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

Meng Hao¹*, Hanxiao Chen¹*, Hongwei Li¹,², Chenkai Weng², Yuan Zhang¹, Haomiao Yang¹, and Tianwei Zhang³

¹School of Computer Science and Engineering, University of Electronic Science and Technology of China
²Northwestern University
³Nanyang Technological University

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 $P$ to convince a verifier $V$ that a public program (i.e., statement) is correctly evaluated on $P$’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 $P$) to prove to clients (as $V$) 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 ML¹, 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...

¹ML 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.

*The authors contributed equally and conducted this work at NTU.
Corresponding author.
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., \( 3 \sim 11 \)K 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\(^1\) (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 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 \( \{x_i, y_i = f(x_i)\} \forall x_i \in \mathbb{F}_p \). 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 \( \textit{structured} \) tables, namely decomposability or low-degree extension structures (refer to their paper for more details), which cannot be extended to our setting.

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

\(^{1}\)Similarly, the state-of-the-art ZK proofs [2, 57] also design their protocols in the amortized setting.
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 $\ell$. (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 $x_i \in \{0, 1\}^{\ell}$ 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 $\ell$-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]: $1\{x < c\} = 1\{x_1 < c_1\} + 1\{x_{k-1} < c_{k-1}\} \cdot 1\{x_0 < c_0\}$, 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 $z^i_0 = 1\{x_i < c_i\}$ and $z^i_{eq} = 1\{x_i = c_i\}$ by calling two table lookups. (2) $1\{x < c\}$ is verified by using $z^0_0$ and $z^0_{eq}$ 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 $z = z^0_0 \ldots z^{k-1}_0 z^{eq}_0 \ldots z^{eq}_{k-1}$ 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 $z^i_0$ 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 $\ell$, it outputs $x_1$ such that $x = x_1 \| x_0$ and $x_0 \in \{0, 1\}^\ell$. 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 $x_0 = 1$ then $x_0 = 0$ for all $i > y$. Alternatively, we have $2^y \leq x \leq 2^{y+1} - 1$. This function is currently explored in secure multi-party com-
We use $L_1$ to denote sampling $x$ uniformly at random from a finite set $S$. $\text{negl}(\cdot)$ denotes a negligible function such that $\text{negl}(k) = O(\kappa^{-c})$ for every positive constant $c$.

**Fixed-Point Representation.** Same as prior works [40,57], we encode a real number $r \in \mathbb{R}$ 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.

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

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 $[-p^{-1}2^{s-1}, p^{-1}2^{s-1}]$.

**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 $[x]_\Delta$, 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 $[x]_\Delta$, it means that $\mathcal{P}$ holds $(x, M_x)$ and $\mathcal{V}$ holds $(\Delta, K_x)$. IT-MACs are *additively* homomorphic, meaning that given public constants $c_0, c_1, \ldots, c_n \in \mathbb{F}_p$ and commitments $[x_1]_\Delta$, $[x_2]_\Delta$, $\ldots$, $[x_n]_\Delta$ from $\mathcal{P}$ and $\mathcal{V}$, check that $[x_1]_\Delta + [x_2]_\Delta + \cdots + [x_n]_\Delta$ is valid and abort if not. Compute $y = c_0 + \sum_{i=1}^{n} c_i \cdot x_i$ in $\mathbb{F}_p$, store $y$, and send $[y]_\Delta$ to $\mathcal{P}$ and $\mathcal{V}$.

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

**Functionality $\mathcal{F}_{\text{IZK}}$**

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

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

**Affine Combination:** On receiving $(\text{Affine}, c_0, c_1, \ldots, c_n, [x_1]_\Delta, \ldots, [x_n]_\Delta)$ from $\mathcal{P}$ and $\mathcal{V}$, check that $[x_1]_\Delta$, $[x_2]_\Delta$, $\ldots$, $[x_n]_\Delta$ are valid and abort if not. Compute $y = c_0 + \sum_{i=1}^{n} c_i \cdot x_i$ in $\mathbb{F}_p$, store $y$, and send $[y]_\Delta$ to $\mathcal{P}$ and $\mathcal{V}$.

