGD can invoke the simulators of these building blocks to generate proof messages and commitments indistinguishable from a real PoGD protocol execution.

C COMMITMENT AGGREGATION SCHEME

In this section, we first provide a formal definition of a polynomial commitment aggregation scheme. We then present our construction in Protocol 4 and sketch the security proof. An aggregation scheme consists of a prover $\mathcal{P}$ and a verifier algorithm $\mathcal{V}$ such that on input a set of commitment/evaluation instances $\{(\sigma_i, x_i, y_i)\}_{i=1}^k$ the protocol ends with an aggregated instance (σ^*, x^*, y^*). Each instance input (σ_i, x_i, y_i) includes a commitment σ_i to a polynomial f_i, which is supposed to satisfy $y_i = f_i(x_i)$. Similarly, σ^* is a commitment to a polynomial f^* with evaluation $y^* = f^*(x^*)$. As the witness, the prover holds input polynomials $\{f_i\}$ and their corresponding commitment randomnesses. In the end, the prover gets the output polynomial f^* and its commitment randomness.

If one can substantiate that the aggregated instance (σ^*, x^*, y^*) is satisfied, we can conclude that input instances (σ_i, x_i, y_i) are also satisfied if the aggregation verifier has returned accept. Furthermore, aggregation messages should not reveal any information about each input polynomial f_i other than its evaluation points (x_i, y_i). We next present the security notions more formally.

Definition 3. (Zero-knowledge aggregation scheme) Given a knowledge sound zero-knowledge polynomial commitment PCS with parameters pp, an aggregation scheme $\mathsf{AGG} = (\mathcal{P}, \mathcal{V})$ satisfies:

- ***Completeness.*** Given satisfying instances $S = \{(\sigma_i, x_i, y_i)\}_{i=1}^k$ with witness $W = \{(f_i, r_i)\}_{i=1}^k$, the following probability is 1.

$$\Pr\left[b_0 \wedge b_1 = 1 \;\middle|\; \begin{array}{c}(\sigma^*,x^*,y^*),(f^*,r^*),\pi_0 \leftarrow \mathcal{P}(S,W,\mathsf{pp});\\ b_0 \leftarrow \mathcal{V}(S,(\sigma^*,x^*,y^*),\pi_0,\mathsf{pp});\\ (y^*,\pi_1) \leftarrow \mathsf{PCS.Open}(f^*,r^*,x^*,\mathsf{pp});\\ b_1 \leftarrow \mathsf{PCS.Verify}(\sigma^*,x^*,y^*,\pi_1,\mathsf{pp});\end{array}\right]$$

- ***Knowledge soundness.*** For any PPT adversary prover $\mathcal{A}$, there exist a PPT extractor algorithm $\xi^{\mathcal{A}}$ such that

$$\begin{aligned}&\Pr\left[\begin{array}{c}\{(f_i,r_i)\}_{i=1}^k,(f^*,r^*) \leftarrow \xi^{\mathcal{A}}(\mathsf{pp}):\\ f^*(x^*) = y^* \wedge f_i(x_i) = y_i \qquad \forall i \in [k]\end{array}\right]\\ \geq \Pr&\left[\begin{array}{c}\{(\sigma_i,x_i,y_i)\}_{i=1}^k,(\sigma^*,x^*,y^*),\pi_0,\pi_1 \leftarrow \mathcal{A}(\mathsf{pp});\\ \mathcal{V}(\{(\sigma_i,x_i,y_i)\}_{i=1}^k,(\sigma^*,x^*,y^*),\pi_0,\mathsf{pp}) = 1;\\ \mathsf{PCS.Verify}(\sigma^*,x^*,y^*,\pi_1,\mathsf{pp}) = 1;\end{array}\right] - \mathsf{negl}(\lambda)\end{aligned}$$

- ***Zero knowledge.*** Given a set of tuples $F = \{\{(f_i, x_i, y_i)\}_{i=1}^k$, where each f_i is a polynomial with evaluation $f_i(x_i) = y_i$, there exists a simulator $\mathcal{S}$ such that given oracle access to the algorithms of PCS, for every adversary distinguisher $\mathcal{A}$ the following two ideal world and real world experiments are indistinguishable.

Protocol 3. Proof of Gradient Descent Protocol

Parameters: *Let* GKR *be the Virgo++ [62] implementation of a a generic GKR-style zero-knowledge proof system, ,* $\mathsf{Sum}_{\mathsf{Mat}}$ *be the zero-knowledge sublinear sumcheck protocol for matrix multiplication [53], and* $\mathsf{Sum}_{\mathsf{Conv}}$ *be the zero-knowledge sublinear sumcheck protocol for convolution [45]. Furthermore, let* $\mathsf{PCS} = (\mathsf{KeyGen}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Verify})$ *be the Orion [60] polynomial commitment scheme with public parameters* pp. *Suppose that the model has* L *layers.*

Initialization: *On inputs weights* $\boldsymbol{W}_{i-1} = \{\boldsymbol{W}_{i-1,\ell}\}_{\ell=1}^{L}$ *and a batch* $\boldsymbol{B}_{i-1}$, $\mathcal{P}$ *runs gradient descent (cf. Algorithm 1) to evaluate outputs* $\boldsymbol{U}_i = \{\boldsymbol{U}_{i,\ell}\}_{\ell=1}^{L}$, *gradients* $\boldsymbol{G}_i = \{\boldsymbol{G}_{i,\ell}\}_{\ell=1}^{L}$ *and* $\boldsymbol{R}_i = \{\boldsymbol{R}_{i,\ell}\}_{\ell=1}^{L}$, *weights* $\boldsymbol{W}_i = \{\boldsymbol{W}_{i,\ell}\}_{\ell=1}^{L}$, *and auxiliary inputs* $\mathbf{AUX}_i = \{\mathbf{AUX}_{i,\ell}\}_{\ell=1}^{L}$. $\mathcal{P}$ *sends* $\mathcal{V}$ *the commitments* $\sigma_{W_i} \leftarrow \mathsf{PCS.Commit}(\widetilde{\boldsymbol{W}}_i, r_{W_i})$, $\sigma_{W_{i-1}} \leftarrow \mathsf{PCS.Commit}(\widetilde{\boldsymbol{W}}_{i-1}, r_{W_{i-1}})$, $\sigma_{B_{i-1}} \leftarrow \mathsf{PCS.Commit}(\widetilde{\boldsymbol{B}}_{i-1}, r_{B_{i-1}})$, *and* $\sigma_{\mathbf{AUX}_i} \leftarrow \mathsf{PCS.Commit}(\widetilde{\mathbf{AUX}}_i, r_{\mathbf{AUX}_i})$.

Phase 1: Proof of update. For $\ell = 1$ to L ***do the following steps.***

(1) $\mathcal{V}$ *samples* $r_{\mathsf{out},\boldsymbol{W}_{i,\ell}}$ *and request the evaluation* $\widetilde{\boldsymbol{W}}_{i,\ell}(r_{\mathsf{out},\boldsymbol{W}_{i,\ell}})$.

(2) Having $\widetilde{\boldsymbol{W}}_{i,\ell}(r_{\mathsf{out},\boldsymbol{W}_{i,\ell}})$, *parties run* GKR *on* $\boldsymbol{W}_{i,\ell} = \boldsymbol{W}_{i-1,\ell} - \eta \cdot \overline{\boldsymbol{G}}_{i,\ell}$, *which ends with evaluations* $\widetilde{\boldsymbol{W}}_{i-1,\ell}(r_{1,\boldsymbol{W}_{i-1,\ell}})$, $\widetilde{\boldsymbol{G}}_{i,\ell}(r_{1,\boldsymbol{G}_{i,\ell}})$.

