Scalable Zero-knowledge Proofs for Non-linear Functions in Machine Learning

Meng Hao, Hanxiao Chen, and Hongwei Li, *School of Computer Science and Engineering, University of Electronic Science and Technology of China;* Chenkai Weng, *Northwestern University;* Yuan Zhang and Haomiao Yang, *School of Computer Science and Engineering, University of Electronic Science and Technology of China;* Tianwei Zhang, *Nanyang Technological University*

https://www.usenix.org/conference/usenixsecurity24/presentation/hao-meng-scalable

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.

Scalable Zero-knowledge Proofs for Non-linear Functions in Machine Learning

Meng Hao[1,*], Hanxiao Chen[1,*], Hongwei Li[1,✉], Chenkai Weng[2], Yuan Zhang[1], Haomiao Yang[1], and Tianwei Zhang[3]

[1]School of Computer Science and Engineering, University of Electronic Science and Technology of China
[2]Northwestern University
[3]Nanyang Technological University
{menghao303, chenhanxiao.chx}@gmail.com, {hongweili, zhangyuan, haomyang}@uestc.edu.cn, chenkaiweng2024@u.northwestern.edu, tianwei.zhang@ntu.edu.sg

Abstract

Zero-knowledge (ZK) proofs have been recently explored for the integrity of machine learning (ML) inference. However, these protocols suffer from high computational overhead, with the primary bottleneck stemming from the evaluation of *non-linear* functions. In this paper, we propose the first systematic ZK proof framework for non-linear mathematical functions in ML using the perspective of *table lookup*. The key challenge is that table lookup cannot be directly applied to non-linear functions in ML since it would suffer from inefficiencies due to the intolerably large table. Therefore, we carefully design several important building blocks, including digital decomposition, comparison, and truncation, such that they can effectively utilize table lookup with a quite small table size while ensuring the soundness of proofs. Based on these building blocks, we implement complex mathematical operations and further construct ZK proofs for current mainstream non-linear functions in ML such as ReLU, sigmoid, and normalization. The extensive experimental evaluation shows that our framework achieves $50 \sim 179\times$ runtime improvement compared to the state-of-the-art work, while maintaining a similar level of communication efficiency.

1 Introduction

Machine-learning-as-a-service (MLaaS) provides powerful platforms for ML-based inference and prediction as a paid service. However, the inference process is black-boxed to clients and hence it is difficult to validate the service integrity, e.g., the inference results are evaluated by legitimate ML models with a correct inference specification. Recently, to address this problem, several works explore to design zero-knowledge (ZK) proofs in MLaaS, especially for the integrity of ML inference [16, 22, 24, 34, 37, 40, 57, 63]. Generally speaking, ZK proofs allow a prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ that a public program (i.e., statement) is correctly evaluated on $\mathcal{P}$'s secret input w (i.e., witness) without revealing additional information about w. Correspondingly, the goal of ZK proofs within MLaaS is to enable the service provider (as $\mathcal{P}$) to prove to clients (as $\mathcal{V}$) that the service is of high quality and inference is correctly evaluated by the particular model with secret parameters (as w), while preserving the model's privacy, more seriously, the intellectual property.

Unfortunately, the advanced ZK protocols for ML [40, 57] remain impractical and inefficient, particularly when applied to real-world complicated models such as convolutional neural networks (CNNs) [28, 35] or recently promising Transformer-based large language models (LLMs) like GPT [46, 53]. Upon meticulous examination, we identify that the primary bottleneck lies in the computation cost, primarily stemming from evaluating non-linear layers of ML[1], because the evaluation of those layers involves complex *non-linear mathematical functions* like comparison, exponentiation, division, and reciprocal square root. As exemplified in Mystique [57], the state-of-the-art ZK proofs for ML, the evaluation of non-linear functions consumes about 8 minutes for one inference on the ResNet-101 model, which accounts for more than 80% of the total inference runtime. Therefore, it is critical to design new techniques to solve the performance bottleneck of ZK proofs for non-linear functions, thereby facilitating the scalability and adoption of ZK protocols in ML. Furthermore, these ZK proofs have significant value beyond the field of ML and essentially can be used in any application involving non-linear evaluation, e.g., software vulnerabilities [8], program analysis [15] and database querying [39].

In this paper, we aim to address the above open problem by designing efficient ZK proofs for non-linear functions. Our key observation is that the adoption of heavy arithmetic-Boolean conversion is the main root cause for the inefficiency. In particular, current ZK proofs for ML evaluate linear layers on arithmetic circuits in a prime field $\mathbb{F}_p$. However, when evaluating non-linear functions, these arithmetic outputs have to

*The authors contributed equally and conducted this work at NTU.
✉Corresponding author.

[1]ML is comprised of alternating *linear* and *non-linear* layers. The former includes convolutional and fully connected layers, while the latter includes ReLU, GELU, Softmax, Maxpooling, and batch/layer normalization.

be converted to Boolean values via various bit decomposition techniques such as zk-edaBits [2, 57], so that the non-linear functions can be evaluated using general Boolean circuits in the ZK environment. Unfortunately, these conversion proofs are high cost and cause at least $O(\log p)$ multiplication complexity[2], due to the invocation of modulo-addition circuits in the Boolean field [57]. In addition, the subsequent function evaluation in Boolean circuits also causes a substantial overhead ($O(\log p)$ with a big constant), e.g., 3∼11K multiplication gates are needed for exponentiation, division, and reciprocal square root [57].

To tackle this issue, we propose a novel scalable ZK proof framework for non-linear mathematical functions. Our main insight is to explore *table lookup*-based ZK proofs. Building upon this technique, our framework can avoid the expensive arithmetic-Boolean conversion and Boolean circuit evaluation, by building a table to map the input to the output over arithmetic values. However, the key challenge is that table lookup cannot be directly applied to non-linear functions in ML since it would suffer from inefficiencies due to the oversized table (Section 2.1). Our solution starts by decomposing inputs of large bitlength into several smaller digits to significantly reduce the table size. Nevertheless, it is non-trivial to utilize these digits for function evaluation due to result correctness and proof soundness issues. To this end, we design several important building blocks from scratch, such as comparison and truncation, with new constructions. As a result, our building blocks have an asymptotic multiplication complexity of $O(1)$ in the amortized setting[3] (Section 2.2).

Based on these efficient building blocks, we construct ZK proofs for various non-linear mathematical functions in ML. We conduct extensive experiments to evaluate these protocols and the results show unprecedented efficiency breakthroughs (Section 7). Compared to the state-of-the-art Mystique [57], our protocols for widely used non-linear functions in ML, such as ReLU, sigmoid, GELU, obtain $50 \sim 179\times$ runtime improvement, while achieving $1.2 \sim 4.8\times$ better communication cost. Our contributions can be summarized as follows.

- We propose the first systematic ZK proof framework for non-linear mathematical functions in ML using the perspective of table lookup.
- We present several building blocks with newly proposed table lookup-based techniques, which have $O(1)$ multiplication complexity in the amortized setting.
- We apply these building blocks to various non-linear mathematical functions of ML and conduct extensive evaluation. The results show that our protocols achieve $50 \sim 179\times$ runtime improvement compared with the state-of-the-art work while maintaining a similar level of communication efficiency.

[2]Note that multiplication gates, including both Boolean AND and arithmetic multiplication, dominate the computation overhead of ZK proofs [57–60].

[3]Similarly, the state-of-the-art ZK proofs [2, 57] also design their protocols in the amortized setting.

2 Technical Overview

2.1 New perspective from table lookup

We explore using table lookup for ZK proofs of non-linear functions. Our table lookup-based ZK proof works as follows. The prover $\mathcal{P}$ and the verifier $\mathcal{V}$ pre-compute a public table that stores all legitimate input-output pairs of the evaluated non-linear function, and then $\mathcal{P}$ can prove to $\mathcal{V}$ that the computed output along with its input exists in this table. We instantiate our table lookup protocol by taking recent techniques from ZK proofs of read-only memory (ROM) access [10, 18, 60], which originally aimed to perform batched memory accesses in a verifiable manner (Section 3.5).

However, constructing ZK protocols for non-linear functions from table lookup is not straightforward and challenging. Specifically, to faithfully evaluate a non-linear function $y = f(x)$, the lookup table requires storing all possible input-output pairs $\{x_i, y_i = f(x_i)\}_{x_i \in \mathbb{F}_p}$. Unfortunately, for a typically used arithmetic field $\mathbb{F}_p$, e.g., a 61-bit prime p [2, 57], the table size $T \approx 2^{61}$ would become intolerably large. Such a size will directly lead to an explosion in the number of arithmetic multiplications, thereby a sharp drop in performance.

We address the above challenge by first decomposing inputs with large bitlength into a constant number of smaller digits, e.g., $5 \sim 12$ bits. We note that the size of each resulting table is only $2^5 \sim 2^{12}$ and it does not impose a burden on storage costs. Nevertheless, it is non-trivial to utilize these small tables for the ZK-based non-linear mathematical functions due to result correctness and proof soundness issues. To solve these problems, we further design a series of novel protocols (Section 2.2), such as comparison and truncation, leveraging the table lookup technique.

Remark. There are also some other potential methods called lookup arguments [14, 20, 45, 51, 61, 62], such as Caulk [61] and Lasso [51], to instantiate the table lookup-based ZK proofs. We currently choose the technique from ZK proofs of ROM since they are computation-efficient, especially in batched lookup settings. This is suitable for ML scenarios that repeatedly evaluate a large number of mathematical functions at each non-linear layer. We re-run the source code of Caulk [61], and the results show that the ZK lookup protocol of Caulk costs 177.451ms for each amortized access on a table of size 2^{12}, while our protocol from ZK-ROM only requires 0.069ms (about $2571\times$ better) for the same setting. Lasso [51] is customized for restricted *structured* tables, namely decomposability or low-degree extension structures (refer to their paper for more details), which cannot be extended to our setting.

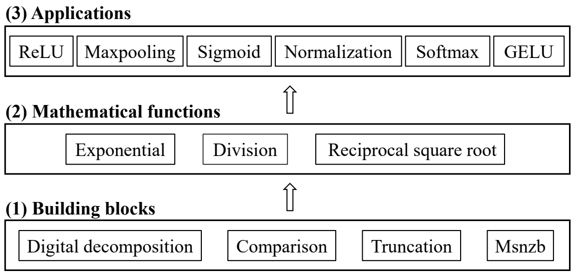

Figure 1: A high-level overview of our framework.

2.2 Novel table lookup-based protocols

Our framework can be summarized into three hierarchical levels, as depicted in Figure 1. Below, we outline the protocols inside in a bottom-up manner. (1) *Fundamental building blocks.* As the most important basis of our framework, this level consists of four operations, i.e., digital decomposition, comparison, truncation, and most significant non-zero-bit (Section 4). All enjoy an asymptotic multiplication complexity of $O(1)$ in the amortized setting, independent of the input bitlength $\log p$. (2) *Complex mathematical functions.* Building on the developed building blocks, we design efficient ZK proofs for complex mathematical functions including exponential, division, and reciprocal square root (Section 5). (3) *Comprehensive ML applications.* Relying on the above ZK proofs for mathematical functions as well as our building blocks, we provide efficient ZK protocols for a lot of widely used non-linear functions in ML (Section 6), including ReLU, Maxpooling, Sigmoid, Normalization, Softmax, and GELU.

Due to being important and challenging, below we introduce the insights of four fundamental building blocks.

Digital decomposition. To address the prohibitively large size of lookup tables as discussed in Section 2.1, a natural idea is to decompose the input x into a constant number k of small digits $x_0,\ldots,x_{k-1}$. We formalize this operation as digital decomposition, which importantly runs throughout our entire framework. Formally, given an input $x \in \mathbb{F}_p$, this operation outputs $x_0,\ldots,x_{k-1} \in \mathbb{F}_p$ such that $x = x_{k-1}\|\ldots\|x_0$ and $x_i \in \{0,1\}^{d_i}$ for $i \in [0,k-1]$.

Unfortunately, a subtle issue arises. Specifically, the malicious prover may provide incorrect $x_0,\ldots,x_{k-1}$ satisfying that $x_{k-1}\|\ldots\|x_0 = x+p$ instead of x. This incorrect decomposition would be successfully verified because the verification $x_{k-1}\|\ldots\|x_0 - x = 0$ is performed in $\mathbb{F}_p$, meaning that all operations are taken over modulo p. The root of the problem is that for an ℓ-bit prime p for $\mathbb{F}_p$, the prover could decompose x in a larger range $[0,2^\ell-1]$ rather than $[0,p-1]$, where $p < 2^\ell$. Therefore, an additional effort is required to check that $x_{k-1}\|\ldots\|x_0 < p$. This operation will be handled later by our comparison protocol. Moreover, we also present a positive digital decomposition protocol, which is customized to positive inputs, to bypass this issue.

Comparison. ZK proofs of comparison verification are used to verify whether $x < c$ holds, where $x \in \mathbb{F}_p$ is the prover's secret witness and $c \in \mathbb{F}_p$ is a public constant. Our idea is to convert the comparison verification on the entire x and c with large bitlength into a set of carefully designed operations on smaller digits. We take inspiration from the observation [21, 48]: $1\{x<c\} = 1\{x_1<c_1\} + 1\{x_1 = c_1\}\cdot 1\{x_0<c_0\}$, where $x = x_1\|x_0$ and $c = c_1\|c_0$. Further, this relation can be recursively invoked when $x = x_{k-1}\|\ldots\|x_0$ and $c = c_{k-1}\|\ldots\|c_0$.

