ZENO: A Type-based Optimization Framework for Zero Knowledge Neural Network Inference

Boyuan Feng, Zheng Wang, Yuke Wang, Shu Yang, Yufei Ding
{boyuan,zheng_wang,yuke_wang,shuyang1995,yufeiding}@ucsb.edu
University of California, Santa Barbara
USA

Abstract

Zero knowledge Neural Networks draw increasing attention for guaranteeing computation integrity and privacy of neural networks (NNs) based on zero-knowledge Succinct Non-interactive ARgument of Knowledge (zkSNARK) security scheme. However, the performance of zkSNARK NNs is far from optimal due to the million-scale circuit computation with heavy scalar-level dependency. In this paper, we propose a type-based optimizing framework for efficient zero-knowledge NN inference, namely ZENO (**ZE**ro knowledge **N**eural network **O**ptimizer). We first introduce ZENO language construct to maintain high-level semantics and the type information (*e.g.*, privacy and tensor) for allowing more aggressive optimizations. We then propose privacy-type driven and tensor-type driven optimizations to further optimize the generated zkSNARK circuit. Finally, we design a set of NN-centric system optimizations to further accelerate zkSNARK NNs. Experimental results show that ZENO achieves up to 8.5× end-to-end speedup than state-of-the-art zkSNARK NNs. We reduce proof time for VGG16 from 6 minutes to 48 seconds, which makes zkSNARK NNs practical.

Keywords: ZKP, Neural Networks, Privacy

ACM Reference Format:
Boyuan Feng, Zheng Wang, Yuke Wang, Shu Yang, Yufei Ding. 2024. ZENO: A Type-based Optimization Framework for Zero Knowledge Neural Network Inference. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'24).* ACM, New York, NY, USA, 15 pages. https://doi.org/XXXXXXX.XXXXXXX

ASPLOS'24, April 27–May 01, 2024, San Diego, CA, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Zero Knowledge Neural Networks [24, 25, 28, 44, 46] draw increasing attention to solving privacy issues of neural networks. Leveraging zero-knowledge Succinct Non-interactive ARgument of Knowledge (zkSNARK) security scheme [2, 7–9, 13, 14, 48, 52, 56], zkSNARK NNs protect the privacy of user data and model weights. When protecting data privacy, zkSNARK NNs allow users to prove their identity without uploading face images to remote servers. When protecting model privacy, zkSNARK NNs allow companies to prove the model accuracy without releasing weights which are usually treated as important intellectual properties.

zkSNARK is a security scheme where, given an arithmetic function $F(\vec{in})$ and an output y, the prover proves the existence of input $\vec{in}$ such that $F(\vec{in}) = y$ while not revealing its value. In zkSNARK NNs, arithmetic function is a plaintext neural network described with multiplication and addition, while $\vec{in} = (\vec{w}, \vec{x})$ includes both weights $\vec{w}$ and data $\vec{x}$. zkSNARK NNs allow the prover to specify the privacy type of inputs. For example, when protecting data privacy, the prover can set $\vec{x}$ (e.g., a face image) as private and weight $\vec{w}$ as public while proving $F(\vec{w}, \vec{x}) = y$ holds for a public y (e.g., a person name).

To prove such computation, zkSNARK NNs first transform a complex arithmetic function into a simple circuit by mapping each addition and multiplication operation into an addition or multiplication gate, as demonstrated in Fig. 1. The second step is *circuit computation* which condenses such circuit into uniform-format constraints (Eq. 1). Finally, *security computation* further compresses constraints into a fixed-size proof (*e.g.*, 192 bytes [30]) which can be used to verify the computation. While zkSNARK NNs provide privacy properties, existing works usually cannot scale to large neural networks. For example, based on a popular zkSNARK framework Arkworks [4], it takes hundreds of seconds to prove zkSNARK LeNet on a single face image while non-zkSNARK LeNet usually requires less than 100 ms on the same hardware. We summarize three key challenges that hinder deeper system optimizations for zkSNARK NNs.

Failing to maintain high-level semantics during proof generation. Existing zkSNARK systems [4, 13, 25, 33, 62] map an arbitrary arithmetic function into a low-level arithmetic circuit. During this procedure, NN semantics such as privacy and tensor are not preserved and hard to recover. For

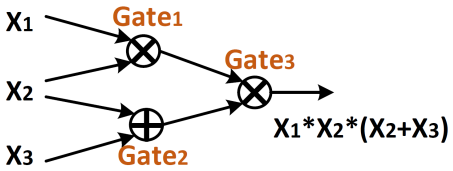

Figure 1. Circuit in zero-knowledge proof for an arithmetic function $X_1 * X_2 * (X_2 + X_3)$. Here, all values are stored in finite-fields (*e.g.*, 254-bit integer [45]) for privacy.

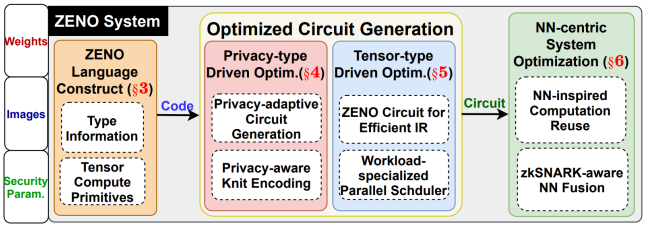

Figure 2. Overview of ZENO.

example, reconstructing tensor semantic by scanning and parsing the assembly-style circuit would introduce heavy runtime overhead. Therefore, zkSNARK systems can only consider individual gates in circuits and fail to exploit high-level NN-specialized optimization opportunities.

Lack of semantic-aware optimizations during compiling zkSNARK NNs. Most zkSNARK optimizations [4, 10, 51]) focus on individual scalar gates and support only local circuit optimization at small scale. These scalar gates usually show heavy dependency and prevent parallel computation. For example, in Fig. 1, parent gates (*e.g.*, $Gate_3$) cannot be computed until all children gates (*e.g.*, $Gate_1$ and $Gate_2$) have been computed. However, most NN computations are conducted at the tensor level (*e.g.*, convolution layers and fully connected layers) and provide abundant parallelization opportunities. Moreover, NN computation usually requires floating-point values (*e.g.*, single-precision or half-precision) or small integers [20, 55, 57, 58, 61], such as `int8` or even `int1`, while zkSNARK operates on finite field (*e.g.*, $\approx 2^{254}$ in case of BLS12-381 [11]) to provide security guarantees. Naively representing these small values from NNs with finite field elements may lead to extra memory and computation overhead.

Lack of NN-centric system optimizations. Neural networks usually contain abundant computation reuse opportunities. For example, a zkSNARK NN shares the same circuit when proving on different images. Existing works usually focus on proving individual images and repeatedly generate redundant constraints. Moreover, fusing NN layers can usually save the number of addition and multiplication computation. This can potentially save the number of constraints in zkSNARK NNs. However, kernel fusion from existing NN systems usually cannot directly bring benefits to zkSNARK NNs. For example, ReLU is usually fused with convolution layer in plaintext NNs but cannot be fused in zkSNARK NNs.

In this paper, we propose a type-based optimizing framework for efficient zero knowledge neural network inference, namely **ZENO** (**ZE**ro knowledge **N**eural network **O**ptimizer). We show the overview of ZENO in Fig. 2. ***First***, we introduce a *ZENO language construct* to maintain high-level semantics (*e.g.*, privacy and tensor) during zkSNARK proof generation. Our key insight is that, instead of parsing an assembly-style circuit, we maintain the privacy type and structured tensor computation to guide efficient zkSNARK proof generation. We further propose a set of compute primitives to effectively express zkSNARK NNs.

Second, we design an *optimized circuit generation* that reduces both computation complexity and the number of computations by exploiting high-level semantics. Our optimized circuit generation includes a privacy-type driven optimization and a tensor-type driven optimization. The privacy-type driven optimization reduces the number of constraints while maintaining zkSNARK NN semantics. We propose a knit encoding to efficiently represent multiple `uint8` NN computation with a single finite field (*e.g.*, 254 bits [11]) to reduce the number of zkSNARK computation. The tensor-type driven optimization exploits tensor computation semantics in zkSNARK NNs to generate a ZENO circuit with minimized dependency. We use ZENO circuit as an in-place replacement for arithmetic circuit to reduce dependency.

Third, we propose *NN-centric system optimizations* to further accelerate zkSNARK NNs. We first propose *NN-inspired computation reuse* to identify the computation reuse opportunities within images and cross images by exploiting NN semantics. Then, we propose a *zkSNARK-aware NN fusion* to fuse NN layers while considering both NN and zkSNARK properties. Our zkSNARK-aware NN fusion can save the number of constraints for reducing zkSNARK NN latency.

Overall, we make the following contributions:

- We propose ZENO, a framework that can deeply optimize zero-knowledge NNs with a synergy between NN semantics and zkSNARK workload properties.
- We propose a set of zkSNARK NN tailored system optimizations. In particular, we design ZENO language construct (§3) to expose high-level semantics, an optimized circuit generation by exploiting privacy type (§4) and tensor type (§5), and NN-centric system optimizations (§6) to further accelerate zkSNARK NNs.
- We extensively evaluate ZENO using six zkSNARK NNs on multiple datasets. We achieve 8.5× end-to-end speedup over state-of-the-art systems.

2 Related Work and Motivation

In this section, we will first give an in-depth discussion on background and related work of zkSNARK Neural Networks

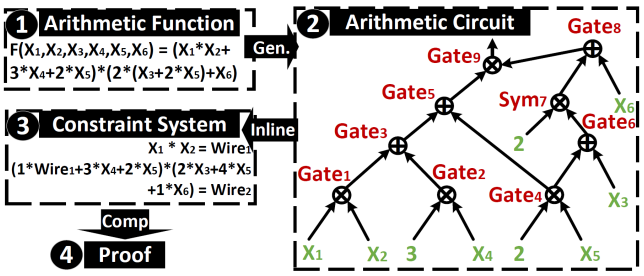

Figure 3. Workflow of generating zero-knowledge proof. All values are stored in ciphertext for privacy.

