ZKML: An Optimizing System for ML Inference in Zero-Knowledge Proofs

Bing-Jyue Chen
bjchen4@illinois.edu
UIUC
USA

Suppakit Waiwitlikhit
suppakit@stanford.edu
Stanford University
USA

Ion Stoica
istoica@berkeley.edu
UC Berkeley
USA

Daniel Kang
ddkang@illinois.edu
UIUC
USA

Abstract

Machine learning (ML) is increasingly used behind closed systems and APIs to make important decisions. For example, social media uses ML-based recommendation algorithms to decide what to show users, and millions of people pay to use ChatGPT for information every day. Because ML is deployed behind these closed systems, there are increasing calls for transparency, such as releasing model weights. However, these service providers have legitimate reasons not to release this information, including for privacy and trade secrets. To bridge this gap, recent work has proposed using zero-knowledge proofs (specifically a form called ZK-SNARKs) for certifying computation with private models but has only been applied to unrealistically small models.

In this work, we present the first framework, ZKML, to produce ZK-SNARKs for realistic ML models, including state-of-the-art vision models, a distilled GPT-2, and the ML model powering Twitter's recommendations. We accomplish this by designing an optimizing compiler from TensorFlow to circuits in the `halo2` ZK-SNARK proving system. There are many equivalent ways to implement the same operations within ZK-SNARK circuits, and these design choices can affect performance by 24×. To efficiently compile ML models, ZKML contains two parts: gadgets (efficient constraints for low-level operations) and an optimizer to decide how to lay out the gadgets within a circuit. Combined, these optimizations enable proving on a wider range of models, faster proving, faster verification, and smaller proofs compared to prior work.

EuroSys '24, April 22–25, 2024, Athens, Greece

ACM ISBN 979-8-4007-0437-6/24/04…$15.00
https://doi.org/10.1145/3627703.3650088

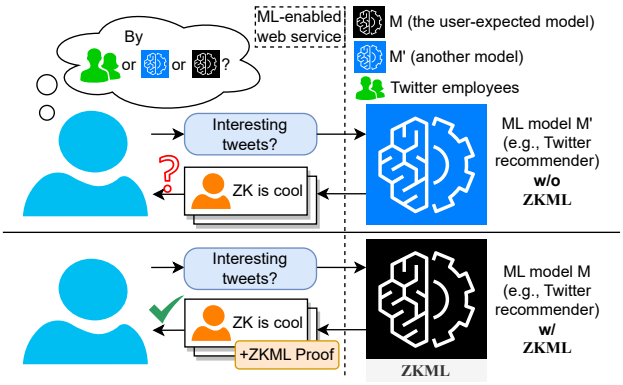


Figure 1. Example use of ZKML in verifying that tweets are ranked honestly from Twitter.

1 Introduction

Machine learning (ML) and artificial intelligence (AI) techniques are becoming increasingly integrated into our world. They now power social media feeds [32], product recommendations [8], medical decisions [5], and even provide day-to-day advice (e.g., ChatGPT) [33]. As these ML systems become increasingly important, there have been increasing calls for transparency [34].

One way to provide this transparency would be to release the model weights and user data. However, the companies deploying these ML models have valid reasons to withhold model weights and data. Releasing certain kinds of data (e.g., medical data) would violate user privacy, and model weights are often trade secrets (e.g., GPT-4 purportedly costs over $100M to train [21]).

To address this, recent work has proposed using zero-knowledge proofs (specifically ZK-SNARKs, zero-knowledge succinct non-interactive arguments of knowledge [6]) to execute ML models without revealing their weights [10, 26, 44]. ZK-SNARKs allow a *prover* to produce a computationally binding proof that some computation happened honestly. In the context of ML, a prover can produce a ZK-SNARK that ensures a specific set of weights were used on some input

known to the prover. For example, ZK-SNARKs could be used to verify that tweets were generated honestly from a fixed recommendation model (Figure 1).

Unfortunately, all prior work on ZK-SNARKs for ML can only produce proofs for unrealistically small models, such as those for MNIST and CIFAR-10 [10, 26, 44]. However, potential users of ZK-SNARKs for ML are largely interested in realistic, production models (e.g., ImageNet-scale models, language models). Furthermore, this prior work only applies to convolutional neural networks (CNNs). This limits the practical applicability of ZK-SNARKs for ML transparency.

In this work, we substantially advance the frontier of possible models, both in terms of diversity and complexity, possible to ZK-SNARK. To do so, we build ZKML, an optimizing compiler that transforms TensorFlow models to ZK-SNARK circuits. ZKML's first component are gadgets that perform basic operations for ML models, such as dot products, softmax, and pointwise non-linearities. Given these gadgets, ZKML's second component is its circuit layouter, which optimizes the circuit layout for a given hardware target. ZKML is designed to be modular, allowing developers to add gadgets.

As mentioned, all of the prior work in this space only applies to CNNs. As such, they only optimize linear layers (convolutions and fully connected layers) and the rectified linear unit (ReLU) non-linearity. These operations are not sufficient for more general models, including recommendation systems, language models, and more.

We design and implement a large number of gadgets that allow for a wider range of models. These gadgets include variable integer division, the maximum operator, the softmax operator, pointwise non-linearities, and more. Combined, these allow ZKML to ZK-SNARK substantially more diverse models, including the mentioned recommendation systems (the Twitter algorithm), language models (DistillGPT-2), small diffusion models, and more. Furthermore, ZKML's gadgets are designed to be optimally laid out for various circuit configurations, resulting in high performance.

Given these gadgets, ZKML still must decide on the circuit size, circuit layout, and which of the gadgets to use. In order to do so, we designed and implemented a circuit layout optimizer. ZKML's optimizer simulates the circuit layout process for a given configuration and uses a cost model to determine which layout is optimal for a given neural network. Compared to using a fixed configuration, ZKML's optimizer can improve performance by up to 131%.

Combined, our optimizations allow ZKML to construct proofs of ML models that are up to 5× larger than prior work, in addition to a wider range of models. Furthermore, ZKML can prove models at a fixed accuracy faster than prior work while achieving up to 5× faster verification and 22× smaller proofs on MNIST and CIFAR-10, the only datasets prior work considers. Although much work remains for widespread deployment, ZKML substantially advances the state-of-the-art of ZK-SNARKs for ML models.

2 Use Cases

Before we describe ZKML, we first describe how ZK-SNARKs can be used in ML applications. All of the applications we describe must be combined with other techniques to be fully secure (e.g., trusted databases [47] or sensors). As we focus on ZK-SNARKs for ML in this work, we simply sketch the end-to-end application. These applications use the same underlying ZK-SNARK technology applied in different ways.

Beyond these applications, ZK-SNARKs can be used to prove any computable function over the ML model outputs. This can include fairness properties or other auditable properties of the ML models. Several of these use cases have been explored in prior work [23, 27].

Trustless audits. ML models are becoming increasingly powerful and trained on private user data. For example, OpenAI has not released the weights for their commercial models (`gpt-3.5-turbo`, `gpt-4`, etc.) at the time of writing. As another example, recommender system models (e.g., as used by Twitter) are trained on private user data, so the model weights cannot be made public. However, users often wish to perform audits over these systems [3, 23, 40], leading to tensions between privacy (of user data) and trust (whether or not the model provider is cheating).

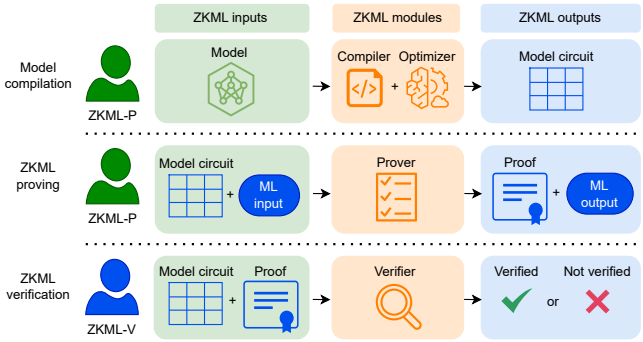
ZK-SNARKs allow the model provider to commit to a fixed model and prove that an output was generated from that fixed model. These ZK-SNARKs can be combined with other techniques, such as verified databases, to perform end-to-end, trustless audits. Such audits have been previously proposed [23], but no work has succeeded in creating ZK-SNARKs of realistic models.

We show a system diagram of how to perform an end-to-end audit in Figure 2.