Phase 2: Proof of backward pass. For $\ell = 1$ to L ***do the following steps.***

(1) Having $\widetilde{\boldsymbol{G}}_{i,\ell}(r_{1,\boldsymbol{G}_{i,\ell}})$ *from phase 1, parties run* $\mathsf{Sum}_{\mathsf{Conv}}$ *on* $\boldsymbol{G}_{i,\ell} = (\boldsymbol{R}_{i,\ell+1} \circ \boldsymbol{T}'_{i,\ell}) * \boldsymbol{U}_{i,\ell-1}$ *if the* ℓ *layer is convolutional or otherwise* $\mathsf{Sum}_{\mathsf{Mat}}$ *on* $\boldsymbol{G}_{i,\ell} = (\boldsymbol{R}_{i,\ell+1} \circ \boldsymbol{T}'_{i,\ell}) \cdot \boldsymbol{U}_{i,\ell-1}^{T}$ *if the layer is dense. At the end of the run, parties receive evaluations* $\widetilde{\boldsymbol{T}'}_{i,\ell+1}(r^{(1)}_{2,\boldsymbol{T}'_{i,\ell+1}})$, $\widetilde{\boldsymbol{U}}_{i,\ell}(r^{(1)}_{2,\boldsymbol{U}_{i,\ell}})$, *and* $\widetilde{\boldsymbol{R}}_{i,\ell+1}(r^{(1)}_{2,\boldsymbol{R}_{i,\ell+1}})$.

(2) For $\ell \neq 1$, *having* $\widetilde{\boldsymbol{R}}_{i,\ell}(r_{2,\boldsymbol{R}_{i,\ell}})$ *from the previous iteration, parties run* $\mathsf{Sum}_{\mathsf{Conv}}$ *on* $\boldsymbol{R}_{i,\ell} = \mathsf{pad}(\boldsymbol{W}_{i-1,\ell}^{T}) * (\boldsymbol{R}_{i,\ell+1} \circ \boldsymbol{T}'_{i,\ell})$ *if the layer if convolutional or* $\mathsf{Sum}_{\mathsf{Mat}}$ *on* $\boldsymbol{R}_{i,\ell} = \boldsymbol{W}_{i-1,\ell}^{T} \cdot (\boldsymbol{R}_{i,\ell+1} \circ \boldsymbol{T}'_{i,\ell})$ *if the layer is dense, ending with evaluations* $\widetilde{\boldsymbol{R}}_{i,\ell+1}(r^{(2)}_{2,\boldsymbol{R}_{i,\ell+1}})$, $\widetilde{\boldsymbol{W}}_{i-1,\ell}(r^{(2)}_{2,\boldsymbol{W}_{i-1,\ell}})$, *and* $\widetilde{\boldsymbol{T}'}_{i,\ell+1}(r^{(2)}_{2,\boldsymbol{T}'_{i,\ell+1}})$.

(3) Parties combine evaluations $\widetilde{\boldsymbol{T}'}_{i,\ell+1}(r^{(1)}_{2,\boldsymbol{R}_{i,\ell+1}})$ *and* $\widetilde{\boldsymbol{T}'}_{i,\ell}(r^{(2)}_{2,\boldsymbol{T}'_{i,\ell}})$, *by running an evaluation reduction sumcheck, i.e., Equation* (5). *Having the combination, parties run* GKR *on* $\boldsymbol{T}'_{i,\ell} = \boldsymbol{Q}'_{i,\ell} \circ (\partial \mathsf{act}_\ell(\boldsymbol{T}_{i,\ell})/\partial \boldsymbol{T}_{i,\ell})$ *followed by another run on* $\boldsymbol{Q}'_{i,\ell} = \partial \mathsf{pool}_\ell(\mathsf{act}_\ell(\boldsymbol{T}_{i,\ell}))/\partial \mathsf{act}_\ell(\boldsymbol{T}_{i,\ell})$ *if the layer is convolutional or otherwise son* $\boldsymbol{T}'_{i,\ell} = \partial \boldsymbol{U}_{i,\ell}/\partial \boldsymbol{T}_{i,\ell} = \partial \mathsf{act}_\ell(\boldsymbol{T}_{i,\ell})/\partial \boldsymbol{T}_{i,\ell}$ *if the layer is dense. At the end, parties receive evaluations* $\widetilde{\boldsymbol{T}}_{i,\ell}(r^{(3)}_{2,\boldsymbol{T}_{i,\ell}})$ *and* $\widetilde{\mathbf{AUX}}_{i,\ell}(r^{(3)}_{2,\mathbf{AUX}_{i,\ell}})$.

(4) Having $\widetilde{\boldsymbol{T}}_{i,\ell}(r^{(3)}_{2,\boldsymbol{T}_{i,\ell}})$, *parties tun* $\mathsf{Sum}_{\mathsf{Conv}}$ *on* $\boldsymbol{T}_{i,\ell} = \boldsymbol{W}_{i-1,\ell} * \boldsymbol{U}_{i,\ell-1}$ *if the layer is convolution or* $\mathsf{Sum}_{\mathsf{Mat}}$ *on* $\boldsymbol{T}_{i,\ell} = \boldsymbol{W}_{i-1,\ell} \cdot \boldsymbol{U}_{i,\ell-1}$ *if the layer is dense. At the end, parties received evaluations* $\widetilde{\boldsymbol{U}}_{i,\ell-1}(r^{(4)}_{2,\boldsymbol{U}_{i,\ell-1}})$ *and* $\widetilde{\boldsymbol{W}}_{i-1,\ell}(r^{(4)}_{2,\boldsymbol{W}_{i-1,\ell}})$. *For* $\ell = 1$, $\widetilde{\boldsymbol{U}}_{i,0}(r^{(4)}_{2,\boldsymbol{U}_{i,0}})$ *is viewed as* $\widetilde{\boldsymbol{B}}_{i-1}(r^{(4)}_{2,\boldsymbol{B}_{i-1}})$.

(5) For $\ell = L$, *parties combine* $\widetilde{\boldsymbol{R}}_{i,L+1}(r^{(1)}_{2,\boldsymbol{R}_{i,L+1}})$ *and* $\widetilde{\boldsymbol{R}}_{i,L+1}(r^{(2)}_{2,\boldsymbol{R}_{i,L+1}})$ *by an evaluation reduction sumcheck. Having the combinations, parties run* GKR *on* $\boldsymbol{R}_{i,L+1} = \partial \mathcal{L}(\boldsymbol{U}_{i,L}, Y_i)/\partial \boldsymbol{U}_{i,L}$. *At the end, parties receive evaluations* $\widetilde{\boldsymbol{B}}_{i-1}(r^{(5)}_{2,\boldsymbol{B}_{i-1}})$, $\widetilde{\boldsymbol{U}}_{i,L}(r^{(5)}_{2,\boldsymbol{U}_{i,L}})$, *and* $\widetilde{\mathbf{AUX}}_{i,\ell}(r^{(5)}_{2,\mathbf{AUX}_{i,\ell}})$.

(6) Consistency checks and scalings are proved by running sumchecks on Equation (3) *and Equation* (4), *which ends with several evaluations of auxiliary inputs* $\widetilde{\mathbf{AUX}}_{i,\ell}$. *The evaluations are combined together into a single evaluation* $\widetilde{\mathbf{AUX}}_{i,\ell}(r^{(6)}_{2,\mathbf{AUX}_{i,\ell}})$.

(7) For $\ell = L$, *evaluations of* $\widetilde{\boldsymbol{W}}_{i-1,\ell}$, $\widetilde{\boldsymbol{B}}_{i-1}$, $\widetilde{\boldsymbol{U}}_{i,\ell}$, $\widetilde{\mathbf{AUX}}_{i,\ell}$, *from steps 1-5, are combined into* $\widetilde{\boldsymbol{W}}_{i-1,\ell}(r_{2,\boldsymbol{W}_{i-1,\ell}})$, $\widetilde{\boldsymbol{U}}_{i,\ell}(r_{2,\boldsymbol{U}_{i,\ell}})$, $\widetilde{\boldsymbol{B}}_{i-1}(r_{2,\boldsymbol{B}_{i-1}})$, $\widetilde{\mathbf{AUX}}_{i,\ell}(r_{2,\mathbf{AUX}_{i,\ell}})$.

Phase 3: Proof of forward pass. For $\ell = L$ to 1 ***do the following steps.***

(1) Let $\widetilde{\boldsymbol{U}}_{i,L}(r_{3,\boldsymbol{U}_{i,L}}) = \widetilde{\boldsymbol{U}}_{i,L}(r_{2,\boldsymbol{U}_{i,L}})$. *Parties combine* $\widetilde{\boldsymbol{U}}_{i,\ell}(r_{3,\boldsymbol{U}_{i,\ell}})$ *received from the previous iteration (if* $\ell \neq L$*) and* $\widetilde{\boldsymbol{U}}_{i,\ell-1}(r_{2,\boldsymbol{U}_{i,\ell-1}})$ *from phase 2. Having the combined evaluation, parties run* GKR *on* $\boldsymbol{U}_{i,\ell} = \mathsf{pool}_\ell(\boldsymbol{Q}_{i,\ell})$ *followed by another run on* $\boldsymbol{Q}_{i,\ell} = \mathsf{act}_\ell(\boldsymbol{T}_{i,\ell})$ *if the layer is convolutional or on* $\boldsymbol{U}_{i,\ell} = \mathsf{act}_\ell(\boldsymbol{T}_{i,\ell})$ *if the layer is dense, ending with* $\widetilde{\boldsymbol{T}}_{i,\ell}(r^{(1)}_{3,\boldsymbol{T}_{i,\ell}})$ *and* $\widetilde{\mathbf{AUX}}_{i,\ell}(r^{(1)}_{3,\mathbf{AUX}_{i,\ell}})$.