With the above relation, our basic idea for ZK proofs of comparison verification consists of two steps. (1) For each $i \in [0,k-1]$, given the digits (x_i,c_i) with bitlength d_i, we evaluate $z_i^{lt} = 1\{x_i<c_i\}$ and $z_i^{eq} = 1\{x_i=c_i\}$ by calling two table lookups. (2) $1\{x<c\}$ is verified by using z_i^{lt} and z_i^{eq} for $i \in [0,k-1]$ according to the above relation. However, the computation cost of this solution is still large due to the requirement of $2k$ table lookups and k multiplications. Therefore, we further make two important improvements. First, rather than recursively invoking k multiplications to verify $1\{x<c\}$ in step (2), we use only one table lookup by carefully constructing a table containing $z = z_{k-1}^{lt}\|\ldots\|z_0^{lt}\|z_{k-1}^{eq}\|\ldots\|z_0^{eq} \in \{0,1\}^{2k}$ and the corresponding result of $1\{x<c\}$ from z. Second, with the first insight, we further reduce $2k$ invocations of table lookup of step (1) into k times, by designing a compact encoding method that combines z_i^{lt} and z_i^{eq} into one value. Totally, our optimized proofs only invoke $k+1$ table lookups.

Truncation. Truncation on positive values x can be directly evaluated by invoking our positive digital decomposition protocol. Namely, given the truncation bitlength t, it outputs x_1 such that $x = x_1\|x_0$ and $x_0 \in \{0,1\}^t$. However, when we consider the general case that supports arbitrary inputs in $\mathbb{F}_p$, the ZK proofs of truncation become challenging. The main reason is that a negative value x is embedded in $\mathbb{F}_p$ as $p-|x|$ and hence given $p-|x| = x_1\|x_0$, the digit x_1 decomposed from $p-|x|$ is not a correct truncation result.

To address this challenge, an important observation is that the truncation operation conducts an arithmetic right shift on the 2's complement representation of the real value x, rather than its embedded field representation. Hence our insight is that we can convert the truncation on a negative x into the positive one by flipping all bits of x in the 2's complement. Specifically, the result y below is still the correct output of truncation on a negative x, if we (1) compute positive $\bar{x}$ by flipping all the bits of x in the 2's complement, i.e., 0 to 1 and 1 to 0, (2) compute $\bar{y}$ by performing positive truncation on $\bar{x}$, and (3) obtain y by flipping all the bits of $\bar{y}$ in the 2's complement. Therefore, the ZK proofs of general truncation can be achieved by invoking our comparison and positive truncation protocols.

Msnzb. The most significant non-zero-bit (Msnzb) computes the index y on a positive input x, such that if $x_y = 1$ then $x_i = 0$ for all $i > y$. Alternatively, we have $2^y \le x \le 2^{y+1}-1$. This function is currently explored in secure multi-party computation (MPC) works [47]. In their protocol, the input x is decomposed into several digits $x_0,\dots,x_{k-1}$, and the Msnzb is computed on each x_i of these digits. Finally, the output corresponds to $y = \mathsf{Msnzb}(x_i) + i \cdot d$ if $x_i \neq 0$ and $x_j > 0$ for all $j > i$, where d is the bitlength of each digit. Although this method can be directly migrated to the ZK-based evaluation, utilizing our digital decomposition and table lookup protocols, the cost is significantly high due to the requirement of multiple Msnzb, comparison, and multiplication operations.

We re-think this function and find that the property $2^y \leq x \leq 2^{y+1} - 1$ is overlooked. We further observe that except y, we could let the prover provide additional values $z_0 = 2^y$ and $z_1 = 2^{y+1} - 1$, and then verify that $z_0 \leq x \leq z_1$. This verification can be implemented via the table lookup technique, where a table L is constructed containing (y, z_0, z_1) for all possible y. It is worth noting that the table L is quite small with size $\lceil \log p \rceil - 1$. However, simply constructing this table as above is unreasonable, since when y is large, z_1 will overflow the range of positive values in $\mathbb{F}_p$. We address this problem by replacing the overflowed z_1 with $\frac{p-1}{2}$ without affecting correctness, based on a rigorous analysis.

Functionality $\mathcal{F}_{\mathsf{ZK}}$

This functionality is parameterized by a prime p such that $p \geq 2^\lambda$.

Input: On receiving (Input, x) from $\mathcal{P}$, store x and send $[x]_p$ to $\mathcal{P}$ and $\mathcal{V}$.

Affine Combination: On receiving $(\mathsf{Affine}, c_0, c_1, \dots, c_n, [x_1]_p, \dots, [x_n]_p)$ from $\mathcal{P}$ and $\mathcal{V}$, check that $[x_1]_p, \dots, [x_n]_p$ are valid and abort if not. Compute $y = c_0 + \sum_{i \in [1,n]} c_i \cdot x_i$ in $\mathbb{F}_p$, store y, and send $[y]_p$ to $\mathcal{P}$ and $\mathcal{V}$.

Multiply: On receiving $(\mathsf{Mult}, [x]_p, [y]_p)$ from $\mathcal{P}$ and $\mathcal{V}$, check that $[x]_p, [y]_p$ are valid and abort if not. Compute $z = x \cdot y$ in $\mathbb{F}_p$, store z, and send $[z]_p$ to $\mathcal{P}$ and $\mathcal{V}$.

Output: On receiving $(\mathsf{Output}, [z]_p)$ from $\mathcal{P}$ and $\mathcal{V}$, check if $[z]_p$ is valid and abort if the check fails, otherwise send z to $\mathcal{V}$.

Figure 2: Ideal functionality for ZK proofs.

3 Preliminaries

3.1 Notation

We use κ and λ to denote the computational and statistical security parameters, respectively. For $a, b \in \mathbb{Z}$ with $a < b$, we use $[a, b]$ to denote $\{a, \dots, b\}$ and $(a, b]$ to denote $\{a + 1, \dots, b\}$. $a := b$ denotes that a is assigned by b. We use $x \leftarrow S$ to denote sampling x uniformly at random from a finite set S. $\mathsf{negl}(\cdot)$ denotes a negligible function such that $\mathsf{negl}(\kappa) = o(\kappa^{-c})$ for every positive constant c.

Fixed-Point Representation. Same as prior works [40,57], we encode a real number $\hat{x} \in \mathbb{R}$ as a field element $x \in \mathbb{F}_p$ using their fixed-point representation. The representation in $\mathbb{F}_p$ is parameterized by a fixed *scale* variable, s, which refers to the fractional bitlength. We define two mappings for mutual conversion between reals and their field representation.

- $\mathsf{R2F} : \mathbb{R} \to \mathbb{F}_p$. The mapping from reals to its field representation is $\mathsf{R2F}(x, p, s) = \lfloor x \cdot 2^s \rfloor \bmod p$.
- $\mathsf{F2R} : \mathbb{F}_p \to \mathbb{R}$. The mapping from the field representation to reals is $\mathsf{F2R}(x, p, s) = (x - c \cdot p)/2^s$, where the operations are over $\mathbb{R}$ and $c = 1\{x > (p-1)/2\}$.

We sometimes omit s, meaning that $s = 0$, i.e., the conversions are between signed integers and their field representation. Hence, $\mathbb{F}_p$ can encode signed integers between $[-\frac{p-1}{2}, \frac{p-1}{2}]$.

3.2 Information-theoretic MACs

We commit values in $\mathbb{F}_p$ using information-theoretic message authentication codes (IT-MACs) [4,43]. Let $\Delta \in \mathbb{F}_p$ be a uniform global key known only to the verifier $\mathcal{V}$. A commitment on a message $x \in \mathbb{F}_p$ is denoted by $[x]_p$, meaning that the prover $\mathcal{P}$ holds $x \in \mathbb{F}_p$ and a MAC $M_x \in \mathbb{F}_p$, and the verifier $\mathcal{V}$ holds a uniform local key $K_x \in \mathbb{F}_p$ such that $M_x = K_x + \Delta \cdot x$ in $\mathbb{F}_p$. When we say both parties hold $[x]_p$, it means that $\mathcal{P}$ holds (x, M_x) and $\mathcal{V}$ holds (Δ, K_x). IT-MACs are *additively homomorphic*, meaning that given public constants $c_0, c_1, \dots, c_n \in \mathbb{F}_p$ and commitments $[x_1]_p, \dots, [x_n]_p$, $\mathcal{P}$ and $\mathcal{V}$ can locally compute $[y]_p := c_0 + \sum_{i \in [1,n]} c_i \cdot [x_i]_p$. A commitment $[x]_p$ can be opened by having $\mathcal{P}$ send (x, M_x) to $\mathcal{V}$, who checks $M_x = K_x + \Delta \cdot x$ with the probability of forging an incorrect x at most $1/|\mathbb{F}_p|$. When opening n all zero values $[x_1]_p, \dots, [x_n]_p$, called $\mathsf{CheckZero}$, $\mathcal{P}$ computes $h := \mathsf{H}(M_{x_1}, \dots, M_{x_n})$ and sends h to $\mathcal{V}$, who checks whether $h = \mathsf{H}(K_{x_1}, \dots, K_{x_n})$, where $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\kappa$ is a hash function modeled as a random oracle. The soundness error of $\mathsf{CheckZero}$ is at most $1/|\mathbb{F}_p| + q/2^\kappa$, where q is the number of queries to the random oracle [9]. Random IT-MAC commitments can be efficiently generated using the recent LPN-based Vector Oblivious Linear Evaluation (VOLE) protocols [5,49], which have communication complexity sublinear in the number of commitments.

3.3 Zero-knowledge proofs

Zero-knowledge proofs (of knowledge) are interactive two-party protocols that allow the prover $\mathcal{P}$ to convince a verifier $\mathcal{V}$ that a certain statement is true on a private witness. Instead of using the classical definition by Goldwasser et al. [27], we define it as an ideal functionality $\mathcal{F}_{\mathsf{ZK}}$ for circuit satisfiability in Figure 2. It belongs to the commit-and-prove paradigm and uses information-theoretic MACs as commitments. The functionality directly implies the standard properties, i.e., completeness, knowledge soundness, and zero knowledge. The functionality can be instantiated using several existing ZK protocols, but in our implementation, we use the recent VOLE-based interactive designated-verifier ZK proofs [56–59] due to their fast prover time and small memory footprint.

3.4 ZK proofs of read-only memory access

We introduce ZK proofs for read-only memory access (ROM) [11, 18, 60], which are the basis of our table lookup. This functionality allows the prover $\mathcal{P}$ to commit a size-T memory containing $m_0, \ldots, m_{T-1}$, and then when $\mathcal{P}$ accesses an element at address $i \in [0, T-1]$ from the memory, $\mathcal{P}$ proves to the verifier $\mathcal{V}$ that the read value is m_i. Note that recent works are customized for batch ROM settings, meaning that $\mathcal{P}$ proves that N accesses on a size-T memory are correct.

We review the state-of-the-art ZK ROM protocol in arithmetic circuits [60]. The protocol contains three phases: setup, access, and cleanup. (1) In the setup phase, $\mathcal{P}$ and $\mathcal{V}$ initialize two triple vectors, reads and writes, in which each triple consists of an access address, an access value, and a metadata called version. Then, for the i-th element in memory, where $i \in [0, T-1]$, $\mathcal{P}$ and $\mathcal{V}$ append $([i]_p, [m_i]_p, [0]_p)$ to writes. (2) The access phase can be performed N times. In each access of the j-th element in memory for $j \in [0, T-1]$, $\mathcal{P}$ and $\mathcal{V}$ append $([j]_p, [m_j]_p, [v_j]_p)$ to reads, where v_j is the latest version of m_j in writes, while appending $([j]_p, [m_j]_p, [v_j+1]_p)$ to writes. (3) Finally, in the cleanup phase, for the i-th element in memory, where $i \in [0, T-1]$, $\mathcal{P}$ and $\mathcal{V}$ append $([i]_p, [m_i]_p, [v_i]_p)$ to reads, where v_i is the latest version of m_i in writes. The insight is that each access to ROM is correct if and only if reads is a permutation of writes. The permutation proof costs $2 \cdot (T+N)$ multiplications. Note that this protocol can be generalized to support a key-value store (i.e., the address space is an arbitrary set) and multiple values (i.e., the value space includes an arbitrary number of values). We refer the reader to Section 4 and Appendix C of the work [60] for detailed protocols and soundness analysis.

3.5 ZK proofs of table lookup from ZK-ROM

Table lookup is our main insight to evaluate non-linear functions, in which $\mathcal{P}$ and $\mathcal{V}$ pre-compute a public table storing all legitimate input-output pairs of the evaluated non-linear function, and then $\mathcal{P}$ proves to $\mathcal{V}$ that the computed output along with its input exists in this table. We extend the ZK functionality of Figure 2 with table lookup, and present the augmented functionality in Figure 3. We also include the procedure of range check, which is a simplified variant of table lookup with the exception that the output is empty.

We instantiate these protocols using ZK proofs from the aforementioned ZK-ROM, where the access address and value in ZK-ROM now respectively correspond to the input and output. The detailed table lookup protocol is shown in the full version. For N lookups on a size-T table, the computation complexity is $T + 2 \cdot N$ multiplication gates, reducing T multiplications compared to the original ZK-ROM. The reason is that our table is always public, and hence in the setup phase, we can append T tuples in plaintext into writes. In our ML application, it holds $N \gg T$, thus the amortized computation complexity per lookup is 2 multiplications. We emphasize that batch lookup is very reasonable in ML, since ML repeatedly evaluates non-linear functions a large number of times, e.g., 800K ReLUs in a layer of ResNet50 [28, 48].

Functionality $\mathcal{F}_{\mathsf{ZK}}^{Lookup}$

This functionality extends the instructions in $\mathcal{F}_{\mathsf{ZK}}$.