(NNs). Then, we will demonstrate the unique optimization opportunities for zkSNARK NNs.

2.1 zkSNARK

Zero-Knowledge Succinct Non-interactive Argument of Knowledge (zkSNARK) [2, 7–9, 13, 14, 30, 48, 52, 56] is a security scheme where, given a function $F(x)$ and a target output y, the *prover* shows that the prover knows a specific value x such that $F(x) = y$ while not revealing such value x. Here, the function $F(\cdot)$ can describe an arbitrary computation. One specific example is that, given a function $F(x_1, x_2, x_3, x_4, x_5, x_6) = (x_1x_2 + 3x_3)(2(x_4 + 2x_5) + x_6)$, the prover can generate a proof that the prover knows a set of secrete values $(x_1, x_2, x_3, x_4, x_5, x_6)$ such that $F(x_1, x_2, x_3, x_4, x_5, x_6) = y$ with a public value y while not revealing the exact values of $(x_1, x_2, x_3, x_4, x_5, x_6)$.

zkSNARK Workflow. zkSNARK involves *prover* and *verifier* as two essential participants. During proof generation, the prover knows both private data x and public data y such that $F(x) = y$, and generates a *proof* showing the equality between $F(x)$ and y. This proof could be shared publicly since the private data x cannot be recovered from the proof and the public data y. During proof verification, the verifier checks if the proof is valid under the public data y and is convinced that the prover knows private data x such that $F(x) = y$. Here, the verifier only knows the public data y and the proof, while not knowing the value of the private data x. Proof verification takes only a few milliseconds which are several orders of magnitudes faster than proof generation with second-level latency [30]. To this end, we build ZENO to accelerate the proof generation.

Zero-knowledge Proof Generation. We illustrate zero-knowledge proof generation in Fig. 3. There are three steps in proof generation. The first step is **Generate**, which takes a given *arithmetic function F(x)* ❶ and generates a circuit ❷. In this step, each scalar addition and multiplication in arithmetic function is mapped to a addition gate (*e.g.*, $Gate_3$) and a multiplication gate (*e.g.*, $Gate_1$) in the circuit, respectively. For a large arithmetic function with millions of computation (*e.g.*, zkSNARK NNs [24, 25, 28, 44, 46]), the circuit contains millions of gates. The latency of zkSNARK is proportional to this number of gates such that million-level gates can easily lead to hour-level latency.

The second step is **Circuit Computation** that condenses the *circuit* ❷ into constraints ❸, which is a specialized mathematical format:

$$(\sum_{i=1}^{n} a_{j,i}X_i) * (\sum_{i=1}^{n} b_{j,i}X_i) = Wire_j, \quad j \in \{1, 2, ..., m\} \qquad (1)$$

Here, X_i are private input values (*e.g.*, private NN weights) and $Wire_j$ are private output values which can be used in following constraints. n is the number of private values including both private input values X_i and private output $Wire_j$. m is the number of multiplication between private values (*e.g.*, X_1 and X_2) or linear combination (LC) of private values (*e.g.*, $1 * Wire_1 + 3 * X_4 + 2 * X_5$). The zkSNARK proof generation latency is proportional to the number of private values n and the number of constraints m. For a realistic arithmetic function (*e.g.*, a neural network), both m and n could be million-level. We note several properties in the constraints. First, privacy plays an important role where multiplying a public value and a private value (*e.g.*, $3 * X_4$) does not lead to constraints. Second, the addition is "free" in zkSNARK in terms of not introducing constraints, since a large number of additions can be expressed in a single linear combination (*e.g.*, adding $1*Wire_1$, $3*X_4$, and $2*X_5$) by incorporating into the linear combination of private values. Third, in the circuit computation, children gates (*e.g.*, $Gate_1$ to $Gate_4$) need to be computed before parent gates (*e.g.*, $Gate_5$). This leads to heavy dependency in circuits and is major bottleneck in zkSNARK NNs (see Fig. 4).

The third step is **Security Computation**. Given a constraint system with a large number of n private values and m constraints, *security computation* generates a small fixed-size (e.g., 192 bytes) proof for efficient verification. The key idea is to add carefully crafted random noises [30] upon n private values for generating encrypted values. With these encrypted values, the m constraints still holds but you cannot derive original private value from encrypted values. We remark that noises could be added only when the format of constraints follows a simple math formula (Eq. 1). The latency of this step depends on the number of n private values and m constraints such that we can accelerate this step by reducing m and n.

Bit Size in zkSNARK. zkSNARK relies on well-accepted cryptography techniques such as finite field and pairing-friendly elliptic curves to provide cryptographical security. These cryptography techniques involve computations on large bit-size integers. Popular elliptic curves for zkSNARK include BN254 [6, 53] with 254 bits and BLS12-381 [12] with 381 bits. ZENO is generic over diverse elliptic curves since the choice of elliptic curves does not affect the zkSNARK computation and optimizations.

2.2 zkSNARK Neural Networks

Neural Networks. Neural network (NN) [23, 32, 43, 54] is a function $F(W, X) = Y$ that maps an input image $X \in uint8^{H\times W\times 3}$ and weights W to a prediction $Y \in \mathbb{R}^d$, where d is the number of labels (*e.g.*, $n = 2$ when only distinguish cat and dog). NN is usually defined as the composition of a sequence of NN layers $F(W, X) = F_1 \circ F_2 \cdots \circ F_n(W, X)$. Popular layers include convolution, fully connected, pooling, and ReLU, where each layer computes at tensor level. For example, the convolution layer and fully connected layer take two inputs: activation $X^{(k)}$ and weight W. Then, these two layers compute the output activation $X^{(k+1)} = W \cdot X^{(k)} + b$.

zkSNARK NNs and Killer Applications. zkSNARK NNs [24, 25, 28, 44, 46] draw increasing attention in recent years by proving certain properties while protecting the privacy of images or weights. These zkSNARK NNs treat a NN as a function $F(W, X)$ and generate proof following the workflow in Fig. 3. One popular application is *World ID* [18] for user identity which protects the privacy of biometric image X. In particular, a user (the prover) generates proof with her eye iris image to prove that she is a unique and real person. This proof serves as her digital passport or password while keeping anonymity. This proof is submitted to servers where other companies (the verifier) could verify the proof. Another application is *Leela vs the World* [40] which allows users to play against an AI chess model and uses zkSNARK NNs to prove each move of this model while protecting the privacy of weight W and not leaking the model.

To facilitate the development of zkSNARK programs, several frameworks have been proposed such as Arkworks [4, 13], Bellman [62] and Ginger [33]. However, these frameworks usually focus on scalar computation and ignore optimization opportunities from tensor-level computations, leading to prohibitive latency. In this paper, we propose ZENO to exploit tensor-level computation and privacy type for efficient zkSNARK NN inference.

Diverse Types of zkSNARK NN Layers. Besides addition and multiplication, zkSNARK NNs could also support other operations such as Pooling layers and > in ReLU at a higher cost [25]. Take the ReLU circuit as an example. While zkSNARK does not directly support comparing two integers, it efficiently supports checking bit equality. Thus ReLU circuit first decomposes an integer into n(=254) bits and checks whether the first bit equals zero, deciding the sign of this integer and the output of ReLU layer.

In zkSNARK NNs, a NN is first trained with Stochastic Gradient Descent (SGD) [49] in plaintext and then proofs are generated for NN inference with zkSNARK. Following existing zkSNARK NN application scenarios, ZENO focuses on inference and does not support SGD for training.

2.3 Opportunities and Challenges

In this section, we introduce optimization opportunities and challenges in enabling efficient zkSNARK NNs.

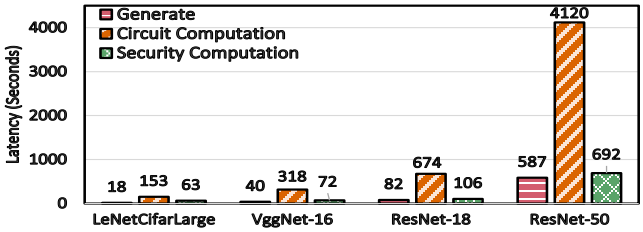

Figure 4. Proof latency: *private* images and *public* weights.

We show the latency of individual proof generation steps in Fig. 4 for private images and public NN weights. We have similar observations on other privacy settings (*e.g.*, private weights and private images, or private weights and public images). We profile this latency based on state-of-the-art zkSNARK framework, Arkworks [4], on a single image. Note that these three steps need to be executed sequentially and the total time is the sum of individual steps. We have three major observations. First, the total time of zkSNARK NN can easily exceed 5000 seconds, while the corresponding non-zkSNARK NNs usually take less than 1 second to compute. Second, the latency of *circuit computation* increases significantly as NN sizes increase. Third, the latency of *security computation* also increases as NN sizes increase.

Opportunities. There are two major opportunities to accelerate zkSNARK NNs. The first opportunity is to exploit privacy types (*e.g.*, private weights or public weights, as discussed in §3). Our investigation shows a significant impact from privacy types, which motivates privacy-driven optimizations. The second opportunity is to exploit tensor computation in NNs for optimizing circuits and exploiting parallelism. This opportunity has not been explored in existing zkSNARK frameworks that focus on scalar operations.

Challenges. Although these ideas sound promising, the efforts to capitalize on their benefits are non-trivial due to several challenges. First, while tensor operations may provide optimization opportunities, it is highly non-trivial to identify and reconstruct such high-level semantics from assembly-style circuits. We need a language construct to maintain these high-level semantics and facilitate optimizations for zkSNARK NNs. Second, the zkSNARK computation procedure usually shows complex dependency across Gates and synergy between privacy types. For example, circuits (as discussed in Fig. 3) are inherently sequential since earlier computation results (*e.g.*, $Gate_1$) may be used by later computation (*e.g.*, $Gate_3$). We need specialized optimizations based on type information in zkSNARK NNs to reduce the computation workload and mitigate the dependency.