(2) Having $\widetilde{\boldsymbol{T}}_{i,\ell}(r^{(1)}_{3,\boldsymbol{T}_{i,\ell}})$, *parties run* $\mathsf{Sum}_{\mathsf{Conv}}$ *on* $\boldsymbol{T}_{i,\ell} = \boldsymbol{W}_{i-1,\ell} * \boldsymbol{U}_{i,\ell-1}$ *or* $\mathsf{Sum}_{\mathsf{Mat}}$ *on* $\boldsymbol{T}_{i,\ell} = \boldsymbol{W}_{i-1,\ell}\boldsymbol{U}_{i,\ell-1}$. *At the end, parties receive evaluations* $\widetilde{\boldsymbol{W}}_{i-1,\ell}(r_{3,\boldsymbol{W}_{i,\ell}})$ *and* $\widetilde{\boldsymbol{U}}_{i,\ell-1}(r_{3,\boldsymbol{U}_{i,\ell-1}})$. *If* $i = 1$, $\widetilde{\boldsymbol{U}}_{i,0}(r_{3,\boldsymbol{U}_{i,0}})$ *is viewed as* $\widetilde{\boldsymbol{B}}_{i-1,\ell}(r_{3,\boldsymbol{B}_{i-1,\ell}})$. *For consistency checks and scalings, parties run sumchecks on Equation* (3) *and Equation* (4), *ending with several evaluations combined together and with* $\widetilde{\mathbf{AUX}}_{i,\ell}(r^{(1)}_{3,\mathbf{AUX}_{i,\ell}})$ *into* $\widetilde{\mathbf{AUX}}_{i,\ell}(r_{3,\mathbf{AUX}_{i,\ell}})$.

Phase 4: Proof of evaluation reduction.

(1) Parties combine received evaluations of form $\widetilde{\boldsymbol{W}}_{i-1,\ell}(r_{1,\boldsymbol{W}_{i-1,\ell}})$, $\widetilde{\boldsymbol{W}}_{i-1,\ell}(r_{2,\boldsymbol{W}_{i-1,\ell}})$, *and* $\widetilde{\boldsymbol{W}}_{i-1,\ell}(r_{3,\boldsymbol{W}_{i-1,\ell}})$ *into a single evaluation* $\widetilde{\boldsymbol{W}}_{i-1}(r_{\mathsf{in},\boldsymbol{W}_{i-1}})$, *evaluations of form* $\widetilde{\boldsymbol{B}}_{i-1,\ell}(r_{2,\boldsymbol{B}_{i-1,\ell}})$, $\widetilde{\boldsymbol{B}}_{i-1,\ell}(r_{3,\boldsymbol{B}_{i-1,\ell}})$ *into a single evaluation* $\widetilde{\boldsymbol{B}}_{i-1}(r_{\mathsf{in},\boldsymbol{B}_{i-1}})$ *and evaluations of form* $\widetilde{\mathbf{AUX}}_{i,\ell}(r_{1,\mathbf{AUX}_{i,\ell}})$, $\widetilde{\mathbf{AUX}}_{i,\ell}(r_{2,\mathbf{AUX}_{i,\ell}})$, *and* $\widetilde{\mathbf{AUX}}_{i,\ell}(r_{3,\mathbf{AUX}_{i,\ell}})$ *into a single evaluation* $\widetilde{\mathbf{AUX}}_{i}(r_{\mathsf{in},\mathbf{AUX}_{i}})$. *The final combined evaluations can be verified by invoking the evaluation opening algorithm of the commitment scheme* $\mathsf{PCS.Open}(\widetilde{\boldsymbol{W}}_i, r_{W_i}, r_{\mathsf{in},\mathbf{AUX}_i}, \mathsf{pp})$, $\mathsf{PCS.Open}(\widetilde{\boldsymbol{W}}_{i-1}, r_{W_{i-1}}, r_{\mathsf{in},\boldsymbol{W}_{i-1}}, \mathsf{pp})$, $\mathsf{PCS.Open}(\widetilde{\boldsymbol{B}}_{i-1}, r_{B_{i-1}}, r_{\mathsf{in},\boldsymbol{B}_{i-1}}, \mathsf{pp})$, *and* $\mathsf{PCS.Open}(\widetilde{\mathbf{AUX}}_i, r_{\mathbf{AUX}_i}, r_{\mathsf{in},\mathbf{AUX}_i}, \mathsf{pp})$ *or applying the aggregation in the recursive setting (cf. Section 6).*

Protocol 4. Polynomial Commitment Aggregation

Parameters: *Let* $\mathsf{PCS} = (\mathsf{KeyGen}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Verify})$ *be the Orion [60] polynomial commitment scheme with public parameters* pp. *Let* $f_1, f_2, \ldots, f_k$ *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. $\boldsymbol{M}_i$ *denotes coefficients of* f_i *parsed as a square matrix and* $\boldsymbol{C}_{i,1}, \boldsymbol{C}_{i,2}$ *denote the associated encoded matrices generated during the Orion commitment procedure (cf. Section 5.2).*

Phase 1: Evaluation Aggregation

(1) $\mathcal{P}$ *and* $\mathcal{V}$ *interpolate a polynomial* $L : \mathbb{F} \to \mathbb{F}^\ell$ *such that* $L(i) = x_i$.

(2) $\forall i = 1$ *to* k, $\mathcal{P}$ *samples random univariate polynomials* $h_{i,1} \ldots, h_{i,\ell}$*, each of degree* $d\ell$*, sets* $h_i(z_1, \ldots, z_\ell) \leftarrow h_{i,1}(z_1) + \cdots + h_{i,\ell}(z_\ell)$*, and sends* $v_i \leftarrow h_i(x_i)$ *and* $\sigma'_i \leftarrow \mathsf{PCS.Commit}(h_i, r'_i, \mathsf{pp})$. $\boldsymbol{M}'_i$ *denotes the coefficient matrix of* h_i *and* $\boldsymbol{C}'_{i,1}, \boldsymbol{C}'_{i,2}$ *denote the associated encoded matrices.*

(3) $\forall i = 1$ *to* k, $\mathcal{V}$ *responds with a random challenge* β_i.

(4) $\forall i = 1$ *to* k, $\mathcal{P}$ *sends the polynomial* $g_i \leftarrow (f_i + \beta_i h_i) \circ L$.

(5) $\forall i = 1$ *to* k, $\mathcal{V}$ *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) $\mathcal{P}$ *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) $\mathcal{P}$ *sends* $\sigma^* \leftarrow \mathsf{PCS.Commit}(f^*, r^*, \mathsf{pp})$. $\boldsymbol{M}^*$ *denotes the coefficient matrix of* f^* *and* $\boldsymbol{C}^*_1, \boldsymbol{C}^*_2$ *denote the associated encoded matrices.*

(2) $\mathcal{V}$ *randomly samples and sends ta subset of indices* I *of size* $t \in \Theta(\lambda)$.