CheckLookup: On receiving $(\mathsf{Lookup}, L, [x]_p, [y]_p)$ from $\mathcal{P}$ and $\mathcal{V}$, check if $[x]_p, [y]_p$ are valid and $(x, y) \in L$, and output abort to $\mathcal{V}$ if the check fails, otherwise output success.

CheckRange: On receiving $(\mathsf{Range}, R, [x]_p)$ from $\mathcal{P}$ and $\mathcal{V}$, check if $[x]_p$ is valid and $x \in R$, and output abort to $\mathcal{V}$ if the check fails, otherwise output success.

Figure 3: Ideal functionality for ZK proofs of table lookup and range check.

4 Building Blocks

In this section, we present several crucial building blocks and outline their ideal functionalities in Figure 4. These components serve as the foundational elements for ZK proofs of non-linear mathematical functions introduced later.

4.1 Digital decomposition

As discussed in Section 2.1, to avoid the oversized lookup table, a natural idea is decomposing the input x into k small digits $x_0, \ldots, x_{k-1}$ before employing table lookup. Formally, the digital decomposition operation decomposes $x \in \mathbb{F}_p$ into $x_0, \ldots, x_{k-1} \in \mathbb{F}_p$ such that $x = x_{k-1} \| \ldots \| x_0$ and $x_i \in \{0,1\}^{d_i}$ for $i \in [0, k-1]$. This operation can be viewed as a generalized form of bit decomposition [40, 57] when all d_i's are set as 1. However, it is worth noting that the number k of output digits is a *constant*, rather than the prime bitlength $\lceil \log p \rceil$ in bit decomposition. This advantage effectively ensures a constant asymptotic multiplication complexity of our protocols. In the following, we first detail how to perform digital decomposition specific to positive inputs, and then discuss general digit decomposition for arbitrary values.

Positive digital decomposition. To decompose an input $x \in [0, \frac{p-1}{2}]$, we ask the prover to provide the decomposed dig-

Functionality $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$

This functionality extends the instructions in $\mathcal{F}_{\mathsf{ZK}}$.

Positive digital decomposition: On input (DigitDec, $[x]_p$, $d_0,\dots,d_{k-1}$) from $\mathcal{P}$ and $\mathcal{V}$, where $x \in [0, \frac{p-1}{2}]$, check that $[x]_p$ is valid and abort if not. Decompose x to $(x_0,\dots,x_{k-1})$ such that $x = x_{k-1}\|\dots\|x_0$ and $x_i \in \{0,1\}^{d_i}$ for $i \in [0,k-1]$. Then, for $i \in [0,k-1]$, store x_i and send $[x_i]_p$ to $\mathcal{P}$ and $\mathcal{V}$.

Comparison verification: On input (VrfyCmp, $[x]_p, c$) from $\mathcal{P}$ and $\mathcal{V}$, where $x \in \mathbb{F}_p$, check that $[x]_p$ is valid and abort if not. Check whether $x < c$, and output abort to $\mathcal{V}$ if the check fails, otherwise output $\mathsf{success}$.

Comparison: On input (Cmp, $[x]_p, c$) from $\mathcal{P}$ and $\mathcal{V}$, $x \in \mathbb{F}_p$, check that $[x]_p$ is valid and abort if not. Compute $y = 1\{x < c\}$, store y and send $[y]_p$ to $\mathcal{P}$ and $\mathcal{V}$.

Positive truncation: On input (PosTrunc, $[x]_p, t$) from $\mathcal{P}$ and $\mathcal{V}$, where $x \in [0, \frac{p-1}{2}]$, check that $[x]_p$ is valid and abort if not. Compute $y = \mathsf{R2F}(\mathsf{F2R}(x,p)/2^t, p)$, store y, and send $[y]_p$ to $\mathcal{P}$ and $\mathcal{V}$.

General truncation: On input (Trunc, $[x]_p, t$) from $\mathcal{P}$ and $\mathcal{V}$, where $x \in \mathbb{F}_p$, check that $[x]_p$ is valid and abort if not. Compute $y = \mathsf{R2F}(\mathsf{F2R}(x,p)/2^t, p)$, store y, and send $[y]_p$ to $\mathcal{P}$ and $\mathcal{V}$.

Most significant non-zero-bit: On input (Msnzb, $[x]_p$) from $\mathcal{P}$ and $\mathcal{V}$, where $x \in (0, \frac{p-1}{2}]$, check that $[x]_p$ is valid and abort if not. Compute y such that $2^y \le x \le 2^{y+1}-1$, store y, and send $[y]_p$ to $\mathcal{P}$ and $\mathcal{V}$.

Figure 4: Ideal functionality for ZK proofs of our building blocks.

its $\{x_0,\dots,x_{k-1}\}$ of x. Then the protocol verifies that (1) for $i \in [0,k-1]$, $x_i \in \{0,1\}^{d_i}$, by invoking the $\mathsf{CheckRange}$ procedure of functionality $\mathcal{F}_{\mathsf{ZK}}^{Lookup}$, and (2) $\{x_0,\dots,x_{k-1}\}$ constitute the digit decomposition of x, by determining whether $x_0 + \sum_{i\in[1,k-1]} 2^{\sum_{j\in[0,i-1]} d_j} x_i = x$ based on the $\mathsf{CheckZero}$ procedure. The detailed protocol Π_{DigitDec} is illustrated in Figure 5. The dominant cost of this protocol is k range checks, which consume $2k$ multiplication gates.

General digital decomposition. Before presenting a general digital decomposition construction, it is essential to address why protocol Π_{DigitDec} cannot be directly applied to arbitrary values in $\mathbb{F}_p$. For an ℓ-bit prime p, where $p < 2^\ell$ obviously, a malicious prover could decompose x in a larger range $[0, 2^\ell - 1]$ instead of $[0, p-1]$ such that $x_{k-1}\|\dots\|x_0 = x + p$, rather than x. Nonetheless, these results would still pass the verification strategy in protocol Π_{DigitDec} mainly because the $\mathsf{CheckRange}$ procedure is taken over modulo p. This malicious behavior does not occur in the positive case. The reason is that $x_{k-1}\|\dots\|x_0$ should be represented by at most $\ell-1$ bits, namely $x_{k-1}\|\dots\|x_0 < 2^{\ell-1}$ due to $x \le \frac{p-1}{2} < 2^{\ell-1}$. It is a contradiction that $x_{k-1}\|\dots\|x_0 = x + p$ since $p > 2^{\ell-1}$ and should be represented by ℓ bits. To address this issue, it is necessary to add an extra check to ensure that the output digits $x_{k-1}\|\dots\|x_0 < p$. This is precisely addressed with our comparison verification protocol detailed in Section 4.2. Consequently, the ZK proofs of general digital decomposition can be straightforwardly derived by integrating our positive digital decomposition protocol in Figure 5 and comparison verification protocol in Figure 6. Note that we only provide the positive digital decomposition protocol here because it suffices for our work.

Protocol Π_{DigitDec}

Parameters: A finite field $\mathbb{F}_p$, a constant k.

Input: $\mathcal{P}$ and $\mathcal{V}$ have an authenticated value $[x]_p$ and digital bitlengths $d_0,\dots,d_{k-1}$, where $x \in [0, \frac{p-1}{2}]$.

Protocol execution: $\mathcal{P}$ and $\mathcal{V}$ compute $[x_0]_p,\dots,[x_{k-1}]_p$ such that $x = x_{k-1}\|\dots\|x_0$ and $x_i \in \{0,1\}^{d_i}$ for $i \in [0,k-1]$ as follows:

1. $\mathcal{P}$ decomposes x into $(x_0,\dots,x_{k-1})$ such that $x = x_{k-1}\|\dots\|x_0$ and $x_i \in \{0,1\}^{d_i}$ for $i \in [0,k-1]$.
2. $\mathcal{P}$ sends $(\mathsf{Input}, x_0,\dots,x_{k-1})$ to functionality $\mathcal{F}_{\mathsf{ZK}}$, which returns $([x_0]_p,\dots,[x_{k-1}]_p)$ to $\mathcal{P}$ and $\mathcal{V}$.
3. For $i \in [0,k-1]$, $\mathcal{P}$ and $\mathcal{V}$ send $(\mathsf{Range}, R_i, [x_i]_p)$ to functionality $\mathcal{F}_{\mathsf{ZK}}^{Lookup}$, where $R_i = \{0,1\}^{d_i}$, to verify that $x_i \in \{0,1\}^{d_i}$.
4. $\mathcal{P}$ and $\mathcal{V}$ compute $[z]_p = [x_0]_p + \sum_{i\in[1,k-1]} 2^{\sum_{j\in[0,i-1]} d_j}[x_i]_p - [x]_p$ and execute the $\mathsf{CheckZero}$ procedure on $[z]_p$.
5. If any of the above checks fails, $\mathcal{V}$ aborts. Otherwise, $\mathcal{P}$ and $\mathcal{V}$ output $[x_0]_p,\dots,[x_{k-1}]_p$.

Figure 5: Protocol for positive digital decomposition.

Observe that the main cost of protocol Π_{DigitDec} is dominated by the $\mathsf{CheckRange}$ procedure on $[x_i]_p$ with bitlength d_i. This overhead can be optimized when d_i is large. Briefly, instead of directly performing $\mathsf{CheckRange}$ on $[x_i]_p$ for large d_i, we can iteratively invoke the digital decomposition functionality on $[x_i]_p$. For convenience, we set an upper bound B (e.g., $B = 12$) and perform $\mathsf{CheckRange}$ only when $d_i \le B$.

Theorem 1. *Protocol* Π_{DigitDec} *UC-realizes the* $\mathsf{DigitDec}$ *command of functionality* $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$ *against static and malicious adversaries in the* $(\mathcal{F}_{\mathsf{ZK}}, \mathcal{F}_{\mathsf{ZK}}^{Lookup})$*-hybrid model.*

The proof of this theorem can be found in Appendix B.1.

4.2 Comparison

We consider two useful ZK proofs of comparison. One is the comparison verification to verify that $x < c$ holds, where $x \in \mathbb{F}_p$ is the prover's secret witness and $c \in \mathbb{F}_p$ is a public constant. The other is the general comparison operation that computes $y = 1\{x < c\}$. These proofs can be directly used to compute whether x is positive by setting $c = \frac{p+1}{2}$. Existing comparison proofs either utilize heavy bit decomposition [57] or introduce strong assumptions about the input range [2, 40]. We explore using the table lookup technique to address the efficiency problem without introducing additional assumptions. Below, for clarity, we focus on verifying that $x < c$ holds, and defer the general comparison in the full version.

Basic solution. As illustrated in Section 2.2, our solution recursively exploits the observation [21, 48]:

$$1\{x < c\} = 1\{x_1 < c_1\} + 1\{x_1 = c_1\} \cdot 1\{x_0 < c_0\}, \qquad (1)$$

where $x = x_1 \| x_0$ and $c = c_1 \| c_0$. Thus, given $x = x_{k-1} \| \ldots \| x_0$ and $c = c_{k-1} \| \ldots \| c_0$, a straightforward protocol to verify $1\{x < c\}$ is as follows. (1) For $i \in [0, k-1]$, given (x_i, c_i) with bitlength d_i, the prover and verifier evaluate $z_i^{lt} = 1\{x_i < c_i\}$ and $z_i^{eq} = 1\{x_i = c_i\}$ by calling table lookups. (2) After obtaining all z_i^{lt} and z_i^{eq}, we can compute $1\{x < c\}$ recursively based on Equation 1 by calling functionality $\mathcal{F}_{\mathsf{ZK}}$. Note that the corrupted server could provide an incorrect decomposition from $x + p$ instead of x. In this case, the verification will abort since $x + p < c$ must not be held. The overhead of this solution remains costly, primarily stemming from $2k$ evaluations of table lookup in step (1) and k multiplication gates in step (2). In the following, we show how to improve the basic method via two important insights.

Improved construction. The first insight is that in step (2), rather than recursively evaluating Equation 1 based on z_i^{lt} and z_i^{eq} for $i \in [0, k-1]$, we utilize table lookup by constructing a table L containing (z, y), where $z = z_{k-1}^{lt} \| \ldots \| z_0^{lt} \| z_{k-1}^{eq} \| \ldots \| z_0^{eq} \in \{0,1\}^{2k}$ and y is computed based on z according to Equation 1. Note that z is obtained via $z = 2^{2k-1} \cdot z_{k-1}^{lt} + \ldots + 2^k \cdot z_0^{lt} + 2^{k-1} \cdot z_{k-1}^{eq} + \ldots + 2^0 \cdot z_0^{eq}$. Intuitively, the table L consists of 2^{2k} entries, including all possible $z \in \{0,1\}^{2k}$. However, it is important to emphasize that the number of entries in L is explicitly 3^k. The reason is that for each pair (z_i^{lt}, z_i^{eq}), there can only be three possible cases, namely, $\{(0,0), (0,1), (1,0)\}$, since $x_i < c_i$ and $x_i = c_i$ can not hold simultaneously. If we overlook this point, these incorrect but still considered values might be maliciously manipulated to compromise soundness.

The second insight is that in step (1), for $i \in [0, k-1]$, we can employ table lookup only once by combining z_i^{lt} and z_i^{eq} into a single value. To this end, we design a compact encoding as follows

$$z_i = \underbrace{0 \ldots 0 \| \overbrace{z_i^{lt} \| 0 \ldots 0}^{i+1}}_{k} \| \underbrace{0 \ldots 0 \| \overbrace{z_i^{eq} \| 0 \ldots 0}^{i+1}}_{k}, \qquad (2)$$

where z_i consists of two parts, each has k bits. Except for the i-th position in each part where z_i^{lt} or z_i^{eq} is placed, the remaining $k-1$ bits are all 0. This encoding has two advantages. First, with this encoding, we can reduce $2k$ invocations of table lookup into k times. Second, when generating z as input of table lookup in the first optimization, $\mathcal{P}$ and $\mathcal{V}$ only need simple summations with this encoding, without constant multiplications. Note that z_i will not exceed the range of $\mathbb{F}_p$ by appropriately setting the value of k.