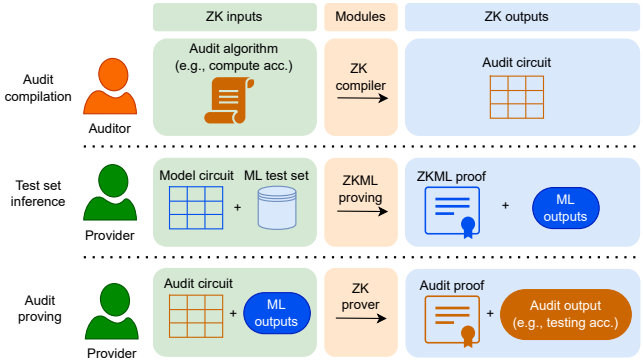
Private biometric authentication. Increasingly, online services want assurances that their users are real people. With the rise of generative AI, this becomes increasingly challenging, and existing biometric authentication solutions violate privacy.

ZK-SNARKs, combined with *attested sensors* (i.e., a sensor that contains a hardware unit to digitally sign the sensor data) [9, 13], make it possible to perform trustless biometric identification. Namely, the user would take a photo (ideally with a depth camera) and use an ML model to verify that their face matches a previously identified face.

Trustless credit scores. Beyond audits of ML systems, ZK-SNARKs for ML (also combined with trusted data access) allow for the computation and verification of trustless credit scoring [30]. Trustless credit scores are particularly interesting in the blockchain setting, where users may wish to draw undercollateralized loans. The credit score can be computed by summarizing the user's on-chain history and executing a machine learning model to determine the credit-worthiness from this history. By using ZK-SNARKs, both the borrower



(a) Diagram of ZKML proving and verification.



(b) Diagram of an end-to-end audit.

Figure 2. Diagram of using ZKML in an audit.

Term	Definition
Circuit	Representation of computation within the ZK-SNARK.
Lookup table	Injective mapping of input values to output values (similar to a hash map).
Constraint	Restriction on the values of the grid.
Selector	Variable used to determine which polynomial constraint to apply to a row.
Polynomial constraint	Enforces a polynomial of a row of the grid is zero.
Copy constraint	Enforces two cells of the grid are equal.
Lookup constraint	Enforces a tuple of cells is in a lookup table.

Table 1. List of terms used to describe ZK-SNARK circuits in this work.

and lender can be assured that the credit score was computed honestly.

3 ZK-SNARKs and ML

In this work, we focus on the `halo2` proving system, which supports the Plonkish randomized AIR (arithmetic intermediate representation) with preprocessing [4, 12]. A full description of the proving system is outside of the scope of this manuscript, but we describe properties salient to the performance of ML circuits.

Circuit representation. Logically, we can view circuits as 2D grids where the values of the cells are in some large prime, finite field, $\mathbb{F}_p$. The number of rows *must be a power of two*. The cells are constrained in three ways:

1. Enforcing that an arbitrary polynomial where the variables correspond to the cells within one or more adjacent rows evaluates to zero (custom gates/constraints).
2. Enforcing that arbitrary cells within the grid are equal (copy/permutation constraint).
3. Enforcing that a specific pattern of cells within a row is within some table (lookup constraint).

We provide a list of terms in Table 1.

Representing computations. ZK-SNARKs can represent arbitrary computations: this can be seen as polynomial constraints and equality checks are universal [42]. However, there are many ways to represent the same computation within a ZK-SNARK.

As a concrete example, consider performing the ReLU operation, which is the function $f(x) = \max(0, x)$. Assume that x is represented using fixed-point with b total bits in the representation. One common way that prior work represents the ReLU is by performing a full bit decomposition using polynomial constraints, setting the sign bit to zero, and recomposing x (with the sign bit set to zero) [27]. This method of computing ReLU does not require any lookup arguments, but requires $b + 2$ cells.

We can perform the ReLU using only two cells with a lookup table, where the table consists of two columns with $(x, f(x))$. This table has at least 2^b rows.

Although the second representation uses fewer cells, the relative costs of both methods depend heavily on the number of ReLU operations performed. If only a single ReLU operation is performed, the cost of the table may be higher than the cost of using the expanded representation.

As we will see, the choice of representing computation affects costs dramatically.

Performance. The dominant cost for ZK-SNARKs is the proving time: as we will show, verification is orders of magnitude cheaper than proving.

Many factors affect the proving latency for `halo2`. Roughly, the proving latency scales with the total number of rows and columns, when accounting for the selector columns, the lookup columns, the total number of constraints, and the maximum degree of all of the constraints. However, the operations done internally include polynomial operations, Fast Fourier Transformations (FFTs), and Multi-Scalar Multiplications (MSMs) over large finite fields or elliptic curves. As such, it is difficult to predict the proving latency exactly, but this can be benchmarked relatively easily as the operations are the same for a fixed circuit configuration.

	ZEN	vCNN	zkCNN	ZKML
CNN	✔	✔	✔	✔
CNN training	✗	✗	✗	✔
GPT	✗	✗	✗	✔
Twitter model	✗	✗	✗	✔
Diffusion	✗	✗	✗	✔

Table 2. List of models supported by ZKML and by prior work.

	ZEN	vCNN	zkCNN	ZKML
Conv2D	✔	✔	✔	✔*
FullyConnected	✔	✔	✔	✔*
Pooling	✔	✔	✔	✔*
ReLU	✔	✔	✔	✔
DepthwiseConv2D	✗	✗	✗	✔
BatchMatMul	✗	✗	✗	✔
Softmax	✗	✗	✗	✔
Other non-linearities	✗	✗	✗	✔

Table 3. A non-exhaustive list of layers supported by ZKML and by prior work. The * indicates that ZKML provides additional optimized implementations through various combinations of low-level gadgets. Many of these layers, such as BatchMatMul and Softmax are required for modern ML models, such as GPT.

To understand the complexities of performance optimization for `halo2` circuits, consider a toy circuit with 10 columns and 2^{20} rows where $2^{20} - 100$ of the rows are occupied by arithmetic operations (i.e., arithmetic constraints are already present). Let us further assume that there is a single ReLU operation that must be performed with 16 bits of precision. Consider three ways of performing this single ReLU:

1. Defining a new selector with a lookup table with the ReLU inputs/outputs. This adds three columns (the selector, input lookup column, and output lookup column) and an additional constraint, and takes up one additional row.
2. Defining a new selector that spans four rows and performs the bit-decomposition. This adds one column (the selector) and an additional constraint, and takes up four additional rows.
3. Using the existing arithmetic constraints to perform the bit decomposition. Assuming one bit per row, this takes up an additional 32 rows.

In our toy example, surprisingly, the *third* method is the most efficient. However, if we instead performed 2^{18} ReLU operations, the table may be more efficient: it takes up substantially fewer rows (the second and third methods would take 2^{20} and 2^{23} rows, respectively).

Thus, the most efficient circuit layout depends *globally* on the operations within the ML model.

4 ZKML Architecture and Overview

We describe ZKML's high-level architecture in this section.

4.1 Overview

Recall that ZKML compiles ML model specifications to ZK-SNARK circuits. To do so, ZKML uses a standard method of representing ML model computation: data are tensors (n-dimensional arrays) and operations (i.e., layers) take as inputs tensors and outputs tensors. In this work, we represent all values of the tensors as fixed-point numbers, where ZKML chooses the scale factor. Choosing the scale factor appropriately is critical for high performance.

ZKML contains two main components: low-level gadgets and an optimizer to select the circuit layout for a given ML model. The optimizer composes the gadgets to form the layers. ZKML is designed to be modular, allowing developers to add gadgets for increased flexibility and performance. Furthermore, its optimizer can target different hardware backends, such as servers with a varying number of CPUs and RAM. We show the overall architecture diagram of ZKML in Figure 3.

ZKML supports a much wider range of models compared to prior work (Table 2). Currently, ZKML supports CNNs, LSTMs, Transformers, MLPs, diffusion models, and other commonly used ML models. Because `halo2` is universal (i.e., can represent any computation) it is hypothetically possible to support any ML model. However, ZKML currently does not support branching or variable-length loops. Thus, ZKML requires fixed-length inputs for NLP models. Loops and branches will be unrolled.

4.2 Components

ZKML contains several components, including low-level gadgets, an optimizer, and a transpiler from TensorFlow.