3 ZENO Language Construct

In this section, we introduce ZENO language construct to facilitate the zkSNARK NN development and maintain the semantic information during zkSNARK computation.

Table 1. ZENO Type Information.

	Type	Description
Standard	**Const**	Public constant value in λ-bit finite field.
	Variable	Private scalar value in circuit for input.
	Gate	Private scalar value in circuit for intermediate results.
	Wire	Private scalar value in constraint system.
	LC	Linear Combination of wires in constraint system.
ZENO	**Privacy**	'private' or 'public'
	Tensor	A tensor of finite field data.
	zkTensor	Tuple (T, P) where "T" is a Tensor and "P" specifies privacy.

Type Information with Tensor and Privacy. The goal of ZENO type information is to express the two important information in zkSNARK NN – *tensor* and *privacy*. We summarize ZENO type information in Table 1. There are two complications in zkSNARK systems. First, previous zkSNARK systems contain only scalar-level data types, which make it complicated to implement zkSNARK NNs with intensive tensor computations. Second, individual scalar data types have different privacy properties. This makes it challenging to manually set the privacy type for scalar values in zkSNARK NNs.

To tackle these challenges, we introduce tensor-level data types to directly express zkSNARK NN tensor computation and hide the complexity of privacy selection. *zkTensor* is the basic data unit in zkSNARK NNs, which can represent weight tensors and feature tensors in NNs. When "P" is public, "T" is a tensor of Const scalars for public constant values. When "P" is private, "T" is a tensor of *Variable*, *Gate*, *Wire*, and *LC*, where the specific type can be inferred automatically. Our type information abstracts details of zkSNARK implementations and enables users to focus on complex NN structures.

Tensor Compute Primitives. We propose a set of tensor-level compute primitives respecting the privacy and tensor types. The goal of tensor compute primitives is to maintain the high-level semantics of zkSNARK NN computation and maps directly to gate-level circuits. In particular, the tensor compute primitives hide the complexity of scalar-level operations and expose tensor computation capability, which is the building block of many zkSNARK NNs. The tensor compute primitives also allow users to easily specify the privacy type of images and weights, which mitigates the manual efforts in specifying the privacy of each scalar. The tensor compute primitives directly support `dotProduct` which consumes most computation in neural networks. This high-level `dotProduct` can be directly mapped to gate-level circuits with optimized circuit generation (discussed in §4 and §5). We then introduce `fullyConnected`, `convolution`, `pool`, and `ReLU` to support popular layers in NNs. We also provide `addTensor` and `mulTensor` to facilitate user-defined NN operations such as residual connection [32].

4 Privacy-type Driven Optimization

In this section, we propose privacy-type driven optimizations. Our key insight is that fully exploiting privacy of input data can significantly reduce the number of constraints (Eq. 1), which leads to proportional performance improvement for zkSNARK NNs. To this end, we propose *privacy-adaptive circuit generation* and *privacy-aware knit encoding* to squeeze the number of constraints.

4.1 Privacy-adaptive Circuit Generation

We propose privacy-adaptive circuit generation to reduce the number of constraints in zkSNARK NNs. We observe that many zkSNARK NNs algorithmic designs [24, 25, 28, 44, 46] only require one of features or weights to be private. For example, ZEN [25] only keeps privacy of NN weights and use a public dataset to prove the NN accuracy. A naive implementation usually ignores privacy type of input data and generate constraints for each multiplication in zkSNARK NN, which leads to a large number of constraints and high latency. Our key insight is that privacy comes with costs in zkSNARK. We should introduce privacy only when necessary and exploit as many "free" operations as possible to reduce cost. For example, multiplying a public scalar and a private scalar are "free" but multiplying two private scalars costs 1 constraint. The number of constraints largely decides the latency. So, we exploit privacy types of features and weights to minimize the number of constraints.

We present our privacy-adaptive circuit generation for dot products which can be easily applied to many zkSNARK NN layers (*e.g.*, fully-connected, convolution, and average pooling). Formally, we consider a weight vector $W = [w_1, w_2, ..., w_n]$ with privacy p_w and a feature vector $X = [x_1, x_2, ..., x_n]$ with privacy p_X where p_W and p_X are user-specified privacy type ("private" or "public"). zkSNARK first computes a *reference value ref* in plaintext according to dot product. Then, zkSNARK proves in constrains that $ref = \sum_{i=1}^{n} w_i * x_i$. In the last layer of zkSNARK NN, *ref* is the NN prediction such as a "cat" or "dog". We show the mapping from high-level dot product computation $\sum_{i=1}^{n} x_i * w_i$ to low-level constraints $(\sum_{i=1}^{n} a_{j,i}X_i) * (\sum_{i=1}^{n} b_{j,i}X_i) = Wire_j, \quad j \in \{1, 2, ..., m\}$ where X_i and $Wire_j$ are private values, and $a_{j,i}$ and $b_{j,i}$ are public coefficients (see background in Eq. 1).

Both private feature and private weights. When both feature and weights are private, we have n multiplications between private scalars w_i and x_i and $n-1$ addition to sum the multiplication output. Since both w_i and x_i are private values, we generate one constraint for each multiplication $w_i * x_i = Wire_i$. Formally, each multiplication can be written

as constraints $(1 * w_i) * (1 * x_i) = Wire_i$. This leads to n constraints for multiplying private scalars. Then, we generate a linear combination $LC = \sum_{i=1}^{n} 1 * Wire_i$ to represent the computation result in zkSNARK and check the equality between LC and a reference value ref for dot product $W \cdot X$. Intuitively, this circuit checks that the dot product of private input W and X equals to ref without releasing the value of W and X. Checking equality leads to an extra constraint. Formally, we have $n + 1$ constraints:

$$\begin{aligned} &(1 * w_i) * (1 * x_i) = Wire_i, \quad i \in \{1, 2, ..., n\} \\ &(\sum_{i=1}^{n} 1 * Wire_i + (-1) * ref) * (1 * D_1) = D_0 \end{aligned} \tag{2}$$

where $D_1 = 1$, $D_0 = 0$, and -1 is conducted on finite field.

Either private feature or private weights. We consider public weight W and private feature X since the design can be easily applied to the case with public weight and private feature. When weight W is private and feature X is public, we have n multiplications between private weight scalar w_i and public feature scalar x_i and $n - 1$ additions to sum the multiplication output. One naive design is to still generate one constraint for each multiplication. However, our key insight is that the public weight scalar w_i can be treated as public coefficients in Eq. 1 which eliminates unnecessary constraints. To this end, we can directly generate a linear combination $LC = \sum_{i=1}^{n} w_i * x_i$ with public scalars w_i as coefficients and check equality with ref. This design requires only 1 constraint

$$(\sum_{i=1}^{n} w_i * x_i + (-1) * ref) * (1 * D_1) = D_0 \tag{3}$$

This is significantly smaller than $n + 1$ constraints required for both private feature and private weights.

4.2 Privacy-aware Knit Encoding

We propose privacy-aware knit encoding to further reduce the number of constraints when only features or weights are private. This could significantly reduce the latency of *security computation* phase which is proportional to the number of constraints. Knit encoding combines multiple low-bit (*e.g.*, 8-bit) scalars into a high-bit (*e.g.*, 254-bit) scalar to check equality simultaneously and reduce the number of equality checks. This leads to a lower number of constraints and better performance. The key insight is that the output scalars of a NN layer are usually low-bit (*e.g.*, 8 bits) while zkSNARK natively supports large bits (*e.g.*, 254 bits). For example, suppose we prove the computation over two dot products in a NN layer, naive encoding needs 2 equality checks leading to 2 constraints. In knit encoding, we can use the "free" addition to combine the results of these two dot products and introduce only 1 equality check.

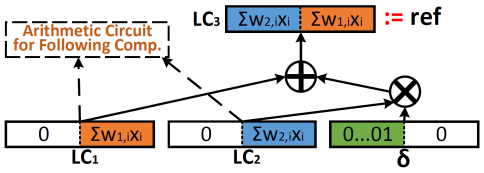

Figure 5. Knit encoding with batch size $s = 2$. LC_1 and LC_2 are two finite fields with leading bits as 0. $\delta = 2^{2*b_{in}+\lceil log(c_{in}) \rceil}$ is a finite field such that multiplying δ is equivalent to bit shifting. ":=" indicates equality check.

Naive encoding. Consider a fully connected layer with a public weight $W = [W_1, W_2] \in \texttt{uint8}^{2 \times c_{in}}$, a private feature $X = [x_1, x_2, ..., x_{c_{in}}] \in \texttt{uint8}^{c_{in}}$, and the output $Y = [y_1, y_2] \in \texttt{uint8}^2$. The fully-connected layer can be treated as two dot products $y_i = W_i \cdot X, i \in \{1, 2\}$. One naive approach is to independently encode individual dot products following Eq. 3. This approach leads to 1 constraint for each dot product and require 2 constraints for the fully connected layer. However, this approach encodes low-bit quantized neural network values (*e.g.*, `uint8`) with high-bit finite fields (*e.g.*, 254-bit), which leads to extra constraints and higher latency.

Knit encoding with batch size $s = 2$**.** We propose to batch multiple low-bit values (*e.g.*, `uint8`) into one high-bit finite field (*e.g.*, 254-bit) to reduce the number of constraints, as illustrated in Fig. 5. We first generate two LCs

$$LC_1 = \sum_{i=1}^{c_{in}} w_{1,i} * x_i, \quad LC_2 = \sum_{i=1}^{c_{in}} w_{2,i} * x_i$$

Generating LC_1 and LC_2 does not introduce constraints since we are multiplying public scalars with private scalars. Here, both LC_1 and LC_2 are finite fields. We note that only $2 * b_{in} + \lceil log(c_{in}) \rceil$ bit of each LC are non-zero values where b_{in}(=8) is the bit width of weights and features.