(3) $\forall i = 1$ *to* k, $\forall \mathsf{idx} \in I$, $\mathcal{P}$ *sends* $s^* = \boldsymbol{C}^*_2[\mathsf{idx}]$, $s_i = \boldsymbol{C}_{i,2}[\mathsf{idx}]$*, and* $s'_i = \boldsymbol{C}'_{i,2}[\mathsf{idx}]$ *with their corresponding Merkle proofs.*

(4) $\forall i = 1$ *to* k, $\forall \mathsf{idx} \in I$, $\mathcal{V}$ *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.*

$\underline{\mathsf{Real}_{\mathcal{A},F}(1^\lambda, \mathsf{pp})}$
(1) $\forall i \in \{1, 2, \ldots, k\}$, $\sigma_i \leftarrow \mathsf{PCS.Commit}(f_i, r_i, \mathsf{pp})$
(2) Let $S = \{(\sigma_i, x_i, y_i)\}_{i=1}^k$ and $W = \{(f_i, r_i)\}_{i=1}^k$.
(3) $(\sigma^*, x^*, y^*), (f^*, r^*), \pi_0 \leftarrow \mathcal{P}(S, W, \mathsf{pp})$
(4) $(y^*, \pi_1) \leftarrow \mathsf{PCS.Open}(f^*, r^*, x^*, \mathsf{pp})$;
(5) $b \leftarrow \mathcal{A}(S, (\sigma^*, x^*, y^*), \pi_0, \pi_1, \mathsf{pp})$
(6) Return b

$\underline{\mathsf{Ideal}_{\mathcal{A},\mathcal{S}}(1^\lambda, \mathsf{pp})}$
(1) $S = \{(\sigma_i, x_i, y_i)\}_{i=1}^k, (\sigma^*, x^*, y^*), \pi_0, \pi_1 \leftarrow \mathcal{S}(1^\lambda, \mathsf{pp})$
(2) $b \leftarrow \mathcal{A}(S, (\sigma^*, x^*, y^*), \pi_0, \pi_1, \mathsf{pp})$
(3) Return b

In particular, for any $\mathcal{A}$, we have

$$|\Pr[\mathsf{Real}_{\mathcal{A},F}(1^\lambda, \mathsf{pp}) = 1] - \Pr[\mathsf{Ideal}_{\mathcal{A},\mathcal{S}}(1^\lambda, \mathsf{pp}) = 1]| \leq \mathsf{negl}(\lambda)$$

Proof of Theorem 2. Completeness is followed immediately. The soundness proof proceeds in two steps. First, we must prove the soundness with respect to commitment aggregation, i.e., if we can prove the well-formedness of σ^*, the verifier is convinced that each σ_i is well-formed, and the linear combination relation holds. Second, we should prove the soundness with respect to evaluation aggregation, i.e., if a prover can substantiate (σ^*, x^*, y^*) is satisfied, the verifier is convinced that $\forall i : (\sigma_i, x_i, y_i)$ is satisfied.

Consider the case of commitment aggregation. Let δ be the distance of the underlying code. First, suppose that the linear combination relation holds. Assume there exists some σ_i or σ'_i, which is not well-formed, i.e., there exists a cth column of $\boldsymbol{C}_{i,2}$ or $\boldsymbol{C}'_{i,2}$, which is not a valid codeword. Then, over the choice of random challenges α_i and β_i, the $\boldsymbol{c}$th column of $\boldsymbol{C}_2$ is not a valid codeword with overwhelming probability [32, Claim 1], which yields σ^* to be not well-formed. Similarly, if there exist an rth row of $\boldsymbol{C}_{i,1}$ or $\boldsymbol{C}'_{i,1}$, which is not code-word, it would cause the rth row of $\boldsymbol{C}_1$ to be an invalid code-word, yielding the verifier to return reject during the well-formedness check of σ^*. We should then show that the verifier returns reject if the linear combination does not hold. Suppose that this case occurs, and we have $\boldsymbol{M}^* = \boldsymbol{M} + \sum_i \alpha_i \boldsymbol{M}_i + \alpha_i \beta_i \boldsymbol{M}'_i$, where $\boldsymbol{M}$ is a non-zero matrix, i.e., there exist an index $(\mathsf{idx}_r, \mathsf{idx}_c)$ such that $\boldsymbol{M}[\mathsf{idx}_r, \mathsf{idx}_c] \neq 0$. If they apply the row-wise encoding, we have $\boldsymbol{C}^*_1 = \boldsymbol{C}_1 + \sum_i \alpha_i \boldsymbol{C}_{i,1} + \alpha_i \beta_i \boldsymbol{C}'_{i,1}$, where the row $\boldsymbol{C}_1[\mathsf{idx}_r, :]$ has at least δ non-zero entries. Then, if we apply the column-wise encoding, we have $\boldsymbol{C}^*_2 = \boldsymbol{C}_2 + \sum_i \alpha_i \boldsymbol{C}_{i,2} + \alpha_i \beta_i \boldsymbol{C}'_{i,2}$, where $\boldsymbol{C}_2$ includes δ^2 non-zero entries. In other words, the two-step application of the encoding scheme propagates the inconsistency through the matrix. Let $\gamma = \frac{\delta}{p}$, be the relative distance of the code, where p is the codeword size. By opening $\Theta(\lambda)$ entries, all opened values are zero with a negligible probability of at most $(1 - \gamma^2)^{\Theta(\lambda)}$.

Then, for the case of evaluation aggregation, suppose that we have $\exists i : f_i(x_i) \neq y_i$. If the prover sends a correct $g_i = (f_i + \beta_i h_i) \circ L$, then with overwhelming probability over the choice of the β_i, we should have $y_i + \beta_i v_i \neq g_i(i)$, and the verifier aborts. Therefore, the prover must send some $g_i \neq (f_i + \beta_i h_i) \circ L$ to pass the verification. However, with overwhelming probability over the choice of the α_i,

$$g = \sum_i \alpha_i g_i \neq \sum_i \alpha_i (f_i + \beta_i h_i) \circ L = f \circ L$$

where $f = \sum_{i=1}^k \alpha_i f_i + \alpha_i \beta_i h_i$. The Schwartz-Zippel lemma then implies $(f \circ L)(r) \neq g(r)$ except with negligible probability.

Knowledge soundness follows immediately from the above. Our aggregation scheme requires the prover to give the verifier oracle access to matrices C_2^*, $C_{i,2}$, and $C'_{i,2}$ by their Merkle root. It is well known that for such Merkle-based oracle accesses, one can build a straight-line extractor to extract the witness committed to by its Merkle root [9, 48]. Once matrices are extracted, we can recover coefficients by running the decoding procedure first to columns and then to rows of each matrix.

Zero knowledge holds since each polynomial $h_{i,j}$ is uniformly sampled from the set of polynomials in $\mathbb{F}[X]$ of degree $d\ell$. The simulator $\mathcal{S}$ first samples random polynomials $\{h^*_{i,j}\}^{k,\ell}_{i=1,j=1}$, and sets $h^*_i(z_1,\ldots,z_\ell) = h^*_{i,1}(z_1) + \cdots + h^*_{i,\ell}(z_\ell)$. Then, $\mathcal{S}$ commits to each h^*_i and the sends σ'^*_i and the evaluation $v^*_i = h^*_i(x)$ to $\mathcal{A}$. As $h^*_{i,j}$ in the ideal world and $h_{i,j}$ in the real world both are sampled uniformly from the same distribution, they are indistinguishable, and hence, polynomials h^*_i and h_i and the evaluations v_i and v^*_i are indistinguishable. Moreover, as PCS is zero-knowledge σ'_i and σ'^*_i are indistinguishable. $\mathcal{S}$ then samples input polynomials $\{f^*_i\}^k_{i=1}$ such that $f^*_i(x_i) = y_i$ and sends σ^*_i, a commitment to f^*_i to the verifier, which is indistinguishable from σ_i. $\mathcal{S}$ also sends polynomials $g^*_i = (f^*_i + \beta_i h^*_i) \circ L$. We note that each $f_i \circ L$ is a univariate polynomial from the set $\mathbb{F}[X]$ of degree at most $dk\ell$, and $h_i \circ L$ is also a univariate polynomial over the field with the same degree. Therefore, each coefficient of the polynomial $f_i \circ L$, and hence g_i, is masked by a coefficient of $h_i \circ L$. Then, as masks h^*_i and h_i are indistinguishable, thus g^*_i and g_i are also indistinguishable.

D RECURSIVE SUMCHECK FRAMEWORK

In this section, we present our recursive sumcheck framework in Protocol 5 and sketch its security proof.

