ZK-Torch: Compiling Machine Learning Models to Zero-Knowledge Proofs

Ari Fiorino[1], Lilia Tang[1], Bing-Jyue Chen[1], and Daniel Kang[1]

University of Illinois Urbana-Champaign

Abstract. In recent years, AI has become increasingly prevalent in our daily lives. As AI models become ubiquitous, it is increasingly important for users to have security guarantees about the outputs they receive from these models. Yet, the owner of the model does not want to reveal the weights, as they are considered trade secrets. To solve this problem, researchers have turned to zero-knowledge proofs of ML model inference. These proofs convince the user that the ML model output is correct, without revealing the weights of the model to the user.

Past work on these provers can be placed into two categories. The first method compiles the ML model into a low-level circuit, and proves the circuit using a ZK-SNARK. The second method are custom cryptographic protocols designed only for a specific class of models. Unfortunately, the first method is highly inefficient, making it impractical for the large models used today. And the second method does not generalize well, making it difficult to update in the rapidly changing field of machine learning.

To solve this, we propose `ZkTorch`, an end-to-end proving system that compiles ML models into base cryptographic operations called basic blocks, which are proved via custom protocols. These basic blocks are easily swapped out and updated when a new machine learning model is designed. By directly implementing these basic blocks in zero knowledge, as opposed to compiling down to a low level circuit, we can achieve 38× speedup over previous general purpose zkml provers.

Keywords: Machine Learning · Zero Knowledge Proofs · Cryptographic Protocols

1 Introduction

In recent years, AI has become increasingly capable and prevalent, with models ranging from ResNet-50 [14] for image classification to LLMs for text generation [26]. As these models become ubiquitous, so has the calls for *audits* of these models [27]. However, audits face a trade-off: transparency into the models vs protecting the trade secrets that are the weights of the models.

To resolve this tension, researchers have turned to zero-knowledge proofs (typically ZK-SNARKs) of ML model inference to perform trustless and weight privacy-preserving audits [27, 15, 5, 17, 28, 8, 25, 18]. These ZK-SNARKs allow a

model provider to prove that a specific model passed an audit while preserving the trade secrets of the weights.

These ZK-SNARKs turn ML model inference into cryptographic operations in two general ways. The first set of techniques turns the ML model computation into a general arithmetic or logical circuit using a general-purpose ZK-SNARK system [15], such as `halo2` [29]. The second set of techniques designs bespoke cryptographic protocols for specific forms of models, such as CNNs or LLMs [17, 28, 25, 18].

Unfortunately, these methods are either highly inefficient or not general and error-prone. General-purpose ZK-SNARK systems are not specialized for ML model operations, leading to extreme inefficiencies. Proving a single inference for a model like Llama3-8B could take hundreds of hours across many servers, making it impractical for real-world use. Bespoke cryptographic protocols do not capture the range of ML models widely used. For example, zkCNN does not support LLMs [18] and zkLLM does not support CNNs [25]. Furthermore, as implemented, these protocols can be insecure (Section 2.3).

To resolve this tension, we propose `ZkTorch`, an end-to-end system that compiles ML models into base cryptographic operations with high efficiency. `ZkTorch` consists of three major components: a base set of cryptographic protocols (which we call *basic blocks*), a transpiler from ML model layers to basic blocks, and a compiler that optimizes ML models for cryptographic computation.

`ZkTorch` achieves efficiency by directly using cryptographic primitives, resulting in up to 38× end-to-end speedups compared to general purpose circuits. To understand the intuition, consider adding two matrices. This operation is used in nearly every modern AI/ML model using a "residual" or "skip" connection [14]. Adding two KZG-committed vectors is 620× faster than adding two vectors in `halo2`, due to the overheads of the general-purpose cryptographic machinery in `halo2`.

`ZkTorch` achieves security by carefully selecting the cryptographic basic blocks to support. We implement 17 basic blocks, of which four are novel, two are modifications from protocols in the literature, and the remainder are taken from existing protocols. Security follows form analyzing our six new and modified basic blocks and proving that the composition of these operations remains secure. We implement a total of 43 commonly used ML model layers with our 17 basic blocks.

Our optimizations, including the compiler, results in 38× faster proving, 5× more supported layers compared to prior work, and support for all of the MLPerf Inference models. In the remainder of this manuscript, we describe in detail how `ZkTorch` works.

2 ML Models and ZK-SNARKs

In this section, we provide the relevant background on ML models and how they relate to ZK-SNARKs. We further discuss how specific considerations in ML models relate to the security of ZK-SNARKs for ML.

2.1 ZK-SNARKs

ZK-SNARKs are a class of cryptographic protocols that provide the following properties: 1) zero-knowledge, 2) non-interactivity, 3) succinctness, 4) knowledge soundness, and 5) completeness [20]. There are many different ways of constructing ZK-SNARKs, which range from general-purpose circuits [29] to specialized protocols for specific operations, such as ML models [18] or identity protocols [19].

Although protocols vary in their implementations, all ZK-SNARK protocols operate over finite fields and lower the computation they prove into an intermediate representation that is compiled to cryptographic operations. For example, the `halo2` protocol uses a 256-bit field (several are admissible) and a randomized arithmetic intermediate representation (AIR) with preprocessing to represent computation [9]. Other proving systems use 64- or 32-bit fields, and other intermediate representations such as arithmetic circuits.

Thus, there are two critical aspects of security for ZK-SNARKs: the security of the underlying cryptographic protocol and that the computation to be proved is accurately represented in the intermediate representation.

2.2 ML Models and ZK-SNARKs

ML models consist of highly structured computation that are typically described as *layers* that take as input one or more *tensors*. These layers include operations ranging from matrix multiplication to the ReLU non-linearity [16]. Some of the tensors are fixed ahead of time (i.e., are *weights*) and others must be computed dynamically depending on the input. [16] provides a detailed description of ML model computation for further reading.

ML models vary in their computational patterns, but within an ML model, computational patterns are often repeated. For example, the widely used ResNet series of models alternates between convolutions, residual layers, and the ReLU non-linearity [14]. LLMs use feed-forward networks (matrix multiplications) and attention heads (matrix multiplication, point-wise multiplication, non-linearities, and the softmax) [26].

A key insight we leverage to construct `ZkTorch` is that the total number of operations used by common ML models is relatively small. Consider the widely used MLPerf inference benchmark [21], which covers a wide range of ML models. Across all of the models in the edge category, only 61 operations are used. These operations further cover other popular models, such as the Llama series of LLMs.

ML model computation is natively computed in floating-point arithmetic, which is not natively representable in finite fields. To address this issue, ZK-SNARK systems for ML approximate the floating-point values with fixed-point arithmetic, where a number is represented by a value and a scale factor [5].

2.3 Security and Performance for ML Models and ZK-SNARKs

When constructing protocols for ML model inference, there are many important considerations in accurately assessing security and performance. These consid-

erations span across the whole stack, including the accuracy of the ML model when converted to a ZK-SNARK, the mapping of the ML model computation to the cryptographic operations, and the measurement of the cryptographic performance itself.

We analyze zk-llm, a recent system that claims to securely prove LLM inference [25]. Unfortunately, there are three major issues with this claim. We further note that other systems may have similar issues and we choose zk-llm as a case study.

The first issue is that the design of tlookup in zk-llm is very restrictive, and doesn't allow for proving many common operations used in ML inference. Concretely, tlookup is a lookup argument introduced in zk-llm which proves that every element in a vector v is in a table T. The tlookup construction, as stated by the authors of zk-llm, requires that the cardinality of T is less than the cardinality of v. This is a problem as often times the input vector v is small, which requires the table T to be small as well. One layer in modern LLMs, the RMSNorm layer, requires proving a vector of length n is in some $[0, \texttt{SF})$, where n is the number of tokens and `SF` is the scale factor used in fixed point arithmetic. In order to achieve their high perplexity measurements, they use a scale factor of 2^{12}. Therefore zk-llm requires the token length of the input to be at least 2^{11}, which is rarely the case in most conversations with LLM agents.

The second issue is that the zk-llm implementation is not end-to-end, and it's claimed performance numbers are only estimates. In order to arrive at their proof times, they first timed their implementation of tlookup and matrix multiplication. Then they used these estimates to arrive at an estimate for the full proof. This means that their implementation cannot be used in practice, since it is not end-to-end.