Instead of naively introducing constraints for checking equality between LC_i and y_i, we further encode these two LCs into one LC:

$$\begin{aligned} LC_3 &= LC_1 + LC_2 * \delta \\ &= \sum_{i=1}^{c_{in}} w_{1,i} * x_i + \sum_{i=1}^{c_{in}} (w_{2,i} * \delta) * x_i \end{aligned}$$

Here, $\delta = 2^{2*b_{in}+\lceil log(c_{in}) \rceil}$ is sufficiently large to ensure the correctness of encoding. δ is also a public scalar such that generating LC_3 does not introduce constraints.

Finally, we compute the encoded output value $ref = y_1 + y_2 * \delta$ and introduce 1 constraint to check equality of these two dot products simultaneously

$$(\sum_{i=1}^{c_{in}} w_{1,i} * x_i + \sum_{i=1}^{c_{in}} (w_{2,i} * \delta) * x_i + (-1) * ref) * (1 * D_1) = D_0$$

Table 2. Comparing knit encoding and stranded encoding. We consider 8 bits for features and weights and 254 bits for finite fields.

	Knit Encoding	Stranded Encoding [25]
Max Constraint Saving	8×	4×
Encoding Overhead	0 Constraint	0 Constraint
Decoding Overhead	0 Constraint	632 Constraints
Privacy	One private	Both Private

This constraint bitwisely checks equality such that $W_1 \cdot X = y_1$ and $W_2 \cdot X = y_2$ when δ is sufficiently large.

Knit encoding for arbitrary batch size s. Knit encoding can be generalized to arbitrary batch size s. Formally, knit encoding takes a public weight $W = [W_1, W_2, ..., W_s] \in \texttt{uint8}^{s \times c_{in}}$, a private feature $X = [x_1, x_2, ..., x_{c_{in}}] \in \texttt{uint8}^{c_{in}}$, and the output $Y = [y_1, y_2, ..., y_s] \in \texttt{uint8}^s$. We first generates s LCs for dot products

$$LC_j = \sum_{i=1}^{c_{in}} w_{j,i} * x_i, \quad j \in \{0, 1, ..., s-1\}$$

Then, we encode s LCs into one LC

$$LC_s = \sum_{j=0}^{s-1} \sum_{i=1}^{c_{in}} (w_{j,i} * \delta^j) * x_i$$

Since we only require multiplication between public scalars and private scalars, we do not introduce constraints when generating these LCs. Finally, we can compute the encoded output value $ref = \sum_{j=0}^{s-1} y_j * \delta^j$ and use 1 constraint to bitwisely check the euqality of s dot products:

$$(\sum_{j=0}^{s-1} \sum_{i=1}^{c_{in}} (w_{j,i} * \delta^j) * x_i + (-1) * ref) * (1 * D_1) = D_0$$

Security Analysis. Relying on a well-known "free" addition property of zkSNARK [30], knit encoding is cryptographically secure if and only if bit overflow is avoided for all possible input weights and features. Bit overflow happens when batching too many low-bit values into a high-bit finite field using a large batch size s.

ZENO automatically selects the batch size s to maximize the performance while avoiding bit overflow. Formally, given the vector length n, input data bitwidth b_{in}, and finite field bitwidth b_{out}, each dot product requires $2 * b_{in} + \lceil \log n \rceil$ bits and all s dot products require $s*(2*b_{in}+\lceil \log n \rceil)$ bits. To avoid bit overflow and maximize benefits, we select a batch size as the largest integer satisfying $s \leq b_{out}/(2 * b_{in} + \lceil \log n \rceil)$. For example, on dot product with b_{in} = 8-bit data, b_{in} = 8-bit weight, b_{out} = 254-bit finite field, and length n = 1024, we select s = 9 to maximize benefits while avoiding bit overflow.

Comparing with Stranded Encoding. Existing work [25] proposed stranded encoding which shares similar high-level motivation as our knit encoding. It focuses on the case with private weights and private features by reducing the number of multiplications. However, stranded encoding and knit encoding are significantly different in multiple perspectives, as summarized in Table 2. Stranded encoding can be applied when both features and weights are private while knit encoding can be applied when only features or weights is private. By exploiting privacy type, knit encoding can save more constraints with significantly reduced decoding overhead.

5 Tensor-type Driven Optimization

In this section, we propose tensor-type driven optimizations. We first propose *ZENO circuit* as an efficient intermediate representation (IR) between high-level NN layers and low-level constraints. Then, we propose *workload-specialized parallel scheduler* to identify parallel computation opportunities in ZENO circuit across NN layers. All these optimizations focus on the system level and do not introduce any cryptographical changes, thus guaranteeing security.

5.1 ZENO Circuit for Efficient IR

We present our ZENO circuit as an efficient intermediate representation (IR) from high-level zkSNARK NN arithmetic function to low-level constraints. Since low-level constraints require a specialized mathematical format $(\sum_{i=1}^{n} a_{j,i}X_i) * (\sum_{i=1}^{n} b_{j,i}X_i) = Wire_j$ (see Eq. 1), it is challenging to manually write constraints for an arbitrary arithmetic function. Existing work [4] utilizes circuit as an intermediate representation to automatically map arithmetic functions into constraints during the *circuit computation* phase. However, it is designed for scalar computations and ignores intrinsic tensor types in zkSNARN NNs which leads to unsatisfactory performance. We first analyze the bottleneck in circuit and then propose ZENO circuit as an efficient intermediate representation.

Circuit. Circuit first breaks an arbitrary arithmetic function into a sequence of scalar multiplication and scalar addition operations. Then, it maps each operation to a corresponding multiplication gate and addition gate, as discussed in §2.1. We show an example of circuit for dot product in Fig. 6(a).

Consider a weight vector $W = [w_1, w_2, w_3, w_4]$, a feature vector $X = [x_1, x_2, x_3, x_4]$, and an arithmetic function

$$F(W, X) = w_1 * x_1 + w_2 * x_2 + w_3 * x_3 + w_4 * x_4$$

Circuit first maps each multiplication to a multiplication gate (*e.g.*, $Gate_1$ and $Gate_2$) and maps each addition to an addition gate (*e.g.*, $Gate_3$ and $Gate_5$), leading to 4 multiplication gates and 3 addition gates. Here, all computations related to private variables are symbolic since the circuit describes computation in the arithmetic function regardless of specific values. For example, public weight and private feature indicate that features X are symbolic variables but weights W are numeric coefficients.

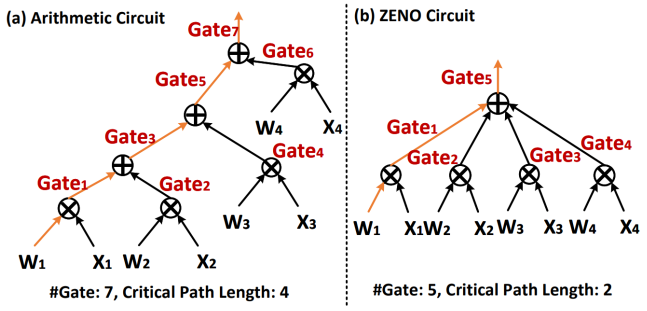

Figure 6. Illustration of ZENO IR for dot product of $\mathbf{W} \cdot \mathbf{X} = [W_1, W_2, W_3, W_4] \cdot [X_1, X_2, X_3, X_4]$.

Given this circuit, we need to conduct *circuit computation* which converts individual gates into constraints with a specialized mathematical format (Eq. 1). Without loss of generality, we consider public weight and private feature here. We can first check privacy of each scalar and generate a tuple where public input w_i is coefficient (*i.e.*, $a_{j,i}$ and $b_{j,i}$ in constraints Eq. 1) and private input x_i is a symbolic variable:

$$(1, Gate_i) = (w_i, x_i), \quad i \in \{1, 2, 4, 6\}$$
$$(1, Gate_i) = (c_{i,1}, Gate_{i-2}) + (c_{i,2}, Gate_{i-1}), \quad i \in \{3, 5, 7\}$$

For addition gates, we have $c_{i,1} = c_{i,2} = 1$ as the coefficient for dot product, which can be an arbitrary integer in general.

Then, we need to recursively expand children gates for an addition gate, if one of its children gates is still an addition gate. This recursive expansion from "binary" gates to "multi-input" format (Eq. 1) leads to $O(n^2)$ costs where n is the vector length. For example, suppose we have expanded $Gate_5$ as

$$(1, Gate_5) = (w_1, x_1) + (w_2, x_2) + (w_3, x_3)$$

where $Gate_5$ has three coefficients w_1, w_2, and w_3. When expanding $Gate_7 = (c_{7,1}, Gate_5) + (c_{7,2}, Gate_6)$, there are $O(n)$ multiplications since we need to multiply the coefficient $c_{7,1}$ of $Gate_5$ with all expanded coefficients, including w_1, w_2, and w_3. Since we need to repeat for all n addition gates, this expansion costs $O(n^2)$, leading to prohibitive latency for zkSNARK NNs with millions of gates in circuit.