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

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

**Figure 2:** Ideal functionality for ZK proofs.
pleteness, 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 $P$ to commit a size-$T$ memory containing $m_0, \ldots, m_{T-1}$, and then when $P$ accesses an element at address $i \in [0, T-1]$ from the memory, $P$ verifies the ROM and output the real value that the read address is $m_i$. Note that recent works are customized for batch ROM settings, meaning that $P$ proves that $N$ accesses on a size-$T$ memory block 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, $P$ and $V'$ 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 $i \in [0, T-1]$, $P$ and $V'$ append $(\langle i \rangle_p, \langle m_i \rangle_p, \langle 0 \rangle_p)$ to writes. (2) The access phase can be performed $N$ times. In each access of the $j$-th element in memory, where $j \in [0, T-1]$, $P$ and $V'$ append $(\langle j \rangle_p, \langle m_j \rangle_p, \langle v_j \rangle_p)$ to reads, where $v_j$ is the latest version of $m_j$ in writes, while appending $(\langle j \rangle_p, \langle m_j \rangle_p, \langle v_j + 1 \rangle_p)$ to writes. (3) Finally, in the cleanup phase, for the $i$-th element in memory, where $i \in [0, T-1]$, $P$ and $V'$ append $(\langle i \rangle_p, \langle m_i \rangle_p, \langle v_i \rangle_p)$ 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 $P$ and $V'$ pre-compute a public table storing all legitimate input-output pairs of the evaluated non-linear function, and then $P$ proves to $V'$ 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 $T + 2 \cdot N$ multiplications, 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].

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 $x \in [0, \frac{k-1}{d_i}]$, we ask the prover to provide the decomposed dig-
Functionality $F_{\text{ZK}}^{\text{BuildBlock}}$

This functionality extends the instructions in $F_{\text{ZK}}$.

**Positive digital decomposition:** On input (DigitDec, $[x]_p$, $d_0, \ldots, d_{k-1}$) from $\mathcal{P}$ and $\mathcal{P}'$, where $x \in [0, \ell - 1]$, check that $[x]_p$ is valid and abort if not. Decompose $x$ to $(x_0, \ldots, x_{k-1})$ such that $x = x_{k-1} \ldots \| x_0$ 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 $\mathcal{P}'$.

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

**Comparison:** On input (Cmp, $[x]_p, c$) from $\mathcal{P}$ and $\mathcal{P}'$, $x \in \mathbb{F}_p$, 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 $\mathcal{P}'$.

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

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

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

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

its $\{x_0, \ldots, x_{k-1}\}$ of $x$. Then the protocol verifies that (1) for $i \in [0, k-1]$, $x_i \in \{0, 1\}^{d_i}$, by invoking the CheckRange procedure of functionality $F_{\text{ZK}}^{\text{BuildBlock}}$, and (2) $\{x_0, \ldots, x_{k-1}\}$ constitute the digit decomposition of $x$, by determining $x_0 + \sum_{i=1}^{k-1} 2^{d_i} x_i = x$ based on the CheckZero procedure. The detailed protocol $\Pi_{\text{DigitDec}}$ 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 $\Pi_{\text{DigitDec}}$ cannot be directly applied to arbitrary values in $\mathbb{F}_p$. For an $\ell$-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 $x_{k-1} \ldots \| x_0 = x + p$, rather than $x$. Nonetheless, these results would still pass the verification strategy in protocol $\Pi_{\text{DigitDec}}$ mainly because the CheckRange procedure is taken over modulo $p$. This malicious behavior does not occur in the positive case. The reason is that $x_{k-1} \ldots \| x_0$ should be represented by at most $\ell - 1$ bits, namely $x_{k-1} \ldots \| x_0 < 2^\ell - 1$ due to $x \leq \ell - 1 < 2^\ell - 1$. It is a contradiction that $x_{k-1} \ldots \| x_0 = x + p$ since $p > 2^\ell - 1$ and should be represented by $\ell$ bits. To address this issue, it is necessary to add an extra check to ensure that the output digits $x_{k-1} \ldots \| x_0 < p$. 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.

Figure 5: Protocol for positive digital decomposition.

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

**Theorem 1.** Protocol $\Pi_{\text{DigitDec}}$ UC-realizes the DigitDec command of functionality $F_{\text{ZK}}^{\text{BuildBlock}}$ against static and malicious adversaries in the $(\mathbb{F}_p, F_{\text{ZK}}^{\text{LookUp}})$-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 \([21, 48]\):

\[
1 \{ x < c \} = 1 \{ x_1 < c_1 \} + 1 \{ x_1 = c_1 \} \cdot 1 \{ x_0 < c_0 \},
\]

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 \( z_{i}^{0} = 1 \{ x_i < c_i \} \) and \( z_{i}^{\text{eq}} = 1 \{ x_i = c_i \} \) by calling table lookups. (2) After obtaining all \( z_{i}^{0} \) and \( z_{i}^{\text{eq}} \), we can compute \( 1 \{ x < c \} \) recursively based on Equation 1 by calling functionality \( \text{VrfyCmp} \).

