Efficient 3PC for Binary Circuits with Application to Maliciously-Secure DNN Inference

Yun Li[*‡], Yufei Duan[*], Zhicong Huang[†], Cheng Hong[‡], Chao Zhang[*], Yifan Song[*]

[*]*Tsinghua University*, [†]*Alibaba Group*, [‡]*Ant Group*

liyun19@mails.tsinghua.edu.cn, *duanyufi@foxmail.com*, *zhicong.hzc@alibaba-inc.com*,
vince.hc@antgroup.com, *chaoz@tsinghua.edu.cn*, *yfsong@mail.tsinghua.edu.cn*

Abstract

In this work, we focus on maliciously secure 3PC for binary circuits with honest majority. While the state-of-the-art (Boyle et al. CCS 2019) has already achieved the same amortized communication as the best-known semi-honest protocol (Araki et al. CCS 2016), they suffer from a large computation overhead: when comparing with the best-known implementation result (Furukawa et al. Eurocrypt 2017) which requires 9× communication cost of Araki et al., the protocol by Boyle et al. is around 4.5× slower than that of Furukawa et al.

In this paper, we design a maliciously secure 3PC protocol that matches the same communication as Araki et al. with comparable concrete efficiency as Furukawa et al. To obtain our result, we manage to apply the distributed zero-knowledge proofs (Boneh et al. Crypto 2019) for verifying computations over $\mathbb{F}_2$ by using *prime* fields and explore the algebraic structure of prime fields to make the computation of our protocol friendly for native CPU computation.

Experiment results show that our protocol is around 3.5× faster for AES circuits than Boyle et al. We also applied our protocol to the binary part (e.g. comparison and truncation) of secure deep neural network inference, and results show that we could reduce the time cost of achieving malicious security in the binary part by more than 67%.

Besides our main contribution, we also find a hidden security issue in many of the current probabilistic truncation protocols, which may be of independent interest.

1 Introduction

Secure multiparty computation(MPC) has been an appealing and blooming field of research. Informally, MPC protocols allow distrustful parties jointly computing a function without leaking any information about their inputs (except that can be inferred from the output of this function).

Yao's work [48] is a seminal research of secure computation, on which many cryptographic primitives thrive for building MPC protocols, such as garbled circuits [5, 31, 37, 39, 41, 48–50] and secret sharing [2, 3, 14, 22, 27, 32, 38, 44, 47].

Among them, three party computation against one corrupted party (i.e., in an honest majority) is particularly promising due to its potential of being highly efficient and practical. However, in malicious setting where the corrupted party can arbitrarily deviate from the protocol, achieving high concrete efficiency becomes much more challenging, especially for those computing binary circuits.

Prior works have explored efficient three party computations on binary circuits against malicious adversaries in the honest majority setting. Representative works [2, 21], based on the replicated secret sharing scheme, follow the Beaver multiplication triple method [4] and rely on the cut-and-choose paradigm [13] to generate verified AND triples. The protocol in [21] needs to communicate 10 bits per party to compute an AND gate[1], and [2] further reduces the number to 7 bits per party by taking a smaller bucket size but requires an additional shuffle process.

Despite the progress achieved in [2, 21], the maliciously secure protocol still requires 7 times of the communication complexity compared with the best-known semi-honest protocol [3], which only needs to communicate 1 bit per AND gate per party. In a beautiful work [7], Boyle et al. explore a different way to achieve malicious security by utilizing distributed zero-knowledge proofs [6]. As a result, their protocol achieves 1 bit per AND gate per party in an amortized way, matching the communication complexity of [3]. However, the distributed-zero knowledge proofs [6] require the size of the underlying field to be large enough so that the achieved soundness error, which is proportional to the inverse of the field size, is negligible. For the particular case of binary circuits, the protocol in [7] has to lift the data from the binary field $\mathbb{F}_2$ to an extension field $\mathbb{F}_{2^\kappa}$ (where κ is the security parameter). The large computational overhead due to the distributed-zero knowledge proofs (especially due to the need of doing arithmetic operations over extension fields) significantly hurts the concrete efficiency of the protocol in [7]. Compared with [21]

[1] The implementation in MP-SPDZ [28] improves the cost of [21] to 9 bits per party per AND gate.

where most of computation is done over $\mathbb{F}_2$[2], the saving in the communication leads to 4.5× slow down in the running time in the LAN setting. On the other hand, the saving in the communication indeed benefits the WAN setting where network resources are limited.