The reason that ZKML can support a wide range of models is due to our implementation of low-level gadgets that can be composed to compute a variety of ML layers. We show a comparison of prior work and ZKML in Table 3. ZKML's low-level gadgets broadly fall under four categories:

1. Shape operations
2. Arithmetic operations
3. Pointwise non-linearities
4. Specialized operations

We describe the gadgets in detail in Section 5. However, we highlight one design choice to "future-proof" ZKML: each constraint for a gadget is only within a single row. Although the existing `halo2` library allows for constraints across rows, we limit ZKML to single-row constraints because new forms of proving systems (that have not yet been integrated into `halo2`) only allow single-row constraints. Furthermore, we have found that this does not significantly affect performance.

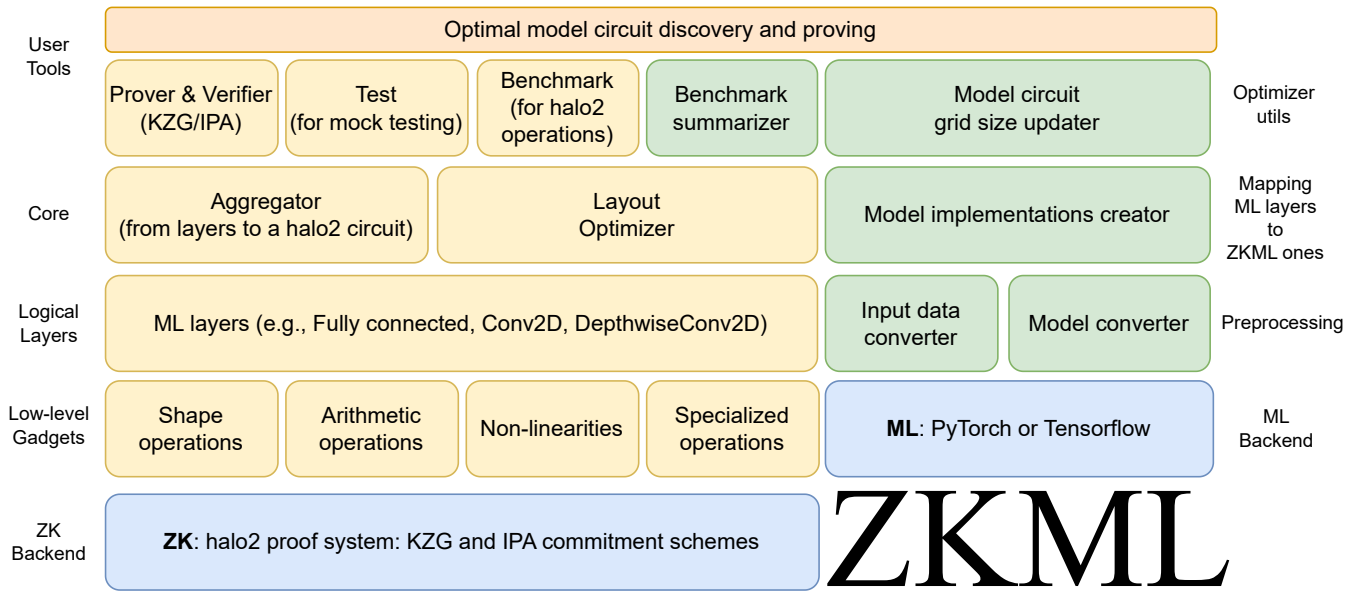


Figure 3. ZKML architecture diagram. ZKML is built on the `halo2` proving system. The `halo2` related submodules implemented in Rust are colored in yellow. The submodules implemented in Python are colored in green, which are largely ML-related. We provide a simple bash interface for users to find and prove optimal circuit with ZKML, which is colored in orange.

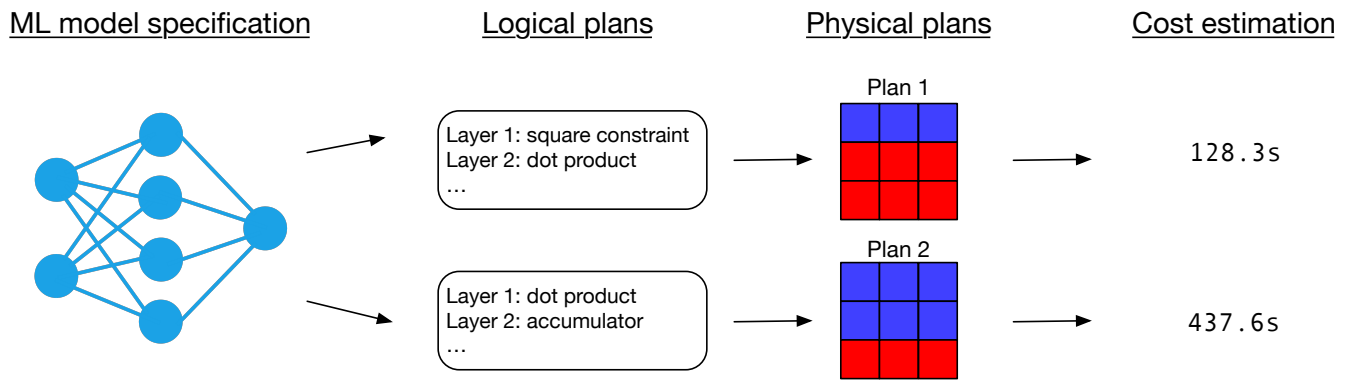


Figure 4. ZKML optimizer architecture diagram. The ZKML optimizer compiles ML model specifications to logical plans, lays out physical plans, and chooses the optimal physical plan based on cost estimates.

ZKML's optimizer takes as input an ML model specification and outputs an optimized circuit layout. To do so, ZKML composes the low-level gadgets into layers. As we describe in Section 7, ZKML must choose which gadgets to use, the size of the circuit, and the layout. These choices are dependent on the hardware target.

We show a high-level architecture diagram of ZKML's optimizer in Figure 4.

4.3 Security

ZKML inherits all of the assumptions and security properties of the underlying proving system. In particular, it inherits all security properties of `halo2`, including zero-knowledge, completeness, and knowledge soundness [6]. Importantly, the knowledge soundness property intuitively means that the prover must know the input and the weights.

`halo2` supports two underlying commitment schemes: KZG [20] and IPA (inner-product argument) [7]. The security assumptions depend on the underlying commitment scheme we use, but only require general, widely used cryptographic assumptions.

The KZG commitment scheme requires a one-time, universal trusted setup (i.e., a single trusted setup that can subsequently be used for *all* models and proofs). This trusted setup has been completed in a distributed manner by Privacy Scaling Explorations [35] for 2^{28} with over 75 participants.

We use this trusted setup in our work, which removes the need for users of ZKML to run their own trusted setup. They can also contribute to the existing trusted setup for further security.

The IPA commitment scheme is transparent, meaning that there is no trusted setup required. However, the IPA commitment schema requires larger proofs and higher verification time. Users of ZKML can choose the KZG or IPA version of ZKML depending on the security properties they desire.

4.4 Limitations

As with all prior work for ZK-SNARKs and ML, we are limited by the memory consumption and hardware resources for reasonable proving times. The largest model we can prove with under 1TB of RAM is the distilled version of GPT-2 we consider in Section 9. Furthermore, we do not focus on proofs of training. Nonetheless, ZKML is capable of proving a much larger variety of models and much larger models compared to prior work. We believe the path towards higher performance largely lies in creating more efficient proving systems.

ZKML requires that the model architecture (but not weights) is revealed. Given the recent trends in open-source models (e.g., variations of Llama), we do not believe this is a significant constraint for many applications.

Another major concern for all ZK-SNARKs is that the representation of the computation within the ZK-SNARK is equivalent to the original computation (i.e., correctness of the circuit). A formal proof of correctness is outside the scope of this work.

5 ZKML Gadgets

We now describe ZKML's gadgets.

5.1 Overview

ZKML contains gadgets that are designed to represent the majority of common operations in widely used ML models. Broadly, these gadgets fall under four categories: shape operations, arithmetic operations, pointwise non-linearities, and specialized operations. We describe these gadgets below.

Shape operations. Because tensors can hold references to previously assigned cells, shape operations can be done logically. Namely, any operation that changes the shape, subsets, or concatenates one or more tensors can be performed by creating a new tensor with references to previously created tensors. These operations are "free" with respect to the proving time since new cells are not assigned.

Arithmetic operations. There are many arithmetic operations that are commonly performed in ML models, including addition, multiplication, division, and squaring. These operations are typically a part of a larger computation, such as a convolution or fully connected layer.