ZENO Circuit for Dot Product. To address this problem, we propose a ZENO circuit to minimize the number of gates and reduce the computation complexity to $O(n)$ where n is the vector length, leading to reduced latency during the *circuit computation* phase. The core idea is to exploit the commutative property of addition gates in zkSNARK. In particular, the order of addition gates can be exchanged while the order between two multiplication gates and the order between a multiplication gate and an addition gate need to be maintained. We show ZENO circuit for dot product of length 4 in Fig. 6(b). We introduce 4 multiplication gatess ($Gate_1$, $Gate_2$, $Gate_3$, and $Gate_4$) and only one addition gate ($Gate_5$). Note that ZENO circuit has the same number of multiplication gates but a significantly smaller number of addition gates. This reduced number of addition gates significantly saves the number of computations during *circuit computation*. On ZENO circuit, we can skip *circuit computation* operation for addition gates and directly generate constraints. In particular, we only need 5 operations for converting ZENO circuit while requiring 12 operations for converting circuit. We also note that ZENO circuit shows short critical path length (=2) than conventional circuit with length (=4).

Formally, given two vectors $[w_1, w_2, ..., w_n]$ and $[x_1, x_2, ..., x_n]$ of length n, ZENO circuit contains binary multiplication gates and multi-child addition gates. The binary multiplication gate takes two input gates. To support dot product on two vectors of length n, we need n multiplication gates for each $w_i * x_i$. The multi-child addition gate takes n inputs where n can be arbitrarily large number. This gate efficiently supports summation over a large number of scalars in dot product and significantly reduces the number of addition gates. In comparison, conventional circuit for dot product requires $n-1$ binary addition gates. In total, ZENO circuit for dot product generates $n+1$ gates while conventional circuit generates $2n-1$ gates. We also stress that both ZENO circuit and conventional circuit generate the same constraint systems. Thus ZENO circuit maintains the semantic and can be used as an in-place replacement of conventional circuit.

ZENO circuit for fully connected, convolution, and pooling layers. We propose ZENO circuit for fully connected, convolution, and pooling layers as an extension to ZENO circuit for dot product. Fully-connected layer takes two input tensors $\mathbf{W} \in \mathbb{R}^{m \times n}$ and $\mathbf{X} \in \mathbb{R}^n$ and generates one output tensor $\mathbf{Y} = \mathbf{WX} \in \mathbb{R}^m$. With the help of *img2col* algorithm [17], convolution layer can also be transformed into a matrix-matrix multiplication. It takes two input tensors $\mathbf{W} \in \mathbb{R}^{m \times n}$ and $\mathbf{X} \in \mathbb{R}^{n \times k}$ and computes an output tensor $\mathbf{Y} = \mathbf{WX} \in \mathbb{R}^{m \times k}$. Since fully connected and convolution layer can be viewed as m and mk independent dot products, we simply duplicate dot product circuits for m and mk times as ZENO circuit for fully-connected and convolution layers, respectively. For the pooling layer, we focus on average pool following state-of-the-art zkSNARK NN security scheme [25]. Given an input tensor of shape $m \times n$ and a constant s, average pool splits the tensor into small grids of shape $s \times s$ and computes the average value in each grid. Thus average pool can be viewed as a dot product between a *one* vector $\mathbf{1}$ of length s^2 (*i.e.*, all elements are 1's) and a vector of all values in a grid. On the ReLU layer, ZENO shares the same circuit as scalar-level zkSNARK frameworks since ReLU contains only elementwise comparison.

Theoretical benefit analysis. We summarize theoretical benefits of ZENO circuit in Table 3. One significant result is that ZENO circuit requires $O(n)$ computation for dot product while conventional circuit requires $O(n^2)$ computation. This generalizes to fully connected, convolution, and pool layers with significantly reduced complexity. This saving

Table 3. NN layer complexity comparison between conventional circuit and proposed ZENO circuit.

IR	Layer	Input Shape	# Gate	# Wire	# LC	len(CriticalPath)	Computation
Arithmetic Circuit	Dot Product	(n, n)	$2n-1$	n	$n-1$	n	$O(n^2)$
	Fully Connected	$(m \times n, n)$	$m(2n-1)$	mn	$m(n-1)$	n	$O(mn^2)$
	Convolution	$(m \times n, n \times k)$	$mk(2n-1)$	mkn	$mk(n-1)$	n	$O(mkn^2)$
	Pool	$(m \times n)$, s	$\frac{mn}{s^2}(s^2-1)$	0	$\frac{mn}{s^2}(s^2-1)$	s^2-1	$O(mns^2)$
ZENO Circuit	Dot Product	(n, n)	$n+1$	n	1	2	$O(n)$
	Fully Connected	$(m \times n, n)$	$m(n+1)$	mn	m	2	$O(mn)$
	Convolution	$(m \times n, n \times k)$	$mk(n+1)$	mkn	mk	2	$O(mkn)$
	Pool	$(m \times n)$, s	$\frac{mn}{s^2}$	0	$\frac{mn}{s^2}$	1	$O(mn)$

leads to significant performance improvement on zkSNARK NNs with millions of gates. ZENO circuit also introduces a constant critical path length of 2, in contrast to the length n in conventional circuit. This exposes parallel opportunities that can hardly be identified in conventional circuit due to complex dependency.

5.2 Workload-specialized Parallel Scheduler

Workload-specialized parallel scheduler identifies the parallel computation opportunities during *circuit computation* phase and exploits these opportunities for speedup. While NNs have parallel opportunities in the same NN layer (*e.g.*, fully connected layer), NNs are also intrinsically sequential across layers where leading layer needs to be computed before following layers. This cross-layer dependency still hurdles paralleling zkSNARK NN computation even with ZENO circuit that improves parallelism within NN layer. Naively parsing the circuit at NN level still leads to heavy overhead.

We propose a lightweight *dependency-aware workload scheduler* to identify cross-layer dependency in circuit and map parallel workloads to individual threads. We have two major observations. First, gates in the same zkSNARK NN layer usually can be computed independently while gates in later layers depend on gates in leading layers. Second, the number of gates for a NN layer is proportional to the number of computation in this layer. To this end, we propose a three-step design. First, based on the plaintext NN with specific layer shapes, we first count the number of addition and multiplication in each layer. For example, given a fully connected layer with shape $M \times N$, there are $M \times N$ multiplications and $M \times (N-1)$ additions. Then, based on this number of computation, we directly identify the gates for each NN layer since each addition and multiplication is mapped to exactly one gate in the circuit. Finally, we evenly assign gates in the same layer to each thread for acceleration.

6 NN-centric System Optimization

In this section, we propose *NN-centric system optimization* to further accelerate zkSNARK NN computation.

6.1 NN-inspired Computation Reuse

ZENO identifies redundant computation in zkSNARK NNs and removes such redundancy for improving performance. In particular, we identify two types of computation reuse opportunities – *frequency-based cache service* for mitigating redundancy when computing a single image and *batch-specialized constraint system sharing* for mitigating redundancy when computing a batch of images.

Frequency-based Cache Service. We build a lightweight cache service to cache computation results of frequent operand pairs during *circuit computation* phase, such as public weights and constant coefficients in average pooling. We have two insights behind this design. First, zkSNARK NNs usually compute with `uint8` values since zkSNARK supports only computation on finite fields (*e.g.*, 254-bit integers). Since there are at most 256 values for `uint8`, the same value appears frequently. Second, NN weights and features usually follow Normal distribution where many weights and features are around zero, as widely observed in the NN algorithmic area [31, 34]. This distribution makes many values around zero appear frequently. To this end, cache service can improve performance by reducing the number of expensive computations on λ-bit finite fields ($\lambda \geq 254$). Since we only apply cache service to public data, this does not lead to security vulnerabilities such as timing side channels.

To mitigate the runtime overhead, we adopt a two-phase design. During the offline profiling phase, we evaluate the plaintext NN on a small set (=100) of images and profile the frequency of addition and multiplication operand pairs. We rank all pairs by frequency and keep the top-k(=5) values and the computation results in a hash table. This offline profiling introduces negligible overhead since it is only conducted once on a plaintext NN. During the online computation phase, for each weight and data pair, we first search the pair in the hash table and reuse the results in the hash table. In this way, we can mitigate expensive security computation for a large number of weight and data pairs that appear frequently.

Batch-specialized Constraint System Sharing. We share the constraint system across images when using the same

zkSNARK NN to process a batch of images. Our key insight is that the constraint system is a description of the zkSNARK NN computation. Since we usually use the same zkSNARK NNs to process a batch of images, the same computation applies to each image such that the constraint system can be shared. One specific example is the accuracy scheme in ZEN [25], where the same zkSNARK NN is used to process $n(= 100)$ images for proving the accuracy of the zkSNARK NN. To this end, ZENO provides a batch mode that takes a zkSNARK NN and a batch of images. The *generate* and *circuit computation* steps are only conducted once and the constraint system is reused for different images, leading to improved overall performance. In particular, in the same constraint system, we assign different values to input variables according to images.

6.2 zkSNARK-aware NN Fusion

We propose *zkSNARK-aware NN fusion* to further reduce the number of constraints for performance improvement. Our key insight is that the number of constraints is proportional to the number of computation in zkSNARK NNs. While fusion has been utilized to accelerate non-zkSNARK NNs [16, 26, 59], there are several intrinsic differences in tensor fusion for zkSNARK NNs. *First*, fusion in non-zkSNARK NNs usually target reducing memory access by avoiding saving intermediate results in memory. In zkSNARK NNs, we target reducing the number of computations which decides the number of constraints and the latency of generating zero-knowledge proofs. *Second*, fusion in non-zkSNARK NNs usually fuses all element-wise computation (*e.g.*, relu) with convolution layers. However, many element-wise computation cannot be fused in zkSNARK NNs. For example, relu layer cannot be fused since relu requires expensive comparison operator with hundreds of constraints in zkSNARK.

To this end, we propose *pre-computation-based fusion* to reduce computation in zkSNARK NNs. Many NN layers involve injective computation such as one-to-one scale and addition. We can fuse such injective layers with convolution and fully-connected layers. For example, consider a fully connected layer $Y = WX$ and a batch normalization layer $BN(Y) = \gamma * Y + \beta$ which is an injective layer. Naive approach is to independently prove the computation of these two layers which leads to extra constraints. Instead, we can precompute the fused weight value $\gamma * W$ and directly prove the computation of $(\gamma * W)X + \beta$ to save constraints.