Proof of Theorem 3. For completeness, consider a satisfying tuple $(i-1, z_0, z_{i-1}, \pi_{i-1})$, i.e., we have $\mathcal{V}(i-1, z_0, z_{i-1}, \pi_{i-1}, \mathsf{pp}) = 1$. Given ω_{i-1}, we must show that the the updated instance (i, z_0, z_i, π_i), returned by the prover is also satisfying. $\mathcal{P}$ executes the ith iteration and generates $\pi_i = (\pi_{i,G}, \pi_{i,A}, \pi_{i,E}, \mathsf{agg}_i, \mathsf{wp}_i, \mathsf{agg}_{i+1}, \mathsf{wp}_{i+1})$. First, as $\mathcal{V}(i-1, z_0, z_{i-1}, \pi_{i-1}, \mathsf{pp}) = 1$, the verification steps, and in turn, the entire computation of $\mathcal{F}_A$ is completed successfully. $\pi_{i,G}$ is generated by GKR.$\mathcal{P}$ and the sumcheck protocol, and $\pi_{i,A}$ is generated by AGG.$\mathcal{P}$. If the iteration is a final iteration $\pi_{i,E}$ is generated by PCS.$\mathcal{P}$ as an opening to agg_{i+1}, otherwise it is returned by AGG.$\mathcal{P}$ as a witness to agg_{i+1}. Moreover, wp_{i+1} is the combined predicate evaluation by applying evaluation reduction sumcheck. Following the completeness of the GKR scheme, the sumcheck protocol, AGG proof messages $\pi_{i,G}$ and $\pi_{i,A}$ are satisfying. The completeness of AGG or PCS ensures that $\pi_{i,E}$ is also satisfying. The correctness of wp_{i+1} follows the completeness of the sumcheck protocol. Therefore, the ith proof π_i is satisfying.

For knowledge soundness, let ξ_{AGG} be the extractor of AGG. The adversary $\mathcal{A}$ returns a satisfying tuple (i, z_0, z_i, π_i), i.e., we have $\mathcal{V}(i, z_0, z_i, \pi_i, \mathsf{pp}) = 1$. We must construct an efficient extractor ξ that can extract witnesses that correctly satisfy $\mathcal{F}$ for the entire computation. We show inductively that for all $j = i-1$ to 1, ξ can construct $\xi^{(j)}$ that outputs z_j and ω_j. For the base case, i.e., $j = i-1$, recall that $\pi_i = (\pi_{i,G}, \pi_{i,A}, \pi_{i,E}, \mathsf{agg}_i, \mathsf{wp}_i, \mathsf{agg}_{i+1}, \mathsf{wp}_{i+1})$. As $\pi_{i,A}$ is a valid proof for agg_{i+1}, by running ξ_{AGG}, $\xi^{(i-1)}$ can extract the vector $\boldsymbol{u}_i = i||z_0||z_{i-1}||\omega_{i-1}||\pi_{i-1}$, which is committed to by σ_i. Therefore, z_{i-1} and ω_{i-1} are extracted. Soundness of GKR, PCS, AGG, and the protocol sumcheck implies that the extracted inputs $\boldsymbol{u}_i$, so z_{i-1} and ω_{i-1} are satisfying and $z_i = \mathcal{F}(z_{i-1}, \omega_{i-1})$. For the case $j < i-1$, $\xi^{(j)}$ runs $\xi^{(j+1)}$ to extract $\boldsymbol{u}_{j+1} = j+1||z_0||z_j||\omega_j||\pi_j$. As $\boldsymbol{u}_{j+1}$ is satisfying, proof messages in π_j, including $\pi_{j,A}$ are valid. By running ξ_{AGG}, $\xi^{(j)}$ can then extract $\boldsymbol{u}_j$ and get z_{j-1} and ω_{j-1}. This completes the recursive construction of ξ from ξ_{AGG}.

We prove zero-knowledge with respect to the *final* verifier. In particular, we construct a simulator $\mathcal{S}$ that can generate proof messages commitments indistinguishable from the real proofs and commitments. In particular, $\mathcal{S}$ samples a random initial input z_0 and auxiliary inputs $\{\omega_j\}^{i-1}_{j=0}$. With access to the $\mathcal{F}$, $\mathcal{S}$ executes the IVC prover algorithm i iterations. At the end, $\mathcal{S}$ returns the final tuple (i, z_0, z_i, π_i). If the instantiations of GKR, PCS, AGG, and the sumcheck protocol satisfy zero-knowledge, proof messages and commitments included in π_i reveals no additional information about the private auxiliary inputs $\{\omega_j\}^{i-1}_{j=0}$.

E PROOF OF TRAINING

In this section, we first define zero-knowledge proof-of-training. We then present the Kaizen construction in Protocol 6.

E.1 Definition

A zero-knowledge proof-of-training must satisfy completeness, knowledge soundness, and zero-knowledge defined formally as below. Let C be the training circuit defined in Section 7.2.

Definition 4. (Zero-Knowledge Proof-of-Training) A tuple of (KeyGen, DataCom, WeiCom, BatchOpen, Prove, Verify) is a zkPoT if the following hold. Let $\mathsf{pp} \leftarrow \mathsf{KeyGen}(1^\lambda)$.

- ***Completeness.*** Given a dataset $\mathcal{D}$, $\rho_{\mathcal{D}} \leftarrow \mathsf{DataCom}(\mathcal{D}, \mathsf{pp}, r_{\rho_{\mathcal{D}}})$, $\boldsymbol{W}_0$ and $\boldsymbol{W}_{i-1}$ with commitments $\sigma_{W_0} \leftarrow \mathsf{WeiCom}(\boldsymbol{W}_0, r_{\sigma_{W_0}}, \mathsf{pp})$ and $\sigma_{W_{i-1}} \leftarrow \mathsf{WeiCom}(\boldsymbol{W}_{i-1}, r_{\sigma_{W_{i-1}}}, \mathsf{pp})$, respectively, the and opening $\boldsymbol{B}_{i-1}, p_{\boldsymbol{B}_{i-1}} \leftarrow \mathsf{BatchOpen}(\mathcal{D}, r_{\rho_{\mathcal{D}}}, i, \mathsf{pp})$, and a satisfying proof π_{i-1} the following equals to 1. Let $t_i = (i, \rho_{\mathcal{D}}, \sigma_{W_0}, \sigma_{W_{i-1}})$.

$$\Pr\left[b = 1 \;\middle|\; \begin{array}{c} \boldsymbol{W}_i, \pi_i \leftarrow \mathsf{Prove}(t_i, \boldsymbol{W}_{i-1}, \boldsymbol{B}_{i-1}, p_{\boldsymbol{B}_{i-1}}, \pi_{i-1}, \mathsf{pp}); \\ \boldsymbol{W}_i \leftarrow C(\boldsymbol{W}_{i-1}, \boldsymbol{B}_{i-1}, (i, \rho_{\mathcal{D}}, \boldsymbol{p}_{\boldsymbol{B}_{i-1}})); \\ \sigma_{W_i} \leftarrow \mathsf{WeiCom}(\boldsymbol{W}_i, r_{\sigma_{W_i}}, \mathsf{pp}); \\ b \leftarrow \mathsf{Verify}(i, \rho_{\mathcal{D}}, \sigma_{W_0}, \sigma_{W_i}, \pi_i, \mathsf{pp}); \end{array}\right]$$

- ***Knowledge Soundness.*** For any adversary $\mathcal{A}$, there exists an extractor $\xi^{\mathcal{A}}$ such that the following holds.

$$\Pr\left[\begin{array}{c} \{\boldsymbol{B}_k, p_{\boldsymbol{B}_k}, \boldsymbol{W}_k\}^i_{k=0} \leftarrow \xi^{\mathcal{A}}(\mathsf{pp}) : \\ \forall k \in \{1,\ldots,k\} : \boldsymbol{W}_k = C(\boldsymbol{W}_{k-1}, \boldsymbol{B}_{k-1}, k, \rho_{\mathcal{D}}, \boldsymbol{p}_{\boldsymbol{B}_{k-1}}); \\ \sigma_{W_i} = \mathsf{WeiCom}(\boldsymbol{W}_i, r_{\sigma_{W_i}}, \mathsf{pp}); \sigma_{W_0} = \mathsf{WeiCom}(\boldsymbol{W}_0, r_{\sigma_{W_0}}, \mathsf{pp}); \end{array}\right]$$
$$\geq \Pr\left[\begin{array}{c} i, \rho_{\mathcal{D}}, \sigma_{W_0}, \sigma_{W_i}, \pi_i \leftarrow \mathcal{A}(\mathsf{pp}) : \\ \mathsf{Verify}(i, \rho_{\mathcal{D}}, \sigma_{W_0}, \sigma_{W_i}, \pi_i, \mathsf{pp}) = 1 \end{array}\right] - \mathsf{negl}(\lambda)$$

- ***Zero knowledge.*** There exists a simulator $\mathcal{S}$, having oracle access to C, such that for every adversary $\mathcal{A}$, iteration counter i, dataset $\mathcal{D}$ with a commitment $\rho_{\mathcal{D}}$, initial weights $\boldsymbol{W}_0$, batches of data points and openings $\{(\boldsymbol{B}_k, p_k)\}^i_{k=0}$ satisfying the circuit C, the two following experiments Real, Ideal are indistinguishable.