Based on the above discussion, we provide the detailed comparison verification protocol Π_{VrfyCmp} in Figure 6. This protocol mainly consists of $k+1$ table lookups, which consume $2k+2$ multiplication gates.

Theorem 2. *Protocol Π_{VrfyCmp} UC-realizes the* VrfyCmp *command of functionality $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$ against static and malicious adversaries in the $(\mathcal{F}_{\mathsf{ZK}}, \mathcal{F}_{\mathsf{ZK}}^{Lookup})$-hybrid model.*

The proof of this theorem can be found in Appendix B.2.

4.3 Truncation

Truncation (also known as arithmetic right shift) is widely used in fixed-point operations, especially after multiplication to maintain the fixed fractional precision. Given an input x and truncation bitlength t, the truncation operation outputs $y = \mathsf{R2F}(\mathsf{F2R}(x, p)/2^t, p)$. Below we provide two truncation protocols for positive and arbitrary values, respectively.

Positive truncation. We first present the truncation protocol on positive inputs. Our insight is that for a positive value $x \in [0, \frac{p-1}{2}]$, a t-bit truncation can be achieved by directly dropping out x_0 with t-bit and outputting x_1, where $x = x_1 \| x_0$. Thus, as illustrated in Figure 7, we can instantiate this protocol by simply leveraging the functionality of positive digital decomposition. Observe that this protocol has the same cost as positive digital decomposition, and hence requires $2k$ multiplication gates. We emphasize that this positive protocol is useful in several non-linear functions, in which there is some prior knowledge about the input domain. For example, the outputs of exponential are always positive and their multiplication can directly invoke this positive truncation.

General truncation. We further extend the positive truncation protocol into the general case to support arbitrary inputs $x \in \mathbb{F}_p$. This is challenging because a negative value x is embedded in $\mathbb{F}_p$ as $p - |x|$, and hence given $x_1 \| x_0 = p - |x|$, the digit x_1 decomposed from x is an incorrect result. Existing works do not address this challenge effectively. They either utilize expensive Boolean circuits [57] to evaluate this operation or only support positive values [40].

Building upon the insight illustrated in Section 2.2, we provide a novel protocol for general truncation in Figure 8. Specifically, we first invoke our comparison protocol to determine whether x is positive or negative. This is done to execute different operations for negative and positive values separately. For clarity, we below focus on the truncation of

Protocol Π_{VrfyCmp}

Parameters: A finite field $\mathbb{F}_p$, a constant k.

Input: $\mathcal{P}$ and $\mathcal{V}$ have an authenticated value $[x]_p$ and a constant c, where $x, c \in \mathbb{F}_p$.

Protocol execution: $\mathcal{P}$ and $\mathcal{V}$ verify that $x < c$ holds as follows:

1. $\mathcal{P}$ decomposes x into $(x_0, \dots, x_{k-1})$ such that $x = x_{k-1} \| \dots \| x_0$ where $x_i \in \{0,1\}^{d_i}$ for $i \in [0, k-1]$. $\mathcal{P}$ sends $(\mathsf{Input}, x_0, \dots, x_{k-1})$ to $\mathcal{F}_{\mathsf{ZK}}$, which returns $([x_0]_p, \dots, [x_{k-1}]_p)$ to $\mathcal{P}$ and $\mathcal{V}$.
2. $\mathcal{P}$ and $\mathcal{V}$ compute $[t]_p = [x_0]_p + \sum_{i\in[1,k-1]} 2^{\sum_{j\in[0,i-1]} d_j}[x_i]_p - [x]_p$, and execute the $\mathsf{CheckZero}$ procedure on $[t]_p$.
3. $\mathcal{P}$ and $\mathcal{V}$ locally decompose c into $(c_0, \dots, c_{k-1})$ such that $c = c_{k-1} \| \dots \| c_0$ and $c_i \in \{0,1\}^{d_i}$. For $i \in [0, k-1]$, $\mathcal{P}$ computes $z_i = 2^{k+i} \cdot 1\{x_i < c_i\} + 2^i \cdot 1\{x_i = c_i\}$, and sends $(\mathsf{Input}, z_0, \dots, z_{k-1})$ to $\mathcal{F}_{\mathsf{ZK}}$, which returns $([z_0]_p, \dots, [z_{k-1}]_p)$ to $\mathcal{P}$ and $\mathcal{V}$.
4. For $i \in [0, k-1]$, $\mathcal{P}$ and $\mathcal{V}$ send $(\mathsf{Lookup}, L_i, [x_i]_p, [z_i]_p)$ to functionality $\mathcal{F}_{\mathsf{ZK}}^{Lookup}$, where $L_i = \{(x_i, 2^{k+i} \cdot 1\{x_i < c_i\} + 2^i \cdot 1\{x_i = c_i\})\}_{x_i \in \{0,1\}^{d_i}}$.
5. $\mathcal{P}$ compute $y_0 = 1\{x_0 < c_0\}$ and $y_i = 1\{x_i < c_i\} + 1\{x_i = c_i\} \cdot y_{i-1}$ for $i \in [1, k-1]$, and set $y = y_{k-1}$. $\mathcal{P}$ sends (Input, y) to $\mathcal{F}_{\mathsf{ZK}}$, which returns $[y]_p$ to $\mathcal{P}$ and $\mathcal{V}$.
6. $\mathcal{P}$ and $\mathcal{V}$ compute $[z]_p = \sum_{i\in[0,k-1]} [z_i]_p$ and send $(\mathsf{Lookup}, L, [z]_p, [y]_p)$, where $L = \{(\sum_{i\in[0,k-1]} 2^{k+i} \cdot 1\{x_i < c_i\} + 2^i \cdot 1\{x_i = c_i\}, y_{k-1})\}_{x_i \in \{0,1\}^{d_i}}$. Here, $y_0 = 1\{x_0 < c_0\}$ and $y_i = 1\{x_i < c_i\} + 1\{x_i = c_i\} \cdot y_{i-1}$ for $i \in [1, k-1]$.
7. $\mathcal{P}$ and $\mathcal{V}$ execute the $\mathsf{CheckZero}$ procedure on $[y]_p - 1$.
8. If any of the above checks fails, $\mathcal{V}$ aborts, otherwise $\mathcal{V}$ outputs success.

Figure 6: Protocol for comparison verification.

negative values. For each negative $x \in [-\frac{p-1}{2}, -1]$, we first compute $\bar{x} = -1 \cdot x - 1$, which is the value that flips the bits of x in the 2's complement. Then, we compute $\bar{y}$ by invoking our positive truncation on $\bar{x}$. Note that $\bar{x} \in [0, \frac{p-1}{2} - 1]$ lies in the range of positive values, and hence it is correct for the positive truncation evaluation. Finally, we compute $y = -1 \cdot \bar{y} - 1$ that corresponds to flipping the bits of $\bar{y}$ in the 2's complement. This protocol mainly consists of comparison, multiplications, and positive truncation, which totally costs $4k + 4$ multiplication gates.

Theorem 3. *Protocol Π_{PosTrunc} UC-realizes the* $\mathsf{PosTrunc}$ *command of functionality $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$ against static and malicious adversaries in the $(\mathcal{F}_{\mathsf{ZK}}, \mathcal{F}_{\mathsf{ZK}}^{Lookup})$-hybrid model.*

Protocol Π_{PosTrunc}

Parameters: A finite field $\mathbb{F}_p$.

Input: $\mathcal{P}$ and $\mathcal{V}$ have an authenticated value $[x]_p$ and a truncation bitlength t, where $x \in [0, \frac{p-1}{2}]$.

Protocol execution: $\mathcal{P}$ and $\mathcal{V}$ compute $[y]_p$ such that $y = \mathsf{R2F}(\mathsf{F2R}(x, p)/2^t, p)$ as follows:

1. $\mathcal{P}$ and $\mathcal{V}$ send $(\mathsf{DigitDec}, [x]_p, t, m-t)$ to functionality $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$, which returns $([x_0]_p, [x_1]_p)$ such that $x = x_1 \| x_0$, $x_0 \in \{0,1\}^t$ and $x_1 \in \{0,1\}^{m-t}$, where $m = \lceil \log p \rceil - 1$.
2. $\mathcal{P}$ and $\mathcal{V}$ output $[y]_p = [x_1]_p$.

Figure 7: Protocol for positive truncation.

The proof of this theorem can be found in Appendix B.3.

Theorem 4. *Protocol Π_{Trunc} UC-realizes the* Trunc *command of functionality $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$ against static and malicious adversaries in the $(\mathcal{F}_{\mathsf{ZK}}, \mathcal{F}_{\mathsf{ZK}}^{BuildBlock})$-hybrid model.*

The proof of this theorem can be found in Appendix B.4.

4.4 Most significant non-zero-bit

Given a positive input $x \in (0, \frac{p-1}{2}]$, the most significant non-zero-bit (Msnzb) outputs y such that if $x_y = 1$ then $x_i = 0$ for all $i > y$, namely $2^y \le x \le 2^{y+1} - 1$. This operation is necessary to normalize the inputs of our ZK-based mathematical functions detailed in Section 5, such as division and reciprocal square root. Our initial idea for Msnzb is to directly use the inequality $2^y \le x \le 2^{y+1} - 1$ to check the correctness of y. Specifically, we ask the prover to provide additional values z_0 and z_1, and then the protocol verifies that (1) $z_0 = 2^y$ and $z_1 = 2^{y+1} - 1$, and (2) $x \in [z_0, z_1]$. The former can be achieved via table lookup with a table L containing $(y, 2^y, 2^{y+1} - 1)$ for $y \in (0, \lceil \log p \rceil - 2]$[4]. The latter can be regarded as checking whether $x - z_0 < \frac{p+1}{2}$ and $z_1 - x < \frac{p+1}{2}$ by invoking our comparison verification protocol.

The above solution seems correct since it verifies all the conditions that y should satisfy. However, we found that the construction of table L in step (1) is unreasonable based on the following observation. Given $\ell = \lceil \log p \rceil$, let's pay attention to the last entry $(\ell - 2, 2^{\ell-2}, 2^{\ell-1} - 1)$ in the table L. We observe that $2^{\ell-1} - 1$ may exceed the range of positive values in $\mathbb{F}_p$, because $2^{\ell-1} - 1 \ge \frac{p-1}{2}$. This can be demonstrated by a contradiction, that is, assuming $2^{\ell-1} - 1 < \frac{p-1}{2}$ we have $p > 2^\ell - 1$, which contradicts the definition of an

[4] Since the input x is positive, the $(\lceil \log p \rceil - 1)$-th bit of x is always 0.

Protocol Π_{Trunc}

Parameters: A finite field $\mathbb{F}_p$.

Input: $\mathcal{P}$ and $\mathcal{V}$ have an authenticated value $[x]_p$ and a truncation bitlength t, where $x \in \mathbb{F}_p$.

Protocol execution: $\mathcal{P}$ and $\mathcal{V}$ compute $[y]_p$ such that $y = \mathsf{R2F}(\mathsf{F2R}(x,p)/2^t, p)$ as follows:

1. $\mathcal{P}$ and $\mathcal{V}$ send (Cmp, $[x]_p, \frac{p+1}{2}$) to functionality $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$, which returns $[b]_p$ such that $b = 1\{x < \frac{p+1}{2}\}$.
2. $\mathcal{P}$ and $\mathcal{V}$ compute $[\bar{x}]_p = (2 \cdot [b]_p - 1) \cdot [x]_p - (1 - [b]_p)$ by calling functionality $\mathcal{F}_{\mathsf{ZK}}$, and send (PosTrunc, $[\bar{x}]_p, t$) to functionality $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$, which returns $[\bar{y}]_p$.
3. $\mathcal{P}$ and $\mathcal{V}$ compute $[y]_p = (2 \cdot [b]_p - 1) \cdot [\bar{y}]_p - (1 - [b]_p)$ by calling functionality $\mathcal{F}_{\mathsf{ZK}}$.
4. $\mathcal{P}$ and $\mathcal{V}$ output $[y]_p$.

Figure 8: Protocol for general truncation.

Protocol Π_{Msnzb}

Parameters: A finite field $\mathbb{F}_p$.

Input: $\mathcal{P}$ and $\mathcal{V}$ have an authenticated value $[x]_p$, where $x \in (0, \frac{p-1}{2}]$.

Protocol execution: $\mathcal{P}$ and $\mathcal{V}$ compute $[y]_p$ such that $2^y \le x \le 2^{y+1} - 1$ as follows:

1. $\mathcal{P}$ computes y such that $2^y \le x \le 2^{y+1} - 1$ and $z_0 = 2^y$, and set $z_1 = \frac{p-1}{2}$ if $y = \lceil \log p \rceil - 2$, otherwise $z_1 = 2^{y+1} - 1$. $\mathcal{P}$ sends $(\mathsf{Input}, y, z_0, z_1)$ to functionality $\mathcal{F}_{\mathsf{ZK}}$, which returns $([y]_p, [z_0]_p, [z_1]_p)$ to $\mathcal{P}$ and $\mathcal{V}$.
2. $\mathcal{P}$ and $\mathcal{V}$ send $(\mathsf{Lookup}, L, [y]_p, [z_0]_p, [z_1]_p)$ to functionality $\mathcal{F}_{\mathsf{ZK}}^{Lookup}$, where L consists of $(y, 2^y, 2^{y+1} - 1)$ for $y \in [0, \lceil \log p \rceil - 3]$ and $(y, 2^y, \frac{p-1}{2})$ for $y = \lceil \log p \rceil - 2$.
3. $\mathcal{P}$ and $\mathcal{V}$ send (VrfyCmp, $[x]_p - [z_0]_p, \frac{p+1}{2}$) and (VrfyCmp, $[z_1]_p - [x]_p, \frac{p+1}{2}$) to functionality $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$.
4. If any of the above checks fails, $\mathcal{V}$ aborts. Otherwise, $\mathcal{P}$ and $\mathcal{V}$ output $[y]_p$.