Name	Operation
Add(x, y)	$x + y$
Sub(x, y)	$x - y$
Mul(x, y)	$x \cdot y$
Div(x, y)	$\lfloor x/y \rfloor$
DivRound(x, y)	Round(x/y)
Square(x)	x^2
SquaredDiff(x, y)	$(x - y)^2$
Sum$(\vec{x})$	$\sum_i x_i$
DotProd$(\vec{x}, \vec{y})$	$\vec{x} \cdot \vec{y}$

Table 4. Partial list of arithmetic operations ZKML supports.

An important consideration for performance is how to implement these operations. For example, suppose we have a dot product constraint. This constraint could be used to implement pairwise addition and multiplication of tensors, in addition to squaring a tensor element-wise. However, all of these operations (addition, multiplication, squaring) could also be implemented with fewer cells using additional constraints.

We implement all of these operations (and more) as gadgets that the optimizer can choose from. A partial list of arithmetic gadgets is in Table 4.

A final consideration in implementing these gadgets is to account for the fixed-point representation of values within the finite field. As such, multiplication and division operations must account for the fixed-point scaling factor. In particular, we can implement division with a multiplication and rescaling.

Pointwise non-linearities. Another common set of operations within ML models are pointwise non-linearities. These are commonly referred to as activation functions. Pointwise non-linearities include the ReLU, ELU, sigmoid, exponential, and tanh functions. With the exception of the ReLU function, these functions are difficult to approximate with polynomial constraints efficiently.

Thus, to efficiently perform pointwise non-linearities, we use lookup tables. Since the table size can be at most the grid length, the range of inputs to the non-linearities constrains the precision of the fixed-point representation. As we will see, this constraint is important for the optimizer to decide the grid size.

Specialized operations. We refer to all other operations as specialized operations as they require custom constraints. In this work, all specialized operations arise from the need to perform the softmax operation. As we will see, a high-performance softmax requires the maximum operation, a scaled exponential operation, and a variable division operation. In this section, we describe how to perform the individual operations and describe how to combine them into a high-performance softmax in Section 6.

First, consider the maximum operation: $c = \max(a, b)$. In order to constrain c to be the maximum, we first constrain c to be equal to one of a or b using the polynomial constraint $(c - a)(c - b) = 0$. We can then constrain c to be greater than or equal to both a and b using two lookup constraints: $c - a \in [0, ..., N)$ and $c - b \in [0, ..., N)$. The table $[0, ..., N)$ is required for the pointwise non-linearities so it can be reused for the maximum operator.

Second, the scaled exponentiation refers to the operation $y = \exp(x) \cdot \text{SF}$, where SF is the scale factor. This can be done in the same way as the pointwise non-linearities are done.

Third, consider variable division: $\text{VarDiv}(a, b) = \text{Round}(b/a)$, where $a, b \in [0, ..., N)$. Here, b and a are unknown ahead of time (unlike the fixed-point scaling factor). In order to perform variable (rounded) division, we first describe how to perform standard integer division. Standard integer division is equivalent to the following equation holding:

$$b = c \cdot a + r$$

for $r \in [0, ..., a)$. Thus, we can use the polynomial constraint $b - c \cdot a - r = 0$ and constrain that $a - r \in [0, ..., N)$. To perform rounded division, we note that $c = \text{Round}(b/a)$ is equivalent to

$$c = \left\lfloor \frac{2b + a}{2a} \right\rfloor = \left\lfloor \frac{b}{a} + \frac{1}{2} \right\rfloor$$

but the first equation is equivalent to standard integer division with $2b + a$ as the numerator and $2a$ as the denominator. Finally, we note that if a is larger than N, we can decompose a into "limbs" of size 2^N to allow for higher precision. r can similarly be decomposed.

5.2 Examples

We now provide concrete examples of gadgets. Throughout this section, denote the number of columns as N.

Sum. The first example we consider is the sum of a fixed size vector $\text{Sum}(\vec{x}) = \sum_i^n x_i$, where $n = N - 1$. We can lay out the elements of the vector and the result $z = \text{Sum}(\vec{x})$ in a row as follows:

$$x_1 | \cdots | x_n | z.$$

The constraint is:

$$z - \sum_i^n x_i = 0.$$

Dot product without bias. Consider a dot product of fixed size without a bias. Namely,

$$\text{DotProd}(\vec{x}, \vec{y}) = \sum_i^n x_i \cdot y_i.$$

For the gadget, we let $n = \lfloor \frac{N-1}{2} \rfloor$.

To compute the dot product, we lay out $\vec{x}$ and $\vec{y}$ and the result $z = \text{DotProd}(\vec{x}, \vec{y})$ as follows:

$$x_1 | \cdots | x_n | y_1 | \cdots | y_n | z$$

If N is even, we leave a cell empty. The constraint is simply:

$$z - \sum_i^n x_i \cdot y_i = 0.$$

Suppose we had two vectors $\vec{x}$ and $\vec{y}$ of cardinality $m > n$. We can decompose the overall dot product into $\lceil \frac{m}{n} \rceil$ dot products. We can then use the sum gadget from above to add the partial results. As we will see, there are many ways to perform a large dot product.

Dot product with bias. Consider a dot product of fixed size with a bias: $\text{DotProd}(\vec{x}, \vec{y}, b) = b + \sum_i^n x_i \cdot y_i$. Here, $n = \lfloor \frac{N-2}{2} \rfloor$. We can lay out the row as follows:

$$x_1 | \cdots | x_n | y_1 | \cdots | y_n | b | z$$

and use the constraint

$$z - b - \sum_i^n x_i \cdot y_i = 0.$$

In order to compose a larger dot product, we can decompose the dot product into $\lceil \frac{m}{n} \rceil$ dot products with biases. The first bias is set to zero and the remainder of the biases are set to the accumulation. This method of computing a larger dot product does not require the sum gadget.

As shown in this example, there are a number of ways to perform the same operation. The efficiency will depend on a large number of factors, including the total size of the circuit and the size of the dot products.

ReLU. As a final example, consider computing the ReLU function pointwise over a vector $\vec{x}$. Here, $|\vec{x}| = \lfloor \frac{N}{2} \rfloor$. We can simply lay out the row as

$$x_1 | \text{ReLU}(x_1) | \cdots | x_n | \text{ReLU}(x_n)$$

The constraints ensure that pairs of columns $(x_i, \text{ReLU}(x_i)) \in T$ for a table T that contains the domain and range of the ReLU function. Other pointwise non-linearities can be performed similarly.

6 ZKML Layers

We now describe how ZKML implements ML layers.

6.1 Overview

Given low-level gadgets, ZKML will compose these into implementing ML model layers within a given circuit. ZKML's optimizer will decide on the specific layout and choices of gadgets for a given ML model. However, we first describe several concrete instantiations and optimizations for commonly used ML layers.

ZKML currently supports 43 layers, which broadly fall under linear layers, arithmetic layers, activation layers, and softmax. Several of these layers can be computed as compositions of base layers. We show a non-exhaustive list of layers in Table 3.

Linear layers. In this work, we refer to a linear operation as a layer that takes one or more tensors as input and outputs a tensor composed of linear combinations of the input (potentially with a bias term). Linear layers include convolutional layers, fully connected layers, and batched matrix multiplication layers.

Consider a fully connected layer, which computes a matrix multiplication of W (the weights) and A (the input). Naively computing the matrix multiplication takes $O(n^3)$ operations. We can perform naive matrix multiplication by computing each element of the result (B) by performing the dot product of W_i and A_j^T.

However, we can perform matrix multiplication asymptotically more efficiently by using Freivalds' algorithm to verify matrix multiplication [11]. In particular, we can compute B "outside" of the circuit and simply verify that $B = WA$. To perform the verification, we can take a random vector r and verify that $Br = WAr$, which is $O(n^2)$. Similarly, the result can be computed with a series of dot products. The random vector r must be generated after the matrix and results are committed.

Other linear layers (convolutions, batched matrix multiplication) can be accelerated with Frievald's algorithm as well.

Arithmetic layers. Arithmetic layers perform arithmetic operations between two or more tensors. These operations include addition, subtraction, multiplication, division, and computing the squared difference between two tensors.

These layers can be implemented with custom gadgets or by repurposing the dot product gadget. The overall efficiency within the circuit depends on the relative costs of adding an extra constraint compared to the inefficiency of using the dot product gadget.