This leads to our following question: *Can we build efficient 3 party computation for binary circuits with malicious security that matches the communication complexity of the best-known semi-honest protocol [3] while enjoys a comparable concrete efficiency as the cut-and-choose based approach [2, 21]?*

1.1 Our Contributions

Efficient 3PC for Binary Circuits with Malicious Security. In this work, we answer the above question affirmatively by constructing a concretely efficient three-party computation protocol for binary circuits that is secure (with abort) against malicious adversaries in the honest majority setting. Our protocol can be viewed as a variant of [7] tailored for binary circuits by designing an efficient verification protocol on the binary field $\mathbb{F}_2$ using the distributed zero-knowledge proofs [6]. Our verification protocol features *sub-linear communication cost and round complexity*, as well as *lower computational cost* compared with the approach based on extension fields [6, 7].

At a high level, the improvement comes from the following two techniques.

- First, we initiate the approach of using a *prime* field $\mathbb{F}_p$ (In our experiment, we set $p = 2^{61} - 1$) in the distributed zero-knowledge proof to verify *binary computation* instead of using the binary extension field as [7].

This is motivated by the fact that arithmetic operations over a binary extension field are less efficient than those over a prime field with comparable size. In Appendix E.1, our experiments show that arithmetic operations over the 64-bit extension field $\mathbb{F}_{2^{64}}$ is about 7.6× slower than those over the 61-bit Mersenne prime field $\mathbb{F}_{2^{61}-1}$. However transforming the verification of binary computation to prime fields is not straightforward. We show how to mitigate the cost during the transformation so that the number of multiplications we need to verify in prime fields is only 2× of that in the binary fields. See Section 3.2 for more details.

- Second, we exploit the algebraic structure of prime fields to further improve the *computation* complexity of the distributed zero-knowledge proof technique.

In the standard distributed zero-knowledge technique [6, 7], to check m multiplication triples over $\mathbb{F}_p$, the computation complexity is $O(m)$ operations over $\mathbb{F}_p$. Since in our case, we transform AND triples to multiplication triples over $\mathbb{F}_p$, we show how to reduce most of the computation from $\mathbb{F}_p$ to the binary field $\mathbb{F}_2$. Furthermore, by exploiting the algebraic structure of prime fields, we can batch the computation using bit-wise XOR and AND operations, which are very efficient and friendly for native CPU computation. See Section 4.2 for more details.

We put everything together and test the AES circuits using the MP-SPDZ framework [28]. The results show that our protocol is at least 3.5× faster in computation than [7] with comparable communication complexity. Our computation time is only 1.25× of the cut-and-choose based approach [21] whose communication is around 9× of ours.

Applications to Maliciously Secure DNN Inference. Nowadays Privacy-Preserving Machine Learning (PPML) has attracted a surge of interests from researchers. Plenty of works [12, 16, 17, 32, 35, 36, 38, 43, 45, 46] have built secure computation frameworks that are customized for machine learning primitives. It is established in previous works [17, 35] that computing non-linear functions (e.g., comparison, truncation) is more efficient using binary circuits. As non-linear operations are pervasively used in machine learning tasks, the protocol for securely computing binary circuits significantly influences the overall performance (see Section 5 for more details).

We have implemented a mixed protocol for maliciously secure three-party DNN inference using the MP-SPDZ [28] framework. Our experiment results show that our protocol reduces the overhead for obtaining malicious security in binary part by more than 67% compared to [7]. When comparing with the cut-and-choose based approach [21], our protocol not only reduces the communication complexity by 9×, but also achieves a faster running time.

A Security Issue in Probabilistic Truncation Protocols. As an extra contribution, we have found, as far as we know, for the first time a hidden security issue in many of the current probabilistic truncation protocols [11, 17, 32, 35, 36, 38]. Briefly put, the root cause of the security issue is that the same randomness is used for both protecting the privacy of the secret value to be truncated, and sampling the 1-bit rounding error for the truncated value probabilistically. This is problematic when considering the joint distribution of the messages exchanged by the parties and the output of the protocol (which is required according to the well accepted definition of MPC [9]). As a result, much more binary gates have to be introduced due to the need of using exact truncation protocols. We refer the readers to Section 5.2 for more discussion.

1.2 Related Work

A classical way of securely computing binary circuits against malicious adversaries is compiling Yao's semi-honest garbled

[2] We choose to compare with [21] rather than the optimized version [2] because in MP-SPDZ, the implementation of [21] is faster than that of [2]. And the work [2] does not open the source of their implementation.

circuit protocol [48] with the cut-and-choose paradigm [13, 30, 33, 34]. It requires the garbler sending many independent garbled version of the same circuit for the evaluator to randomly check. These garbled circuit based approaches have a constant number of rounds, which is an inherited benefit, but they also inevitably have high communication cost.