Figure 9: Protocol for most significant non-zero-bit.

ℓ-bit prime p. In this case, when we need to check $z_1 - x$ given $z_1 = 2^{\ell-1} - 1$ in step (2), the result of $z_1 - x$ is either negative or incorrectly overflows to a positive value. To address this issue, we carefully set the last entry of the table L to $(\ell - 2, 2^{\ell-2}, \frac{p-1}{2})$, because x is restricted to be positive and cannot exceed $\frac{p-1}{2}$. The soundness is guaranteed, because all z_0 and z_1 are positive. Overall, our complete Msnzb protocol is detailed in Figure 9. This protocol mainly consists of comparison verification and table lookup, which totally costs $4k + 6$ multiplication gates.

Theorem 5. *Protocol Π_{Msnzb} UC-realizes the* Msnzb *command of functionality $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$ against static and malicious adversaries in the $(\mathcal{F}_{\mathsf{ZK}}, \mathcal{F}_{\mathsf{ZK}}^{Lookup})$-hybrid model.*

The proof of this theorem can be found in Appendix B.5.

5 Mathematical Functions

In this section, we detail efficient ZK protocols for complex mathematical functions based on the above building blocks, including exponential, division, and reciprocal square root. The ideal functionalities are summarized in Figure 10.

5.1 Exponential

The exponential operation $y = (\frac{1}{e})^x$ is widely used in various ML functions, such as softmax and GELU. A straightforward approach to evaluate this function is to directly invoke table lookup once, where the table includes all possible inputs and their exponential results. However, such a table would be exceedingly large as discussed in Section 2.1, resulting in poor performance in the check phase. To tackle this problem, inspired by [47], our main idea is to first decompose x into several smaller digits $x_0, \ldots, x_{k-1}$ using our digital decomposition protocol, such that $x = x_{k-1} \| \ldots \| x_0$, and then perform exponential on each of these digits using table lookup. Finally, the outputs are multiplicatively combined to recover the real result. The detailed protocol is provided in Figure 11, where the input x is assumed to be non-negative.

Note that we can reduce the number of truncations in step 4 of Figure 11. The observation is that each $y_i \in [0, 2^s]$ is small due to the range of exponential, and hence we can perform truncation after multiple multiplications only when the result is about to exceed p.

5.2 Division

The division operation $y = \frac{1}{x}$ is typically used in softmax and sigmoid in ML. There are mainly two categories of algorithms for this operation [7, 47, 57]: general Boolean circuits and functional iterations. The state-of-the-art ZK proof [57] employs the former method, but its performance is undesirable as described in Section 1. In our framework, we explore the functional iterations, more specifically Goldschmidt's algorithm [26], following prior secure multi-party computation works [7, 47, 54]. Note that this algorithm has not been pre-

Functionality $\mathcal{F}_{\mathsf{ZK}}^{Math}$

This functionality extends the instructions in $\mathcal{F}_{\mathsf{ZK}}$.

Exponential: On input (Exp, $[x]_p$) from $\mathcal{P}$ and $\mathcal{V}$, where $x \in [0, \frac{p-1}{2}]$, check that $[x]_p$ is valid and abort if not. Compute $y = \mathsf{PtExp}(x)$, store y, and send $[y]_p$ to $\mathcal{P}$ and $\mathcal{V}$.

Division: On input (Div, $[x]_p$) from $\mathcal{P}$ and $\mathcal{V}$, where $x \in (0, \frac{p-1}{2}]$, check that $[x]_p$ is valid and abort if not. Compute $y = \mathsf{PtDiv}(x)$, store y, and send $[y]_p$ to $\mathcal{P}$ and $\mathcal{V}$.

Reciprocal square root: On input (RSqrt, $[x]_p$) from $\mathcal{P}$ and $\mathcal{V}$, $x \in (0, \frac{p-1}{2}]$, check that $[x]_p$ is valid and abort if not. Compute $y = \mathsf{PtRSqrt}(x)$, store y, and send $[y]_p$ to $\mathcal{P}$ and $\mathcal{V}$.

Figure 10: Ideal functionality for our mathematical functions. The three plaintext procedures, i.e., PtExp, PtDiv, and PtRSqrt, are defined in the full version of our work.

viously adopted in ZK proofs, because its operations and verification are complex in general arithmetic or Boolean circuits. Fortunately, with the help of our building blocks and table lookup techniques, we can naturally construct the ZK proofs of division based on Goldschmidt's algorithm.

Our division protocol is detailed in Figure 12, which consists of four steps. (1) Normalize the input. This algorithm needs to iterate on a good initial approximation, which requires the input $x \in [1,2)$. Thus, we normalize x to z that satisfies this constraint, by invoking our Msnzb and CheckLookup protocols. (2) Compute the initial approximation. The initial approximation y' can be expressed as $y' = a - b \cdot z_0$ [32], where $z = z_1 \| z_0$ and (a,b) are coefficients determined by z_1. We utilize the table lookup technique where a table L is constructed containing all possible pairs (a,b) corresponding to z_1. (3) Perform Goldschmidt's iteration. The initial approximation is further tuned to optimize the result's accuracy [26]. This step can be iteratively achieved by invoking multiplication and the proposed truncation protocol. (4) Normalize the output. The output's range is adjusted to compensate for the normalization of the input in step (1).

5.3 Reciprocal square root

The reciprocal square root $y = \frac{1}{\sqrt{x}}$ with $x > 0$ is used in the normalization layer of ML models. Same as the division protocol, we still choose Goldschmidt's algorithm iterating on a precise initial approximation [47] to evaluate this function. Due to space limitations, we provide this protocol in the full version.

Protocol Π_{Exp}

Parameters: A finite field $\mathbb{F}_p$, a constant k, and scale s.

Input: $\mathcal{P}$ and $\mathcal{V}$ have an authenticated value $[x]_p$, where $x \in [0, \frac{p-1}{2}]$.

Protocol execution: $\mathcal{P}$ and $\mathcal{V}$ compute $[y]_p$ such that $y = \mathsf{PtExp}(x)$ as follows:

1. $\mathcal{P}$ and $\mathcal{V}$ send (DigitDec, $[x]_p, d_0, \ldots, d_{k-1}$) to functionality $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$, which returns $[x_0]_p, \ldots, [x_{k-1}]_p$ such that $x = x_{k-1} \| \ldots \| x_0$ and $x_i \in \{0,1\}^{d_i}$ for $i \in [0, k-1]$.
2. $\mathcal{P}$ computes $y_i = \mathsf{R2F}((\frac{1}{e})^{2^{\sum_{j \in [0,i-1]} d_j} \cdot \hat{x}_i}, p, s)$ for $i \in [1, k-1]$ and $y_i = \mathsf{R2F}((\frac{1}{e})^{\hat{x}_i}, p, s)$ for $i = 0$, where $\hat{x}_i = \mathsf{F2R}(x_i, p, s)$, and sends (Input, y_i) to functionality $\mathcal{F}_{\mathsf{ZK}}$, which returns $[y_i]_p$ to $\mathcal{P}$ and $\mathcal{V}$.
3. For $i \in [0, k-1]$, $\mathcal{P}$ and $\mathcal{V}$ send (Lookup, L_i, $[x_i]_p, [y_i]_p$) to functionality $\mathcal{F}_{\mathsf{ZK}}^{Lookup}$, where $L_i = \{x_i, \mathsf{R2F}((\frac{1}{e})^{2^{\sum_{j \in [0,i-1]} d_j} \cdot \hat{x}_i}, p, s)\}_{x_i \in \{0,1\}^{d_i}}$ for $i \in [1, k-1]$ and $L_i = \{x_i, \mathsf{R2F}((\frac{1}{e})^{\hat{x}_i}, p, s)\}_{x_i \in \{0,1\}^{d_i}}$ for $i = 0$, with $\hat{x}_i = \mathsf{F2R}(x_i, p, s)$.
4. $\mathcal{P}$ and $\mathcal{V}$ set $[z_0]_p = [y_0]_p$. For $i \in [1, k-1]$, $\mathcal{P}$ and $\mathcal{V}$ compute $[z_i]_p = [z_{i-1}]_p \cdot [y_i]_p$ by calling functionality $\mathcal{F}_{\mathsf{ZK}}$ and send (PosTrunc, $[z_i]_p, s$) to functionality $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$, which returns $[z_i]_p$ to $\mathcal{P}$ and $\mathcal{V}$.
5. If the above check fails, $\mathcal{V}$ aborts; otherwise, $\mathcal{P}$ and $\mathcal{V}$ output $[y]_p = [z_{k-1}]_p$.

Figure 11: Protocol for exponential.

6 Machine Learning Applications

To explore the applicability of the proposed ZK protocols, in this section, we apply them to mainstream non-linear functions of ML models, including ReLU, maxpooling, sigmoid, softmax, GELU, and normalization. Due to space limitations, the detailed protocols are provided in the full version.

ReLU. ReLU is a widely used non-linear activation function, especially in CNNs. Given an input x, ReLU computes

$$y = \mathsf{Max}(x, 0) = x \cdot 1\{x \geq 0\}. \tag{3}$$

Hence, the ZK proofs of this function can be implemented by invoking our comparison protocol.

Maxpooling. Maxpooling is an essential operation in CNNs to reduce the spatial dimensions of feature maps and select the most relevant features. Given inputs $(x_0, \ldots, x_{n-1})$, maxpooling computes

$$y = \mathsf{Max}(x_0, \ldots, x_{n-1}). \tag{4}$$

In this protocol, the prover is required to provide the result y, and then the protocol verifies that (1) $y - x_i \geq 0$

Protocol Π_{Div}

Parameters: A finite field $\mathbb{F}_p$, upper bound of input bitlength n, scale s, number of iterations I, and bitlength m for lookup.

Input: $\mathcal{P}$ and $\mathcal{V}$ have an authenticated value $[x]_p$, where $x \in (0, 2^n - 1]$.

Protocol execution: $\mathcal{P}$ and $\mathcal{V}$ compute $[y]_p$ such that $y = \mathsf{PtDiv}(x)$ as follows:

Step 1. Normalize the input:

1. $\mathcal{P}$ and $\mathcal{V}$ send $(\text{Msnzb}, [x]_p)$ to functionality $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$, which returns $[k]_p$ such that $2^k \leq x \leq 2^{k+1} - 1$.
2. $\mathcal{P}$ computes $d = 2^{n-1-k}$ and sends (Input, d) to functionality $\mathcal{F}_{\mathsf{ZK}}$, which returns $[d]_p$ to $\mathcal{P}$ and $\mathcal{V}$.
3. $\mathcal{P}$ and $\mathcal{V}$ send $(\mathsf{Lookup}, L, [k]_p, [d]_p)$ to functionality $\mathcal{F}_{\mathsf{ZK}}^{Lookup}$, where $L = \{k, 2^{n-1-k}\}_{k \in [0,n-1]}$.
4. $\mathcal{P}$ and $\mathcal{V}$ compute $[z]_p = [x]_p \cdot [d]_p$ by calling functionality $\mathcal{F}_{\mathsf{ZK}}$.

Step 2. Compute the initial approximation:

1. $\mathcal{P}$ and $\mathcal{V}$ send $(\text{DigitDec}, [z]_p, n-1-m, m+1)$ to functionality $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$, which returns $[z_0]_p, [z_1]_p$ such that $z = z_1 \| z_0$, $z_0 \in \{0,1\}^{n-1-m}$, and $z_1 \in \{0,1\}^{m+1}$.
2. $\mathcal{P}$ computes $a = \mathsf{R2F}(\frac{2^{-m-1} + \sqrt{\hat{z}_1 \cdot (\hat{z}_1 + 2^{-m})}}{\hat{z}_1 \cdot (\hat{z}_1 + 2^{-m})}, p, s+n-1) \in \{0,1\}^{s+n-1}$ and $b = \mathsf{R2F}(\frac{1}{\hat{z}_1 \cdot (\hat{z}_1 + 2^{-m})}, p, s) \in \{0,1\}^s$, where $\hat{z}_1 = \mathsf{F2R}(z_1, p, m)$, and sends (Input, a, b) to functionality $\mathcal{F}_{\mathsf{ZK}}$, which returns $[a]_p, [b]_p$ to $\mathcal{P}$ and $\mathcal{V}$.
3. $\mathcal{P}$ and $\mathcal{V}$ send $(\mathsf{Lookup}, L, [z_1]_p, [a]_p, [b]_p)$ to functionality $\mathcal{F}_{\mathsf{ZK}}^{Lookup}$, where $L = \{z_1, \mathsf{R2F}(\frac{2^{-m-1} + \sqrt{\hat{z}_1 \cdot (\hat{z}_1 + 2^{-m})}}{\hat{z}_1 \cdot (\hat{z}_1 + 2^{-m})}, p, s+n-1), \mathsf{R2F}(\frac{1}{\hat{z}_1 \cdot (\hat{z}_1 + 2^{-m})}, p, s)\}_{z_1 \in \{0,1\}^{m+1}}$ with $\hat{z}_1 = \mathsf{F2R}(z_1, p, m)$.
4. $\mathcal{P}$ and $\mathcal{V}$ compute $[t']_p = [a]_p - [b]_p \cdot [z_0]_p$ by calling functionality $\mathcal{F}_{\mathsf{ZK}}$ and send $(\text{PosTrunc}, [t']_p, n-1)$ to functionality $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$, which returns $[t]_p$ to $\mathcal{P}$ and $\mathcal{V}$.