Activation layers. Activation layers apply pointwise non-linearities to tensors. As we described in Section 5, all of the pointwise non-linearities in this work (with the exception of ReLU) are difficult to approximate with polynomial constraints. As such, they require lookup tables.

Softmax. The softmax function is a non-linear, vector-valued function, which computes $y_i = \frac{e^{x_i}}{\sum e^{x_i}}$. Because the softmax is a vector-valued function, it would require a lookup table that is unrealistically large (for a vector of size n, it would require a table of size SF^n).

To compute the softmax, we use a standard trick for numeric stability. Namely, we compute the softmax of $x_i - \max_j x_j$. This rescaling reduces the range of the exponentials and produces the same result as the softmax is shift invariant. We can compute the maximum of the vector x using the maximum gadget described in Section 5.

Given the exponentiated vector, we can compute the sum and divide by the sum. However, naively doing this will result in catastrophic loss of precision. To understand why, it is clear that the sum of e^{x_i} is greater than each e^{x_i}. Using the standard integer division gadget will result in all values except potentially one to be rounded to zero.

To address the numerical instability we could divide the sum by the fixed-point scale factor. However, this will also result in reduced precision. Instead, we scale the numerator by the scale factor. As we show, these optimizations combined lead to a high-performance softmax.

6.2 Examples

We provide an example of composing gadgets to produce layers to illustrate the choices ZKML's optimizer must make. Consider a simple fully connected layer with a bias. The inputs are matrices A and B, and a bias b. The output is the result $C = A \cdot B + b$ when computed in fixed-point. For simplicity, let B be a vector (so the operation is instead a matrix-vector multiplication).

We can decompose the matrix multiplication into a series of dot products by computing the dot product of the rows of A with B. Assuming the number of columns A is larger than the number of columns in the circuit, we must split the dot product across rows. To do so, we can use the dot product without a bias, accumulate, and add the bias using the addition gadget. However, we can also use the dot product with a bias and have the first bias be b. The optimal choice of gadget depends on the size of the matrices and the size of the circuit.

We must further perform the fixed-point scaling after the multiplication. If this is followed by a non-linearity, we can fuse these operations.

As we can see, there are a large number of choices when compiling even a single layer from gadgets.

7 ZKML Optimizer

We now describe ZKML's optimizer.

7.1 Overview

ZKML's optimizer takes as input an ML model specification and outputs an optimized `halo2` circuit layout. As described in Section 4, ZKML currently takes fixed-function ML models. Operations such as branching and looping will be unrolled.

Given a set of layers in an ML model, ZKML has many choices of circuit layout. Fully profiling each possible layout is infeasible, especially for larger ML models: producing a single proof could take hours.

Unfortunately, it is also difficult to choose the optimal layout since individual choices affects the global performance. As a simple example, because the number of rows in a circuit must be a power of two, increasing the number of rows to reduce the number of columns can have dramatic effects.

Thus, instead of exhaustive profiling, ZKML instead performs the following procedure to optimize circuit layout:

Algorithm 1 Optimizer for ZKML

```
Input : ML Model M, Operation statistics stat
Output : Best physical layout B
stat ← BenchmarkOperations(hardware) ▷ Once only
function OptimizeLayout(M, stat)
    L ← GenerateLogicalLayouts(M)
    B ← NULL
    cost ← ∞
    for ℓ in L do
        nCols ← N_min
        while nCols ≤ N_max do
            b ← GeneratePhysicalLayout(ℓ, nCols)
            k ← FindOptimalK(b, nCols)
            b̂ ← UpdatePhysicalLayout(b, nCols, k)
            T ← EstimateCost(b̂, nCols, k, stat)
            if T < cost then
                cost ← T
                B ← b̂
            end if
            nCols ← nCols + 1
        end while
    end for
    return B
end function
```

1. ZKML will generate candidate choices of gadgets based on the operations within the ML model.
2. ZKML will generate circuit layouts of varying grid sizes from the gadget choices, where the number of rows is optimal relative to the number of columns.
3. ZKML will use cost estimation to determine the highest performance layout.

At all steps, ZKML will use heuristics to prune suboptimal plans.

In the first step, ZKML will generate *logical* circuit layouts, which specify how individual layers should be implemented but not the physical instantiation. For example, it may specify to use an accumulation gadget but not the number of columns.

In the second step, ZKML will generate *physical* circuit layouts, which specify exact grid sizes and layouts.

We show the overall algorithm in Algorithm 1 and describe these steps in turn.

7.2 Generating logical layouts

The first step simply generates a candidate list of laying out individual layers within the circuit. For example, the linear layers could be implemented with a separate aggregation constraint or with just the dot product constraint. Similarly, the squared layer can be implemented with a separate constraint or with the multiplication constraint.

An exhaustive enumeration of layer implementations would result in exponentially many configurations with the depth of the neural network. To reduce this, ZKML uses a heuristic pruning method that enforces the same implementation for every layer per configuration. This is because adding a constraint is more expensive than adding a column, and the gains from using separate implementations are rarely worth the cost of adding a column.

7.3 Generating physical layouts

The second step is for ZKML to generate physical instantiations of the logical layouts. Namely, for each logical layout, ZKML will generate physical circuit layouts while varying the number of columns. Because the number of rows must be 2^k, ZKML will only keep the grids with a minimal number of rows for each k (per configuration).

In order to generate grids, we implemented a circuit simulator which produces row-exact simulations of circuits given a logical layout and number of columns. Using this simulator, ZKML can exactly compute the number of rows needed at a given number of columns.

7.4 Cost estimation

Given a set of physical layouts, ZKML will perform cost estimation on the remaining layouts for the giving proving hardware. Performing exact cost estimation is difficult because many kinds of computational operations are involved in producing a proof. However, the dominant cost of proof generation are Fast Fourier Transformations (FFTs) and Multi-Scalar Multiplications (MSMs). Besides these operations, the creation of lookup columns and the calculation of quotient polynomial are the dominant factors in proving time.

As a result, we focus on estimating the costs of these four operations. The cost of these operations depends on three factors: the size of the input, the number of times the operations are done, and the hardware resources available. For example, much work has been done to accelerate MSMs on hardware accelerators [17, 29].

For a given proving hardware configuration, ZKML requires costs estimates for: 1) a single FFT of size 2^k for $k \in \{18, ..., 30\}$, 2) a single MSM of size 2^k for $k \in \{18, ..., 28\}$, 3) the creation of a lookup table of size 2^k for $k \in \{18, ..., 28\}$, 4) the time to perform a single field multiplication and addition. This is because the existing trusted setup only supports circuits with at most 2^{28} rows. These estimates need only be produced once per hardware configuration.

The size of each individual FFT depends both on the maximum degree of the custom constraints and the number of rows. The number of FFTs depends on the number of columns and copy constraints. We can compute these statistics given a physical circuit layout. Namely, there are two sizes of FFTs performed for a given physical circuit layout, so we can

estimate the cost as:

$$C_{\text{FFT}} = n_{\text{FFT}} \cdot t_{\text{FFT}}(k) + n'_{\text{FFT}} \cdot t_{\text{FFT}}(k'). \quad (1)$$

Here, k is such that the number of rows is 2^k and $k' = k + \log_2(d_{\max} - 1)$, where $d_{\max}$ is the maximum degree of the constraints in the circuit. The FFTs corresponding to n_{FFT} and n'_{FFT} are for computing the quotient polynomial in `halo2` [45]. Specifically, n_{FFT} is for converting each column polynomial/constraint to coefficient form, and n'_{FFT} is for converting each column polynomial/constraint to expanded evaluation form. They can be computed from the number/type of columns in the circuit, which includes the number of permutation arguments, lookups, and maximum degree of the custom gates. If we denote the number of instance (public) columns as N_i, the number of advice (private) columns as N_a, the number of lookups as N_{lk}, and the number of permutation constraints as N_{pm}, then the number of FFTs is

$$n_{\text{FFT}} = N_i + N_a + N_{\text{lk}} * 3 + \frac{N_{\text{pm}} + d_{\max} - 3}{d_{\max} - 2} \quad (2)$$

and $n'_{\text{FFT}} = n_{\text{FFT}} + 1$, in which the additional one is to transform the overall evaluation form back to the coefficient form.