Kikuchi et al. [29] propose the idea of dynamically changing the underlying fields to improve efficiency. Their construction allows the parties to use a field of small size for efficient (semi-honest) computation, and switch to a field of large size for verification to ensure the desired statistical error. However, their approach also relies on extension fields and simply maps shares over a base field to its extension field for field transformation. Our protocol, unlike theirs, transforms computations over binary fields to (Mersenne) prime fields rather than extension fields, and thus allows more efficient constructions due to faster arithmetic operations.

The work of Polychroniadou and Song [40] first achieves unconditional security (with abort) against a malicious adversary in the n-party honest majority setting for binary circuits. They rely on Reverse Multiplication-Friendly Embeddings (RMFE) [10] and conceive a new way to authenticate secrets to detect malicious behaviors with an amortized communication complexity of $O(n)$ bits per gate. However, their construction only focuses on asymptotic performance and its concrete efficiency is worse than [7], and the RMFE scheme still requires extension fields, which introduces heavier computational overhead compared with prime fields.

Other representative works [25, 26] explore Shamir secret sharing [44] for arithmetic circuits in the n-party setting. They offer a construction with an amortized communication complexity of $O(n)$ field elements per multiplication gate. The main technique [25] they rely on to achieve malicious security can be viewed as a variant of [6, 7]. However, when using their construction for binary circuits, they still have to raise the verification to the binary extension field as [7].

2 Preliminaries

2.1 Notations

Let $\mathcal{P} = \{P_0, P_1, P_2\}$ be a set of three parties. We use $P_i, i \in \{0,1,2\}$ to denote a certain party in this set, and $P_{i\pm1}$ denotes the next $(+)$ or previous $(-)$ party with wrap around.

We use $\mathbb{Z}_{2^k}$ to denote the ring of modulus 2^k. We use $\mathbb{F}_2$ to refer to the binary field, and $\mathbb{F}_p$ to denote a finite field of prime order p having bit length $\sigma = \log p$. For simplicity, we write $[n]$ to denote the set $\{1, 2, \ldots, n\}$. We use $\vec{u}$ to denote a vector and u_j the j-th element of the vector.

Let κ be a security parameter. We use $\mathcal{F} = \{\mathrm{F_K} \mid \mathrm{K} \in \{0,1\}^\kappa, \mathrm{F_K} : \{0,1\}^\kappa \to R\}$ for a family of pseudo-random functions [24]. Given a key $\mathrm{K} \in \{0,1\}^\kappa$, the pseudo-random function $\mathrm{F_K}$ takes as input a string $x \in \{0,1\}^\kappa$ and outputs a pseudo-random string $y \in R$ where R is $\mathbb{Z}_{2^k}, \mathbb{F}_2$, or $\mathbb{F}_p$.

2.2 Secret Sharing

2.2.1 Additive Secret Sharing

Additive secret sharing is a basic cryptographic primitive in many secure computation protocols. In an additive secret sharing scheme over ring $\mathbb{Z}_{2^k}$ for n parties, a secret $x \in \mathbb{Z}_{2^k}$ is split into n random values $x_0, \cdots, x_{n-1} \in \mathbb{Z}_{2^k}$ s.t. $x = \sum_{i=0}^{n-1} x_i$, and each party P_i for $i \in \{0, \cdots, n-1\}$ holds an additive share x_i. Clearly, the secret x can be reconstructed iff all these n parties reveal their share x_i and then sum them up. Thus, the additive secret sharing scheme has perfect secrecy against $n-1$ corrupted parties.

2.2.2 Replicated Secret Sharing

Replicated secret sharing (RSS) [27] is an extension of additive secret sharing. We focus on the RSS scheme for three parties, where the secret $x \in \mathbb{Z}_{2^k}$ is split into three random values $x_0, x_1, x_2 \in \mathbb{Z}_{2^k}$ s.t. $x = x_0 + x_1 + x_2$, and each party P_i for $i \in \{0,1,2\}$ now holds two additive shares (x_i, x_{i-1}) as its share of x. We denote the RSS scheme over ring $\mathbb{Z}_{2^k}$ as $[\![\cdot]\!]^{\mathrm{R}}$. For simplicity, we sometimes write the sharing of $x \in \mathbb{Z}_{2^k}$ as $[\![x]\!]^{\mathrm{R}} = (x_0, x_1, x_2)$.

Reconstruction and Opening. To reconstruct a secret x to some party P_i, the other two parties P_{i+1}, P_{i-1} both send x_{i+1} to P_i. Then P_i checks if the values received from the two parties are consistent. If the consistency holds, P_i reconstructs x by computing $x := x_0 + x_1 + x_2$. Otherwise P_i aborts. We define this procedure as $\mathsf{reconstruct}([\![x]\!]^{\mathrm{R}}, i)$.

To reveal a secret x to all, the parties can simply run $\mathsf{reconstruct}([\![x]\!]^{\mathrm{R}}, i)$ for each $i \in \{0,1,2\}$ respectively.