Note that the truncated server could provide an incorrect decomposition from \( x + p \) instead of \( x \). In this case, the verification will abort since \( x + p \) 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 \( z_{i}^{0} \) and \( z_{i}^{\text{eq}} \) for \( i \in [0, k-1] \), we utilize table lookup by constructing a table \( L \) containing \((z, y)\), where \( z = z_{k-1}^{0} | \ldots | z_{0}^{0} \), \( y = z_{k-1}^{\text{eq}} | \ldots | z_{0}^{\text{eq}} \in \{0, 1\}^{2k} \) and \( y \) is computed based on \( z \) according to Equation 1. Note that \( z \) is obtained via \( z = 2^{2k} \cdot z_{k-1}^{0} + \ldots + 2^{k} \cdot z_{0}^{0} + 2^{k-1} \cdot z_{k-1}^{\text{eq}} + \ldots + 2^{0} \cdot z_{0}^{\text{eq}}. \)

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 \((z, y)\), there can only be three possible cases, namely, \((0,0), (0,1), (1,0)\), since \( x_i < c_i \) and \( x_i = c_i \) cannot 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 \( z_{i}^{0} \) and \( z_{i}^{\text{eq}} \) into a single value. To this end, we design a compact encoding as follows

\[
z_i = 0 \cdots 0 | z_i^0 | 0 \cdots 0 | z_i^{eq} | 0 \cdots 0,
\]

where \( z_i \) consists of two parts, each has \( k \) bits. Except for the \( i \)-th position in each part where \( z_i^0 \) 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 \( \Pi_{\text{VrfyCmp}} \) in Figure 6. This protocol mainly consists of \( k+1 \) table lookups, which consume \( 2k + 2 \) multiplication gates.

**Theorem 2.** Protocol \( \Pi_{\text{VrfyCmp}} \) UC-realizes the VrfyCmp command of functionality \( \mathcal{F}_{\text{BuildBlock}} \) against static and malicious adversaries in the \((\mathcal{F}_Z, \mathcal{F}_{\text{Lookup}})\)-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 \( y = \text{R2F}(F2R(x, p)/2^t, p) \). 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 \( x_0 \) 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 \( x \| x_0 = p - |x| \), 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
Parameters: A finite field $F_p$, a constant $k$.

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

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

1. $\mathcal{P}$ decomposes $x$ into $(x_0, \ldots, x_{k-1})$ such that $x = x_{k-1} \cdot \cdots \cdot x_0$ where $x_i \in \{0, 1\}$ for $i \in [0, k-1]$.
2. $\mathcal{P}$ sends $(\text{Input}, x_0, \ldots, x_{k-1})$ to $j_{\mathcal{FKZ}}$, which returns $(\ldots, [\bar{x}_{i-1}], \ldots)$ to $\mathcal{P}$ and $\mathcal{V}'$.
3. $\mathcal{P}$ and $\mathcal{V}'$ locally decompose $c$ into $(c_0, \ldots, c_{k-1})$ such that $c = c_{k-1} \cdot \cdots \cdot c_0$ and $c_i \in \{0, 1\}$ for $i \in [0, k-1]$.
4. $\mathcal{P}$ computes $z_i = 2^{k-1} \cdot \lceil x_i \rceil / 2^i$, for $i \in [0, k-1]$, and sends $(\text{Input}, [z_0, \ldots, z_{k-1}])$ to $j_{\mathcal{FKZ}}$, which returns $([z_0], \ldots, [z_{k-1}])$ to $\mathcal{P}$ and $\mathcal{V}'$.
5. $\mathcal{P}$ sends $(\text{Input}, y)$ to $j_{\mathcal{FKZ}}$, which returns $[y]_p$ to $\mathcal{P}$ and $\mathcal{V}'$.
6. $\mathcal{P}$ and $\mathcal{V}'$ compute $\bar{y} = \sum_{i=0}^{k-1} z_i [y]_p$ and send $(\text{Lookup}, y_j [y]_p)$, where $L = \{(\sum_{i=0}^{k-1} 2^{k-1-i} \cdot \lceil x_i \rceil + 2^{i-1} \cdot \lceil x_{i-1} \rceil)_{y_i} = \{0, 1\}^{16}\}$. Here, $y_0 = 1 \cdot \lceil x_0 \rceil$ and $y_i = 1 \cdot \lceil x_i \rceil + 1 \cdot \lceil x_{i-1} \rceil$, for $i \in [1, k-1]$. $\mathcal{P}$ sends $(\text{Input}, y)$ to $j_{\mathcal{FKZ}}$, which returns $[y]_p$ to $\mathcal{P}$ and $\mathcal{V}'$.
7. $\mathcal{P}$ and $\mathcal{V}'$ execute the CheckZero procedure on $[y]_p - 1$.
8. If any of the above checks fails, $\mathcal{V}'$ aborts, otherwise $\mathcal{V}'$ outputs success.