The third issue is that zk-llm does not measure all of the parts of the proof, leading to highly underestimated proving times. As an example, consider the fixed-point scaling. Given a scaled input a, a scale factor s, and the rescaled $\bar{a}$, zk-llm proves that

$$a = \bar{a} \cdot s + r$$

for some remainder r such that $r \in [0, s)$. zk-llm only measures the time to commit a and $\bar{a}$, *not* r. This results in proving at least 33% lower than the actual proving time, along with smaller proofs than actually needed to securely prove LLM inference. In addition, the current zk-llm implementation does not prove the blinding of any polynomials, so it is not zero knowledge.

We hope that this analysis emphasizes the complexities around proving modern AI/ML inference and that future work carefully considers these factors when constructing proving systems.

3 Technical Overview

`ZkTorch` is an optimizing system that compiles ML model inference to zero-knowledge cryptographic operations to preserve weight or input privacy. It consists of a set of cryptographic primitives (basic blocks) that are composed to

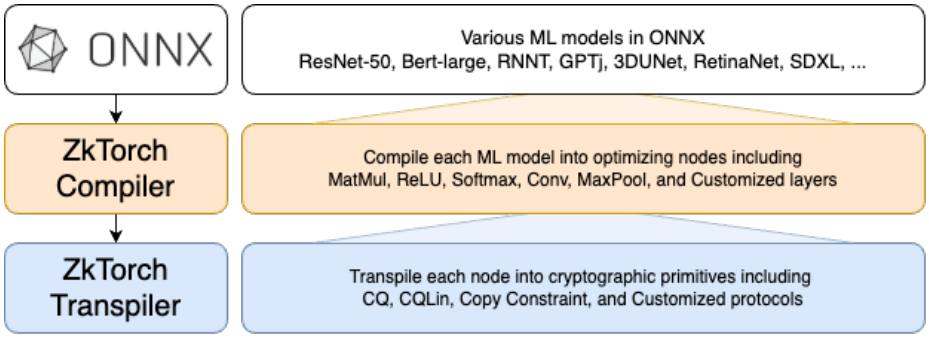

Fig. 1. Architecture diagram of `ZkTorch`

represent ML model layers, a transpiler from the ML model layers to the basic blocks, and an optimizing compiler that transforms ML model representations to be optimized for our basic blocks. We show an architecture diagram of `ZkTorch` in Figure 1, highlighting the various steps.

Through `ZkTorch`, one can easily transform an ONNX graph (each node is an ML operation) into a `ZkTorch` graph (each node is a cryptographic basic block). `ZkTorch` offers graph-level APIs for users to directly do witness generation, witness encoding, proving, and verification based on a given input tensor for the ONNX model. Specifically, witness generation is similar to ML model inference, but `ZkTorch` scales the tensors up to work over the defined prime field of cryptographic protocols. Then, `ZkTorch` encodes the witness tensors (i.e., the input, output, and all intermediate results of the model) by viewing the values in the last dimension as the evaluations of a polynomial and calculating the Kate commitment for it. With the commitment and witness, `ZkTorch` proves the graph one basic block by one and collects the proof along the procedure. `ZkTorch` also verifies if each proof passes one in the same order.

An important consideration for `ZkTorch` is the selection of the scale factors to achieve high accuracy. As mentioned in Section 2.3, it is critical that cryptographic operations are measured properly. A major consideration is the size of the table: the size of the table determines the range of values before scaling down, similar to how standard quantization works. A large table size is required for high accuracy, but if it is too large, there are memory issues. Therefore we need to carefully balance the table sizes to account for this.

Finally, we highlight that `ZkTorch` can prove a range of models on a single server, including all of the MLPerf edge inference models. This is in contrast to prior work that is either highly specialized to specific models or generic but inefficient.

If a fixed scale factor is used throught the computation, there are often issues with table size. An example of this is that after a matrix multiplication, the values get very large, and once they are passed into a nonlinearity, that nonlinearity is required to be implemented using a table. So the input of the table is

Table 1. List of basic blocks rules we currently implement in `ZkTorch`.

Basic Block	Category
Add	Arithmetic Operations
Sub	Arithmetic Operations
BooleanCheck	Arithmetic Operations
Concat	Shape Operations
CopyConstraint	Shape Operations
CQ	Tables
CQ2	Tables
Eq	Arithmetic Operations
MatMul	Matrix Multiplication
MaxProof	Arithmetic Operations
Mul	Arithmetic Operations
MulConst	Arithmetic Operations
MulScalar	Arithmetic Operations
OneToOne	Shape Operations
Ordered	Shape Operations
Permute	Shape Operations
Sum	Arithmetic Operations

very large which requires the table to be unfeasibly large in terms of memory. Therefore we propose a method of selecting different scale factors at different points along the computation which keeps the accuracy high and prevents values from falling out of range of the table.

4 Basic Blocks

The first major component of `ZkTorch` are its basic blocks. These basic blocks prove targeted operations that are common in ML model layers.

We show a list of basic blocks in Table 1 along with a categorization of the kinds of basic blocks. These include arithmetic operations (e.g., vector addition), matrix multiplication, tables (e.g., pointwise non-linearities), and shape operations (e.g., permutations). Of our 17 basic blocks, 4 are novel and 2 are modifications of previous protocols. The remaining 11 are taken from prior ZK-SNARK protocols.

In the remainder of this section, we first prove that composing basic blocks satisfying certain properties remains secure. We then prove the security of our individual basic blocks.

4.1 Security of Composing Basic Blocks

We represent the computation of an ML model's inference as a directed acyclic graph (DAG), where vertices are basic blocks and edges denote where the inputs flow. Take Fig. 2 and 3 as examples, we use the `ZkTorch` compiler and transpiler

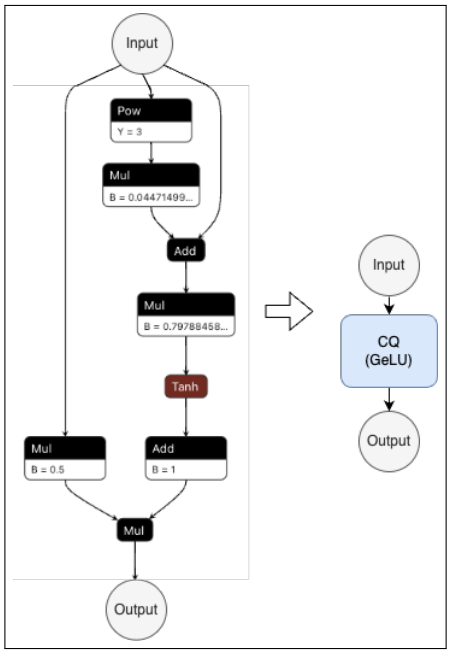

Fig. 2. One CQ for GeLU

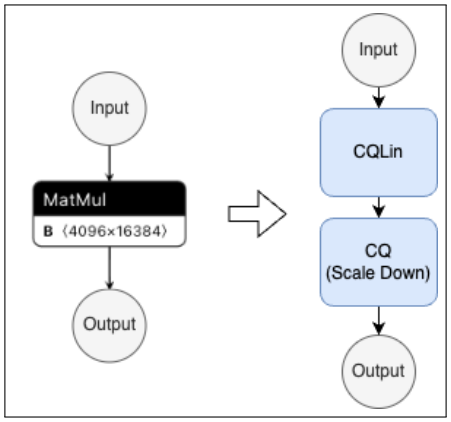

Fig. 3. A CQLin and a CQ for MatMul

to transform the orignal ONNX graph into a `ZkTorch` DAG that consists of basic blocks.

Our zero knowledge proofs rely on a structured reference string $\texttt{srs} = [x^0]_1, \ldots, [x^n]_1$ for some uniform trapdoor $x \in \mathbb{F}$. This structured reference string is generated by group of parties via a powers of tau ceremony as described in [3]. Once the `srs` is generated, as long as one of the parties is honest, no party can recover the value of the trapdoor x [3].