Step 3. Perform Goldschmidt's iteration:

1. $\mathcal{P}$ and $\mathcal{V}$ compute $[a_0']_p = 2^{n-1+s} - [z]_p \cdot [t]_p$, and send $(\text{PosTrunc}, [a_0']_p, n-1)$ to functionality $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$, which returns $[a_0]_p$ to $\mathcal{P}$ and $\mathcal{V}$. $\mathcal{P}$ and $\mathcal{V}$ set $[b_0]_p = 2^s + [a_0]_p$ and $[c_0]_p = [b_0]_p$.
2. For $i \in [1, I]$, $\mathcal{P}$ and $\mathcal{V}$ (1) compute $[a_i']_p = [a_{i-1}]_p \cdot [a_{i-1}]_p$ by calling functionality $\mathcal{F}_{\mathsf{ZK}}$ and send $(\text{PosTrunc}, [a_i']_p, s)$ to functionality $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$, which returns $[a_i]_p$, (2) compute $[b_i]_p = 2^s - [a_i]_p$, (3) compute $[c_i']_p = [c_{i-1}]_p \cdot [b_i]_p$ by calling functionality $\mathcal{F}_{\mathsf{ZK}}$ and send $(\text{PosTrunc}, [c_i']_p, s)$ to functionality $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$, which returns $[c_i]_p$.

Step 4. Normalize the output:

1. $\mathcal{P}$ computes $e = 2^{n-k}$ and sends (Input, e) to functionality $\mathcal{F}_{\mathsf{ZK}}$, which returns $[e]_p$ to $\mathcal{P}$ and $\mathcal{V}$.
2. $\mathcal{P}$ and $\mathcal{V}$ send $(\mathsf{Lookup}, L, [k]_p, [e]_p)$ to functionality $\mathcal{F}_{\mathsf{ZK}}^{Lookup}$, where $L = \{k, 2^{n-k}\}_{k \in [0,n-1]}$.
3. $\mathcal{P}$ and $\mathcal{V}$ compute $[y']_p = [c_I]_p \cdot [e]_p$ by calling functionality $\mathcal{F}_{\mathsf{ZK}}$, and send $(\text{PosTrunc}, [y']_p, n-s)$ to functionality $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$, which returns $[y]_p$ to $\mathcal{P}$ and $\mathcal{V}$.
4. If the above check fails, $\mathcal{V}$ aborts; otherwise, $\mathcal{P}$ and $\mathcal{V}$ output $[y]_p$.

Figure 12: Protocol for division.

for $i \in [0, n-1]$, to ensure that y is the maximum value of $(x_0, \ldots, x_{n-1})$, and (2) $\prod_{i=0}^{n-1}(y - x_i) = 0$, to ensure that $y \in \{x_0, \ldots, x_{n-1}\}$. The former can be achieved using our comparison verification protocol while the latter is implemented through the $\mathsf{CheckZero}$ procedure.

Sigmoid. Sigmoid is a commonly used activation function in ML models, which maps any input value to a range between 0 and 1. Given an input x, sigmoid computes

$$y = \frac{1}{1 + e^{-x}}. \tag{5}$$

It can be written as $y = \frac{1}{1+e^{-x}}$ if $x \geq 0$ and $y = e^{-|x|} \cdot \frac{1}{1+e^{-|x|}}$ if $x < 0$ [47]. Hence, this ZK proof can be built by invoking our comparison, exponential, and division protocols.

Softmax. Softmax plays a fundamental role in ML models. In CNNs, it is used for generating a probability distribution over different classes. In LLMs, it is used for computing language attention scores and text generation. Given inputs $(x_0, \ldots, x_{n-1})$, softmax computes $(y_0, \ldots, y_{n-1})$ such that for $i \in [0, n-1]$, it holds

$$y_i = \frac{e^{x_i - x_{\mathsf{max}}}}{\sum_{j \in [0,n-1]} e^{x_j - x_{\mathsf{max}}}}, \tag{6}$$

where $x_{\mathsf{max}} = \mathsf{Max}(x_0, \ldots, x_{n-1})$. Note that we can utilize the known input range to reduce the overhead of division. The real input $\sum_{i \in [0,n-1]} e^{x_j - x_{\mathsf{max}}}$ to division is bounded by n, and hence the maximum input bitlength is $s + \lceil \log n \rceil$. For example, when $n = 256$ and $s = 12$, it only requires performing division on 20 bits.

GELU. GELU activation is used in LLMs. Given an input x, GELU computes

$$y = 0.5 \cdot x \cdot \left(1 + \mathsf{Tanh}\left[\sqrt{2/\pi} \cdot \left(x + 0.044715 \cdot x^3\right)\right]\right), \quad (7)$$

where $\mathsf{Tanh}(x) = 2 \cdot \mathsf{Sigmoid}(2x) - 1$. Thus, this ZK proof can be implemented by invoking the Sigmoid protocol.

Normalization. Normalization, e.g., batch normalization and layer normalization, plays a crucial role in CNNs and LLMs to stabilize training and improve model generalization. Given inputs $(x_0, \ldots, x_{n-1})$, normalization computes $(y_0, \ldots, y_{n-1})$ such that for $i \in [0, n-1]$, it holds

$$y_i = \gamma \cdot \frac{x_i - \mu}{\sqrt{\sigma}} + \beta, \quad (8)$$

where (γ, β) are trained affine transform parameters, $\mu = \frac{\sum_{i \in [0,n-1]} x_i}{n}$ and $\sigma = \frac{\sum_{i \in [0,n-1]} (x_i - \mu)^2}{n}$. We observe that this function can be evaluated by invoking our reciprocal square root protocol.

Other applications beyond ML. Our ZK proofs for mathematical functions are general and essentially can be used in any application involving the evaluation of non-linear functions. Specifically, not only well-known ZK proofs for ML inference but also some other applications can benefit from our constructions, such as software vulnerabilities [8], program analysis [15], and database querying [39]. For example, the ZK proof of conditional statements, e.g., if else, in the programming language [15] could be realized utilizing our comparison protocol. Also, database querying contains a series of set operations like sort, disjoint, and aggregation. The ZK-based set sort and disjoint relies on the set equality check and the generic comparison circuits [39], which can be instantiated by our comparison protocol. The ZK proof of aggregation can be easily implemented by invoking our division construction.

7 Evaluation

7.1 Experiment setup

We implemented our framework built on top of the EMP toolkit [55] in C++. Same as Mystique [57], we simulate the network connection with different bandwidths, including 200Mbps, 500Mbps, and 1Gbps. Unless otherwise specified, the bandwidth is set to 500Mbps. All experiments are performed using a single thread on AWS c5.9xlarge instances with Intel Xeon 8000 series CPUs at 3.6GHz.

Implementation details. Following prior works [56, 57, 59], our implementations set the computational security parameter $\kappa = 128$ and the statistical security parameter $\lambda \geq 40$ over a 61-bit field where $p = 2^{61} - 1$ is a Mersenne prime. The default number of instances is 10^5 and the scale is 12 in our evaluation. When constructing a lookup table, we decompose the original input into multiple smaller digits, each with 12 bits except for the most significant digit.

Baselines. Our baseline is Mystique [57], the state-of-the-art ZK proofs for ML. Mystique provides comprehensive ZK protocols of non-linear functions in ML. The implementation of its protocols is provided in the EMP toolkit [55]. For a fair comparison, we re-run these protocols under the same network environment and experimental setup as our framework.

Table 1: Runtime (μs) and communication (KB) overhead of our building blocks in the amortized setting.

Protocol	**Runtime (μs) on different bandwidths**			**Comm. (KB)**
	200 Mbps	500 Mbps	1 Gbps	
DigitDec	10.320	9.058	8.946	0.159
VrfyCMP	15.862	14.314	14.358	0.230
CMP	20.662	18.918	18.569	0.301
PosTrunc	10.352	8.990	8.951	0.159
Trunc	32.488	28.899	28.814	0.475
Msnzb	34.806	30.360	30.224	0.508

7.2 Performance evaluation

We evaluate the performance of our framework from the three hierarchical levels as shown in Figure 1.

Results of building blocks. We test the performance of our key building blocks, and report the runtime and communication overhead under varying network bandwidths in the amortized setting in Table 1. We can observe that all of our building blocks are highly efficient. For example, when the bandwidth is 200Mps, our building blocks only take around $10 \sim 34$ μs. Moreover, the communication performance is also satisfactory. As the bandwidth reduces from 1Gbps to 200Mbps, the runtime only slightly increases, due to the high communication efficiency of our protocols. In Table 2, we explore the impact of different numbers of instances on runtime and communication performance. Although our protocols are better suited to the amortized setting, we observe that they are still highly efficient even with a small number of evaluations. For example, evaluating 10^3 instances requires only $8 \sim 33$ ms. Further, we study the impact of different scales in Table 3. We can observe that the scale only affects the performance of truncation operations, i.e., PosTrunc and Trunc, since it only represents the truncation bitlength. In addition, there is a notable increase in runtime from the scale of 12 to 14. The reason is that each lookup table in our building blocks contains 2^{12} items associated with a 12-bit digit, as detailed in Section 7.1. Thus, when the scale is 14 or 16 in our evaluation, two lookup tables are required to complete the protocols.

Results of mathematical functions. In Table 4, we report

Table 2: Runtime (sec) and communication (MB) overhead of our building blocks with the different number of instances (10^3, 10^4, 10^5).

Protocol	10^3		10^4		10^5	
	Time	**Comm.**	**Time**	**Comm.**	**Time**	**Comm.**
DigitDec	0.008	0.215	0.113	1.588	0.906	15.571
VrfyCMP	0.023	0.599	0.164	2.591	1.431	22.504
CMP	0.008	0.217	0.109	2.140	1.892	29.374
PosTrunc	0.008	0.215	0.115	1.588	0.899	15.571
Trunc	0.017	0.448	0.204	3.881	2.890	46.471
Msnzb	0.033	0.868	0.327	5.262	3.036	49.586

Table 3: Runtime (sec) and communication (MB) overhead of our building blocks with different scales (12, 14, 16).

Protocol	12		14		16	
	Time	**Comm.**	**Time**	**Comm.**	**Time**	**Comm.**
DigitDec	0.906	15.571	0.906	15.571	0.906	15.571
VrfyCMP	1.431	22.504	1.431	22.504	1.431	22.504
CMP	1.892	29.374	1.892	29.374	1.892	29.374
PosTrunc	0.899	15.571	1.157	18.576	1.158	18.565
Trunc	2.890	46.471	3.118	49.476	3.130	49.465
Msnzb	3.036	49.586	3.036	49.586	3.036	49.586

the performance of our mathematical functions and give the comparison with Mystique [57]. In our protocols, we set the parameters following the prior work [47], i.e., the number of iterations $I = 0$ for division and $I = 1$ for reciprocal square root, and the lookup bitlength $m = 5$ for division and $m = 6$ for reciprocal square root. We can observe that our protocols achieve significant performance gains from $61 \sim 130\times$ on the runtime. Furthermore, our communication cost is at a similar level as Mystique, and precisely, it is $1.4 \sim 2.9\times$ better.

Results of ML applications. We apply the proposed protocols to widely used non-linear functions in ML models, and the performance is reported in Table 5. We can observe that the runtime of our non-linear function evaluation is efficient. For example, for the ReLU activation in CNNs, we can complete 10^5 evaluations in 2 seconds with the bandwidth 1Gbps, outperforming Mystique by about $100\times$. For the GELU activation in LLMs, we achieve $77 \sim 86\times$ improvement. What's more, for the softmax function, we even obtain a $179\times$ gain. The main reason is that as discussed in Section 1, in Mystique, the ZK proofs of these functions require multiple arithmetic-Boolean conversions, as well as heavy Boolean circuit evaluation. In addition, we also achieve $1.2 \sim 4.8\times$ better communication performance.

Table 4: Comparison with the state-of-the-art Mystique [57] on runtime (sec) and communication (MB) overhead of mathematical functions.

Protocol	**Runtime (sec) on different bandwidths**			**Comm. (MB)**
	200 Mbps	500 Mbps	1 Gbps	
Exponential				
Ours	9.877	8.696	8.652	99.020
Mystique	1184.240 (119.901×)	1130.020 (129.948×)	1118.570 (129.280×)	291.435 (2.943×)
Division				
Ours	10.378	9.837	9.798	110.684
Mystique	636.038 (61.287×)	617.690 (62.792×)	619.162 (63.193×)	160.428 (1.449×)
Reciprocal square root				
Ours	13.406	11.836	11.804	147.903
Mystique	836.267 (62.379×)	824.639 (69.674×)	823.949 (69.803×)	212.211 (1.435×)

8 Related Works

There has been advanced progress in the efficiency and scalability of ZK protocols derived from various techniques, such as zk-SNARKs [44, 50], ZK protocols based on vector oblivious linear evaluation (VOLE) [3, 12, 13, 56, 59], privacy-free garbled circuits [19, 29, 33], and the MPC-in-the-head paradigm [31]. Benefiting from this, several recent works have explored devising verifiable ML services based on various ZK proofs. Most of these studies focus on verifying the integrity of inference [17, 25, 36, 38, 40, 57, 63] on various ML models, including shallow fully-connected neural networks, decision trees, and CNNs. Some recent works have also attempted the possibility of applying ZK proofs for relatively simple ML training tasks, such as training of logistic regression model [23], fairness of training decision trees [52]. Different from our solution, the above works do not specifically focus on complex non-linear functions. However, as shown in Section 1, ZK proofs for non-linear mathematical functions are the main bottleneck especially when applied in reasonably complicated neural network models [57].