Theorem 3. Protocol $\Pi_{\text{PosTrunc}}$ UC-realizes the PosTrunc command of functionality $j_{\mathcal{FKZ}}$ against static and malicious adversaries in the $(j_{\mathcal{FKZ}}, j_{\text{Flip}})$-hybrid model.

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

Theorem 4. Protocol $\Pi_{\text{Trunc}}$ UC-realizes the Trunc command of functionality $j_{\text{BuildBlock}}$ against static and malicious adversaries in the $(j_{\mathcal{FKZ}}, j_{\text{Flip}})$-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 $x \in [0, p^{-1}/2]$, the most significant non-zero-bit (Msnbz) outputs $y$ such that if $x_1 = 1$ then $x_1 = 0$ for all $i > y$, namely $2^y \leq x < 2^{y+1} - 1$. 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 Msnbz is to directly use the inequality $2^y \leq x < 2^{y+1} - 1$ 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 $y \in [0, \lceil \log p \rceil - 2]$. The latter can be regarded as checking whether $x - z_0 < p^{-1}/2$ and $z_1 - x < p^{-1}/2$ 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 $F_p$, because $2^{\ell-1} - 1 \geq p^{-1}/2$, which can be demonstrated by a contradiction, that is, assuming $2^{\ell-1} - 1 < p^{-1}/2$ we have $p > 2^{\ell-1} - 1$, 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.
The exponential operation $y = \left(\frac{1}{2}\right)^x$ 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 $x_0, \ldots, x_k$ using our digital decomposition protocol, such that $x = x_k \cdots x_0$, 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 input $x_i$ of the above checks either $0$ 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.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{protocol.png}
\caption{Protocol for general truncation.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{protocol.png}
\caption{Protocol for most significant non-zero-bit.}
\end{figure}

Theorem 5. Protocol $\Pi_{Msnzb}$ UC-realizes the Msnzb command of functionality $f_{ZK}^{BuildBlock}$ against static and malicious adversaries in the $(f_{ZK}^{BuildBlock}, f_{ZK}^{Lookup})$-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 $y = \left(\frac{1}{2}\right)^x$ 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 $x_0, \ldots, x_k$ using our digital decomposition protocol, such that $x = x_k \cdots x_0$, 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 input $x_i$ of the above checks either $0$ 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.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{protocol.png}
\caption{Protocol for general truncation.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{protocol.png}
\caption{Protocol for most significant non-zero-bit.}
\end{figure}

Theorem 5. Protocol $\Pi_{Msnzb}$ UC-realizes the Msnzb command of functionality $f_{ZK}^{BuildBlock}$ against static and malicious adversaries in the $(f_{ZK}^{BuildBlock}, f_{ZK}^{Lookup})$-hybrid model.

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

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 $\mathcal{F}_{ZK}^{Math}$**

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

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

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

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

Due to space limitations, we provide this protocol in the full version of our work.

![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.](image)

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.

![Figure 11: Protocol for exponential.](image)

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, 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

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

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

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

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 $\Pi_{\text{Div}}$**

**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 $y = \text{PtDiv}(x)$ as follows:

Step 1. Normalize the input:
1. $\mathcal{P}$ and $\mathcal{V}'$ send $\langle \text{Msnzb}, [x]_p \rangle$ to functionality $\mathcal{F}_{\text{BuildBlock}}^ZK$, 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 $(\text{Input}, d)$ to functionality $\mathcal{F}_{\text{ZK}}$, which returns $[d]_p$ to $\mathcal{P}$ and $\mathcal{V}'$.
3. $\mathcal{P}$ and $\mathcal{V}'$ send $\langle \text{Lookup}, L, [k]_p, [d]_p \rangle$ to functionality $\mathcal{F}_{\text{ZK}}^\mathcal{K}$, 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 $\mathcal{F}_{\text{ZK}}$.