We require the following properties for each basic block:

1. **Completeness.** For every `srs` output by $\mathcal{G}(1^\lambda)$ and $(\pi, w) \in \mathcal{R}$,
$$\Pr[\langle \mathcal{P}(\texttt{srs}, w), \mathcal{V}(\texttt{srs})\rangle(\pi) = \text{accept}] = 1$$
2. **ϵ-Soundness.** For every `srs` output by $\mathcal{G}(1^\lambda)$ and $(\pi, w^*) \notin \mathcal{R}$,
$$\Pr[\langle \mathcal{P}(\texttt{srs}, w^*), \mathcal{V}(\texttt{srs})\rangle(\pi) = \text{accept}] \leq \epsilon$$
3. **Knowledge soundness.** For any probabilistic polynomial time (PPT) prover $\mathcal{P}^*$, there exists a PPT extractor $\mathcal{E}$ such that given access to the entire executing process and the randomness of $\mathcal{P}^*$, $\mathcal{E}$ can extract a witness w such that $\texttt{srs} \leftarrow \mathcal{G}(1^\lambda)$, $\pi^* \leftarrow \mathcal{P}^*(\pi, \texttt{srs})$, and $w \leftarrow \mathcal{E}^{\mathcal{P}^*}(\texttt{srs}, \pi, \pi^*)$, the following inequality holds
$$\Pr[(\pi; w) \notin \mathcal{R} \wedge \mathcal{V}(\pi, \pi^*, \texttt{srs}) = \text{accept}] \leq \text{negl}(\lambda)$$

4. **Zero knowledge.** There exists a PPT simulator $\mathcal{S}$ such that for any PPT algorithm $\mathcal{V}^*$, $(\pi; w) \in \mathcal{R}$, $\mathtt{srs}$ output by $\mathcal{G}(1^\lambda)$, it holds that

$$\text{View}\left(\langle \mathcal{P}(\mathtt{srs}, w), \mathcal{V}^*(\mathtt{srs})\rangle(\pi)\right) \approx \mathcal{S}^{\mathcal{V}^*}(x, \mathtt{srs}, \pi)$$

Note that the simulator $\mathcal{S}$ has access to the trapdoor x, which allows for the simulator to generate valid proofs without access to the witness w.

Our goal is to show that the DAG itself satisfies these four properties.

In our work, all of our protocols are perfectly complete and perfectly knowledge sound. Composition follows directly from the definition.

Soundness follows from the union bound. If any node in the DAG is invalid but accepted, the entire DAG is invalid. Denote the probability of node v_i being unsound as e_{v_i}. Then, the probability of any node being invalid is at most $\sum e_{v_i}$ by the union bound. Note that in practice, we have at most 1000 nodes in our DAG, so we can achieve a total soundness error of e by setting the threshold of each individual node at $\frac{1}{1000}e$. In practice, due to our field size, this error is negligible.

Our definition of zero knowledge is equivalent to the black-box simulator zero knowledge defined in [11]. Since our protocol satisfies this definition, it is also auxiliary-input zero knowledge, and therefore has the "composability" property defined in [11] which shows the full DAG is zero knowledge as well. We now turn to proving these properties for each basic block.

4.2 Security of Basic Blocks

4.2.1 PolyCheck

Given a multivariate polynomial $R(x_1, x_2, ..., x_{2N})$, the goal of a prover $\mathcal{P}$ is to convince a verifier $\mathcal{V}$ that they know a set of witness polynomials $f_1(x), f_2(x), ..., f_N(x)$ such that $R(f_1(x), f_2(x), ..., f_N(x), f_1(\omega x), f_2(\omega x), ..., f_N(\omega x)) = 0$ for $x = \omega^i, \forall i \in \{0, 1, ..., n-1\}$. Note that $f_i(x)$ is a blinded polynomial in the form of $f_i(x) = g_i(x) + (x^n - 1) \cdot b_i(x)$, where $g_i(x)$ is the witness polynomial that contains secret information and $b_i(x)$ is low degree random polynomial. The degree of $b_i(x)$ should be greater than the times we will open $f_i(x)$ at some points, which should be 3 in this protocol.

Protocol

We define groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$, a field $\mathbb{F}$, and a bilinear pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. For $t \in \mathbb{F}$, let $[t]_s = g_s^t$, where g_s is the generator for group $\mathbb{G}_s$. We assume the DLOG assumption that one cannot recover t from $[t]_s$ in polynomial time. The protocol proceeds as follows.

1. $\mathcal{P}$ sends the commitments of a set of witness polynomials $[f_1(x)]_1, [f_2(x)]_1, ..., [f_N(x)]_1$ along with the commitment of a quotient polynomial $[Q(x)]_1$ such that $R(f_1(x), f_2(x), ..., f_N(x), f_1(\omega x), f_2(\omega x), ..., f_N(\omega x)) = Q(x) \cdot (x^n - 1)$.
2. $\mathcal{V}$ uniformly samples ζ from $\mathrm{F} - \{\omega^i | \forall i \in \{0, 1, ..., n-1\}\}$, where F is the prime field they are working on. Then, $\mathcal{V}$ sends ζ.

3. $\mathcal{P}$ opens all polynomials at ζ and send $f_1(\zeta), f_2(\zeta), ..., f_N(\zeta), f_1(\omega\zeta), f_2(\omega\zeta), ..., f_N(\omega\zeta)$ and $Q(\zeta)$.
4. $\mathcal{V}$ checks if $R(f_1(\zeta), f_2(\zeta), ..., f_N(\zeta), f_1(\omega\zeta), f_2(\omega\zeta), ..., f_N(\omega\zeta)) = Q(\zeta)\cdot(\zeta^n - 1)$. If so, $\mathcal{V}$ proceeds to ask for the proof of openings; otherwise, $\mathcal{V}$ rejects the proof. $\mathcal{V}$ uniformly samples γ from F. Then, $\mathcal{V}$ sends γ.
5. $\mathcal{P}$ computes $h_1(x) = \frac{Q(x)-Q(\zeta)+\sum_{k=1}^{N}\gamma^k[f_k(x)-f_k(\zeta)]}{x-\zeta}$ and $h_2(x) = \frac{\sum_{k=1}^{N}\gamma^{k-1}[f_k(x)-f_k(\omega\zeta)]}{x-\omega\zeta}$. Then, $\mathcal{P}$ sends $[h_1(x)]_1$ and $[h_2(x)]_1$.
6. $\mathcal{V}$ checks if

$$e([h_1(x)]_1, [x-\zeta]_2) = e\left([Q(x)]_1 - [Q(\zeta)]_1 + \sum_{k=1}^{N}\gamma^k([f_k(x)]_1 - [f_k(\zeta)]_1), [1]_2\right)$$

and if

$$e([h_2(x)]_1, [x-\omega\zeta]_2) = e\left(\sum_{k=0}^{N-1}\gamma^k([f_k(x)]_1 - [f_k(\omega\zeta)]_1), [1]_2\right).$$

If so, $\mathcal{V}$ accepts the proof (i.e., output 1); otherwise, $\mathcal{V}$ rejects it (i.e., output 0).

Security properties

1. **Completeness** If $\mathcal{P}$ really has a set of witness polynomials $f_1(x), f_2(x), ..., f_N(x)$ such that $R(f_1(x), f_2(x), ..., f_N(x), f_1(\omega\zeta), f_2(\omega\zeta), ..., f_N(\omega\zeta)) = 0$ for $x = \omega^i, \forall i \in \{0, 1, ..., n-1\}$, the proof will be accepted with probability 1.
2. **Soundness** If $\mathcal{P}$ does not know witness polynomials $f_1(x), f_2(x), ..., f_N(x)$ that can satisfy $R(f_1(x), f_2(x), ..., f_N(x), f_1(\omega\zeta), f_2(\omega\zeta), ..., f_N(\omega\zeta)) = 0$, then the probability that $\mathcal{P}$ can pass

$$R(f_1(\zeta), f_2(\zeta), ..., f_N(\zeta), f_1(\omega\zeta), f_2(\omega\zeta), ..., f_N(\omega\zeta)) = Q(\zeta)\cdot(\zeta^n - 1)$$

is smaller than $\frac{deg(R(f_1(x),f_2(x),...,f_N(x),f_1(\omega x),f_2(\omega x),...,f_N(\omega x)))}{|\mathrm{F}|}$, where $|\mathrm{F}|$ is the size of the prime field.
3. **Knowledge soundness** For every prover $\mathcal{P}^*$ who is able to convince $\mathcal{V}$ with probability 1, we can build the extractor $\mathcal{E}^{\mathcal{P}^*}$ in the following steps.
   (1) Assume the maximal degree of $f_i(x)$ is M. $\mathcal{E}^{\mathcal{P}^*}$ initializes a counter $m = 0$.
   (2) $\mathcal{P}^*$ sends the commitments of a set of witness polynomials $[f_1(x)]_1, [f_2(x)]_1$, $..., [f_N(x)]_1$ along with the commitment of a quotient polynomial $[Q(x)]_1$ such that $R(f_1(x), f_2(x), ..., f_N(x), f_1(\omega x), f_2(\omega x), ..., f_N(\omega x)) = Q(x)\cdot(x^n - 1)$.
   (3) $\mathcal{E}^{\mathcal{P}^*}$ uniformly samples ζ from $\mathrm{F} - \{\omega^i | \forall i \in \{0, 1, ..., n-1\}\}$, where F is the prime field they are working on. Then, $\mathcal{E}^{\mathcal{P}^*}$ sends ζ.
   (4) $\mathcal{P}^*$ opens all polynomials at ζ and send $f_1(\zeta)$, $f_2(\zeta), ..., f_N(\zeta), f_1(\omega\zeta), f_2(\omega\zeta), ..., f_N(\omega\zeta)$ and $Q(\zeta)$. Then, $\mathcal{E}^{\mathcal{P}^*}$ sets $m = m + 1$.