7 Evaluation

In this section, we comprehensively evaluate ZENO over various datasets and popular NNs.

Baselines. We compare ZENO with Arkworks [4, 13], which is the state-of-the-art zkSNARK framework and widely used in industry zkSNARK products [3, 15, 41]. We also compare with two other representative zkSNARK frameworks, Bellman [62] and Ginger [33] for comprehensive comparison.

Table 4. Neural Networks for Evaluation

Network	Abbr.	#FLOPs (K)	Acc.(%)
ShallowNet	*SHAL*	102	94.91
LeNetCifarSmall	*LCS*	530	55.35
LeNetCifarLarge	*LCL*	7,170	63.68
VggNet-16	*VGG16*	19,917	84.19
ResNet-18	*RES18*	32,355	85.45
ResNet-50	*RES50*	69,191	87.05

Datasets. We evaluate with two popular datasets (MNIST and CIFAR-10) in secure deep learning field [22, 25, 29, 35, 36, 60]. **MNIST** [42] is a large dataset for handwritten digits classification with 60,000 training images and 10,000 testing images in gray-scale with the shape of $28 \times 28 \times 1$. **CIFAR-10** [39] is a classification dataset with 10 classes (e.g., cat and dog). It contains 50,000 training images and 10,000 testing images of shape $32 \times 32 \times 3$.

Models. We evaluate six neural networks, as summarized in Table 4. The evaluation of these six variants demonstrates the performance of ZENO under diverse model sizes. In particular, ShallowNet [25] contains two fully connected layers and one ReLU layer. *LeNetCifarSmall* and *LeNetCifarLarge* are two variants of LeNet [43] with 5 layers but different number of computation. *VggNet-16* [50], *ResNet-18* [32], *ResNet-50* [32] have 16, 18, and 50 NN layers, respectively. We evaluate ShallowNet on MNIST and all other models on CIFAR-10.

Experiment Configuration. All the evaluations run on a server with Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz and 503 GB DRAM.

7.1 End-to-End Evaluation

In this section, we show the end-to-end performance improvement from ZENO on various privacy settings. For example, the privacy setting of private image and public weights can be used when we only protect the user image privacy (*e.g.*, face image) and prove the user's identity on a public NN (*e.g.*, a face recognition based door lock system). The privacy setting of private weights and private images can be used when we aim to protect both privacy-sensitive images (*e.g.*, medical images) and weights (*e.g.*, private NNs as we discussed in §1). We skip the privacy setting of private weights and public images since it shows similar results as private images and public weights. We report the end-to-end execution time summing all three phases for proof generation, including *Generate*, *Circuit Computation*, and *Security Computation*. We measure the proof generation latency of a single image with a batch size of 1.

Overall Speedup. We first show the overall speedup when proving private images and public weights in Fig. 7. Overall, ZENO achieves upto 8.5× speedup than Arkworks.

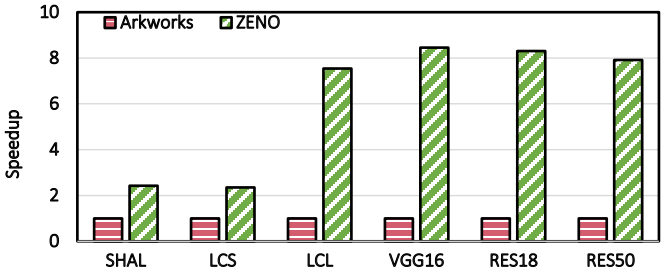

Figure 7. Overall speedup: private image & public weight.

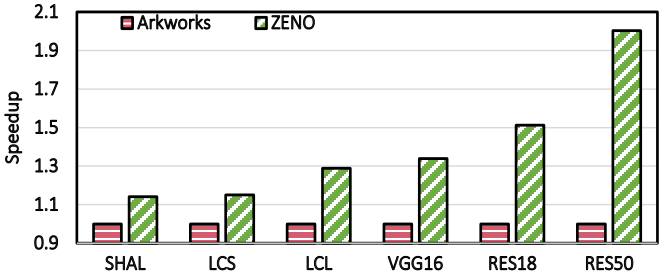

Figure 8. Overall speedup: private image & private weight.

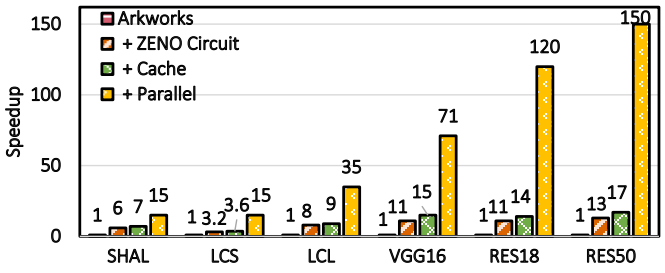

Figure 9. Circuit computation speedup: private image & public weight.

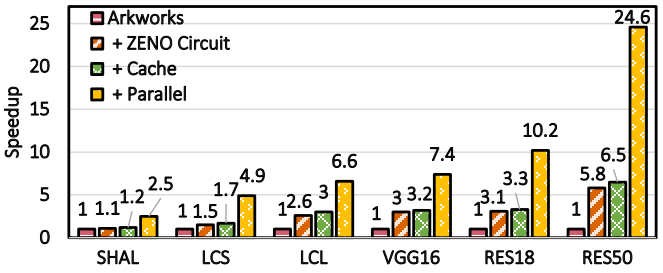

Figure 10. Circuit computation speedup: private image & private weight.

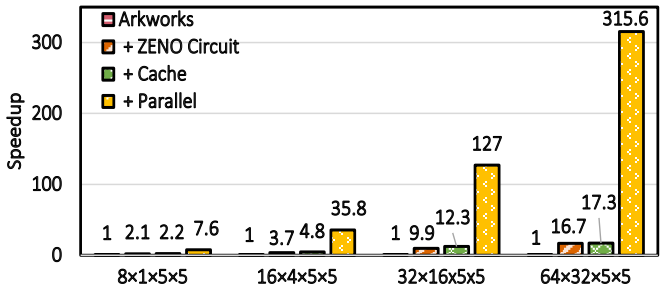

Figure 11. Circuit computation speedup: convolution. Shape: [#*c_out*, #*c_in*, *kernel_width*, *kernel_height*]

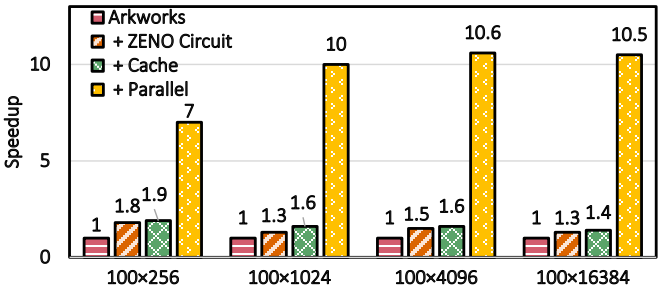

Figure 12. Circuit computation speedup: fully-connected layer. Shape: [#*c_in*, #*c_out*]

This result shows that ZENO can significantly improve the performance of zkSNARK NNs. We also observe that ZENO achieves higher speedup on large NNs (*e.g.*, 8.5× on VGG16) than small NNs (*e.g.*, 2.4× on SHAL). The reason is that tensor-type driven optimization (§5) reduces the quadratic computation complexity to linear complexity for many NN layers (*e.g.*, fully connected, convolution, and pool). We highlight that we reduce the latency of ResNet-50 from 5154 seconds (around 1.5 hours) to 680 seconds (around 11 minutes), which makes it promising to construct practical zkSNARK NNs.

We show overall speedup when proving private NN weights and private images in Fig. 8. We achieve up to 2.01× speedup, which shows the effectiveness of ZENO optimizations. We also observe a similar trend as Fig. 7 that ZENO achieves higher speedup on larger zkSNARK NNs. This validates the benefits of tensor-driven optimizations in reducing the computation complexity. Compared with Fig. 7, we observe a smaller speedup. The reason is that our type-sensitive circuit generation provides more aggressive optimization for the setting with private weights and public images. This shows the importance of considering privacy information (§4) when optimizing zkSNARK NNs.

Raw Latency Comparison. We provide the raw latency on an image in Table 5. We show latency on CPU following popular zkSNARK frameworks. We observe a significant latency reduction from 398 seconds to 48 seconds for VGG16. Although this is still a gap from non-zkSNARK NNs due to the overhead from zkSNARK security scheme such as 254-bit finite fields instead of 8-bit integers, we are the first to bring zkSNARK NN into the practical realm. For example, a user may spend 8.5 seconds for lightweight models such