Step 2. Compute the initial approximation:
1. $\mathcal{P}$ and $\mathcal{V}'$ send $\langle \text{DigitDec}, [z]_p, n-1 - m, m+1 \rangle$ to functionality $\mathcal{F}_{\text{BuildBlock}}^ZK$, which returns $[z_0]_p, [z_1]_p$ such that $z = z_1 || z_0$, $z_0 \in \{0, 1\}^{n-1-m}$, and $z_1 \in \{0, 1\}^m$.
2. $\mathcal{P}$ computes $a = R2F(\frac{2^m - 1}{2^{m+1} - 1}, p, s + n - 1) \in \{0, 1\}^{s+n-1}$ and $b = R2F(\frac{1}{2^{m+1} - 1}, p, s) \in \{0, 1\}^s$, where $\hat{z}_1 = F2R(z_1, p, m)$, and sends $(\text{Input}, a, b)$ to functionality $\mathcal{F}_{\text{ZK}}$, which returns $[a]_p, [b]_p$ to $\mathcal{P}$ and $\mathcal{V}'$.
3. $\mathcal{P}$ and $\mathcal{V}'$ send $\langle \text{Lookup}, L, [z_1]_p, [a]_p, [b]_p \rangle$ to functionality $\mathcal{F}_{\text{ZK}}^\mathcal{K}$, where $L = \{z_1, R2F(\frac{2^m - 1}{2^{m+1} - 1}, p, s + n - 1), R2F(\frac{1}{2^{m+1} - 1}, p, s)\} || z_0 \in \{0, 1\}^{m+1}$ with $\hat{z}_1 = F2R(z_1, p, m)$.
4. $\mathcal{P}$ and $\mathcal{V}'$ compute $[y']_p = [a]_p - [b]_p \cdot [z_0]_p$, by calling functionality $\mathcal{F}_{\text{ZK}}$ and send ($\text{PosTrunc}, [y']_p, n-1$) to functionality $\mathcal{F}_{\text{BuildBlock}}^ZK$, which returns $[y]_p$ to $\mathcal{P}$ and $\mathcal{V}'$.

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

Step 4. Normalize the output:
1. $\mathcal{P}$ computes $e = 2^{n-k}$ and sends $(\text{Input}, e)$ to functionality $\mathcal{F}_{\text{ZK}}$, which returns $[e]_p$ to $\mathcal{P}$ and $\mathcal{V}'$.
2. $\mathcal{P}$ and $\mathcal{V}'$ send $\langle \text{Lookup}, L, [k]_p, [e]_p \rangle$ to functionality $\mathcal{F}_{\text{ZK}}^\mathcal{K}$, 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 $\mathcal{F}_{\text{ZK}}$, and send ($\text{PosTrunc}, [y]_p, n-s$) to functionality $\mathcal{F}_{\text{BuildBlock}}^ZK$, 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 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

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

(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

$$y_i = \frac{e^{x_i - x_{\text{max}}}}{\sum_{j=0}^{n-1} e^{x_j - x_{\text{max}}}},$$  

(6)

where $x_{\text{max}} = \text{Max}(x_0, \ldots, x_{n-1})$. Note that we can utilize the known input range to reduce the overhead of division. The real input $\sum_{i=0}^{n-1} e^{x_i - x_{\text{max}}}$ 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

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

where $\tanh(x) = 2 \cdot \text{Sigmoid}(2x) - 1$. 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

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

where $(\gamma, \beta)$ are trained affine transform parameters, $\mu = \frac{\sum_{i=0}^{n-1} x_i}{n}$ and $\sigma = \sqrt{\frac{\sum_{i=0}^{n-1} (x_i - \mu)^2}{n}}$. 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 $10^3$ 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.