(5) $\mathcal{E}^{\mathcal{P}^*}$ checks if the counter $m = M + 1$ or not. If so, $\mathcal{E}^{\mathcal{P}^*}$ proceeds to the next step. If not, $\mathcal{E}^{\mathcal{P}^*}$ goes back to step 3.
(6) $\mathcal{E}^{\mathcal{P}^*}$ can perform polynomial interpolation to recover all $f_i(x)$ because $\mathcal{E}^{\mathcal{P}^*}$ has $M + 1$ openings for all polynomial $f_i(x)$.
(7) $\mathcal{E}^{\mathcal{P}^*}$ then returns the witness by computing values $g_i(\omega^j) = f_i(\omega^j), \forall i \in \{1, 2, ..., N\},\ j \in \{0, 1, ..., n-1\}$.

Note that $\mathcal{E}^{\mathcal{P}^*}$ only fails when there exists a repetition among the $M + 1$ choices. Therefore, the upperbound of knowledge error is $\frac{(M+1)^2}{2*|\mathrm{F}-\{\omega^i|\forall i\in\{0,1,...,n-1\}\}|}$ ($M << |\mathrm{F} - \{\omega^i|\forall i \in \{0, 1, ..., n-1\}\}|$).

4. **Zero knowledge** The simulator is defined as follows:

$\mathcal{S}(x, \mathtt{srs}, \pi)$:
(1) Sample $f_{1,x}, \ldots, f_{n,x}, f_{1,\omega x}, \ldots, f_{n,\omega x} \leftarrow \mathbb{F}$.
(2) Calculate $Q_x = \frac{R(f_{1,x},...,f_{1,x},f_{1,\omega x},...,f_{1,\omega x})}{x^n-1}$.
(3) Send $[f_{1,x}]_1, \ldots, [f_{n,x}]_1, [Q_x]_1$ to V^*.
(4) Receive ζ from V^*.
(5) Sample $f_{1,\zeta}, \ldots, f_{n,\zeta}, f_{1,\omega\zeta}, \ldots, f_{n,\omega\zeta} \leftarrow \mathbb{F}$.
(6) Calculate $Q_\zeta = \frac{R(f_{1,\zeta},...,f_{1,\zeta},f_{1,\omega\zeta},...,f_{1,\omega\zeta})}{\zeta^n-1}$.
(7) Send $f_{1,\zeta}, \ldots, f_{n,\zeta}, f_{1,\omega\zeta}, \ldots, f_{n,\omega\zeta}, Q_\zeta$ to V^*.
(8) Receive γ from V^*.
(9) Calculate $h_{1,x} = \frac{Q_x - Q_\zeta + \sum_{k=1}^{N} \gamma^k [f_{k,x} - f_{k,\zeta}]}{x-\zeta}$ and $h_{2,x} = \frac{\sum_{k=0}^{N-1} \gamma^{k-1} [f_{k,x} - f_{k,\omega\zeta}]}{x-\omega\zeta}$.
(10) Send $[h_{1,x}]_1$ and $[h_{2,x}]_1$ to V^*.

Proof. The transcript between $\mathcal{S}(x, \mathtt{srs}, \pi)$ and $\mathcal{V}^*(\mathtt{srs}, \pi)$ is indistinguishable from the transcript between $\mathcal{P}(\mathtt{srs}, \pi, w)$ and $\mathcal{V}^*(\mathtt{srs}, \pi)$. According to Lemma 1 in [22], $f_i(x), f_i(\zeta), f_i(\omega\zeta)$ are indistinguishable from random, which is what is generated by $\mathcal{S}$. Also, the values of $[Q(x)]_1, Q(\zeta), [h_1(x)]_1, [h_2(x)]_1$ returned by $\mathcal{P}$ are not distinguishable from the values returned by $\mathcal{S}$ since these values are deterministic given the earlier values in the transcript.

Custom protocols.
We describe four zk-torch protocols that can be built on top of PolyCheck in this subsubsection.

1. **BooleanCheck** The goal of BooleanCheck is to ensure $f(x)$ are either 0 or 1 for $x = \omega^i, \forall i \in \{0, 1, ..., n-1\}$. It can be reduced to PolyCheck in the following steps.
   (a) Set N as 1 and let $f_1(x) = f(x)$.
   (b) Set $R(f_1(x)) = f_1(x)(1 - f_1(x))$.
2. **OrderedCheck** Given two degree-n polynomials $f(x)$ and $g(x)$, we want to prove that the sequence $g(\omega^0), g(\omega^1), \ldots, g(\omega^{n-1})$ is sorted in descending or ascending order with respect to the sequence $f(\omega^0), f(\omega^1), \ldots, f(\omega^{n-1})$. It can be reduced to three checks.
   – **One-to-one Check**: there's a one-to-one mapping between the evaluations of $f(x)$ and $g(x)$. This can be realized by Plonk permutation check.

- **Sub Check**: computes and proves that $d(x) = g(\omega x) - g(x)$.
- **Range Check**: proves that every evaluation of $d(x)$ for $x = \omega^i, \forall i \in \{0, 1, ..., n-1\}$ is not greater/less than 0 if $g(\omega^0), g(\omega^1), ..., g(\omega^{n-1})$ is in descending/ascending order. This can be done by CQ check.

Now, we show the Sub Check can be reduced to PolyCheck.

(a) Set N as 2 and let $f_1(x) = g(x), f_2(x) = d(x)$.
(b) Set $R(f_1(x), f_2(x), f_1(\omega x)) = f_2(x) + f_1(x) - f_1(\omega x)$.

3. **MaxProof** Given two polynomials $f(x)$ and $g(x)$, we want to prove $g(\omega^0)$ is the max element among $f(\omega^0), f(\omega^1), ..., f(\omega^{n-1})$. It can be reduced to three checks.
   - **OrderedCheck**: proves the evaluations of $f'(x)$ are sorted in descending order from $f(x)$. The construction is introduced above.
   - **Sub Check**: proves $d(x) = f'(x) - g(x)$. The reduction to PolyCheck is introduced above.
   - **i-thIsZero Check**: proves $d(\omega^i)$ is 0. In the MaxProof, $i = 0$.

   Denote $L_i(x)$ as a Lagrange polynomial, which is 1 when $x = \omega^i$ and is 0 when $x = \omega^j, j \neq i$. We now show **i-thIsZero Check** can be reduced to PolyCheck.
   (a) Set N as 2 and let $f_1(x) = L_i(x), f_2(x) = d(x)$.
   (b) Set $R(f_1(x), f_2(x)) = f_1(x) * f_2(x)$.
4. **CopyConstraint** Let $H \subset \mathbb{F}$ be a multiplicative subgroup of order $N = \max(m, n)$. We define $L_i(X)$ to be such that L_i for $i \in [\max(m, n)]$ is a Lagrange basis for H.
   Input polynomials: $f_i(X) = \sum_{j=1}^{m} w_{i \cdot m + j} L_{N/m \cdot j}(X)$ for $i \in [p_1]$
   Output polynomials: $g_j(X) = \sum_{k=1}^{n} w_{j \cdot n + p_1 m + k} L_{N/n \cdot k}(X)$ for $j \in [p_2]$ for a witness $(w_i)_{i \in [p_1 m + p_2 n]}$
   Define $(f_{(1)}, \ldots, f_{(p_1 m)}) \in \mathbb{F}^{p_1 m}, (g_{(1)}, \ldots, g_{(p_2 n)}) \in \mathbb{F}^{p_2 n}$ by

$$f_{((j_1-1)m+i_1)} := f_{j_1}(\omega^{i_1}), g_{((j_2-1)n+i_2)} := g_{j_2}(\omega^{i_2})$$

   for $j_1 \in [p_1], i_1 \in [m], j_2 \in [p_2], i_2 \in [n]$
   CopyConstraint ensures that for a set of padding partition indices $\mathsf{pad\_idx}$, mapping pads: $\mathsf{pad\_idx} \to \mathsf{pad\_val}$ and $\sigma : [p_2 n] \to [p_1 m] \cup \mathsf{pad\_idx}$,

$$g_{(\ell)} = \begin{cases} \mathsf{pads}(\sigma(\ell)) & \sigma(\ell) \in \mathsf{pad\_idx} \\ f_{(\sigma(\ell))} & \sigma(\ell) \notin \mathsf{pad\_idx} \end{cases}$$