It is clear that in the above RSS scheme, no single party can learn any information about the secret x, but any two parties can reconstruct the secret x. Therefore the RSS scheme is secure against one corrupted party.

RSS over Binary Field and Finite Field. Besides the ring $\mathbb{Z}_{2^k}$, the RSS scheme applies to the binary field $\mathbb{F}_2$ and the finite field $\mathbb{F}_p$ naturally. In this case, the elements come from $\mathbb{F}_2$ or $\mathbb{F}_p$; the addition and multiplication operations are the ones defined on $\mathbb{F}_2$ and $\mathbb{F}_p$ respectively. We use $[\![\cdot]\!]^{\mathrm{B}}, [\![\cdot]\!]^{\mathrm{F}}$ to explicitly denote the RSS schemes over $\mathbb{F}_2$ and $\mathbb{F}_p$. Sometimes we may use $\oplus$ to denote the addition operation over $\mathbb{F}_2$ explicitly.

2.3 Distributed Zero-Knowledge Proofs

Zero-knowledge proofs on secret-shared data [6] are another key ingredient of our protocols for achieving malicious security. As shown in a fundamental work of Goldreich, Micali, and Wigderson [22], a standard way to compile a semi-honest secure computation protocol into a maliciously secure one is to enforce semi-honest behaviors via zero-knowledge proofs, i.e., requiring each party to provide a zero-knowledge proof along with its message to argue that the message is faithfully computed according to the semi-honest protocol. For secret sharing based protocols, such a statement is distributed among the parties, i.e., the input (e.g., the message) of the statement is secret-shared among the parties. We follow previous works [6–8, 32, 38] to adopt distributed zero-knowledge proofs [6] to achieve malicious security.

The distributed zero-knowledge techniques [6] are built on fully linear proofs [6]. The main underlying technique we rely on is the fully linear interactive oracle proof (FLIOP) which is based on a fully linear probabilistic checkable proof (FLPCP) construction. This proof system works over a finite field and requires the field size to be large enough (since the soundness error is proportional to the inverse of the field size). It adopts a recursive method to halve circuit size in each round, and delegates the evaluation of the halved circuit to the prover again in the next round. After a logarithmic number of rounds, the circuit size is reduced to minimum, and the verifiers can directly evaluate the circuit to finish checking.

As a result, FLIOP achieves logarithmic proof size and round complexity. Our work will use FLIOP to reduce the verification cost of binary computation.

2.4 Ideal Functionalities

We use the standard security model based on the real/ideal paradigm [9, 23] in this work. We focus on three party setting and consider security with abort in an honest majority against malicious adversaries, i.e., at most one out of the three parties can be corrupted by a malicious adversary.

We rely on some functionalities used as building blocks for our protocols, i.e., $\mathcal{F}_{\text{rand}}$ for generating random shares, $\mathcal{F}_{\text{coin}}$ for generating random coins, and $\mathcal{F}_{\text{input}}$ for secure sharing of inputs. We give the formal definition of these functionalities and their realizations in Appendix A.

3 Maliciously Secure 3PC for Binary Circuits

At a very high level, the overall maliciously secure 3PC protocol works as in [7] by (1) first running the semi-honest protocol based on replicated secret sharing, then (2) checking correctness of the semi-honest computations by invoking a functionality $\mathcal{F}_{\text{vrfy}}$. The functionality is formulated for verifying correctness of messages sent by the parties for computing multiplication gates. In this work, we re-use this functionality from [7] on binary circuits for checking correctness of AND computations conducted by the parties. For completeness, we present the overall protocol from [7] in Appendix B.

Recall that in [7], to instantiate the functionality $\mathcal{F}_{\text{vrfy}}$ for binary circuits, the data involved in semi-honest computations over $\mathbb{F}_2$ is lifted into a binary extension field whose size is large enough (to achieve negligible soundness error) and then directly applied to the FLIOP based distributed zero-knowledge proofs [6]. We observe that this is computationally expensive, and thus try to seek for an efficient way to transform the semi-honest computations over $\mathbb{F}_2$ into prime fields (of comparable size to achieve negligible soundness error). Below we first give a brief review of the semi-honest protocol, then present our verification protocol for instantiating $\mathcal{F}_{\text{vrfy}}$ in detail.

3.1 The Overall Protocol

3.1.1 Review: RSS based Semi-Honest Protocol

Our starting point is the RSS based semi-honest protocol over the binary field $\mathbb{F}_2$ from [3]. We briefly review this protocol in the following.