The size of each individual MSM depends on the number of rows, and the number of MSMs depends on the number of columns (including selector columns) and the maximum degree $d_{\max}$. Similarly, we can compute these statistics given a physical circuit layout. The number of MSMs is given by $n_{\text{FFT}} + d_{\max} - 1$ for the KZG commitment scheme and $n_{\text{FFT}} + d_{\max}$ for the IPA commitment scheme. Compared to FFTs, the additional terms come from the generation of evaluation proofs for the quotient polynomial.

Finally, we can estimate the residual cost by estimating the cost of constructing the lookup tables and assorted field element operations. Our estimate for the residual cost is simply the sum of these two estimates.

Given the cost estimates for each of the physical circuit layouts, ZKML will choose the cheapest layout for proving.

8 Implementation

We implemented ZKML in ˜16,000 lines of Rust and Python code. The Rust code contains the gadgets, circuit layouter, and proving aspects. The Python code contains the logic for the optimizer. We also add semantic-preserving optimizations to the `halo2` proving stack, including optimizing the lookup constraint creation, parallelization of FFTs, and reduced memory requirements.

From a user perspective, there are two stages: optimization and proving. In the optimization step, the user provides an ML model specification. Currently, ZKML accepts models in `tflite` format; this could easily be extended to other formats such as `onnx`.

Models	Parameters	Flops
GPT-2	81.3M	188.9M
Diffusion	19.5M	22.9B
Twitter	48.1M	96.2M
DLRM	764.3K	1.9M
MobileNet (ImageNet)	3.5M	601.8M
ResNet-18 (CIFAR-10)	280.9K	81.9M
VGG16 (CIFAR-10)	15.2M	627.9M
MNIST	8.1K	444.9K

Table 5. List of models considered in the evaluation.

Given the ML model specification, ZKML will pick the appropriate set of gadgets and physical layout with its optimizer. The optimizer will produce the optimal gadgets and layout. It will also produce the proving key and verification key, which is specific to the model. Then, to produce a proof, the user must also supply the input.

In order to verify a proof, the user must provide the model configuration (but not the weights), the verifying key, the proof, and public values. Currently, the verifier is implemented as a standalone binary, but could also be implemented in wasm or other frameworks.

Our code is available at https://github.com/uiuc-kang-lab/zkml.

9 Evaluation

We evaluate ZKML on a wide range of modern ML models, spanning language models, CNNs, and recommendation system models. Our results show that ZKML is able to ZK-SNARK a wider range of models than prior work. Finally, we show that all components of ZKML are necessary for high performance.

9.1 Experimental Setup

In our evaluation, we consider the following models:

1. GPT-2: a distilled GPT-2 optimized for inference [39].
2. Diffusion: a small latent text-to-image Stable diffusion model [37].
3. Twitter: MaskNet in the Twitter recommendation system [43].
4. DLRM: a deep recommender proposed by Facebook research [32] and used by MLPerf [36].
5. MobileNet: a MobileNet v2 trained on ImageNet [38]. We used the "1.0, 224" configuration that has an expansion factor of 1.0 and an input resolution of 224.
6. ResNet-18: ResNet-18 on CIFAR-10 [14].
7. VGG-16: VGG-16 on CIFAR-10 [41].
8. MNIST: A CNN on MNIST optimized for accuracy [1].

We show the number of parameters and the flops for each model in Table 5.

In choosing the models in the evaluation, we note an important feature of ML models: performance is dependent on

Model	Proving time (KZG)	Verification time (KZG)	Proof size (KZG)
GPT-2	3651.67 s	18.70 s	28128 bytes
Diffusion	3600.57 s	92.78 ms	28704 bytes
Twitter	358.7 s	22.41 ms	6816 bytes
DLRM	34.4 s	12.26 ms	18816 bytes
MobileNet	1225.5 s	17.67 ms	17664 bytes
ResNet-18	52.9 s	11.84 ms	15744 bytes
VGG16	637.14 s	9.62 ms	12064 bytes
MNIST	2.45 s	6.69 ms	6560 bytes

Table 6. End-to-end proving time for a variety of models when using the ZKML KZG backend.

Model	Proving time (IPA)	Verification time (IPA)	Proof size (IPA)
GPT-2	3949.60 s	11.98 s	16512 bytes
Diffusion	3658.77 s	5.17 s	30464 bytes
Twitter	364.9 s	2.28 s	8448 bytes
DLRM	30.0 s	0.11 s	18816 bytes
MobileNet	1217.6 s	3.34 s	19360 bytes
ResNet-18	46.5 s	0.20 s	17120 bytes
VGG16	619.4 s	2.49 s	17184 bytes
MNIST	2.36 s	22.26 ms	7680 bytes

Table 7. End-to-end proving time for a variety of models when using the ZKML IPA backend.

a large number of factors. These factors include the architecture, amenability to quantization, and others. Due to the flexibility of ZKML, we are able to ZK-SNARK a wider range of models. Prior work focuses on older, outdated models such as VGG-16, which are inefficient in terms of accuracy, ML statistical efficiency, and ZK-SNARK proving time. Instead, we focus on modern models, such as ResNet-18, which achieve higher accuracy and faster proving. As a result, we focus on a key metric of proving time at a given accuracy level.

For consistent benchmarks, we use the `r6i.8xlarge` AWS EC2 instance type for all models except MobileNet, Diffusion, and GPT-2. We use the `r6i.16xlarge` instance for MobileNet, and the `r6i.32xlarge` for GPT-2 and Diffusion due to memory overheads. The `r6i.8xlarge` instance has 32 vCPU cores (threads) and 256 GB of RAM. The `r6i.16xlarge` instance has 64 vCPU cores (threads) and 512 GB of RAM. The `r6i.32xlarge` has 128 vCPU cores and 1 TB of RAM.

9.2 End-to-End Numbers

ZKML proves on a wider range of models. For the models described in Section 9.1, we measure the latency of proving for each model on AWS EC2 instances. We show results in Table 6 for the KZG backend and Table 7 for the IPA backend. As we show, ZKML can prove model types beyond prior work, including real-world models used in recommendation systems, ResNets, and even a distilled GPT-2. The proving time can be as low as 2.5s. The largest model we consider, GPT-2, can be proven in about an hour. In contrast, some prior work can take over an hour to prove small models on CIFAR-10.

Model	FP32 Accuracy	ZKML Accuracy	Difference
MNIST	99.06%	99.06%	0%
VGG16	90.36%	90.37%	+ 0.01%
ResNet-18	91.88%	91.87%	-0.01%

Table 8. Accuracy of ZKML compared to the base FP32 models.

We also show the latency of verification and the proof size in the above tables. As shown, IPA usually has a larger proof size and higher verification time compared to KZG. KZG has faster verification as it only requires a single pairing check, while IPA needs to perform $O(n)$ group operations for verification. IPA generally has faster proving times for small number of rows since the field operations are slightly more efficient than KZG, but slower for larger models since the extra MSM offsets the initial efficiency gains.

Accuracy. Another important part of ZK-SNARK proving is the resulting accuracy. Because the arithmetization changes the output result because of quantization, it is possible that the accuracy changes. To test this, we computed the accuracy of the three vision models (the other models did not have classification benchmarks). We show results in Table 8. As we can see, the accuracy drops by at most 0.01%. In comparison, high-quality quantization is typically measured within 0.1% accuracy [15, 16, 46], or 20× higher.

	ZKML (ResNet-18)	ZKML (VGG-16)	zkCNN	vCNN
Accuracy	**91.9%**	90.4%	90.3%	90.4%*
Proving time	**52.9 s**	584.1 s	88.3 s	31 hours*
Verfication time	**12 ms**	16 ms	59 ms	20 seconds
Proof size	15.3 kB	12.1 kB	341 kB	**0.34 kB**

Table 9. ZKML compared to zkCNN [27] and vCNN [26] on proving time, verification time, and proof size. ZKML outperforms on all metrics except proof size. The proving time for vCNN was estimated by [27]. The accuracy of vCNN is estimated from the float-point accuracy, as the paper does not discuss accuracy.

Model	Proving time (ZKML)	Proving time (fixed)	Improvement
GPT2	3651.7 s	5952.0 s	63%
Diffusion	3600.6 s	4989.7 s	39%
Twitter	358.7 s	464.0 s	29%
DLRM	34.4 s	42.4 s	23%
MobileNet	1225.5 s	2407.8 s	96%
ResNet-18	52.9 s	74.8 s	41%
VGG16	637.1 s	1474.0 s	131%
MNIST	2.5 s	4.4 s	76%

Table 10. Proving times of ZKML and ZKML with a fixed configuration. As shown, ZKML's optimizer can improve proving times by nearly 2.5× compared to a fixed configuration.