### 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 200Mbps, our building blocks only take around 10 $\sim$ 34 $\mu$s. 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>
<thead>
<tr>
<th>Protocol</th>
<th>Runtime ($\mu$s) on different bandwidths</th>
<th>Comm. (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DigitDec</td>
<td>30.360 10.320 9.058 8.946 0.159</td>
<td>33rd USENIX Security Symposium</td>
</tr>
<tr>
<td>VrfyCMP</td>
<td>15.862 14.314 14.358 0.230</td>
<td>USENIX Association</td>
</tr>
<tr>
<td>CMP</td>
<td>20.662 18.918 18.569 0.301</td>
<td>33.806 0.159</td>
</tr>
<tr>
<td>PosTrunc</td>
<td>10.352 8.990 8.951 0.159</td>
<td>34.806 0.475</td>
</tr>
<tr>
<td>Trunc</td>
<td>32.488 28.899 28.814 0.508</td>
<td>30.360 0.524</td>
</tr>
<tr>
<td>MsnzB</td>
<td>34.806 30.360 30.224 0.508</td>
<td>33rd USENIX Security Symposium</td>
</tr>
</tbody>
</table>
Table 2: Runtime (sec) and communication (MB) overhead of our building blocks with the different number of instances ($10^3$, $10^4$, $10^5$).

<table>
<thead>
<tr>
<th>Protocol</th>
<th>$10^3$</th>
<th>$10^4$</th>
<th>$10^5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ours</td>
<td>0.008</td>
<td>0.215</td>
<td>0.113</td>
</tr>
<tr>
<td>Mystique</td>
<td>0.023</td>
<td>0.599</td>
<td>0.164</td>
</tr>
<tr>
<td>CMP</td>
<td>0.008</td>
<td>0.217</td>
<td>0.109</td>
</tr>
<tr>
<td>PosTrunc</td>
<td>0.008</td>
<td>0.215</td>
<td>0.115</td>
</tr>
<tr>
<td>Trunc</td>
<td>0.017</td>
<td>0.448</td>
<td>0.204</td>
</tr>
<tr>
<td>Msnb</td>
<td>0.033</td>
<td>0.868</td>
<td>0.327</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Protocol</th>
<th>12</th>
<th>14</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ours</td>
<td>0.906</td>
<td>15.571</td>
<td>0.906</td>
</tr>
<tr>
<td>Mystique</td>
<td>1.431</td>
<td>22.504</td>
<td>1.431</td>
</tr>
<tr>
<td>CMP</td>
<td>1.892</td>
<td>29.374</td>
<td>1.892</td>
</tr>
<tr>
<td>PosTrunc</td>
<td>0.899</td>
<td>15.571</td>
<td>1.157</td>
</tr>
<tr>
<td>Trunc</td>
<td>2.890</td>
<td>46.471</td>
<td>3.118</td>
</tr>
<tr>
<td>Msnb</td>
<td>3.036</td>
<td>49.586</td>
<td>3.036</td>
</tr>
</tbody>
</table>

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. 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 to 130× on the runtime. Furthermore, our communication cost is at a similar level as Mystique, and precisely, it is 1.4 to 2.9× 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×. For the GELU activation in LLMs, we achieve 77 to 86× improvement. What’s more, for the softmax function, we even obtain a 179× 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 to 4.8× better communication performance.

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

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Runtime (sec) on different bandwidths</th>
<th>Comm. (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>200 Mbps</td>
<td>500 Mbps</td>
</tr>
<tr>
<td>Exponential</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mystique</td>
<td>1184.240</td>
<td>1130.020</td>
</tr>
<tr>
<td>Division</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mystique</td>
<td>636.038</td>
<td>617.690</td>
</tr>
<tr>
<td>Reciprocal square root</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ours</td>
<td>13.406</td>
<td>11.836</td>
</tr>
<tr>
<td>Mystique</td>
<td>856.267</td>
<td>824.639</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Runtime (sec) on different bandwidths</th>
<th>Comm. (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>200 Mbps</td>
<td>500 Mbps</td>
</tr>
<tr>
<td>ReLU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ours</td>
<td>2.107</td>
<td>1.906</td>
</tr>
<tr>
<td>Mystique</td>
<td>200.433</td>
<td>193.797</td>
</tr>
<tr>
<td></td>
<td>(95.113×)</td>
<td>(101.655×)</td>
</tr>
<tr>
<td>Sigmoid</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ours</td>
<td>19.544</td>
<td>17.706</td>
</tr>
<tr>
<td>Mystique</td>
<td>1918.970</td>
<td>1847.300</td>
</tr>
<tr>
<td></td>
<td>(98.188×)</td>
<td>(104.332×)</td>
</tr>
<tr>
<td>GELU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ours</td>
<td>37.628</td>
<td>32.696</td>
</tr>
<tr>
<td>Mystique</td>
<td>2719.110</td>
<td>2711.700</td>
</tr>
<tr>
<td></td>
<td>(72.264×)</td>
<td>(82.936×)</td>
</tr>
<tr>
<td>Maxpooling, dim = 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ours</td>
<td>10.439</td>
<td>9.310</td>
</tr>
<tr>
<td>Mystique</td>
<td>804.715</td>
<td>774.942</td>
</tr>
<tr>
<td></td>
<td>(77.084×)</td>
<td>(83.240×)</td>
</tr>
<tr>
<td>Softmax, dim = 10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ours</td>
<td>87.131</td>
<td>78.289</td>
</tr>
<tr>
<td>Mystique</td>
<td>1497.300</td>
<td>1408.600</td>
</tr>
<tr>
<td></td>
<td>(171.849×)</td>
<td>(179.436×)</td>
</tr>
<tr>
<td>Normalization, dim = 16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ours</td>
<td>192.826</td>
<td>176.140</td>
</tr>
<tr>
<td>Mystique</td>
<td>9667.060</td>
<td>9642.600</td>
</tr>
<tr>
<td></td>
<td>(50.134×)</td>
<td>(54.744×)</td>
</tr>
</tbody>
</table>