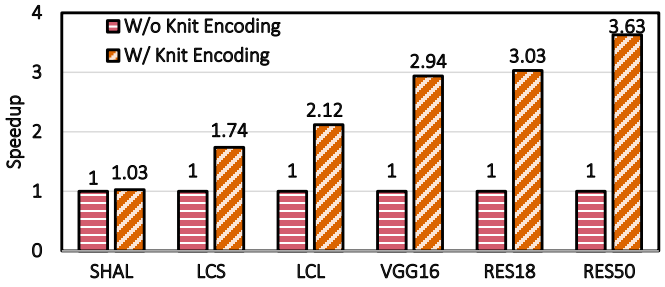

Figure 13. Security comput. speedup from knit encoding

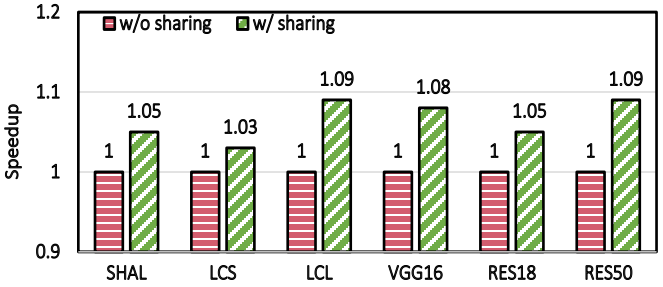

Figure 14. Overall performance: proving n (=100) images

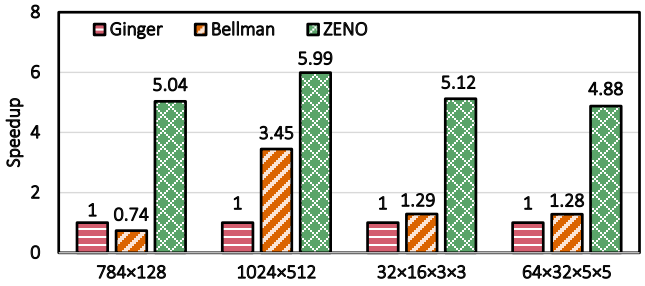

Figure 15. Speedup over Bellman and Ginger

Table 5. Latency measured on Intel Xeon Gold 5218 CPU. Unit: Seconds

Model	Arkworks	ZENO	non-zkSNARK NN
SHAL	5.1	2.1	0.2
LCS	19.6	8.5	0.8
LCL	120	15.3	1.4
VGG16	398	48	4.2
RES18	826	102	8.9
RES50	5440	680	54

as LeNet or 48 seconds for heavy models such as VGG16 to prove his identity without revealing his face image to the access control system. This is a significant improvement in protecting user privacy given the wide deployment of such systems. We note that GPUs can further accelerate zkSNARK by an order of magnitude [27] and may reduce the zkSNARK NN latency to millisecond-level. We leave GPU support as future work.

7.2 Optimization Analysis

In this section, we show speedup from individual ZENO optimizations.

Performance benefits on *circuit computation* step for entire NNs. We show speedup on *circuit computation* step for private images and public weights in Fig. 9. Overall, we achieve speedup of 67.7× on average (from 15× to 150×) for *circuit computation* step. This speedup increases as zkSNARK NN size increases due to our ZENO circuit (§5.1) that reduces quadratic complexity to linear complexity. On individual optimizations, we observe 8.7× speedup from ZENO Circuit (§5.1), 1.2× speedup from frequency-based cache service (§6.1), and 6.2× speedup from workload-specialized parallel scheduler (§5.2). These results show benefits of individual optimizations on reducing zkSNARK NN latency.

We show speedup on *circuit computation* step for private images and private weights in Fig. 10. We have similar observations as the case in private image and public weights. In particular, we observe 9.4× speedup on average (from 2.5× to 24.6×). On individual optimizations, we observe 2.9× speedup from ZENO circuit, 1.1× speedup from frequency-based cache service, and 2.9× speedup from workload-specialized parallel scheduler. Similar to the case in §7.1, this speedup is smaller than the case for private weights and public images. This shows importance of privacy-type driven optimizations (§4) that customize the circuit generation and encoding methods according to privacy types.

Performance benefits on *circuit computation* step at NN layer level. We further show the circuit computation speedup at NN layer level in Fig. 11 and Fig. 12. We focus on the two most time consuming layers – convolution and fully connected layers, under the privacy setting of private images and public weights. We omit the privacy setting of private images and private weights due to page limits. We achieve up to 315.6× speedup on convolution layers and 10.5× speedup on fully connected layers. This result matches up to 150× circuit computation speedup at NN level in Fig. 9. We achieve higher speedup on convolution layers which gain more benefit from ZENO circuit due to the larger number of dot products. We also observe an increasing speedup on both convolution layers and fully connected layers as the layer size increases, thanks to the tensor-driven optimization that reduces computation complexity of *circuit computation* step.

Speedup on *security computation* from knit encoding. We show knit encoding benefits in Fig. 13. We show the result for private weights and public images, as discussed in §4.2. Overall, we achieve up to 3.63× speedup. The reason is that knit encoding can effectively reduce the number of constraints, which decides the latency in *security computation* step. We observe that speedup increases from 1.03× to 3.63×

as NN size increases. The reason is that, in larger zkSNARK NNs, fully-connected, convolution, and pooling layers account for larger portion of *security computation* latency such that knit encoding can bring more benefits.

Benefits from sharing when proving n (=100) images. We show the speedup from batch-specialized constraint system sharing (§6.1) in Fig. 14. While the latency of *circuit computation* step has been significantly reduced, we can still observe 6.5% speedup from this optimization. The reason is that the constraint system represents the computation procedure of a zkSNARK NN with constraints which can be assigned different values for different images.

7.3 Compared with other Frameworks

In this section, we further compare ZENO with two other representative general zkSNARK frameworks – Bellman [62] and Ginger [33]. These two frameworks are general zkSNARK framework and do not provide direct support for zkSNARK NNs. They require constraints (Eq. 1) as inputs and cannot automatically compile arbitrary arithmetic function to constraints. We manually port compiled constraints from ZENO into Bellman and Ginger and compare *security computation* latency. We show results in Fig. 15. We demonstrate the performance on two fully-connected layers with shape [#*in_channels*, #*out_channels*] and two convolution layers with shape [#*out_channels*, #*in_channels*, *kernel_width*, *kernel_height*]. Overall, we observe that ZENO achieves 4.09× speedup over Bellman and 5.26× speedup over Ginger. These benefits come from our NN-tailored optimizations such as privacy-aware knit encoding. Comparing across layers, we observe 1.7× to 6.8× speedup over Bellman and 4.9× to 6× speedup over Ginger. This result demonstrates the consistent benefits from ZENO on various layers.

8 Discussion

Privacy-preserving NN Techniques. To protect diverse aspects of NN privacy, many techniques have been designed. On the training side, MPC [37, 38] enables multiple parties to collaboratively train a NN without sharing training data. Differential privacy [1, 5] prevents extracting sensitive information in training data (e.g., data related to a specific person).

On the inference side, FHE [19, 21, 22] helps private computation outsourcing to remote servers where other persons cannot know the encrypted data or computation results. Instead, zkSNARK NN [24, 25, 28, 44, 46] enables users to generate proof on local machines which could be verified by other persons or companies. This proof could serve as a digital passport, as deployed in World ID.

Practical Applications of zkSNARK NNs. We envision that zkSNARK NN will play a critical role in NN industry given the increasing awareness and regulation of privacy. Existing applications include *World ID* [18] for user identity and *Leela vs the World* [40] for AI chess models. Furthermore, we envision more applications in access control systems where zkSNARK NNs allow users to prove their identity without sharing face images with commercial companies. With ZENO, a laptop can generate proof within 2.1 seconds on a CPU which makes it possible to deploy in access control systems. This latency can be significantly reduced by orders of magnitudes with the help of server GPUs or even mobile GPUs such as Jetson Nano. We leave this as future work. We expect this application to become an industry standard when deploying neural networks in access control systems and more.

9 Conclusion

In this paper, we propose a ZENO (**ZE**ro knowledge **N**eural network **O**ptimizer) framework for efficient zero-knowledge NN inference. Specifically, we design a set of ZENO language constructs to maintain high-level semantics and type information while accommodating a more aggressive compilation from a zkSNARK NN to a gate-level circuit. We then propose several privacy-type driven and tensor-type driven optimizations to further optimize the generated zkSNARK circuit. Finally, we propose NN-centric system optimizations to further accelerate zkSNARK NNs. Extensive experimental results show that ZENO outperforms the state-of-the-art zkSNARK framework across diverse applications.

10 Acknowledgment

We would like to thank Shumo Chu for helpful discussion. We would also like to thank anonymous ASPLOS reviewers, and our shepherd, Mingyu Gao, for helpful feedback. This work was supported in part by NSF-2124039 and CloudBank [47].

References

[1] Martín Abadi, Andy Chu, Ian J. Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *CCS*, pages 308–318. ACM, 2016.

[2] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In *CCS*, pages 2087–2104. ACM, 2017.

[3] Anoma. Anoma network. https://anoma.network/, 2020.

[4] arkworks. arks-snark. https://github.com/arkworks-rs/snark.

[5] Eugene Bagdasaryan, Omid Poursaeed, and Vitaly Shmatikov. Differential privacy has disparate impact on model accuracy. In *NeurIPS*, pages 15453–15462, 2019.

[6] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In *Selected Areas in Cryptography*, volume 3897 of *Lecture Notes in Computer Science*, pages 319–331. Springer, 2005.

[7] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptol. ePrint Arch.*, 2018:46, 2018.

[8] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. Aurora: Transparent succinct arguments for R1CS. In *EUROCRYPT (1)*, volume 11476 of *Lecture Notes in Computer Science*, pages 103–128. Springer, 2019.

[9] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *USENIX Security Symposium*, pages 781–796, 2014.

[10] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for snarks and proof-carrying data. In *Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '13, page 111–120, New York, NY, USA, 2013. Association for Computing Machinery.

[11] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *ASIACRYPT*, volume 2248, pages 514–532, 2001.

[12] Sean Bowe. Bls12-381: New zk-snark elliptic curve construction. https://electriccoin.co/blog/new-snark-curve/.

[13] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. ZEXE: enabling decentralized private computation. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[14] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Gregory Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *IEEE Symposium on Security and Privacy (S&P)*, pages 315–334. IEEE Computer Society, 2018.

[15] Celo. Celo cryptocurrency. https://celo.org/, 2018.

[16] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: an automated end-to-end optimizing compiler for deep learning. In *OSDI*, pages 578–594. USENIX Association, 2018.

[17] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014.

[18] World Coin. Introducing world id and sdk. https://worldcoin.org/blog/announcements/introducing-world-id-and-sdk.