   The partition construction σ is constructed in the same way as in Plonk [10]. To handle padding, all padded indices with the same pad value are contained in the same partition with indices in $\mathsf{pad\_idx}$. Let q be the number of pad values, and pad_i be the i-th pad value. For each $i \in [q]$, let r_i be the smallest value such that pad_i is constrained to be a coefficient of $G_i(X) = f_{r_i}(X)$ or $G_i(X) = g_{r_i - p_1}(X)$. Let t_i be the smallest value such that $G_i(X) L_{t_i}(X) = \mathsf{pad}_i$. If $m < N$ or $n < N$, when $i \mod \frac{N}{m} \neq 0$ or $i \mod \frac{N}{n} \neq 0$, then the element will be in its own singleton partition.
   CopyConstraint reduces to Plonk's Section 5 extended copy constraints [10]. It is sufficient to call the extended copy constraints as a subroutine with

the input and output polynomials and σ described above and perform the following PolyCheck for padding: for $i \in [q]$, $G_i(x)L_{t_i}(x) = \mathsf{pad}_i$.
Overall, CopyConstraint reduces to these PolyChecks:

- **One Check**: $L_1(x)(z(x) - 1) = 0$
- **Permutation Check**:

$$\begin{aligned} &z(x)\prod_{i=1}^{p_1}(f_i(x) + i\beta x + \gamma)\prod_{i=1}^{p_2}(g_i(x) + i\beta x + \gamma) \\ =&z(\omega x)\prod_{i=1}^{p_1}(f_i(x) + \beta S_{\sigma_i}(x) + \gamma)\prod_{i=1}^{p_2}(g_i(x) + \beta S_{\sigma_{p_1+i}}(x) + \gamma) \end{aligned}$$

- **Pad Check**: For $i \in [q]$, $G_i(x)L_{t_i}(x) = \mathsf{pad}_i$

where
$S_{\sigma_i}(X) = \sum_{j=1}^{m} \sigma(i \cdot m + j)L_{N/m \cdot j}(X)$ for $i \in [p_1]$
$S_{\sigma_{p_1+i}}(X) = \sum_{j=1}^{n} \sigma(i \cdot n + p_1 m + j)L_{N/n \cdot j}(X)$ for $i \in [p_2]$

$$z(X) = L_1(X) + \sum_{i=1}^{N-1}\left(L_{i+1}(X)\prod_{j=1}^{i}\prod_{k=1}^{p_1+p_2}\frac{(w_{kN+j} + \beta k\omega^j + \gamma)}{(w_{kN+j} + \sigma(kN + j)\beta + \gamma)}\right)$$

4.2.2 Batch CQ

All range check and lookup queries from polynomials of the same degree and the same table are batched, up to 8 at a time. For $k \leq 8$, we want to check that $\sum_{a\in\mathbb{V}} \frac{m(a)}{x+T(a)} - \sum_{b\in\mathbb{H}}\left(\frac{1}{x+f_1(b)} - \ldots - \frac{1}{x+f_k(b)}\right) = 0$.

Note that $f_1, \ldots, f_k$ are blinded as described in the PolyCheck protocol.

The CQ protocol [6] is modified based on the equations from [13] and the cq^+ protocol in [4].

Setup: Define $z(X) = \frac{Z_{\mathbb{V}}(X)}{Z_{\mathbb{H}}(X)}$, $T(X) = \sum_{i\in[N]} t_i L_i(X)$, $U(X) = X^2 - 1$
$\left\{r_j^{\mathbb{V}}(X) = \frac{L_j^{\mathbb{V}}(X) - L_j^{\mathbb{V}}(0)}{X}U(X)\right\}_{j\in[N]}$, $\left\{r_i^{\mathbb{H}}(X) = \frac{L_i^{\mathbb{H}}(X) - L_i^{\mathbb{H}}(0)}{X}U(X)\right\}_{i\in[n]}$,
$\left\{Q_j(X) = \frac{T(X) - t_j}{Z_{\mathbb{V}}(X)}\right\}_{j\in[N]}$
Compute and output $\left[(r_j^{\mathbb{V}})_{j=1}^N, (r_i^{\mathbb{H}})_{i=1}^n, (Q_j)_{j=1}^N, Z_{\mathbb{V}}\right]_1, [T(x)U(x), Z_{\mathbb{V}}, z(x)U(x)]_2$,.

Round 1:

1. **P** computes the polynomial $m(X) \in \mathbb{F}_{<N}[X]$ defined by setting m_i, for each $i \in [N]$ to the number of times t_i appears in $f_1|_{\mathbb{H}}, \ldots f_k|_{\mathbb{H}}$.
2. **P** samples $\rho_m \xleftarrow{\$} \mathbb{F}$ sends $\mathsf{m} := [m(x)]_1 + \rho_m \cdot [Z_{\mathbb{V}}(x)]_1$.
3. **P** samples $R_S, \rho_S \xleftarrow{\$} \mathbb{F}$ sends $\mathsf{S} := [R_S \cdot X + \rho_S \cdot Z_{\mathbb{V}}(X)]_1$.

Round 2:

1. **V** chooses and sends random $\beta \in \mathbb{F}$.
2. **P** computes $A \in \mathbb{F}_{<N}[X]$ such that for $i \in [N], A_i = m_i/(t_i + \beta)$.
3. **P** samples $\rho_A \xleftarrow{\$} \mathbb{F}$, and computes and sends $\mathsf{a} := [A(X)]_1 + \rho_A \cdot [Z_{\mathbb{V}}(X)]$.
4. **P** samples $\rho_B \xleftarrow{\$} \mathbb{F}$, and computes $B_0(X) \in \mathbb{F}_{<n}[X]$ such that for $i \in [n], B_{0i} = \frac{1}{\beta+f_{1,i}} + \ldots + \frac{1}{\beta+f_{k,i}}$. **P** computes $B(X) := B_0(X) + \rho_B \cdot Z_{\mathbb{V}}(X)$, and computes and sends $\mathsf{b} := [B(x)]_1$.
5. **P** computes $Q_B(X)$ such that
   $B(X)\prod_{i\in[k]}(\beta + f_i(X)) - \left(\sum_{i\in[k]}\prod_{j\neq i}(\beta + f_j(X))\right) = Q_B(X) \cdot Z_{\mathbb{V}}(X)$
6. **P** computes and sends $\mathsf{q}_b := [Q_B(x)]_1$.

Round 3:

1. **V** sends random $\gamma, \eta \in \mathbb{F}$.
2. Let

$$d(X) = \begin{cases} 1 & \text{if } k = 1, \\ f_1(X) + f_2(X) + 2\beta & \text{if } k = 2, \\ (f_1(X) + \beta)\left(\sum_{i\in[k],i>1}\prod_{j\neq i,j>1}(\beta + f_j(\gamma))\right) & \text{if } k > 2 \\ +(f_2(X) + \beta)\prod_{j\neq 1,2}(\beta + f_j(\gamma)) & \end{cases}$$

   P computes $B_\gamma = B(\gamma)$,
   $D(X) = B(X) \cdot \prod_{i\in[k]}(\beta + f_i(\gamma)) - d(X) - Q_B(X) \cdot Z_{\mathbb{H}}(\gamma)$
3. **P** computes $P(X) = \left(D(X) + \sum_{i\in[k]} \eta^i (f_i(X) - f_i(\gamma))\right)/(X-\gamma)$, and computes and sends $\mathsf{p} := [P(X)]_1$
4. **P** computes
   $\mathsf{R_C} := \sum_{m_j\neq 0} A_j \cdot [r_j^{\mathbb{V}}(x)]_1 - n/N \sum_{i=1}^{n} B_i \cdot [r_i^{\mathbb{H}}(x)]_1 + \eta^2 R_S \cdot [U(x)]_1$
5. **P** computes $\mathsf{Q_A} := \sum_{m_j\neq 0} A_j \cdot [Q_j(x)]_1 + [\rho_A(T(x) + \beta) - \rho_m]_1$ and $\mathsf{Q_C} := [\rho_A + n/N\rho_B]_1$
6. **P** computes and sends $\mathsf{Q} := \mathsf{Q_A} + \eta\,\mathsf{Q_C} + \eta^2\,[\rho_S]_1$
7. Let

$$\mathsf{d} = \begin{cases} [1]_1 & \text{if } k = 1, \\ [f_1(x) + \beta]_1 + [f_2(x) + \beta]_1 & \text{if } k = 2, \\ [f_1(x) + \beta]_1\left(\sum_{i\in[k],i>1}\prod_{j\neq i,j>1}(\beta + f_j(\gamma))\right) & \text{if } k > 2 \\ +[f_2(x) + \beta]_1\prod_{j\neq 1,2}(\beta + f_j(\gamma)) & \end{cases}$$