Linear Gates. Linear gates in binary circuit like XOR and AND-by-a-constant, can be locally computed without interactions utilizing the linearity of the RSS scheme. Specifically, given two sharings $[\![x]\!]^{\mathsf{B}} = (x_0, x_1, x_2)$ and $[\![y]\!]^{\mathsf{B}} = (y_0, y_1, y_2)$ where $x, y \in \mathbb{F}_2$, the parties can obtain a sharing of $z := x \oplus y$ by each party P_i locally computing $(z_i, z_{i-1}) := (x_i \oplus y_i, x_{i-1} \oplus y_{i-1})$. Also, given a sharing $[\![x]\!]^{\mathsf{B}}$ and a constant $c \in \mathbb{F}_2$, the parties can obtain a sharing of $z := c \cdot x$ by each party P_i locally setting $(z_i, z_{i-1}) := (c \cdot x_i, c \cdot x_{i-1})$.

AND Gates. Suppose the parties wish to compute $z := x \cdot y$ given input sharings $[\![x]\!]^{\mathsf{B}}, [\![y]\!]^{\mathsf{B}}$. First observe that, holding sharings (x_i, x_{i-1}) and (y_i, y_{i-1}), each party P_i can locally compute $z_i := x_i y_i \oplus x_i y_{i-1} \oplus x_{i-1} y_i$ s.t. $z = z_0 \oplus z_1 \oplus z_2$, which constructs an additive sharing of z. To turn it into a replicated secret sharing, each party needs to pass around its share to the next party, making sure that everyone owns two additive shares of z. To protect privacy of the inputs, each share z_i should be masked before passing around.

To mask the shares $\{z_i\}_{i=0}^{2}$, the parties can generate an additive sharing of 0, i.e., each party P_i holds a random value $\alpha_i \in \mathbb{F}_2$ s.t. $\alpha_0 \oplus \alpha_1 \oplus \alpha_2 = 0$. Then the parties can use the randomness α_i to mask the share z_i, by re-defining

$$z_i := x_i \cdot y_i \oplus x_i \cdot y_{i-1} \oplus x_{i-1} \cdot y_i \oplus \alpha_i. \tag{1}$$

Clearly the new masked values $\{z_i\}_{i=0}^{2}$ still construct an additive sharing of z, and thus the parties can obtain a replicated sharing of z by each party P_i sending its masked share z_i to party P_{i+1}, and setting (z_i, z_{i-1}) as its share of z.

The random values $\alpha_0, \alpha_1, \alpha_2$ are referred as correlated randomnesses in [3], and can be generated in either an information-theoretically secure or computationally secure way. Here we review the latter approach, which is more concretely efficient in communication.

- Each party P_i picks a random key $\mathsf{K}_i \in \{0,1\}^{\kappa}$ and sends it to P_{i+1}.
- Each party P_i sets $\rho_i := \mathsf{F}_{\mathsf{K}_i}(\mathsf{cnt}), \rho_{i-1} := \mathsf{F}_{\mathsf{K}_{i-1}}(\mathsf{cnt})$, and $\alpha_i := \rho_i \oplus \rho_{i-1}$, where F is an agreed pseudo-random

function from the family $\mathcal{F} = \{\mathrm{F_K} | \mathrm{K} \in \{0,1\}^\kappa, \mathrm{F_K} : \{0,1\}^\kappa \to \mathbb{F}_2\}$, and cnt is an agreed public counter that increments each time.

By using a pseudo-random function F with shared keys, each party P_i can locally obtain two (pseudo-)random values ρ_i, ρ_{i-1}. Clearly their differences $\{\alpha_i\}_{i=0}^2$ construct an additive sharing of 0, and can be used as random masks for computing AND gates.

3.1.2 Achieving Malicious Security

To lift the above semi-honest protocol up to a maliciously secure protocol, prior work [7] has shown that it is sufficient to verify honest behaviours of the parties conducted in the semi-honest computations. They abstract a functionality $\mathcal{F}_{\mathrm{vrfy}}$ to check correctness of messages sent by the parties for computing multiplication gates. When working on binary circuits, $\mathcal{F}_{\mathrm{vrfy}}$ checks correctness of messages of the parties for computing AND gates. We introduce $\mathcal{F}_{\mathrm{vrfy}}$ from [7] in Functionality 3.1.1, and elaborate on how it works on binary circuits in the following.

As reviewed in Section 3.1.1, to compute an AND gate with two input sharings $[\![x]\!]^{\mathrm{B}}, [\![y]\!]^{\mathrm{B}}$, each party P_i holds a randomness $\alpha_i = \rho_i \oplus \rho_{i-1}$, where ρ_i, ρ_{i-1} are generated using a pseudo-random function and shared keys, then P_i computes z_i by Equation 1 and sends the share z_i to P_{i+1}.