Protocol 5. Recursive Sumcheck Framework

Parameters: *Let* PCS = (KeyGen, Commit, Open, Verify) *be the Orion [60] commitment with public parameters* pp, AGG = (Prove, Verify) *the non-interactive aggregation scheme (cf. Protocol 4), and* GKR = (Prove, Verify) *be a generic GKR-style non-interactive zero-knowledge proof. Hashes are instantiated with MiMC-p/p [4]. Let $\mathcal{F}$ be the iteration function, z_0 an initial input, π_0 a trivially satisfying proof, and ω_{i-1} the ith auxiliary input.*

$\underline{z_i, \pi_i \leftarrow \mathcal{P}(i, z_0, z_{i-1}, \omega_{i-1}, \pi_{i-1}, \mathsf{pp}) :}$

(1) Parse π_{i-1} as $(\pi_{i-1,G}, \pi_{i-1,A}, \pi_{i-1,E}, \mathsf{agg}_{i-1}, \mathsf{wp}_{i-1}, \mathsf{agg}_i, \mathsf{wp}_i)$, where $\pi_{i-1,G}$ is sumcheck and $\pi_{i-1,A}$ is aggregation messages, agg_{i-1} and agg_i are the last and updated input aggregation instances, wp_{i-1} and wp_i are the last and updated wiring predicate aggregations, and $\pi_{i-1,E}$ is a witness to agg_i.

(2) Compute the augmented iteration function $i, z_0, z_i, \mathsf{agg}_i, \mathsf{wp}_i \leftarrow \mathcal{F}_A(i, z_0, z_{i-1}, \omega_{i-1}, \pi_{i-1} \setminus \{\pi_{i-1,E}\})$ as follows:

(2.1) Evaluate Fiat-Shamir and Merkle hashes required for verification of sumcheck proof messages $\pi_{i-1,G}$ and aggregation proof messages $\pi_{i-1,A}$.

(2.2) On input $i-1, z_0, z_{i-1}, \mathsf{agg}_{i-1}, \mathsf{wp}_{i-1}$, run GKR.$\mathcal{V}$ *and the sumcheck protocol verifier to verify sumcheck messages $\pi_{i-1,G}$. The verification must end with an input commitment/evaluation instance $(\sigma_{i-1}, x_{i-1}, y_{i-1})$ and the updated predicate aggregation wp_i.*

(2.3) On input $\mathsf{agg}_{i-1}, \mathsf{agg}_i, (\sigma_{i-1}, x_{i-1}, y_{i-1})$, run AGG.$\mathcal{V}$ *to verify aggregation messages $\pi_{i-1,A}$.*

(2.4) If verification steps (2.1)-(2.3) passes, compute $z_i \leftarrow \mathcal{F}(z_{i-1}, \omega_{i-1})$, and return $i, z_0, z_i, \mathsf{agg}_i, \mathsf{wp}_i$.

(3) Evaluate a commitment σ_i to the inputs of the running $\mathcal{F}_A$, i.e., given $\boldsymbol{u_i} = i||z_0||z_{i-1}||\omega_{i-1}||\pi_{i-1} \setminus \{\pi_{i-1,E}\}$, we have $\sigma_i \leftarrow$ PCS.Commit$(\widetilde{u}_i, r_i, \mathsf{pp})$.

(4) Generate sumcheck messages for the running $\mathcal{F}_A$ execution by applying hash sumcheck, i.e., sumchecks on Equation (6)*, to hash evaluations and* GKR.$\mathcal{P}$ *to other components. Combine wiring predicate evaluations into single one wp_{i+1} and also the evaluations of $\widetilde{u}_i$ into (x_i, y_i) by applying evaluation reduction sumchecks, i.e., running the sumcheck protocol on Equation* (5)*. Include sumcheck messages as well as wp_{i+1} and (σ_i, x_i, y_i) in $\pi_{i,G}$.*

(5) On input instances agg_i, (σ_i, x_i, y_i) and witnesses $\pi_{i-1,E}$, $(\widetilde{u}_i, r_i)$, run AGG.$\mathcal{P}$*, which ends with agg_{i+1}. Include aggregation messages in $\pi_{i,A}$.*

(6) If the running iteration is the final iteration, run PCS.Open *and include evaluation opening proofs otherwise the witnesses for agg_{i+1} in $\pi_{i,E}$.*

(7) Let $\pi_i \leftarrow (\pi_{i,G}, \pi_{i,A}, \pi_{i,E}, \mathsf{agg}_i, \mathsf{wp}_i, \mathsf{agg}_{i+1}, \mathsf{wp}_{i+1})$, and return z_i, π_i.

$\underline{\{0,1\} \leftarrow \mathcal{V}(i, z_0, z_i, \pi_i, \mathsf{pp}) :}$

(1) Parse π_i as $(\pi_{i,G}, \pi_{i,A}, \pi_{i,E}, \mathsf{agg}_i, \mathsf{wp}_i, \mathsf{agg}_{i+1}, \mathsf{wp}_{i+1})$.

(2) Evaluate Fiat-Shamir and Merkle hashes required for verification of sumcheck proof messages $\pi_{i,G}$ and aggregation proof messages $\pi_{i,A}$.

(3) On input $i, z_0, z_i, \mathsf{agg}_i, \mathsf{wp}_i$ run GKR.$\mathcal{V}$ *and the sumcheck protocol verifier on messages $\pi_{i,G}$, which ends with (σ_i, x_i, y_i) and wp_{i+1}. Having access to wiring predicates, verify wp_{i+1} directly. On input $\mathsf{agg}_{i+1}, \mathsf{agg}_i, (\sigma_i, x_i, y_i)$ run* AGG.$\mathcal{V}$ *to verify messages $\pi_{i,A}$. If it is the final iteration, run* PCS.Verify$(\mathsf{agg}_{i+1}, \pi_{i,E}, \mathsf{pp})$*. Otherwise, check whether $\pi_{i,E}$ is a valid opening witness to agg_{i+1}.*

Let INP be a set, including the dataset, initial weights, batches with their satisfying openings generated by BatchOpen.

$\underline{\mathsf{Real}_{\mathcal{A},\mathsf{INP}}(i, \mathsf{pp})}$
(1) $\rho_{\mathcal{D}} \leftarrow \mathsf{DataCom}(\mathcal{D}, \mathsf{pp}, \rho_{\mathcal{D}})$
(2) $\sigma_{W_0} \leftarrow \mathsf{WeiCom}(\boldsymbol{W}_0, r_{\sigma_{W_0}}, \mathsf{pp})$
(3) $\forall k$, given $t_k = (k, \rho_{\mathcal{D}}, \sigma_{W_0}, \sigma_{W_{k-1}})$:
  (3.1) $\boldsymbol{W}_k, \pi_k \leftarrow \mathsf{Prove}(t_k, \boldsymbol{W}_{k-1}, \boldsymbol{B}_{k-1}, p_{\boldsymbol{B}_{k-1}}, \pi_{k-1}, \mathsf{pp})$
  (3.2) $\sigma_{W_k} \leftarrow \mathsf{WeiCom}(\boldsymbol{W}_k, r_{\sigma_{W_k}}, \mathsf{pp})$
(4) $b \leftarrow \mathcal{A}(i, \rho_{\mathcal{D}}, \sigma_{W_0}, \sigma_{W_i}, \pi_i, \mathsf{pp})$
(5) Return b

$\underline{\mathsf{Ideal}_{\mathcal{A},\mathcal{S}}(i, \mathsf{pp})}$
(1) $\rho_{\mathcal{D}}, \sigma_{W_0}, \sigma_{W_i}, \pi_i \leftarrow \mathcal{S}(\mathsf{pp})$, given oracle access to C
(2) $b \leftarrow \mathcal{A}(i, \rho_{\mathcal{D}}, \sigma_{W_0}, \sigma_{W_i}, \pi_i, \mathsf{pp})$
(3) Return b

In particular, for any instance t_i and adversary $\mathcal{A}$ we have

$$|\Pr[\mathsf{Real}_{\mathcal{A},\mathsf{INP}}(i, \mathsf{pp}) = 1] - \Pr[\mathsf{Ideal}_{\mathcal{A},\mathcal{S}}(i, \mathsf{pp}) = 1]| \leq \mathsf{negl}(\lambda)$$

E.2 Construction

We present the Kaizen construction in Protocol 6. As Kaizen is basically a concrete instantiation of Protocol 5, we extend the recursion protocol to describe the prover and verifier of Kaizen. Also, we sketch the security proof, which follows the same approach as the security proof, presented in Appendix D.