   V computes $\mathsf{D} := \mathsf{b} \cdot \prod_{i\in[k]}(\beta + f_i(\gamma)) - \mathsf{d} - [\mathsf{q}_b]_1 \cdot Z_{\mathbb{H}}(\gamma)$.
8. **V** checks $e(\mathsf{a}, [T(x)]_2)\cdot e((\beta+\eta)\cdot\mathsf{a} - \mathsf{m} + \eta^2\mathsf{s}, [U(X)]_2) = e(\eta\, n/N \cdot \mathsf{b}, [z(X)U(X)]_2)\cdot e(\mathsf{q}, [Z_{\mathbb{V}}(X)U(X)]_2) \cdot e(\eta \cdot \mathsf{R_C}, [x]_2)$
   $e(\mathsf{D} + \sum_{i\in[k]} \eta^i([f_i(X)]_1 - [f(\gamma)]_1), [1]_2) = e(\mathsf{p}, [x - \gamma]_2)$.

Knowledge Soundness and Completeness
The CQ proof still holds for the new equality

$$\sum_{a\in\mathbb{V}} \frac{m(a)}{x+T(a)} - \sum_{b\in\mathbb{H}} \left(\frac{1}{x+f_1(b)} - \ldots - \frac{1}{x+f_k(b)} \right) = 0$$

and the new definition for $B(X)$.

Zero Knowledge
The simulator is defined as follows:

$\mathcal{S}(x, \texttt{srs}, \pi, [T(X)]_1)$:

Round 1:

1. Sample $f'_1(X), f'_2(X), ..., f'_k(X) \leftarrow \mathbb{F}_{\leq n}[X]$.
2. Send $[f'_1(X)]_1, [f'_2(X)]_1, ..., [f'_k(X)]_1$
3. Sample $\rho_m \leftarrow \mathbb{F}$. $m(X) = \rho_m Z_{\mathbb{V}}(X)$
4. Send $[m(X)]_1$

Round 2:

1. Receive β from V^*.
2. Sample $\rho_A \leftarrow \mathbb{F}$. $A(X) = \rho_A Z_{\mathbb{V}}(X)$
3. Sample $\rho_B(X) \leftarrow \mathbb{F}$. $B(X) = \rho_B(X) Z_{\mathbb{V}}(X)$
4. Sample $R_S, \rho_S \leftarrow \mathbb{F}$. $S(X) = R_S X + \rho_S Z_{\mathbb{V}}(X)$
5. $[Q_B(x)]_1 = \left(B(X) \prod_{i\in[k]} (\beta + f_i(X)) - \left(\sum_{i\in[k]} \prod_{j\neq i} (\beta + f_j(X)) \right) \right) \cdot \frac{1}{Z_{\mathbb{H}}(X)}$
6. Send $[A(X)]_1, [B(X)]_1, [Q_B(X)]_1, [S(X)]_1$

Round 3:

1. Receive γ, η from V^*.
2. Let

$$\mathsf{d} = \begin{cases} [1]_1 & \text{if } k = 1, \\ [f_1(x) + \beta]_1 + [f_2(x) + \beta]_1 & \text{if } k = 2, \\ [f_1(x) + \beta]_1 \left(\sum_{i\in[k], i>1} \prod_{j\neq i, j>1} (\beta + f_j(\gamma)) \right) & \text{if } k > 2 \\ + [f_2(x) + \beta]_1 \prod_{j\neq 1,2} (\beta + f_j(\gamma)) & \end{cases}$$

   $[D(X)]_1 = [B(X)]_1 \cdot \prod_{i\in[k]} (\beta + f_i(\gamma)) - \mathsf{d} - [Q_B(X)]_1 \cdot Z_{\mathbb{H}}(\gamma)$.
   Claim $D(\gamma) = 0, f_i(\gamma) = f'_i(\gamma)$ for $i \in [k]$
3. $[P(X)]_1 = \left([D(X)]_1 + \sum_{i\in[k]} \eta^i (f_i(X) - f_i(\gamma)) \right) / (X - \gamma)$
4. $R_C(X) = \eta R_S \cdot U(X)$
5. Choose $[Q(X)]_1 = \rho_A(\beta + \eta + [T(X)]_1) - \eta \cdot n/N \rho_B - \rho_m + \eta^2 \rho_S$.
6. Send $[P(X)]_1, [R_C(X)]_1, [Q(X)]_1, f_i(\gamma)$ for $i \in [k]$

Proof. The transcript between $\mathcal{S}(x, \texttt{srs}, \pi, [T(X)]_1)$ and $\mathcal{V}^*(\texttt{srs}, \pi, [T(X)]_1)$ is indistinguishable from the transcript between $\mathcal{P}(\texttt{srs}, \pi, w, [T(X)]_1)$ and $\mathcal{V}^*(\texttt{srs}, \pi, [T(X)]_1)$. According to Lemma 7.1 in [22], $m(x), A(x), B(x), f_i(x), f_i(\gamma)$ are indistinguishable from random, which is what is generated by $\mathcal{S}$. Also, the values of $[P(x)]_1, [R_C(x)]_1$ returned by $\mathcal{P}$ are not distinguishable from the values returned by $\mathcal{S}$ since these values are deterministic given the earlier values in the transcript.

5 Compiler and Transpiler

Given the basic blocks and an ML model, `ZkTorch` must compile the ML model to the underlying basic blocks. There are two major challenges in constructing such a compiler.

First, some individual operations are complex and require many basic blocks. To understand the complexity, consider the softmax operation. Naively implemented, it consists of an exponential, a sum, and a division, which is at minimum three basic blocks. In order to increase accuracy, we implement softmax using the logsumexp equation which increases the required basic blocks to 7.

Second, we must optimize *across* layers for high performance. For this, we create graph transformation rules that prune the DAG of basic blocks and combine basic blocks as well.

Third, we must choose scale factors to achieve high accuracy while maintaining reasonable table sizes. Choosing a uniform scale factor that is too large results in tables that are too large to prove (e.g., table sizes of 2^{50}) or poor accuracy. Therefore we use different scale factors at different points in the computation.

We now provide an overview of `ZkTorch`'s compiler before discussing its detailed implementation.

5.1 Overview

`ZkTorch`'s compiler takes as input an ML model specification (we use ONNX for our implementation) and outputs a program that proves ML model inference. It consists of three steps: 1) graph transformations to optimize the layers for cryptographic proving as opposed to standard inference, 2) per-layer scale factor selection, and 3) transpilation of the ML models to basic blocks.

The transpilation takes the optimized graph and scale factors as inputs and outputs a program specific to the model at hand. This process must be repeated for different models. Furthermore, several parts of the proving process can be done per-model ahead of time (precomputation) to reduce proving time per inference.

In our current implementation, we use rule-based transformations for the graph transformations and transpilation. We run the model through a sample input to select the scale factors used at different points in the computation. We now describe the different steps for our compiler.

Table 2. List of graph transformation rules we currently implement in `ZkTorch`.

Rule name	Model	Reason	Description
ReshapeTrans	Bert	Use fewer permutes	Replace every Reshape node followed by a Transpose node with a custom ReshapeTrans node.
GeLU	GPTj	Use fewer CQ	Replace every subgraph that represents GeLU with one GeLU node.
MultiHeadMatMul	GPTj	Use no copy constraint	Replace every subgraph that represents MultiHead MatMul with one custom MultiHeadMatMul node.
RoPE	GPTj	Use no copy constraint	Replace every subgraph that represents RoPE operation with two custom nodes (RoPEConst and RoPERotate).
CustomCNN	ResNet, 3DUnet, RetinaNet	Use no copy constraint, cheap aggregation	Execute convlutions by viewing intermediate tensors as channels second. This allows us to use only cqlin and add.
MultiHeadConv	RetinaNet	Use no copy constraint	Replace conv followed by reshape and transpose with a custom MultiHeadConv.
ConcatConv	3DUnet	Use no copy constraint	Replace two conv followed by a concat with a custom ConcatConv to avoid the concat.