We rewrite Equation 1 as

$$z_i = x_i \cdot y_i \oplus x_i \cdot y_{i-1} \oplus x_{i-1} \cdot y_i \oplus \rho_i \oplus \rho_{i-1}. \qquad (2)$$

To verify honest behaviors of P_i, the functionality $\mathcal{F}_{\mathrm{vrfy}}$ needs to check Equation 1 on every AND gate in this circuit.

Given the functionality $\mathcal{F}_{\mathrm{vrfy}}$, we follow [7] to construct the protocol for securely computing binary circuits with abort against malicious adversaries. More specifically, in the verification phase after the semi-honest computation, each party acts as a prover and the other parties act as verifiers to invoke $\mathcal{F}_{\mathrm{vrfy}}$ to check correctness of the messages of the prover. The full description of this protocol appears in Appendix B. We then have the following theorem from [7].

Theorem 3.1.1 ([7]) *Let f be a 3-party functionality computed by a binary circuit $\mathcal{C}$ over $\mathbb{F}_2$. Assume that in Setup step the correlated randomnesses for computing AND gates are generated using pseudo-random functions. Then Protocol B.0.1 computes f with (computational) security with abort in the $(\mathcal{F}_{input}, \mathcal{F}_{vrfy})$-hybrid model in the presence of one malicious party.*

3.2 Instantiating $\mathcal{F}_{\mathrm{vrfy}}$ - Verifying AND Computations

In this section we show our techniques for securely instantiating $\mathcal{F}_{\mathrm{vrfy}}$. Due to page limit, the full description of our protocol is provided in Appendix C.2.1.

Recall that to verify honest behaviours of some party P_i for computing AND gates, the parties need to check Equation 2 for each AND gate in this circuit: where the prover P_i holds input shares $(x_i, x_{i-1}), (y_i, y_{i-1})$, the output share z_i, and (ρ_i, ρ_{i-1}) used to generate correlated randomness.

The key here is that, in the above relation, the data is not exclusively owned by the prover P_i, but shared between the prover and the two verifiers P_{i-1}, P_{i+1}. More specifically, each variable involved in this relation is held by either P_i and P_{i+1} or P_i and P_{i-1}. The two verifiers need to check the relation described in Equation 2 using their own partial data shared with the prover.

Distributed zero-knowledge proofs [6] are naturally fit for this kind of proving tasks where data is shared between a single prover and multiple verifiers. However, the proof systems in [6] require the size of the underlying field to be large enough to achieve negligible soundness error, and are not applicable to relations defined on the binary field $\mathbb{F}_2$. While prior works [6, 7] proposed to lift the data into an extension field to verify relations defined on $\mathbb{F}_2$, the computational cost of such proof systems in extension fields is expensive. Our idea is to transform the verification task to a prime field to reduce the computation overhead of using distributed zero-knowledge proofs.

Step 1: Reduce the Relation. We first consider the case of one AND gate. Starting from Equation 2, we define some new variables as follows. Let

$$\begin{aligned} &a := x_i, c := y_i, e := x_i \cdot y_i \oplus z_i \oplus \rho_i, \\ &b := y_{i-1}, d := x_{i-1}, f := \rho_{i-1}. \end{aligned} \qquad (3)$$

Note that a, c, e are known by both P_i and P_{i+1}, and b, d, f are known by both P_i and P_{i-1}. Now the verification of Equation 2 is reduced to check

$$a \cdot b \oplus c \cdot d \oplus e \oplus f = 0. \qquad (4)$$

Step 2: Transform to Prime Fields. We next transform the relation in Equation 4 from the binary field $\mathbb{F}_2$ to a prime field $\mathbb{F}_p$, rather than lifting it into an extension field. This is mainly motivated by the fact that computing multiplications over an extension field is much more expensive than that over a prime field with comparable size, since the latter only involves integer computations with modular operations (see more discussion in Section E.1).

The insight is to view each value in $\mathbb{F}_2$ as a value in $\mathbb{F}_p$ and then use field operations to simulate XOR and multiplications over $\mathbb{F}_2$. A key formula we utilize here is that, for any $a, b \in \{0,1\}$ and $p \geq 3$, $a \oplus b = a + b - 2ab \bmod p = a(1-2b) +$

FUNCTIONALITY 3.1.1 *($\mathcal{F}_{vrfy}$ - Verifying AND Computations).*

Let $\mathcal{S}$ be the ideal world adversary and P_i the corrupted party controlled by $\mathcal{S}$. $\mathcal{F}_{\text{vrfy}}$ receives an index j of the prover party and a parameter $m \in \mathbb{F}$ from the honest parties. Then