Proof of Theorem 4. For completeness, consider a satisfying instance/proof such that $\mathsf{Verify}(i-1, \rho_{\mathcal{D}}, \sigma_{W_0}, \sigma_{W_{i-1}}, \pi_{i-1}, \mathsf{pp}) = 1$. Given weights $\boldsymbol{W}_{i-1}$ and batch $\boldsymbol{B}_{i-1} \subseteq \mathcal{D}$ with opening $p_{\boldsymbol{B}_{i-1}}$, we show that the updated instance, $(i, \rho_{\mathcal{D}}, \sigma_{W_0}, \sigma_{W_i})$ and proof π_i should be satisfying. $\mathcal{P}$ executes the ith iteration and generates $\pi_i = (\pi_{i,G}, \pi_{i,A}, \pi_{i,E}, \mathsf{agg}_i, \mathsf{wp}_i, \mathsf{agg}_{i+1}, \mathsf{wp}_{i+1})$. Note that as we have $\mathsf{Verify}(i-1, \rho_{\mathcal{D}}, \sigma_{W_0}, \sigma_{W_{i-1}}, \pi_{i-1}, \mathsf{pp}) = 1$, the verification steps of $\mathcal{F}_A$ are completed successfully. Furthermore, the completeness of the Merkle tree ensures that the computation C is also completed successfully. Then, $\pi_{i,G}$ is generated by GKR.$\mathcal{P}$, PoGD.$\mathcal{P}$, and the sumcheck protocol, and $\pi_{i,A}$ is generated by AGG.$\mathcal{P}$. If the iteration is a final iteration $\pi_{i,E}$ is generated by PCS.$\mathcal{P}$, otherwise, by AGG.$\mathcal{P}$ as an opening or witness to agg_{i+1}. Moreover, wp_{i+1} is the combined predicate evaluation by applying evaluation reduction sumcheck. Following the completeness of GKR, PoGD, the sumcheck protocol, and AGG, the messages included in $\pi_{i,G}$ and $\pi_{i,A}$ are satisfying. The completeness of AGG or PCS in the case of the final iteration, ensures that $\pi_{i,E}$ is satisfying. Moreover, the correctness of wp_{i+1} follows the completeness of the sumcheck protocol.

For knowledge soundness, let ξ_{AGG} be the extractor of AGG. The adversary $\mathcal{A}$ returns a satisfying tuple $(i, \rho_{\mathcal{D}}, \sigma_{W_0}, \sigma_{W_i}, \pi_i)$, i.e., we have $\mathsf{Verify}(i, \rho_{\mathcal{D}}, \sigma_{W_0}, \sigma_{W_i}, \pi_i, \mathsf{pp}) = 1$. We must construct

Protocol 6. KAIZEN **Zero-Knowledge Proof-of-Training**

Parameters: *Let* PCS = (KeyGen, Commit, Open, Verify) *be the Orion [60] commitment with public parameters* pp, AGG = (Prove, Verify) *the non-interactive aggregation scheme (cf. Protocol 4),* PoGD = (Prove, Verify) *the non-interactive zero-knowledge proof of gradient descent (cf. Protocol 3), and* GKR *be a non-interactive zero-knowledge generic GKR-style sumcheck-based proof system [62]. Let* $\mathcal{D}$ *be the dataset,* $\boldsymbol{W}_0$ *be an initial weights,* C *be the training iteration circuit as defined in Section 7.2. Let* π_0 *be a trivially satisfying proof.*

Basic procedures: KeyGen *samples public parameters of the commitment scheme by running* PCS.KeyGen, DataCom *generates a commitment* $\rho_{\mathcal{D}}$ *to the dataset* $\mathcal{D}$ *by returning its Merkle root.* WeiCom *commits the multilinear extension of weights by apply* PCS.Commit, *and* BatchOpen *generates an opening proof for a batch* $\boldsymbol{B}$*, consistent with the permutation hard-coded in* C*, by returning Merkle paths of each point included in the batch.*

$\boldsymbol{W}_i, \pi_i \leftarrow \mathsf{Prove}(i, \rho_{\mathcal{D}}, \sigma_{W_0}, \sigma_{W_{i-1}}, \boldsymbol{W}_{i-1}, \boldsymbol{B}_{i-1}, p_{\boldsymbol{B}_{i-1}}, \pi_{i-1}, \mathsf{pp})$:

(1) *Parse* π_{i-1} *as* $(\pi_{i-1,G}, \pi_{i-1,A}, \pi_{i-1,E}, \mathsf{agg}_{i-1}, \mathsf{wp}_{i-1}, \mathsf{agg}_i, \mathsf{wp}_i)$*, where* $\pi_{i-1,G}$ *is sumcheck and* $\pi_{i-1,A}$ *is aggregation messages,* agg_{i-1} *and* agg_i *are the last and updated input aggregation instances,* wp_{i-1} *and* wp_i *are the last and updated wiring predicate aggregations, and* $\pi_{i-1,E}$ *is a witness to* agg_i*.*

(2) *Compute the augmented iteration function* $i, \rho_{\mathcal{D}}, \sigma_{W_0}, \boldsymbol{W}_i, \mathsf{agg}_i, \mathsf{wp}_i \leftarrow \mathcal{F}_A(i, \rho_{\mathcal{D}}, \sigma_{W_0}, \sigma_{W_{i-1}}, \boldsymbol{W}_{i-1}, \boldsymbol{B}_{i-1}, p_{\boldsymbol{B}_{i-1}}, \pi_{i-1} \setminus \{\pi_{i-1,E}\})$ *as follows.*

   (a) *Evaluate Fiat-Shamir and Merkle hashes required for verification of sumcheck proof messages* $\pi_{i-1,G}$ *and aggregation proof messages* $\pi_{i-1,A}$*.*

   (b) *On input* $i-1, \rho_{\mathcal{D}}, \sigma_{W_0}, \sigma_{W_{i-1}}, \mathsf{agg}_{i-1}, \mathsf{wp}_{i-1}$*, verify sumcheck messages* $\pi_{i-1,G}$ *by running* GKR.$\mathcal{V}$*,* PoGD.$\mathcal{V}$*, and the sumcheck verifier. The verification ends with several commitment-evaluation instances of form* $\{(\sigma_{i-1,j}, x_{i-1,j}, y_{i-1,j})\}_j$ *and the updated predicate aggregation* wp_i*, where each* $\sigma_{i-1,j}$ *is a commitment to either* $\widetilde{\boldsymbol{W}}_{i-2}$*,* $\widetilde{\boldsymbol{B}}_{i-2}$*, gradient descent auxiliary inputs* $\widetilde{\mathbf{AUX}}_{i-1}$*, or other inputs, which we denote them by* $u_{i-1} = i-1 || \rho_{\mathcal{D}} || \sigma_{W_0} || \sigma_{W_{i-2}} || p_{\boldsymbol{B}_{i-2}} || \pi_{i-2}$*. Moreover,* $\sigma_{i-1,j}$ *can be a commitment to* $\widetilde{\boldsymbol{W}}_{i-1}$*, which should be the same as* $\sigma_{W_{i-1}}$*.*

   (c) *On input* $\mathsf{agg}_{i-1}, \{(\sigma_{i-1,j}, x_{i-1,j}, y_{i-1,j})\}_j$*, verify aggregation messages* $\pi_{i-1,A}$ *by running* AGG.$\mathcal{V}$*. The verification ends with the updated aggregation instances* agg_i*; note that each type of polynomials, i.e., either* $\widetilde{\boldsymbol{W}}$*,* $\widetilde{\boldsymbol{B}}$*,* $\widetilde{\mathbf{AUX}}$*, or* $\widetilde{u}$*, are aggregated independently.*