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 essential 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 2022YFB3103050, the National Natural Science Foundation of China under Grant 62020106013, the Sichuan Science and Technology Program under Grant 2023ZD0142, 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


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 \( \Pi \) be a protocol. We say that \( \Pi \) UC realizes \( \mathcal{F} \) if for any adversary \( \mathcal{A} \) there exists an ideal-process adversary \( \mathcal{S} \) such that for any environment \( \mathcal{Z} \),

\[
\text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}} \approx \text{REAL}_{\Pi,\mathcal{A},\mathcal{Z}}. \tag{9}
\]

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

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 \( f_{\text{ZK}} \) by recording \( ([x_0], [x_1], \ldots, [x_{k-1}, p]), \) sent to \( f_{\text{ZK}} \) by \( \mathcal{A} \).
2. \( \mathcal{S} \) emulates \( f_{\text{Lookup}} \) with \( \mathcal{A} \). On receiving \( (R_i, [x_i], p) \) for \( i \in \{0, k-1\} \), if \( f_{\text{Lookup}} \) aborts, then \( \mathcal{S} \) sends abort to \( f_{\text{ZK}} \) and aborts.
3. \( \mathcal{S} \) locally computes \( [z]_p = [x_0]_p + \sum_{i=1}^{k-1} [x_i]_p - [x_i]_p. \) Note that \( [x_i]_p \) has been recorded by \( \mathcal{S} \) in previous interactions with \( \mathcal{A} \).
4. $S$ executes the CheckZero procedure with $A$. If the received values are not equal to $[c]_p$ in the above step, then $S$ sends abort to $F_{ZK}^{\text{BuildBlock}}$ and aborts.

5. $S$ sends $([x]_p, [x_0]_p, \ldots, [x_{k-1}]_p)$ to $F_{ZK}^{\text{BuildBlock}}$.

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

B.2 Proof of Theorem 2

Proof. $S$ interacts with adversary $A$ as follows:

1. $S$ emulates $F_{ZK}$ by recording $([x_0], [x_1], \ldots, [x_{k-1}]_p)$, sent to $F_{ZK}$ by $A$.

2. $S$ locally computes $[t]_p = [x_0]_p + \sum_{i=1}^{k-1} 2^{\sum_{j=0}^{i-1} d_i} [x_i]_p - [x]_p$. Note that $[x]_p$ has been obtained by $S$ in previous interactions of $A$ with $F_{ZK}$.

3. $S$ executes the CheckZero procedure with $A$. If the received values are not equal to $[t]_p$ in the above step, then $S$ sends abort to $F_{ZK}^{\text{BuildBlock}}$ and aborts.

4. $S$ emulates $F_{ZK}$ by recording $([x_0], [x_1], \ldots, [x_{k-1}]_p)$, sent to $F_{ZK}$ by $A$.

5. $S$ emulates $F_{ZK}^{\text{Lookup}}$ with $A$. On receiving $(L_i, [x_i]_p, [z_i]_p)$ for $i \in [0, k-1]$, if $F_{ZK}^{\text{Lookup}}$ aborts, then $S$ sends abort to $F_{ZK}^{\text{BuildBlock}}$ and aborts.