- If the prover party P_j is honest:
  - $\mathcal{F}_{\text{vrfy}}$ receives from P_j its input $(x_j^{(\ell)}, x_{j-1}^{(\ell)}, y_j^{(\ell)}, y_{j-1}^{(\ell)}, \rho_j^{(\ell)}, \rho_{j-1}^{(\ell)}, z_j^{(\ell)})$ for $\ell \in [m]$.
  - If $i = j+1$, then $\mathcal{F}_{\text{vrfy}}$ hands $\mathcal{S}$ the index j, the input of P_i, i.e., $(x_j^{(\ell)}, y_j^{(\ell)}, \rho_j^{(\ell)}, z_j^{(\ell)})$ for $\ell \in [m]$.
  - If $i = j-1$, then $\mathcal{F}_{\text{vrfy}}$ hands $\mathcal{S}$ the index j, the input of P_i, i.e., $(x_{j-1}^{(\ell)}, y_{j-1}^{(\ell)}, \rho_{j-1}^{(\ell)})$ for $\ell \in [m]$.
  - $\mathcal{F}_{\text{vrfy}}$ receives a command abort or accept from $\mathcal{S}$, and then hands the command to the honest parties.
- If the prover party P_j is corrupted (i.e., $i = j$):
  - $\mathcal{F}_{\text{vrfy}}$ receives $(x_i^{(\ell)}, y_i^{(\ell)}, \rho_i^{(\ell)}, z_i^{(\ell)})$ from P_{i+1} and $(x_{i-1}^{(\ell)}, y_{i-1}^{(\ell)}, \rho_{i-1}^{(\ell)})$ from P_{i-1} for each $\ell \in [m]$.
  - $\mathcal{F}_{\text{vrfy}}$ hands $\mathcal{S}$ the index i, the input of P_i, i.e., $(x_i^{(\ell)}, x_{i-1}^{(\ell)}, y_i^{(\ell)}, y_{i-1}^{(\ell)}, \rho_i^{(\ell)}, \rho_{i-1}^{(\ell)}, z_i^{(\ell)})$ for $\ell \in [m]$.
  - $\mathcal{F}_{\text{vrfy}}$ checks if for each $\ell \in [m]$, $z_i^{(\ell)} = x_i^{(\ell)} \cdot y_i^{(\ell)} + x_i^{(\ell)} \cdot y_{i-1}^{(\ell)} + x_{i-1}^{(\ell)} \cdot y_i^{(\ell)} + \rho_i^{(\ell)} - \rho_{i-1}^{(\ell)}$. If the equation doesn't hold for any $\ell \in [m]$, $\mathcal{F}_{\text{vrfy}}$ sends abort to the parties; otherwise it receives a command abort or accept from $\mathcal{S}$, hands the command to the honest parties, and then halts.

$b \bmod p$. Then for Equation 4, we have

$$\begin{aligned}
& a \cdot b \oplus c \cdot d \oplus e \oplus f \\
= \; & (a \cdot b \oplus e) \oplus (c \cdot d \oplus f) \\
= \; & (a \cdot b \cdot (1-2e) + e) \oplus (c \cdot d \cdot (1-2f) + f) \\
= \; & (a \cdot b \cdot (1-2e) + e) + (c \cdot d \cdot (1-2f) + f) \\
& -2(a \cdot b \cdot (1-2e) + e) \cdot (c \cdot d \cdot (1-2f) + f) \bmod p \\
= \; & -2(a \cdot c \cdot (1-2e)) \cdot (b \cdot d \cdot (1-2f)) \\
& +(c \cdot (1-2e)) \cdot (d \cdot (1-2f)) + (a \cdot (1-2e)) \cdot \\
& (b \cdot (1-2f)) - \frac{1}{2}((1-2e) \cdot (1-2f)) + \frac{1}{2} \bmod p,
\end{aligned}$$

where $\frac{1}{2}$ denotes the inverse of 2 in $\mathbb{F}_p$.

Now the relation has been transformed to the prime field $\mathbb{F}_p$. In the following we omit "mod p" for simplicity. Similarly to Step 1, we reduce the relation again by categorizing the variables and merging some immediate results that can be computed locally by each party. Recall that a, c, e are known by both P_i, P_{i+1}, and b, d, f are known by both P_i, P_{i-1}.

- P_i, P_{i+1} locally compute $g_1 := -2a \cdot c \cdot (1-2e), g_2 := c \cdot (1-2e), g_3 := a \cdot (1-2e), g_4 := -(1-2e)/2$ in $\mathbb{F}_p$.
- P_i, P_{i-1} locally compute $h_1 := b \cdot d \cdot (1-2f), h_2 := d \cdot (1-2f), h_3 := b \cdot (1-2f), h_4 := 1-2f$ in $\mathbb{F}_p$.

Now the relation we want to verify becomes

$$\sum_{k=1}^{4} g_k \cdot h_k + 1/2 = 0, \qquad (5)$$

which is a length-4 inner product relation defined on the prime field $\mathbb{F}_p$.