   (d) *Compute* $\boldsymbol{W}_i = C(\boldsymbol{W}_{i-1}, \boldsymbol{B}_{i-1}, (i, \rho_{\mathcal{D}}, \boldsymbol{p}_{\boldsymbol{B}_{i-1}}))$*, and return* $i, \rho_{\mathcal{D}}, \sigma_{W_0}, \boldsymbol{W}_i, \mathsf{agg}_i, \mathsf{wp}_i$*.*

(3) *Generate commitments* $\sigma_{i,0} \leftarrow \mathsf{PCS.Commit}(\widetilde{\boldsymbol{W}}_i, r_{\sigma_{W_i}}, \mathsf{pp})$*,* $\sigma_{i,1} \leftarrow \mathsf{PCS.Commit}(\widetilde{\boldsymbol{B}}_{i-1}, r_{\sigma_{B_{i-1}}}, \mathsf{pp})$*,* $\sigma_{i,2} \leftarrow \mathsf{PCS.Commit}(\widetilde{\mathbf{AUX}}_i, r_{\sigma_{\mathbf{AUX}_i}}, \mathsf{pp})$*, where* $\mathbf{AUX}_i$ *is the auxiliary inputs of the running gradient descent iteration, and* $\sigma_{i,3} \leftarrow \mathsf{PCS.Commit}(\widetilde{\boldsymbol{u}}_i, r_{\sigma_{\boldsymbol{u}_i}}, \mathsf{pp})$*. Let* $\sigma_{i,4} = \sigma_{W_{i-1}}$*.*

(4) *Generate sumcheck messages for the running* $\mathcal{F}_A$ *execution by applying hash sumcheck, i.e., sumchecks on Equation* (6)*, to hash evaluations,* PoGD.$\mathcal{P}$ *to the gradient descent computation, and* GKR.$\mathcal{P}$ *to any other components. Moreover, combine received wiring predicate evaluations into* wp_{i+1} *by applying evaluation reduction sumcheck, i.e., sumcheck on Equation* (5)*. Similarly, combine all evaluations of* $\widetilde{u}_i$*. Include sumcheck messages, commitments generated in step (3), and evaluations required for sumcheck messages in* $\pi_{i,G}$*.*

(5) *On input* $\mathsf{agg}_i, \{(\sigma_i, x_i, y_i)\}_j$ *with witnesses* $\widetilde{\boldsymbol{W}}_i$*,* $\widetilde{\boldsymbol{W}}_{i-1}$*,* $\widetilde{\boldsymbol{B}}_{i-1}$*,* $\widetilde{\mathbf{AUX}}_i$*, or* $\widetilde{\boldsymbol{u}}_i$*, and* $\pi_{i-1,E}$*, run* AGG.$\mathcal{P}$ *to get* agg_{i+1}*. Include messages in* $\pi_{i,A}$*.*

(6) *If the running iteration is the final iteration, run* PCS.Open *and include evaluation opening proofs otherwise the witnesses for* agg_{i+1} *in* $\pi_{i,E}$*.*

(7) *Return* $\boldsymbol{W}_i, \pi_i \leftarrow (\pi_{i-1,G}, \pi_{i-1,A}, \pi_{i-1,E}, \mathsf{agg}_{i-1}, \mathsf{wp}_{i-1}, \mathsf{agg}_i, \mathsf{wp}_i)$*.*

$\{0,1\} \leftarrow \mathsf{Verify}(i, \rho_{\mathcal{D}}, \sigma_{W_0}, \sigma_{W_i}, \pi_i, \mathsf{pp})$:

(1) *Parse* π_i *as* $(\pi_{i,G}, \pi_{i,A}, \pi_{i,E}, \mathsf{agg}_i, \mathsf{wp}_i, \mathsf{agg}_{i+1}, \mathsf{wp}_{i+1})$*.*

(2) *Evaluate Fiat-Shamir and Merkle hashes required for verification of sumcheck proof messages* $\pi_{i,G}$ *and aggregation proof messages* $\pi_{i,A}$*.*

(3) *On input* $i, \rho_{\mathcal{D}}, \sigma_{W_0}, \sigma_{W_i}$*, run* GKR.$\mathcal{V}$*,* PoGD.$\mathcal{V}$*, and sumcheck verifier to verify sumcheck messages* $\pi_{i,G}$*, yielding* $\{(\sigma_{i-1,j}, x_{i-1,j}, y_{i-1,j})\}_j$ *and* wp_{i+1} *at the end. Having access to wiring predicates, verify* wp_{i+1} *directly. Moreover, run* AGG.$\mathcal{V}$ *on input* $\mathsf{agg}_i, \{(\sigma_{i-1,j}, x_{i-1,j}, y_{i-1,j})\}_j$ *to verify aggregation messages* $\pi_{i,A}$*, yielding* agg_{i+1} *at the end. If the* i*th iteration is the final iteration, run* PCS.Verify$(\mathsf{agg}_{i+1}, \pi_{i,E})$*. Otherwise, check whether* $\pi_{i,E}$ *is a valid opening witness to* agg_{i+1}*.*

an extractor ξ that can extract witnesses that satisfy C iterations. Inductively, for $j = i-1$ to 1, ξ runs $\xi^{(j)}$ outputting the jth witness. Consider $j = i-1$, s.t. $\pi_i = (\pi_{i,G}, \pi_{i,A}, \pi_{i,E}, \mathsf{agg}_i, \mathsf{wp}_i, \mathsf{agg}_{i+1}, \mathsf{wp}_{i+1})$. As $\pi_{i,E}$ is a valid opening to agg_{i+1}, by running ξ_{AGG}, $\xi^{(i-1)}$ can extract weights $\boldsymbol{W}_i, \boldsymbol{W}_{i-1}$, batch $\boldsymbol{B}_{i-1}$, and $p_{\boldsymbol{B}_{i-1}}$ such that the iteration circuit satisfies $\boldsymbol{W}_i = C(\boldsymbol{W}_{i-1}, \boldsymbol{B}_{i-1}, (j, \rho, \boldsymbol{p}_{\boldsymbol{B}_{i-1}}))$ and also we have $\sigma_{W_i} = \mathsf{WeiCom}(\boldsymbol{W}_0, r_{\sigma_{W_i}}, \mathsf{pp})$. This Follows the soundness of GKR, PCS, AGG, and the sumcheck protocol as well as binding of the Merkle tree. Moreover, $\xi^{(i-1)}$ can extract the inputs of the iteration, i.e., $\boldsymbol{u}_i = i || \rho_{\mathcal{D}} || \sigma_{W_0} || \sigma_{W_{i-1}} || p_{\boldsymbol{B}_{i-1}} || \pi_{i-1}$. For the case $j < i-1$, $\xi^{(j)}$ runs $\xi^{(j+1)}$ to extract $\boldsymbol{u}_{j+1} = j+1$, including proof messages in π_j, so in turn, including $\pi_{j,E}$. By running ξ_{AGG}, we can then extract $\boldsymbol{W}_i, \boldsymbol{W}_{i-1}$, batch $\boldsymbol{B}_{i-1}$, $p_{\boldsymbol{B}_{i-1}}$, and $\boldsymbol{u}_j$ as before. Finally, by running $\xi^{(1)}$, which in turn runs previous extractors $\xi^{(2)}, \ldots, \xi^{(i-1)}$, ξ can output all witnesses for the entire computation, i.e., intermediate model weights and all data points used in training iterations.

We prove zero knowledge with respect to the *final* verifier. $\mathcal{S}$ samples a random dataset $\mathcal{D}^*$ and commits to it by a Merkle root. As Merkle commitments are hiding, the root returned by $\mathcal{S}$ is indistinguishable from ρ. Also, $\mathcal{S}$ samples initial weights $\boldsymbol{W}_0^*$. With access to the C, $\mathcal{S}$ executes the prover algorithm on weights and satisfying batches for i iterations. At the end $\mathcal{S}$ returns the final proof messages and commitments to $\sigma_{W_0^*}$ and $\sigma_{W_i^*}$. As PCS, AGG are zero-knowledge, commitments $\sigma_{W_0^*}$, $\sigma_{W_i^*}$, and other commitments included in the proof, e.g., aggregated commitment, with their openings are indistinguishable from real commitments. Moreover, as AGG, PoGD, GKR, and the sumcheck protocol are zero-knowledge, π_i^* is also indistinguishable for a real proof.