5.2 Graph Transformation Rules

Graph transformations are a widely used method to improve ML model performance. The same operation can be computed in many different ways, which have different performance characteristics on different hardware platforms for standard ML inference. Similarly, the performance characteristics of identical methods of computation widely vary for our basic blocks.

Our graph transformation implementation takes as input an ONNX file and outputs an ONNX file optimized for `ZkTorch`. We use rule-based transformations in our current implementation. The primary goal of our transformations are to use fewer or cheaper cryptographic operations. In this work, we implement seven transformation rules and apply them greedily until the graph does not change. This form of optimization is commonly used in compilers [1].

We provide a summary of the rules we use in Table 2. As an example of a simple rule, the GeLU operation in ONNX is implemented manually, taking 8 operations including Pow and Tanh. We can replace these operations with a single table lookup. This transformation is shown in Figure 2. Our transformation rules can provide up to a 10× speedup for the operations at hand and up to a 2× speedup for real-world models.

ZkTorch's graph transformation rules are easily extensible and can be updated to accommodate newer models.

5.3 Scale Factor Selection

ML models are usually run on floating point numbers, which have a high overhead if proved in zero knowledge. Therefore it is necessary to quantize our model to use fixed point numbers instead. In fixed point, a number a is stored as $\lfloor a \cdot \mathtt{SF} \rceil$ where SF is the scale factor. We have found that using a fixed scale factor across the entire model computation often doesn't work with the model accuracy and available memory requirements we have. The issue is that if the scale factor is too small, we lose too much accuracy, and if the scale factor is too large, we run out of memory because we have to store very large lookup tables, as the fixed point tensors become very large.

Therefore we use a different scale factor at different points in the computation. Intuitively, when a tensor in the computation has larger numbers in it, it should have a larger scale factor. In order to calculate the scale factors, we run the ML model through a sample input, and measure the value of the tensor as is passes through the model. Then we use these values to calculate the scale factors to use.

5.4 Transpilation

Given an optimized graph and the scale factors per layer, ZkTorch's final step is to transpile the graph into an executable cryptographic protocol. Similar to our graph transformation rules, we currently implement our transpiler with rule-based transformations.

We implement 63 layers in transpiler, corresponding to ONNX layers. Due to space limitations, we cannot fully describe all of the layer implementations here. Instead, we describe classes of layers and highlight specific instances for illustrative purposes.

Before describing the classes of layers, we highlight several common patterns. First, any layer that multiplies two numbers must have a rescaling at some point after that layer to correct for the scale factor. Second, due to the way we lay out the committed intermediate values (activations), certain operations can be free. Our graph transformations aim to lay out the activations so that they use as few expensive shape operations as possible.

MatMul-based layers. The Gemm, MatMul, LSTM, Conv, and ConvTranspose layers are implemented with matrix multiplication operations. The default MatMul basic block proves matrix multiplication based on the Aurora protocol [2]. When one matrix is known at compile time, such as the weights in Conv or with a fixed MatMul matrix, we prove using the cqlin protocol [7]. Since the matrix is known at compile time, it can be processed during setup to enable a proving time linear in the input matrix size. Figure 3 shows how a MatMul layer is transformed into basic blocks when one matrix is fixed.

Shape operations. ML models frequently employ layer operations that manipulate array shapes, including Reshape, Concat, Split, and Transpose operations. For both input and output tensors, polynomial commitments are made along the tensor's final axis. When shape operations preserve this final axis, no new polynomial commitments are required for the outputs. However, many such operations still modify the final axis. In this case, transpose and most reshape operations can be proven using our Aurora-based Permute protocol. For more specialized cases involving element copying, we implement CopyConstraint, which offers the most flexibility but comes with higher computational costs.

Pointwise non-linearities. ML models typically contain nonlinearities to capture non-linear relationships and decision boundaries. We also need to introduce scaling operations to correct for scale factors changing after multiplication and division operations. Both of these operations are proven by checking for inclusion in a lookup table using the CQ protocol [6]. When possible, we combine nonlinearities and scaling operations within a layer to reduce the number of lookup operations to prove.

Arithmetic operations. Most arithmetic layers such as Add, Mul are trivially implemented using a corresponding basic block. We describe two examples using custom basic blocks:

- **And**: To implement the And layer, we use BooleanCheck to check that each input array only contains 0s or 1s, and then use Mul to get their product.
- **ArgMax**: To implement ArgMax, we use OrderedCheck to check that the sorted input is in descending order, One-to-oneCheck to check that the sorted elements correspond to the original unsorted input, and CopyConstraint to select the first element.

Bespoke operations. To efficiently prove ML models, `ZkTorch` replaces some specific patterns of ONNX nodes with customized operations. This is because if `ZkTorch` directly encodes and proves based on the original tensors, the proving overhead will be too expensive for practical usage. Here we describe two examples of bespoke operations:

- **CustomConv**: CustomConv achieves the same accuracy as Conv, but it saves considerable memory and proving time because it removes the needs to use any copy constraint in Convolution. Take 2D convolution as an example, rather than feed in an input of size $[batch, in\_channel, height, width]$, `ZkTorch` makes all input sizes become $[batch * height * width, in\_channel]$. Suppose the kernel size is $[out\_channel, in\_channel, h, w]$, `ZkTorch` can view the channel update as $h \times w$ CQLins during the `ZkTorch` graph compilation. This will result in $h \times w$ tensors of shape $[batch * height * width, out\_channel]$. Later, we can select some of the tensors out based on the window sliding mechanism of Convolution and add them together. This selection is free because we only encode the last dimension into commitment. The addition is also almost free because addition is much cheaper than all the other basic blocks .

Model	Part	Parameters	Input dimensions
BERT		110M	1 token
GPT-j		6B	2 tokens
LLaMA-2-7B		7B	seqlen=1, 1 token
RNNT		120M	seqlen=1, batch=1
SDXL	Text encoder	3.5B	1 token
	Text encoder 2		1 token
	VAE decoder		1x4x4x4
ResNet-50		25.6M	1x3x224x224
RetinaNet		34M	1x3x800x800
3D-UNet		19M	1x1x128x128x128

Table 3. List of models considered in the evaluation.

– **GeLU**: As shown in Fig. 2, there are 4 Mul, 1 Pow, and 1 Tanh in the original ONNX GeLU activation function. To prevent using multiple CQ and Mul basic blocks, we use just one CQ lookup to realize it.

6 Experiments

We now turn to evaluating `ZkTorch`. We show that `ZkTorch` can prove inference on a wider range of model that prior work: it can prove all of the MLPerf edge inference models on a single server. We further show that `ZkTorch`'s compiler optimizations result in high performance gains of up to 38×. This was calculated based on our estimate of how long the ZKML framework [5] would take to prove LLaMA-2-7B, versus the proving numbers we measured in `ZkTorch`.

6.1 Experimental Setup

We perform proving and accuracy evaluation on all models in the MLPerf Inference v4.1 Edge benchmark, as well as RNNT from v4.0 and LLaMA-2-7B. These include ResNet50-v1.5, RetinaNet 800x800, BERT, 3D-Unet, GPT-j, and some subnetworks of Stable-Diffusion-XL. The number of parameters and input sizes we used for each model is shown in Table 3.

Proving and verification performance are measured on a single server with an Intel(R) Xeon(R) Platinum 8358, with 64 threads and 4 TB memory.

6.2 End-to-End Results

We measure the proving latency and inference accuracy of our quantization and scale factor scheme on the models and setup described in Section 6.1. We show proving results in Table 4 and accuracy results in Table 5.

Table 4. End-to-end proving time for MLPerf Inference v4.1 Edge models, RNNT, and Llama-V2-7B.