Comparison to prior work. Finally, we compare ZKML to prior work on CNNs. We compare the numbers as presented by zkCNN [27] and vCNN [26] compared to ZKML on the same dataset but with performance-optimized models in addition to VGG-16. Namely, the ResNet-18 we consider achieves higher accuracy than the VGG-16 zkCNN considers. To estimate the cost of proving for prior work, we match the hardware setup used in the prior work as closely as possible. We show comparisons on CIFAR-10 in Table 9. As shown, ZKML is able to achieve faster proving times, 5× faster verification times, and 22× smaller proofs compared to zkCNN. The proving time of 31 hours for vCNN was estimated by [27].

9.3 Ablation Studies

To understand the performance of ZKML, we performed an ablation study. In our first ablation study, we fixed a number of advice columns for all models and considered the objective of optimizing proving time for KZG. The largest model we consider, GPT-2, requires 40 columns to fit within 2^{26} rows (the largest size that can prove in under 1TB of RAM for the KZG version of ZKML). As such, we picked the minimum number of rows for each model with 40 columns. Recall that the number of rows must be a power of two. Finally, since GPT-2 requires 40 columns (for our maximum of using 1TB of RAM), we excluded GPT-2 from this experiment.

We show the proving times for ZKML and for fixed configurations in Table 10. As shown, ZKML can improve proving times by almost 2× compared to using a fixed configuration. One major reason why this is the case is that the number of rows is fixed to a power of two. Even a single extra row over a power of two would cause the proving time to nearly double.

We then conducted an experiment in which we removed the different implementations of gadgets so that each layer only had a fixed implementation. We kept the optimizer to choose the optimal layout. We show the results in Table 11. As shown, the proving time can increase by up to 24× with a fixed set of gadgets.

9.4 Optimizer Savings

Time savings. We investigated whether or not ZKML's optimizers helped in reducing the time needed to choose the optimal configuration. To do so, we measured the end-to-end time for our optimizer to run compared to the time to exhaustively benchmark proofs for the different configurations.

The runtime for our optimizer on the MNIST model was 6.3 seconds for the KZG backend and 6.3 seconds for the IPA backend. In contrast, the time to exhaustively benchmark proofs was 3622 seconds for the KZG backend and 3092 seconds for the IPA backend. Our optimizer is 575× faster for KZG and 491× faster for IPA, even for our smallest model.

We further estimated the time to do exhaustive benchmarking for GPT-2 (our largest model) compared to executing ZKML's optimizer for the KZG backend. ZKML's optimizer took 185.3 seconds compared to an estimated 1,078,449 seconds for exhaustive benchmarking for GPT-2. Our optimizer is an estimated 5900× faster compared to exhaustive benchmarking. Furthermore, as seen, as the models increase in size, the cost savings also increase.

Model	Proving time (ZKML)	Proving time (no extra)	Improvement
MNIST	2.5 s	6.2 s	148%
DLRM	34.4 s	859.5 s	2399%
ResNet-18	52.9 s	812.6 s	1436%

Table 11. Proving times of ZKML and ZKML with a fixed set of gadgets. As shown, the additional gadgets can have an enormous impact on performance, leading to slowdowns of up to 24×.

Model	Pruned runtime	Non-pruned runtime
MNST	6.3 s	9.0 s
ResNet-18	28.1 s	77.5 s
GPT-2	185.3 s	277.2

Table 12. Optimizer runtime with and without pruning for three models. As shown, pruning can significantly reduce optimizer runtime. The same end configuration was used in all cases with and without pruning.

Condition	Proving time
Single-row	18.55 s
Multi-row adder	18.59 s
Multi-row max	18.58 s
Multi-row dot	18.58 s

Table 13. Proving time of models with single-row gadgets vs multi-row gadgets.

Finally, we compared our optimizer with the optimizer without pruning plans. For simplicity, we tested on three models: MNIST, ResNet-18, and GPT-2. We show the *optimizer* runtime in Table 12. In all cases, our optimizer found the same plan as the optimizer without pruning, so the *proving* time is the same for both. As shown, the optimizer runtime can decrease by as much as 2.8× with pruning.

Single-row vs multi-row constraints. We further compared ZKML's use of single-row vs multi-row constraints. To study this, we measured the performance of a fixed model when using single-row constraints vs multi-row constraints. We constructed a model that uses the adder chip, the max chip, and the dot product chip. We further fixed the circuit size to 10 columns for fairness.

We show results in Table 13. In fact, multi-row constraints induce an overhead of up to 2.2% for the proving time, showing that using single-row constraints does not significantly affect runtime.

Case studies. As a first case study, consider the GPT-2 model. Our optimizer chooses 2^{25} rows and 13 columns for KZG, and 2^{24} rows and 25 columns for IPA. As we see, the optimal configuration depends on the hardware and backend.

As a second case study, consider the case of optimizing for proof size instead of proving time. Users may want to minimize proof size when storing large numbers of proofs or using proofs on the blockchain, where storage is expensive. To minimize the proof size, we can minimize the number of columns (which is 10 for our gadgets). We measured the proving time and proof size for our five smallest models with results shown in Table 14. As shown, ZKML can optimize for both proving time and proof size.

	Runtime-optimized		Size-optimized	
Model	Time	Size	Time	Size
MNIST	2.45 s	6560 bytes	2.97 s	4800 bytes
VGG-16	637.14 s	12064 bytes	819.8 s	7680 bytes
ResNet-18	52.9 s	15744 bytes	87.3 s	6112 bytes
Twitter	358.7 s	6816 bytes	544.8 s	5056 bytes
DLRM	34.4 s	18816 bytes	42.2 s	6368 bytes

Table 14. Proving time and proof size of runtime and size optimized ZK-SNARKs.

9.5 Cost Estimation Accuracy

Finally, we measured the accuracy of our cost estimator. For the purposes of choosing a physical layout, the most important factor is that the highest performance layout is the top ranked layout. Because benchmarking all possible physical layouts is extremely expensive for the larger models, we conducted all experiments on the MNIST model.

For the MNIST model, the top ranked physical layout for both the KZG and IPA backends achieved the lowest proving time. Furthermore, we measured Kendall's rank correlation coefficient between our cost estimates and the true proving time. The rank correlation is 0.89 for KZG and 0.88 for IPA, showing that our cost estimator accurately ranks physical layouts.

10 Related Work

ZK-SNARKs for ML. All prior work for ZK-SNARKs for ML focuses on producing proofs for convolutional neural networks (CNNs) [10, 19, 26, 44]. The majority of this work focuses on custom cryptographic arguments for ZK-SNARKs. There are two major drawbacks with this approach. First, none of this work supports other forms of DNNs, including LLMs or DNNs used in recommender systems. Second, this work does not take advantage of new work on faster ZK-SNARK proving systems. ZKML addresses both of these issues by leveraging recent work to produce general-purpose

circuits for a wide range of DNNs. Furthermore, it is more efficient than all prior work.

MPC for ML. Other work in secure ML focuses on multi-party computation (MPC) for ML [22, 24, 25, 31]. MPC requires all parties to be online during the duration of the computation and does not provide validity of the computation to third parties. Furthermore, the majority of work on MPC for ML focuses on the semi-honest adversary setting, where all parties adhere to the protocol. In many circumstances (e.g., for audits, smart contracts, and other situations), the participating parties desire security against malicious adversaries. The work on MPC against malicious adversaries is typically substantially more computationally and network bandwidth-intensive.

HE for ML. Homomorphic encryption allows parties to perform computations on encrypted data without first decrypting the data [2]. This allows for offloading computation in a privacy-preserving manner. However, HE does not satisfy the auditability requirements for the applications we consider. Furthermore, HE is extremely computationally expensive, scaling only to impractically small datasets (MNIST and CIFAR10) [18, 28].

11 Conclusions

In this work, we design and implement ZKML, an optimizing compiler for ML models to ZK-SNARKs. ZKML can produce ZK-SNARKs of substantially more ML model types than prior work, including important classes of models such as LLMs and recommendation systems. We introduce a framework for optimizing circuit layout for ML models in ZK-SNARK proving systems. To the best of our knowledge, ZKML is the first optimizing compiler for ML model inference to ZK-SNARKs. Our optimizations result in up to 24× improved proving speeds. We hope that ZKML will serve as a platform for transparency in ML systems in the future.