[19] Meghan Cowan, Deeksha Dangwal, Armin Alaghi, Caroline Trippel, Vincent T. Lee, and Brandon Reagen. Porcupine: A synthesizing compiler for vectorized homomorphic encryption. 2021.

[20] Dipankar Das, Naveen Mellempudi, Dheevatsa Mudigere, Dhiraj Kalamkar, Sasikanth Avancha, Kunal Banerjee, Srinivas Sridharan, Karthik Vaidyanathan, Bharat Kaul, Evangelos Georganas, Alexander Heinecke, Pradeep Dubey, Jesus Corbal, Nikita Shustrov, Roma Dubtsov, Evarist Fomenko, and Vadim Pirogov. Mixed precision training of convolutional neural networks using integer operations. In *ICLR*, 2018.

[21] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. EVA: an encrypted vector arithmetic language and compiler for efficient homomorphic computation. In *PLDI*, pages 546–561. ACM, 2020.

[22] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin E. Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. CHET: an optimizing compiler for fully-homomorphic neural-network inferencing. In *PLDI*, pages 142–156. ACM, 2019.

[23] Forrest Davis and Marten Van Schijndel. Recurrent neural network language models always learn english-like relative clause attachment. In *ACL*. Association for Computational Linguistics, 2020.

[24] Boxiang Dong, Bo Zhang, and Hui Wang. Veridl: Integrity verification of outsourced deep learning services. *ECML/PKDD*, 2021.

[25] Boyuan Feng, Lianke Qin, Zhenfei Zhang, Yufei Ding, and Shumo Chu. Zen: Efficient zero-knowledge proofs for neural networks. Cryptology ePrint Archive, Report 2021/087, 2021. https://eprint.iacr.org/2021/087.

[26] Boyuan Feng, Yuke Wang, Tong Geng, Ang Li, and Yufei Ding. APNN-TC: accelerating arbitrary precision neural networks on ampere GPU tensor cores. In *SC*, pages 37:1–37:13. ACM, 2021.

[27] filecoin project. Bellperson. https://github.com/filecoin-project/bellperson.

[28] Zahra Ghodsi, Tianyu Gu, and Siddharth Garg. Safetynets: Verifiable execution of deep neural networks on an untrusted cloud. In *NIPS*, pages 4672–4681, 2017.

[29] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin E. Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *ICML*, volume 48, pages 201–210, 2016.

[30] Jens Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT (2)*, volume 9666 of *Lecture Notes in Computer Science*, pages 305–326. Springer, 2016.

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *ICCV*, pages 1026–1034. IEEE Computer Society, 2015.

[32] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778. IEEE Computer Society, 2016.

[33] Horizen. ginger-lib: a rust library for zk-snarks. https://github.com/HorizenOfficial/ginger-lib.

[34] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 448–456. JMLR.org, 2015.

[35] Xiaoqian Jiang, Miran Kim, Kristin E. Lauter, and Yongsoo Song. Secure outsourced matrix computation and application to neural networks. In *CCS*, pages 1209–1222. ACM, 2018.

[36] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *USENIX Security Symposium*, pages 1651–1669. USENIX Association, 2018.

[37] Marcel Keller and Ke Sun. Secure quantized training for deep learning. In *ICML*, volume 162 of *Proceedings of Machine Learning Research*, pages 10912–10938. PMLR, 2022.

[38] Brian Knott, Shobha Venkataraman, Awni Y. Hannun, Shubho Sengupta, Mark Ibrahim, and Laurens van der Maaten. Crypten: Secure multi-party computation meets machine learning. In *NeurIPS*, pages 4961–4973, 2021.

[39] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research). *Tech report*, 2009.

[40] Modulus Labs. Leela vs the world. https://www.leelavstheworld.xyz/.

[41] O. Labs. Mina cryptocurrency. https://minaprotocol.com/, 2017.

[42] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[43] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998.

[44] Seunghwa Lee, Hankyung Ko, Jihye Kim, and Hyunok Oh. vcnn: Verifiable convolutional neural network. *IACR Cryptol. ePrint Arch.*, 2020:584, 2020.

[45] R. Lidl, H. Niederreiter, P.M. Cohn, G.C. Rota, B. Doran, Cambridge University Press, P. Flajolet, M. Ismail, T.Y. Lam, and E. Lutwak. *Finite Fields*. Number v. 20, pt. 1 in EBL-Schweitzer. Cambridge University Press, 1997.

[46] Tianyi Liu, Xiang Xie, and Yupeng Zhang. zkcnn: Zero knowledge proofs for convolutional neural network predictions and accuracy. In *CCS*, pages 2968–2985. ACM, 2021.

[47] Michael L. Norman, Vince Kellen, Shava Smallen, Brian DeMeulle, Shawn Strande, Ed Lazowska, Naomi Alterman, Rob Fatland, Sarah Stone, Amanda Tan, Katherine A. Yelick, Eric Van Dusen, and James Mitchell. Cloudbank: Managed services to simplify cloud access for computer science research and education. In Joseph Paris, Jackie Milhans, Betsy Hillery, Sharon Broude Geva, Patrick Schmitz, and Robert S. Sinkovits, editors, *PEARC '21: Practice and Experience in Advanced Research Computing, Boston, MA, USA, July 18-22, 2021*, pages 45:1–45:4. ACM, 2021.

[48] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on*

Security and Privacy, pages 238–252. IEEE Computer Society, 2013.
[49] Herbert Robbins and Sutton Monro. A Stochastic Approximation Method. *The Annals of Mathematical Statistics*, 22(3):400 – 407, 1951.
[50] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.
[51] Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In Ran Canetti, editor, *Theory of Cryptography*, pages 1–18, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
[52] Riad S. Wahby, Ioanna Tzialla, Abhi Shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zksnarks without trusted setup. In *IEEE Symposium on Security and Privacy*, pages 926–943. IEEE Computer Society, 2018.
[53] Jonathan Wang. Bn254 for the rest of us. https://hackmd.io/@jpw/bn254.
[54] Xin Eric Wang, Vihan Jain, Eugene Ie, William Yang Wang, Zornitsa Kozareva, and Sujith Ravi. Environment-agnostic multitask learning for natural language grounded navigation. In *ECCV*, 2020.
[55] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng. Quantized convolutional neural networks for mobile devices. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4820–4828, 2016.
[56] Tiancheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In *CRYPTO (3)*, volume 11694 of *Lecture Notes in Computer Science*, pages 733–764. Springer, 2019.
[57] Zhaohui Yang, Yunhe Wang, Kai Han, Chunjing Xu, Chao Xu, Dacheng Tao, and Chang Xu. Searching for low-bit weights in quantized neural networks. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.
[58] Ritchie Zhao, Yuwei Hu, Jordan Dotzel, Christopher De Sa, and Zhiru Zhang. Improving neural network quantization without retraining using outlier channel splitting. In *ICML*, volume 97 of *Proceedings of Machine Learning Research*, pages 7543–7552. PMLR, 2019.
[59] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating high-performance tensor programs for deep learning. In *OSDI*, pages 863–879. USENIX Association, 2020.
[60] Yuqing Zhu, Xiang Yu, Manmohan Chandraker, and Yu-Xiang Wang. Private-knn: Practical differential privacy for computer vision. In *CVPR*, June 2020.
[61] B. Zhuang, L. Liu, M. Tan, C. Shen, and I. Reid. Training quantized neural networks with a full-precision auxiliary module. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1485–1494, 2020.
[62] zkcrypto. Bellman. https://github.com/zkcrypto/bellman.

A Artifact Appendix

ZENO is a type-based optimization framework for accelerating zero-knowledge neural network (zkNN) inference. It exploits the privacy type and tensor type information of zkNN to accelerate the computation of zero-knowledge proof generation. ZENO supports diverse neural networks from small models (e.g., LeNet) to large models (e.g., ResNet-50).

A.1 Artifact check-list (meta-information)

- **Compilation**: Rust 1.43.0
- **Hardware**: Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz and 503 GB DRAM
- **How much time is needed to prepare workflow (approximately)?:** 30 minutes
- **How much time is needed to complete experiments (approximately)?:** 10 hours
- **Publicly available?**: Yes

A.2 Description

A.2.1 How to access. The project is open-sourced at Github [1]. It consists of three parts.

- `Arkworks`: A fork of Arkworks implementation for ZK Proof as the baseline.
- `ZENO-engine`: Implementation of ZENO optimizations.
- `zkNN-circuit`: Implementation of zkNN circuits, including both baseline implementations and optimized implementations with knit encoding.

A.2.2 Hardware dependencies. A consumer laptop is sufficient to generate zk proof for small neural networks such as LeNet. To generate zk proof for large neural networks such as ResNet-50, a machine with large (e.g., 256 GB or more) RAM is necessary.

A.3 Installation & Experiment

- Please follow this instruction to install Rust [2].
- We test the code using `rustc 1.43.0`. Use '`rustup override set 1.43.0`' to specify the rust version for compilation.
- Since many Rust dependency packages may not be backward-compatible, we strongly recommend building the code with the provided Cargo.lock. Please refer to `run.sh` file for details.

A.4 Evaluation and expected results

In `zkNN-circuit/`, we include three directories

- `baseline-one-private/`: Baseline with private image and public weight.
- `baseline-both-private/`: Baseline with private image and private weight.
- `all-optimizations/`: Optimized circuit implementation.

In each directory, please use '`sh run.sh`' to run experiments, which show results in '`result/`' directory. For example, on AWS `c6a.48xlarge`, we expect the circuit computation latency of LeNet-Cifar-Large (LCL) to be 47.8 seconds in '`baseline-one-private/result/`' and 1.1 seconds in '`all-optimizations/result`', leading to a 43.5× speedup. This is slightly better than the reported speedup of 35× in Fig. 9.

[1] https://github.com/BoyuanFeng/ZENO
[2] https://www.rust-lang.org/tools/install