We further discuss related works on ZK proofs of non-linear functions. The most relevant work is Mystique [57], which is the state-of-the-art ZK proof for ML inference. Mystique innovatively designed zk-edaBits for converting between arithmetic and Boolean domains within ZK proofs, and then enabled the evaluation of non-linear functions using general Boolean circuits. Moreover, Mystique provides implementations of comprehensive mathematical functions, serving as the baseline for comparison with our work. However, as illustrated in Section 1, the conversion and evaluation with Boolean circuits are expensive and require heavy invocation of multiplication gates. Our evaluation demonstrates improvements of up to two orders of magnitude over Mystique. Moreover, zkCNN [40] also presented efficient ZK proofs

Table 5: Comparison with the state-of-the-art Mystique [57] on runtime (sec) and communication (MB) overhead of widely used non-linear functions in ML models.

Protocol	Runtime (sec) on different bandwidths			Comm. (MB)
	200 Mbps	500 Mbps	1 Gbps	
ReLU				
Ours	2.107	1.906	1.898	30.137
Mystique	200.433 (95.113×)	193.797 (101.655×)	192.360 (101.336×)	58.244 (1.933×)
Sigmoid				
Ours	19.544	17.706	17.715	189.899
Mystique	1918.970 (98.188×)	1847.300 (104.332×)	1830.750 (103.344×)	463.862 (2.443×)
GELU				
Ours	37.628	32.696	32.528	338.182
Mystique	2719.110 (72.264×)	2711.700 (82.936×)	2627.300 (80.769×)	654.685 (1.936×)
Maxpooling, dim = 4				
Ours	10.439	9.310	9.136	95.611
Mystique	804.715 (77.084×)	774.942 (83.240×)	776.658 (85.011×)	184.554 (1.930×)
Softmax, dim = 10				
Ours	87.131	78.289	78.015	816.330
Mystique	14973.300 (171.849×)	14049.600 (179.457×)	13998.800 (179.436×)	3972.490 (4.866×)
Normalization, dim = 16				
Ours	192.826	176.140	175.439	1787.067
Mystique	9667.060 (50.134×)	9642.600 (54.744×)	9279.420 (52.893×)	2219.737 (1.242×)

for CNNs and designed new protocols for maxpooling and ReLU non-linear functions. Their protocols are specialized and may not be readily extended to more complex functions addressed in our paper. Besides, concurrent with Mysitque, Baum et al. [2] also presented zk-edaBits and designed customized protocols for truncation and comparison. However, their protocols assume a gap between the field size and the size of real inputs.

In addition, there is also a line of secure inference work [1, 30, 41, 42, 48] based on secure multi-party computation (called PPML for short). We clarify that ZK-based ML (ZKML) and PPML have significant differences as below. Specifically, (1) Objective: While both ZKML and PPML aim to secure ML inferences without revealing model parameters, ZKML particularly enables the server to prove to clients that the inference is correctly evaluated by the claimed model, which is beyond the scope of PPML. (2) Threat model: ZKML protocols are always secure against malicious adversaries. However, most PPML protocols [30, 41, 48] in the client-server setting only achieve security against semi-honest adversaries. (3) Application: ZKML is typically applied to verifiable scenarios to guarantee integrity. PPML is typically used in privacy-preserving scenarios to guarantee privacy.

9 Conclusion

In this paper, we effectively overcome the runtime bottleneck of non-linear function evaluation in current research, by presenting a scalable ZK proof framework. Based on the new perspective from table lookup and novel protocol designs, our framework achieves up to two orders of magnitude of runtime improvement compared to the state-of-the-art work, while maintaining a similar level of communication efficiency. All of our protocols are performed in the arithmetic field, allowing for seamless integration with ZK-based linear layers in ML to accomplish the whole inference task.

We discuss the potential limitations of our protocols and future works. The main limitation is that our protocols use the fixed-point-based mathematical function evaluation and may result in a slight loss of accuracy compared to the float-point-based protocols [57]. Although this causes a negligible impact on ML tasks [30, 48], it may have limitations for other accuracy-sensitive scenarios. Therefore, it is an interesting future work to address this issue, while maintaining concrete efficiency. In addition, our protocols are general and essentially can be used in any application involving the evaluation of non-linear functions as discussed in Section 6. Thus, exploring the generalized applicability of our protocols is also an interesting future work.

Acknowledgments

We would like to express our deepest gratitude for the invaluable help provided by our shepherd as well as all the reviewers for their constructive comments. This work is supported by the National Key R&D Program of China under Grant 2022YFB3103500, the National Natural Science Foundation of China under Grant 62020106013, the Sichuan Science and Technology Program under Grant 2023ZYD0142, the Fundamental Research Funds for Chinese Central Universities under Grants ZYGX2020ZB027 and Y030232063003002, the National Research Foundation, Singapore, the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN), the Singapore Ministry of Education (MOE) AcRF Tier 2 MOE-T2EP20121-0006, and the Nanyang Technological University (NTU)-DESAY SV Research Program under Grant 2018-0980.

References

[1] Nitin Agrawal, Ali Shahin Shamsabadi, Matt J Kusner, and Adrià Gascón. Quotient: two-party secure neural network training and prediction. In *Proceedings of ACM CCS*, 2019.

[2] Carsten Baum, Lennart Braun, Alexander Munch-Hansen, Benoit Razet, and Peter Scholl. Appenzeller to brie: efficient zero-knowledge proofs for mixed-mode

arithmetic and z2k. In *Proceedings of ACM CCS*, pages 192–211, 2021.

[3] Carsten Baum, Alex J Malozemoff, Marc B Rosen, and Peter Scholl. Mac'n'cheese: Zero-knowledge proofs for boolean and arithmetic circuits with nested disjunctions. In *Proceedings of CRYPTO*, pages 92–122, 2021.

[4] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In *Proceedings of EUROCRYPT*, 2011.

[5] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. Efficient two-round ot extension and silent non-interactive secure computation. In *Proceedings of ACM CCS*, pages 291–308, 2019.

[6] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings of FOCS*, pages 136–145, 2001.

[7] Octavian Catrina and Amitabh Saxena. Secure computation with fixed-point numbers. In *Proceedings of FC*, pages 35–50, 2010.

[8] Santiago Cuéllar, Bill Harris, James Parker, Stuart Pernsteiner, and Eran Tromer. Cheesecloth: Zero-Knowledge proofs of real world vulnerabilities. In *Proceedings of USENIX Security*, 2023.

[9] Ivan Damgård, Jesper Buus Nielsen, Michael Nielsen, and Samuel Ranellucci. The tinytable protocol for 2-party secure computation, or: Gate-scrambling revisited. In *Proceedings of CRYPTO*, pages 167–187, 2017.

[10] Cyprien Delpech de Saint Guilhem, Emmanuela Orsini, Titouan Tanguy, and Michiel Verbauwhede. Efficient proof of ram programs from any public-coin zero-knowledge system. In *Proceedings of SCN*, pages 615–638, 2022.

[11] Cyprien Delpech de Saint Guilhem, Emmanuela Orsini, Titouan Tanguy, and Michiel Verbauwhede. Efficient proof of ram programs from any public-coin zero-knowledge system. In *Proceedings of SCN*, pages 615–638, 2022.

[12] Samuel Dittmer, Yuval Ishai, Steve Lu, and Rafail Ostrovsky. Improving line-point zero knowledge: Two multiplications for the price of one. In *Proceedings of ACM CCS*, pages 829–841, 2022.

[13] Samuel Dittmer, Yuval Ishai, and Rafail Ostrovsky. Line-point zero knowledge and its applications. In *Proceedings of ITC*, 2021.

[14] Liam Eagen, Dario Fiore, and Ariel Gabizon. cq: Cached quotients for fast lookups. ePrint 2022/1763, 2022.

[15] Zhiyong Fang, David Darais, Joseph P Near, and Yupeng Zhang. Zero knowledge static program analysis. In *Proceedings of ACM CCS*, 2021.

[16] Boyuan Feng, Lianke Qin, Zhenfei Zhang, Yufei Ding, and Shumo Chu. Zen: An optimizing compiler for verifiable, zero-knowledge neural network inferences. Cryptology ePrint Archive 2021/087, 2021.

[17] Boyuan Feng, Lianke Qin, Zhenfei Zhang, Yufei Ding, and Shumo Chu. Zen: An optimizing compiler for verifiable, zero-knowledge neural network inferences. *Cryptology ePrint Archive*, 2021.

[18] Nicholas Franzese, Jonathan Katz, Steve Lu, Rafail Ostrovsky, Xiao Wang, and Chenkai Weng. Constant-overhead zero-knowledge for ram programs. In *Proceedings of ACM CCS*, pages 178–191, 2021.

[19] Tore Kasper Frederiksen, Jesper Buus Nielsen, and Claudio Orlandi. Privacy-free garbled circuits with applications to efficient zero-knowledge. In *Proceedings of Eurocrypt*, pages 191–219, 2015.

[20] Ariel Gabizon and Zachary J. Williamson. plookup: A simplified polynomial protocol for lookup tables. ePrint 2020/315, 2020.

[21] Juan Garay, Berry Schoenmakers, and José Villegas. Practical and secure solutions for integer comparison. In *Proceedings of PKC*, pages 330–342, 2007.

[22] Sanjam Garg, Aarushi Goel, Somesh Jha, Saeed Mahloujifar, Mohammad Mahmoody, Guru-Vamsi Policharla, and Mingyuan Wang. Experimenting with zero-knowledge proofs of training. In *Proceedings of ACM CCS*, 2023.

[23] Sanjam Garg, Aarushi Goel, Somesh Jha, Saeed Mahloujifar, Mohammad Mahmoody, Guru-Vamsi Policharla, and Mingyuan Wang. Experimenting with zero-knowledge proofs of training. In *Proceedings of CCS*, 2023.

[24] Sanjam Garg, Abhishek Jain, Zhengzhong Jin, and Yinuo Zhang. Succinct zero knowledge for floating point computations. In *Proceedings of ACM CCS*, pages 1203–1216, 2022.

[25] Zahra Ghodsi, Tianyu Gu, and Siddharth Garg. Safetynets: Verifiable execution of deep neural networks on an untrusted cloud. In *Proceedings of NeurIPS*, 2017.

[26] Robert E Goldschmidt. *Applications of division by convergence*. PhD thesis, Massachusetts Institute of Technology, 1964.

[27] S Goldwasser, S Micali, and C Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of ACM STOC*, pages 291–304, 1985.

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

[29] David Heath and Vladimir Kolesnikov. Stacked garbling for disjunctive zero-knowledge proofs. In *Proceedings of Eurocrypt*, pages 569–598, 2020.

[30] Zhicong Huang, Wen-jie Lu, Cheng Hong, and Jiansheng Ding. Cheetah: Lean and fast secure two-party deep neural network inference. In *Proceedings of the USENIX Security*, 2022.

[31] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In *Proceedings of ACM STOC*, pages 21–30, 2007.

[32] Masayuki Ito, Naofumi Takagi, and Shuzo Yajima. Efficient initial approximation for multiplicative division and square root by a multiplication with operand modification. *IEEE Transactions on Computers*, 46(4):495–498, 1997.

[33] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In *Proceedings of ACM CCS*, pages 955–966, 2013.

[34] Daniel Kang, Tatsunori Hashimoto, Ion Stoica, and Yi Sun. Scaling up trustless dnn inference with zero-knowledge proofs. *arXiv:2210.08674*, 2022.

[35] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. Large-scale video classification with convolutional neural networks. In *Proceedings of CVPR*, 2014.

[36] Julien Keuffer, Refik Molva, and Hervé Chabanne. Efficient proof composition for verifiable computation. In *Proceedings of ESORICS*, pages 152–171, 2018.

[37] Seunghwa Lee, Hankyung Ko, Jihye Kim, and Hyunok Oh. vcnn: Verifiable convolutional neural network based on zk-snarks. Cryptology ePrint Archive 2020/584, 2020.

[38] Seunghwa Lee, Hankyung Ko, Jihye Kim, and Hyunok Oh. vcnn: Verifiable convolutional neural network based on zk-snarks. *Cryptology ePrint Archive*, 2020.

[39] Xiling Li, Chenkai Weng, Yongxin Xu, Xiao Wang, and Jennie Rogers. Zksql: Verifiable and efficient query evaluation with zero-knowledge proofs. In *Proceedings of VLDB*, 2023.

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

[41] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. Delphi: A cryptographic inference service for neural networks. In *Proceedings of USENIX Security*, 2020.

[42] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *Proceedings of IEEE S&P*, 2017.

[43] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In *Proceedings of CRYPTO*, pages 681–700, 2012.

[44] Alex Ozdemir and Dan Boneh. Experimenting with collaborative {zk-SNARKs}:{Zero-Knowledge} proofs for distributed secrets. In *Proceedings of USENIX Security*, pages 4291–4308, 2022.

[45] Jim Posen and Assimakis A. Kattis. Caulk+: Table-independent lookup arguments. ePrint 2022/957, 2022.

[46] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[47] Deevashwer Rathee, Mayank Rathee, Rahul Kranti Kiran Goli, Divya Gupta, Rahul Sharma, Nishanth Chandran, and Aseem Rastogi. Sirnn: A math library for secure rnn inference. In *Proceedings of IEEE S&P*, 2021.

[48] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. Cryptflow2: Practical 2-party secure inference. In *Proceedings of ACM CCS*, 2020.