References

[1] 2023. The minimal neural network that achieves 99% on MNIST. https://github.com/ruslangrimov/mnist-minimal-mnistmodel

[2] Frederik Armknecht, Colin Boyd, Christopher Carr, Kristian Gjøsteen, Angela Jäschke, Christian A Reuter, and Martin Strand. 2015. A guide to fully homomorphic encryption. *Cryptology ePrint Archive* (2015).

[3] Karissa Bell. 2023. What did Twitter's 'open source' algorithm actually reveal? Not a lot. *Engadget* (2023). https://www.engadget.com/what-did-twitters-open-source-algorithm-actually-reveal-not-a-lot-194652809.html

[4] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. 2018. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Paper 2018/046. https://eprint.iacr.org/2018/046 https://eprint.iacr.org/2018/046.

[5] Stan Benjamens, Pranavsingh Dhunnoo, and Bertalan Meskó. 2020. The state of artificial intelligence-based FDA-approved medical devices and algorithms: an online database. *NPJ digital medicine* 3, 1 (2020), 118.

[6] Nir Bitansky, Ran Canetti, Alessandro Chiesa, Shafi Goldwasser, Huijia Lin, Aviad Rubinstein, and Eran Tromer. 2017. The hunting of the SNARK. *Journal of Cryptology* 30, 4 (2017), 989–1066.

[7] Benedikt Bünz, Mary Maller, Pratyush Mishra, Nirvan Tyagi, and Psi Vesely. 2021. Proofs for inner pairing products and applications. In *Advances in Cryptology–ASIACRYPT 2021: 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6–10, 2021, Proceedings, Part III 27*. Springer, 65–97.

[8] Qiwei Chen, Huan Zhao, Wei Li, Pipei Huang, and Wenwu Ou. 2019. Behavior sequence transformer for e-commerce recommendation in alibaba. In *Proceedings of the 1st International Workshop on Deep Learning Practice for High-Dimensional Sparse Data*. 1–4.

[9] Akshay Dua, Nirupama Bulusu, Wu-Chang Feng, and Wen Hu. 2009. Towards trustworthy participatory sensing. In *Proceedings of the 4th USENIX Conference on Hot Topics in Security*. USENIX Association Berkeley, CA, USA, 8–8.

[10] 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* (2021).

[11] Rusins Freivalds. 1977. Probabilistic Machines Can Use Less Running Time.. In *IFIP congress*, Vol. 839. 842.

[12] Ariel Gabizon. 2021. From AIRs to RAPs - how PLONK-style arithmetization works. https://hackmd.io/@aztec-network/plonk-arithmetiization-air. (2021). https://hackmd.io/@aztec-network/plonk-arithmetiization-air

[13] Peter Gilbert, Landon P Cox, Jaeyeon Jung, and David Wetherall. 2010. Toward trustworthy mobile sensing. In *Proceedings of the Eleventh Workshop on Mobile Computing Systems & Applications*. 31–36.

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Identity mappings in deep residual networks. In *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part IV 14*. Springer, 630–645.

[15] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2017. Quantized neural networks: training neural networks with low precision weights and activations. *J. Mach. Learn. Res.* (2017), 6869–6898.

[16] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. 2017. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2017), 2704–2713.

[17] Kimmo Järvinen, Andrea Miele, Reza Azarderakhsh, and Patrick Longa. 2016. Four on FPGA: New Hardware Speed Records for Elliptic Curve Cryptography over Large Prime Characteristic Fields. In *Cryptographic Hardware and Embedded Systems–CHES 2016: 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*. Springer, 517–537.

[18] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. {GAZELLE}: A low latency framework for secure neural network inference. In *27th USENIX Security Symposium (USENIX Security 18)*. 1651–1669.

[19] Daniel Kang, Tatsunori Hashimoto, Ion Stoica, and Yi Sun. 2022. Scaling up Trustless DNN Inference with Zero-Knowledge Proofs. *arXiv preprint arXiv:2210.08674* (2022).

[20] Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. 2010. Constant-size commitments to polynomials and their applications. In *Advances in Cryptology-ASIACRYPT 2010: 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings 16*. Springer, 177–194.

[21] Will Knight. 2023. OpenAI's CEO Says the Age of Giant AI Models Is Already Over. https://www.wired.com/story/openai-ceo-sam-altman-the-age-of-giant-ai-models-is-already-over/

[22] Brian Knott, Shobha Venkataraman, Awni Hannun, Shubho Sengupta, Mark Ibrahim, and Laurens van der Maaten. 2021. Crypten: Secure multi-party computation meets machine learning. *Advances in Neural*

Information Processing Systems 34 (2021), 4961–4973.

[23] Joshua Alexander Kroll. 2015. *Accountable algorithms*. Ph. D. Dissertation. Princeton University.

[24] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. 2020. Cryptflow: Secure tensorflow inference. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 336–353.

[25] Maximilian Lam, Michael Mitzenmacher, Vijay Janapa Reddi, Gu-Yeon Wei, and David Brooks. 2022. Tabula: Efficiently Computing Nonlinear Activation Functions for Secure Neural Network Inference. *arXiv preprint arXiv:2203.02833* (2022).

[26] Seunghwa Lee, Hankyung Ko, Jihye Kim, and Hyunok Oh. 2020. vCNN: Verifiable convolutional neural network based on zk-SNARKs. *Cryptology ePrint Archive* (2020).

[27] Tianyi Liu, Xiang Xie, and Yupeng Zhang. 2021. ZkCNN: Zero knowledge proofs for convolutional neural network predictions and accuracy. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2968–2985.

[28] Qian Lou and Lei Jiang. 2021. Hemet: A homomorphic-encryption-friendly privacy-preserving mobile neural network architecture. In *International conference on machine learning*. PMLR, 7102–7110.

[29] Tao Lu, Chengkun Wei, Ruijing Yu, Chaochao Chen, Wenjing Fang, Lei Wang, Zeke Wang, and Wenzhi Chen. 2022. Cuzk: Accelerating zero-knowledge proof with a faster parallel multi-scalar multiplication algorithm on gpus. *Cryptology ePrint Archive* (2022).

[30] James McGirk. 2023. The State of Zero-Knowledge Machine Learning (zkML). (2023). https://blog.spectral.finance/the-state-of-zero-knowledge-machine-learning-zkml/

[31] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. 2020. Delphi: A cryptographic inference service for neural networks. In *29th USENIX Security Symposium (USENIX Security 20)*. 2505–2522.

[32] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. 2019. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091* (2019).

[33] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems* 35 (2022), 27730–27744.

[34] Mark Pesce. 2023. It's time to reveal all recommendation algorithms – by law if necessary. *The Register* (2023). https://www.theregister.com/2023/04/13/reveal_all_recommendation_algorithms/

[35] PSE. 2023. Perpetual Powers of Tau. https://github.com/privacy-scaling-explorations/perpetualpowersoftau

[36] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, et al. 2020. Mlperf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 446–459.

[37] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. 2022. High-Resolution Image Synthesis With Latent Diffusion Models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[38] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4510–4520.

[39] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. In *NeurIPS EMC^2 Workshop*.

[40] Ali Shahin Shamsabadi, Sierra Calanda Wyllie, Nicholas Franzese, Natalie Dullerud, Sébastien Gambs, Nicolas Papernot, Xiao Wang, and Adrian Weller. 2022. Confidential-PROFITT: Confidential PROof of FaIr Training of Trees. In *The Eleventh International Conference on Learning Representations*.

[41] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

[42] Justin Thaler et al. 2022. Proofs, arguments, and zero-knowledge. *Foundations and Trends® in Privacy and Security* 4, 2–4 (2022), 117–660.

[43] Twitter. 2023. Twitter's Recommendation Algorithm. (2023). https://blog.twitter.com/engineering/en_us/topics/open-source/2023/twitter-recommendation-algorithm

[44] Jiasi Weng, Jian Weng, Gui Tang, Anjia Yang, Ming Li, and Jia-Nan Liu. 2022. pvCNN: Privacy-Preserving and Verifiable Convolutional Neural Network Testing. *arXiv preprint arXiv:2201.09186* (2022).

[45] zcash. 2022. halo2. https://zcash.github.io/halo2/

[46] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. 2018. LQ-Nets: Learned Quantization for Highly Accurate and Compact Deep Neural Networks. In *European Conference on Computer Vision (ECCV)*.

[47] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2017. vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 863–880.