Model	Part	Proving time	Verification time	Proof size
BERT		833.88 s	15.51 s	6 MB
GPT-j		497.14 s	41.8 s	31 MB
LLaMA-2-7B		2646 s	72.4 s	35 MB
RNNT	Joint	0.89 s	0.06 s	12 KB
	Encoder	31.8 s	0.4 s	93 KB
	Prediction	3.18 s	0.11 s	28 KB
	Total	35.9 s	0.57 s	123 KB
SDXL	Text encoder	765 s	2.64 s	2 MB
	Text encoder 2	605 s	8.33 s	6 MB
	VAE decoder	24660 s	392 s	233 MB
ResNet-50		14577 s	11.9 s	37 MB
RetinaNet		89033 s	152 s	707 MB
3D-UNet		143513 s	152 s	10.66 GB

Table 5. Accuracy numbers with `ZkTorch` scaling.

Model	Metric	`ZkTorch` Accuracy	Ref. Accuracy
GPT-j	ROUGE	ROUGE 1 - 42.9865	ROUGE 1 - 42.9865
		ROUGE 2 - 20.1235	ROUGE 2 - 20.1235
		ROUGE L - 29.9881	ROUGE L - 29.9881
BERT	F1 score	90.104%	90.874%
ResNet-50	Accuracy	76.038%	76.456%
RNNT	WER	7.49%	7.45%
3D-UNet	Accuracy	Mean - 86.18%	Mean - 86.17%
		Kidney - 93.47%	Kidney - 93.47%
		Tumor - 78.88%	Tumor - 78.87%
RetinaNet	mAP	0.3753	0.3757

7 Related Work

ZK-SNARK protocols. ZK-SNARKs have been widely studied in the security community. These protocols include protocols with different properties. For example, Groth16 has constant verification time with a per-circuit setup [12], while Spartan has linear proving time but longer proofs [23]. These protocols have also been specialized for other applications, such as general-purpose ZK virtual machines [24] and ML, which we discuss below. Our work draws inspiration from prior protocols, including Aurora [2].

ZK-SNARKs for ML. Researchers have developed ZK-SNARK protocols to perform ML inference. These protocols broadly fall under two categories.

The first category are compilers that take the computation in ML inference and compile them to generic ZK-SNARK protocols. These include ezkl, zkml, and others [30, 5]. Because generic ZK-SNARK protocols are universal, these

systems can support a wide range of ML models. However, they suffer in performance compared to specialized systems.

The second category are specialized protocols for specific kinds of models. These include specialized protocols for CNNs [18] and LLMs [25]. While they are more efficient than generic ZK-SNARK protocols, they are do not support a wider range of models. Furthermore, as we describe in Section 2.3, they can be insecure or result in vacuous accuracy.

In this work, we develop a specialized compiler that generates custom protocols for AI and ML models, bridging the gap between generic ZK-SNARK protocols and specialized protocols for specific model types. To the best of our knowledge, our work is the first to do so in a high-performance manner.

8 Conclusion

This paper introduces `ZkTorch`, a highly efficient general purpose zero knowledge prover for ML model inference. By compiling the ML model into a DAG of basic blocks and proving the basic blocks directly, `ZkTorch` is much faster than past general purpose provers. Secondly, the `ZkTorch` design is much more generalizable than past provers designed for only a single model architecture. `ZkTorch` security is based on the properties of soundness, completeness, and zero knowledge for each of our basic blocks, for which we provide concrete proofs. Our compiler is able to transform the ML model into a DAG of basic blocks, taking into account simplifications in the DAG that can be made to increase efficiency. The compiler also selects varying scale factors at different points in the computation which balances the trade off between model accuracy and required memory. Finally, our experiments on a wide range of ML models shows the benefits `ZkTorch` achieves with respect to speed, accuracy, and generalizability.

References

1. Alfred, V.A., Monica, S.L., Jeffrey, D.U.: Compilers principles, techniques & tools. pearson Education (2007)
2. Ben-Sasson, E., Chiesa, A., Riabzev, M., Spooner, N., Virza, M., Ward, N.P.: Aurora: Transparent succinct arguments for R1CS. Cryptology ePrint Archive, Paper 2018/828 (2018), https://eprint.iacr.org/2018/828
3. Bowe, S., Gabizon, A., Miers, I.: Scalable multi-party computation for zk-SNARK parameters in the random beacon model. Cryptology ePrint Archive, Paper 2017/1050 (2017), https://eprint.iacr.org/2017/1050
4. Campanelli, M., Faonio, A., Fiore, D., Li, T., Lipmaa, H.: Lookup arguments: Improvements, extensions and applications to zero-knowledge decision trees. Cryptology ePrint Archive, Paper 2023/1518 (2023), https://eprint.iacr.org/2023/1518
5. Chen, B.J., Waiwitlikhit, S., Stoica, I., Kang, D.: Zkml: An optimizing system for ml inference in zero-knowledge proofs. In: Proceedings of the Nineteenth European Conference on Computer Systems. pp. 560–574 (2024)
6. Eagen, L., Fiore, D., Gabizon, A.: cq: Cached quotients for fast lookups. Cryptology ePrint Archive (2022)

7. Eagen, L., Gabizon, A.: cqlin: Efficient linear operations on kzg commitments with cached quotients. Cryptology ePrint Archive (2023)
8. Feng, B., Qin, L., Zhang, Z., Ding, Y., Chu, S.: Zen: An optimizing compiler for verifiable, zero-knowledge neural network inferences. Cryptology ePrint Archive (2021)
9. Gabizon, A.: From airs to raps - how plonk-style arithmetization works (2021), https://hackmd.io/@aztec-network/plonk-arithmetiization-air
10. Gabizon, A., Williamson, Z.J., Ciobotaru, O.: Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive (2019)
11. Goldreich, O., Oren, Y.: Definitions and properties of zero-knowledge proof systems. Journal of Cryptology **7**(1), 1–32 (1994)
12. Groth, J.: On the size of pairing-based non-interactive arguments. In: Annual international conference on the theory and applications of cryptographic techniques. pp. 305–326. Springer (2016)
13. Haböck, U.: Multivariate lookups based on logarithmic derivatives. Cryptology ePrint Archive, Paper 2022/1530 (2022), https://eprint.iacr.org/2022/1530
14. He, K., Zhang, X., Ren, S., Sun, J.: 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. pp. 630–645. Springer (2016)
15. Kang, D., Hashimoto, T., Stoica, I., Sun, Y.: Scaling up trustless dnn inference with zero-knowledge proofs. arXiv preprint arXiv:2210.08674 (2022)
16. LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. nature **521**(7553), 436–444 (2015)
17. Lee, S., Ko, H., Kim, J., Oh, H.: vcnn: Verifiable convolutional neural network based on zk-snarks. Cryptology ePrint Archive (2020)
18. Liu, T., Xie, X., Zhang, Y.: Zkcnn: Zero knowledge proofs for convolutional neural network predictions and accuracy. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. pp. 2968–2985 (2021)
19. Pauwels, P., Pirovich, J., Braunz, P., Deeb, J.: zkkyc in defi: An approach for implementing the zkkyc solution concept in decentralized finance. Cryptology ePrint Archive (2022)
20. Petkus, M.: Why and how zk-snark works. arXiv preprint arXiv:1906.07221 (2019)
21. Reddi, V.J., Cheng, C., Kanter, D., Mattson, P., Schmuelling, G., Wu, C.J., Anderson, B., Breughe, M., Charlebois, M., Chou, W., et al.: Mlperf inference benchmark. In: 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). pp. 446–459. IEEE (2020)
22. Sefranek, M.: How (not) to simulate PLONK. Cryptology ePrint Archive, Paper 2024/848 (2024), https://eprint.iacr.org/2024/848
23. Setty, S.: Spartan: Efficient and general-purpose zkSNARKs without trusted setup. Cryptology ePrint Archive, Paper 2019/550 (2019), https://eprint.iacr.org/2019/550
24. Succinct: Succinct (2025), https://www.succinct.xyz/
25. Sun, H., Li, J., Zhang, H.: zkllm: Zero knowledge proofs for large language models. In: Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security. pp. 4405–4419 (2024)
26. Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al.: Llama: Open and efficient foundation language models. arXiv preprint arXiv:2302.13971 (2023)

27. Waiwitlikhit, S., Stoica, I., Sun, Y., Hashimoto, T., Kang, D.: Trustless audits without revealing data or models. arXiv preprint arXiv:2404.04500 (2024)
28. Weng, J., Weng, J., Tang, G., Yang, A., Li, M., Liu, J.N.: pvcnn: Privacy-preserving and verifiable convolutional neural network testing. arXiv preprint arXiv:2201.09186 (2022)
29. zcash: halo2 (2022), https://zcash.github.io/halo2/
30. Zkonduit Inc.: ezkl (2023), https://docs.ezkl.xyz/

A Code

We are planning on releasing our open source implementation soon.