[49] Phillipp Schoppmann, Adrià Gascón, Leonie Reichert, and Mariana Raykova. Distributed vector-ole: Improved constructions and implementation. In *Proceedings of ACM CCS*, pages 1055–1072, 2019.

[50] Srinath Setty. Spartan: Efficient and general-purpose zksnarks without trusted setup. In *Proceedings of CRYPTO*, pages 704–737, 2020.

[51] Srinath Setty, Justin Thaler, and Riad Wahby. Unlocking the lookup singularity with lasso. *Cryptology ePrint Archive*, 2023.

[52] Ali Shahin Shamsabadi, Sierra Calanda Wyllie, Nicholas Franzese, Natalie Dullerud, Sébastien Gambs, Nicolas Papernot, Xiao Wang, and Adrian Weller. Confidential-profitt: Confidential proof of fair training of trees. In *Proceedings of ICLR*, 2023.

[53] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of NeurIPS*, 2017.

[54] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. Falcon: Honest-majority maliciously secure framework for private deep learning. *Proceedings on Privacy Enhancing Technologies*, 1:188–208, 2021.

[55] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

[56] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In *Proceedings of IEEE S&P*, 2021.

[57] Chenkai Weng, Kang Yang, Xiang Xie, Jonathan Katz, and Xiao Wang. Mystique: Efficient conversions for zero-knowledge proofs with applications to machine learning. In *Proceedings of USENIX Security*, 2021.

[58] Chenkai Weng, Kang Yang, Zhaomin Yang, Xiang Xie, and Xiao Wang. Antman: Interactive zero-knowledge proofs with sublinear communication. In *Proceedings of ACM CCS*, 2022.

[59] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. Quicksilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In *Proceedings of ACM CCS*, 2021.

[60] Yibin Yang and David Heath. Two shuffles make a ram: Improved constant overhead zero knowledge ram. In *Proceedings of USENIX Security*, 2024.

[61] Arantxa Zapico, Vitalik Buterin, Dmitry Khovratovich, Mary Maller, Anca Nitulescu, and Mark Simkin. Caulk: Lookup arguments in sublinear time. In *Proceedings of ACM CCS*, pages 3121–3134, 2022.

[62] Arantxa Zapico, Ariel Gabizon, Dmitry Khovratovich, Mary Maller, and Carla Ràfols. Baloo: Nearly optimal lookup arguments. ePrint 2022/1565, 2022.

[63] Jiaheng Zhang, Zhiyong Fang, Yupeng Zhang, and Dawn Song. Zero knowledge proofs for decision tree predictions and accuracy. In *Proceedings of ACM CCS*, pages 2039–2053, 2020.

A Security model

We use the universal composability (UC) framework [6] to prove security in the presence of a probabilistic polynomial time (PPT), static, malicious adversary $\mathcal{A}$. We use the standard simulation-based security definition. Generally, a protocol is said to UC realize an ideal functionality if the process of running the protocol amounts to emulating the ideal process for that ideal functionality.

Definition 1. *Let $\mathcal{F}$ be an ideal functionality and let Π be a protocol. We say that Π UC realizes $\mathcal{F}$ if for any adversary $\mathcal{A}$ there exists an ideal-process adversary $\mathcal{S}$ such that for any environment $\mathcal{Z}$,*

$$\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}} \approx_c \mathsf{REAL}_{\Pi,\mathcal{A},\mathcal{Z}}. \tag{9}$$

Here, $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ denotes the ensemble $\{\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\kappa,x)\}_{\kappa,x}$, which is the output of environment $\mathcal{Z}$ after interacting in the ideal process with adversary $\mathcal{S}$ and ideal functionality $\mathcal{F}$ on security parameter κ and input x. $\mathsf{REAL}_{\Pi,\mathcal{A},\mathcal{Z}}$ denotes the ensemble $\{\mathsf{REAL}_{\Pi,\mathcal{A},\mathcal{Z}}(\kappa,x)\}_{\kappa,x}$, which is the output of environment $\mathcal{Z}$ when interacting with adversary $\mathcal{A}$ and parties running protocol Π on security parameter κ and input x. Besides, we prove the security of our protocols in the $\mathcal{G}$-hybrid model, where the parties execute a protocol with real messages and also have access to an ideal functionality $\mathcal{G}$.

B Proof of ZK Building Blocks

We provide the security proof for our ZK proofs of building blocks. Similar to the previous ZK works [18, 60], the verifier in these protocols does not have any input and only receives messages from the functionalities. Therefore, it is straightforward to prove security for a corrupted verifier, and hence we only focus on the case of a corrupted prover below. In this case, we construct a simulator $\mathcal{S}$, which runs an adversary $\mathcal{A}$ as a subroutine. We then show that no environment $\mathcal{Z}$ can distinguish the real-world execution from the ideal-world execution.

B.1 Proof of Theorem 1

Proof. $\mathcal{S}$ interacts with adversary $\mathcal{A}$ as follows:

1. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{ZK}}$ by recording $([x_0]_p, \dots, [x_{k-1}]_p)$, sent to $\mathcal{F}_{\mathsf{ZK}}$ by $\mathcal{A}$.
2. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{ZK}}^{Lookup}$ with $\mathcal{A}$. On receiving $(R_i, [x_i]_p)$ for $i \in [0, k-1]$, if $\mathcal{F}_{\mathsf{ZK}}^{Lookup}$ aborts, then $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$ and aborts.
3. $\mathcal{S}$ locally computes $[z]_p = [x_0]_p + \sum_{i\in[1,k-1]} 2^{\sum_{j\in[0,i-1]} d_j}[x_i]_p - [x]_p$. Note that $[x]_p$ has been recorded by $\mathcal{S}$ in previous interactions with $\mathcal{A}$.

4. $\mathcal{S}$ executes the CheckZero procedure with $\mathcal{A}$. If the received values are not equal to $[z]_p$ in the above step, then $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$ and aborts.
5. $\mathcal{S}$ sends $([x]_p, [x_0]_p, \ldots, [x_{k-1}]_p)$ to $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$.

The view of adversary $\mathcal{A}$ simulated by $\mathcal{S}$ is perfect, except for the CheckZero procedure. In the real protocol execution, if the value opened by $\mathcal{A}$ is not a valid $[z]_p$ in the CheckZero procedure, then the honest verifier would abort except with probability at most $1/p + \mathsf{negl}(\kappa)$, according to the analysis in Section 3.2. In the ideal world, $\mathcal{S}$ outputs would abort once $[z]_p$ is not valid. Therefore, the view of adversary $\mathcal{A}$ that is simulated by $\mathcal{S}$ is computationally indistinguishable from the view of $\mathcal{A}$ in the real protocol execution.

□

B.2 Proof of Theorem 2

Proof. $\mathcal{S}$ interacts with adversary $\mathcal{A}$ as follows:

1. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{ZK}}$ by recording $([x_0]_p, \ldots, [x_{k-1}]_p)$, sent to $\mathcal{F}_{\mathsf{ZK}}$ by $\mathcal{A}$.
2. $\mathcal{S}$ locally computes $[t]_p = [x_0]_p + \sum_{i \in [1,k-1]} 2^{\sum_{j \in [0,i-1]} d_j} [x_i]_p - [x]_p$. Note that $[x]_p$ has been obtained by $\mathcal{S}$ in previous interactions of $\mathcal{A}$ with $\mathcal{F}_{\mathsf{ZK}}$.
3. $\mathcal{S}$ executes the CheckZero procedure with $\mathcal{A}$. If the received values are not equal to $[t]_p$ in the above step, then $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$ and aborts.
4. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{ZK}}$ by recording $([z_0]_p, \ldots, [z_{k-1}]_p)$, sent to $\mathcal{F}_{\mathsf{ZK}}$ by $\mathcal{A}$.
5. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{ZK}}^{Lookup}$ with $\mathcal{A}$. On receiving $(L_i, [x_i]_p, [z_i]_p)$ for $i \in [0, k-1]$, if $\mathcal{F}_{\mathsf{ZK}}^{Lookup}$ aborts, then $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$ and aborts.
6. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{ZK}}$ by recording $[y]_p$, sent to $\mathcal{F}_{\mathsf{ZK}}$ by $\mathcal{A}$.
7. $\mathcal{S}$ locally computes $[z]_p = \sum_{i \in [0,k-1]} [z_i]_p$. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{ZK}}^{Lookup}$ with $\mathcal{A}$. On receiving $(L, [z]_p, [y]_p)$, if $\mathcal{F}_{\mathsf{ZK}}^{Lookup}$ aborts, then $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$ and aborts.
8. $\mathcal{S}$ executes the CheckZero procedure with $\mathcal{A}$. If the received values are not equal to $[y]_p - 1$ in the above step, then $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$ and aborts.

The view of adversary $\mathcal{A}$ simulated by $\mathcal{S}$ is perfect, except for the CheckZero procedure. Same as the analysis in Appendix B.1, the view of adversary $\mathcal{A}$ that is simulated by $\mathcal{S}$ is computationally indistinguishable from the view of $\mathcal{A}$ in the real protocol execution.

□

B.3 Proof of Theorem 3

Proof. $\mathcal{S}$ interacts with adversary $\mathcal{A}$ as follows:

1. $\mathcal{S}$ emulates the DigitDec command of $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$ with $\mathcal{A}$. On receiving $([x]_p, t, m-t)$, $\mathcal{S}$ aborts if $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$ aborts and sends $[x_0]_p, [x_1]_p$ to $\mathcal{A}$ otherwise, where $x = x_1 \| x_0$, $x_0 \in \{0,1\}^t$ and $x_1 \in \{0,1\}^{m-t}$ with $m = \lceil \log p \rceil - 1$. Here, $[x]_p$ is held by $\mathcal{S}$ from previous interactions with $\mathcal{A}$.
2. $\mathcal{S}$ sends $[x]_p$, $[x_1]_p$ to $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$.

The view of adversary $\mathcal{A}$ simulated by $\mathcal{S}$ is perfect. Therefore, the view of adversary $\mathcal{A}$ that is simulated by $\mathcal{S}$ is identical to the view of $\mathcal{A}$ in the real protocol execution.

□

B.4 Proof of Theorem 4

Proof. $\mathcal{S}$ interacts with adversary $\mathcal{A}$ as follows:

1. $\mathcal{S}$ emulates the Cmp command of $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$ with $\mathcal{A}$. On receiving $([x]_p, \frac{p+1}{2})$, $\mathcal{S}$ aborts if $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$ aborts and sends $[b]_p$ to $\mathcal{A}$ otherwise, where $b = 1\{x < \frac{p+1}{2}\}$. Here, $[x]_p$ has been recorded by $\mathcal{S}$ in previous interactions with $\mathcal{A}$.
2. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{ZK}}$ by receiving $([x]_p, [b]_p)$ from $\mathcal{A}$. $\mathcal{S}$ aborts if $\mathcal{F}_{\mathsf{ZK}}$ aborts and sends $[\bar{x}]_p$ to $\mathcal{A}$ otherwise, where $\bar{x} = (2 \cdot b - 1) \cdot x - (1 - b)$.
3. $\mathcal{S}$ emulates the PosTrunc command of $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$ with $\mathcal{A}$. On receiving $([\bar{x}]_p, t)$, $\mathcal{S}$ aborts if $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$ aborts and sends $[\bar{y}]_p$ to $\mathcal{A}$ otherwise, where $\bar{y} = \mathsf{R2F}(\mathsf{F2R}(\bar{x}, p)/2^t, p)$.
4. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{ZK}}$ by receiving $([\bar{y}]_p, [b]_p)$ from $\mathcal{A}$. $\mathcal{S}$ aborts if $\mathcal{F}_{\mathsf{ZK}}$ aborts and sends $[y]_p$ to $\mathcal{A}$ otherwise, where $y = (2 \cdot b - 1) \cdot \bar{y} - (1 - b)$.
5. $\mathcal{S}$ sends $[x]_p$, $[y]_p$ to $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$.

The view of adversary $\mathcal{A}$ simulated by $\mathcal{S}$ is perfect. Therefore, the view of adversary $\mathcal{A}$ that is simulated by $\mathcal{S}$ is identical to the view of $\mathcal{A}$ in the real protocol execution.

□

B.5 Proof of Theorem 5

Proof. $\mathcal{S}$ interacts with adversary $\mathcal{A}$ as follows:

1. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{ZK}}$ by recording $([y]_p, [z_0]_p, [z_1]_p)$, sent to $\mathcal{F}_{\mathsf{ZK}}$ by $\mathcal{A}$.
2. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{ZK}}^{Lookup}$ with $\mathcal{A}$. On receiving $(L, [y]_p, [z_0]_p, [z_1]_p)$, if $\mathcal{F}_{\mathsf{ZK}}^{Lookup}$ aborts, then $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$ and aborts.
3. $\mathcal{S}$ emulates the VrfyCmp command of $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$ with $\mathcal{A}$. On receiving $([x]_p - [z_0]_p, \frac{p+1}{2})$ and $([z_1]_p - [x]_p, \frac{p+1}{2})$, if $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$ aborts, then $\mathcal{S}$ aborts. Here, $[x]_p$ has been obtained by $\mathcal{S}$ in previous interactions with $\mathcal{A}$ in $\mathcal{F}_{\mathsf{ZK}}$.
4. $\mathcal{S}$ sends $[x]_p$, $[y]_p$ to $\mathcal{F}_{\mathsf{ZK}}^{BuildBlock}$.

The view of adversary $\mathcal{A}$ simulated by $\mathcal{S}$ is perfect. Therefore, the view of adversary $\mathcal{A}$ that is simulated by $\mathcal{S}$ is identical to the view of $\mathcal{A}$ in the real protocol execution.

□