6. $S$ emulates $F_{ZK}$ by recording $[y]_p$, sent to $F_{ZK}$ by $A$.

7. $S$ locally computes $[z]_p = \sum_{i=0}^{k-1} [z_i]_p$. $S$ emulates $F_{ZK}^{\text{Lookup}}$ with $A$. On receiving $(L, [x]_p, [y]_p)$, if $F_{ZK}^{\text{Lookup}}$ aborts, then $S$ sends abort to $F_{ZK}^{\text{BuildBlock}}$ and aborts.

8. $S$ executes the CheckZero procedure with $A$. If the received values are not equal to $[y]_p - 1$ in the above step, then $S$ sends abort to $F_{ZK}^{\text{BuildBlock}}$ and aborts.

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

B.3 Proof of Theorem 3

Proof. $S$ interacts with adversary $A$ as follows:

1. $S$ emulates the DigitDec command of $F_{ZK}^{\text{BuildBlock}}$ with $A$.

On receiving $([x]_p, t, m-t)$, $S$ aborts if $F_{ZK}^{\text{BuildBlock}}$ aborts and sends $[x_0]_p, [x_1]_p$ to $A$ otherwise, where $x = x_1 x_0$, $x_0 \in \{0, 1\}^d$ and $x_1 \in \{0, 1\}^{m-1}$ with $m = \log p - 1$. Here, $[x]_p$ is held by $S$ from previous interactions with $A$.

2. $S$ sends $([x]_p, [x_1]_p)$ to $F_{ZK}$.

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

B.4 Proof of Theorem 4

Proof. $S$ interacts with adversary $A$ as follows:

1. $S$ emulates the Cmp command of $F_{ZK}^{\text{BuildBlock}}$ with $A$. On receiving $([x]_p, [y]_p)$, $S$ aborts if $F_{ZK}^{\text{BuildBlock}}$ aborts and sends $[b]_p$ to $A$ otherwise, where $b = 1\{x < \frac{y + 1}{2}\}$. Here, $[x]_p$ has been recorded by $S$ in previous interactions with $A$.

2. $S$ emulates $F_{ZK}$ by receiving $([x]_p, [b]_p)$ from $A$. $S$ aborts if $F_{ZK}$ aborts and sends $[x]_p$ to $A$ otherwise, where $x = (2b - 1) \cdot x - (1 - b)$.

3. $S$ emulates the PosTrunc command of $F_{ZK}^{\text{BuildBlock}}$ with $A$. On receiving $(\bar{y}, [x]_p, t)$, $S$ aborts if $F_{ZK}^{\text{BuildBlock}}$ aborts and sends $\bar{y}$ to $A$ otherwise, where $\bar{y} = \text{R2F}(2R(\bar{x} p) / 2^i)$.

4. $S$ emulates $F_{ZK}$ by receiving $([y]_p, [b]_p)$ from $A$. $S$ aborts if $F_{ZK}$ aborts and sends $[y]_p$ to $A$ otherwise, where $y = (2b - 1) \cdot \bar{y} - (1 - b)$.

5. $S$ sends $[x]_p, [x_1]_p$ to $F_{ZK}$.

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

B.5 Proof of Theorem 5

Proof. $S$ interacts with adversary $A$ as follows:

1. $S$ emulates $F_{ZK}$ by recording $([y]_p, [z_0]_p, [z_1]_p)$, sent to $F_{ZK}$ by $A$.

2. $S$ emulates $F_{ZK}^{\text{Lookup}}$ with $A$. On receiving $(L, [y]_p, [z_0]_p, [z_1]_p)$, if $F_{ZK}^{\text{Lookup}}$ aborts, then $S$ sends abort to $F_{ZK}^{\text{BuildBlock}}$ and aborts.

3. $S$ emulates the VrfyCmp command of $F_{ZK}^{\text{BuildBlock}}$ with $A$. On receiving $([x]_p, [z_0]_p, [z_1]_p)$ and $([z_1]_p - [x]_p, \frac{p+1}{2})$, if $F_{ZK}^{\text{BuildBlock}}$ aborts, then $S$ aborts. Here, $[x]_p$ has been obtained by $S$ in previous interactions with $A$ in $F_{ZK}$.

4. $S$ sends $[x]_p, [y]_p$ to $F_{ZK}$.

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