Step 3: Batch Verifying Multiple AND Gates. Up to now we have transformed the verification of one AND computation of P_i from the binary field to the prime field $\mathbb{F}_p$. (In our experiment, we choose to use $p = 2^{61} - 1$.) Starting from this point, we can verify correctness of P_i's behaviors by directly applying the FLIOP based distributed zero-knowledge proof system [6] to all AND computation it conducted when evaluating the whole circuit.

However, in [6, 7], the first step is to transform the m inner-product relations to a single length-$4m$ inner-product relation by multiplying random coefficients. This is to reduce the check to a single inner-product relation and the random coefficients guarantee that if one of the original inner-product relation is incorrect, then the resulting inner-product relation is also incorrect with overwhelming probability. We note that in our case, multiplying random coefficients is not necessary. This is because if for some AND triple, Equation 2 does not hold (meaning that P_i deviates from the protocol when computing this AND gate), then $\sum_{k=1}^{4} g_k \cdot h_k + 1/2 = a \cdot b \oplus c \cdot d \oplus e \oplus f = 1$ by construction. Now suppose we have m AND triples to verify, say,

$$\{[\![x^{(\ell)}]\!]^{\mathrm{B}}, [\![y^{(\ell)}]\!]^{\mathrm{B}}, [\![z^{(\ell)}]\!]^{\mathrm{B}}\}_{\ell=1}^{m}.$$

For all $\ell \in [m]$, if P_i follows the protocol, then $\sum_{k=1}^{4} g_k^{(\ell)} \cdot h_k^{(\ell)} + 1/2 = 0$. Otherwise, $\sum_{k=1}^{4} g_k^{(\ell)} \cdot h_k^{(\ell)} + 1/2 = 1$. Now consider the summation in $\mathbb{F}_p$,

$$\sum_{\ell=1}^{m} (\sum_{k=1}^{4} g_k^{(\ell)} \cdot h_k^{(\ell)} + 1/2),$$

which is equal to the number of AND gates where P_i does not follow the protocol modulo p. When $m < p$, if there is one AND gate where P_i does not follow the protocol, then $\sum_{\ell=1}^{m}(\sum_{k=1}^{4} g_k^{(\ell)} \cdot h_k^{(\ell)} + 1/2) \neq 0$, which is what we need. Note that the same trick does not work for the binary extension field since the addition over $\mathbb{F}_{2^\kappa}$ is bit-wise XOR and the errors will be cancelled out as long as a even number of AND triples are incorrect.

Thus, we will choose a prime field $\mathbb{F}_p$ such that $p > m$ (The choice of p also depends on the desired soundness error. See Theorem 3.2.1) and simply verify that

$$\sum_{\ell=1}^{m}(\sum_{k=1}^{4} g_k^{(\ell)} \cdot h_k^{(\ell)} + 1/2) = 0. \tag{6}$$

Now we rearrange the variables and define two vectors as follows.

- $\vec{u} := (g_1^{(1)}, g_2^{(1)}, g_3^{(1)}, g_4^{(1)}, \cdots\cdots, g_1^{(m)}, g_2^{(m)}, g_3^{(m)}, g_4^{(m)})$.
- $\vec{v} := (h_1^{(1)}, h_2^{(1)}, h_3^{(1)}, h_4^{(1)}, \cdots\cdots, h_1^{(m)}, h_2^{(m)}, h_3^{(m)}, h_4^{(m)})$.

Both the two vectors are of length $4m$. And Equation 6 now turns to

$$\sum_{k=1}^{4m} u_k \cdot v_k = -\frac{m}{2}. \tag{7}$$

Rather than verifying m length-4 inner-product relations, we have turned to verify one length-$4m$ inner-product relation over $\mathbb{F}_p$.

In Step 4, we directly follow the FLIOP based distributed zero-knowledge proofs [6] to let the parties check the single one batched inner-product relation over $\mathbb{F}_p$. We slightly extend the techniques [6] to allow more general compression parameters. Due to page limit, we defer this step to Appendix C.1. We have the following theorem.

Theorem 3.2.1 *Assume that the prime field $\mathbb{F}_p$ is sufficiently large, and in Setup step the randomnesses are generated using pseudo-random functions. Then Protocol C.2.1 computes $\mathcal{F}_{vrfy}$ with (computational) security with abort in the presence of one malicious party in the $\mathcal{F}_{coin}$-hybrid model, with statistical error bounded by $\frac{2\lambda(\log_\lambda m+1)+1}{p-\lambda-1}$.*

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

4 Optimizations: Saving Computational Cost

In this section, we show how our protocol can take advantage of the algebraic structure of prime fields to improve the computation complexity in the distributed zero-knowledge proof technique [6,7]. We note that the most expensive step of this technique is in its first round of the recursion, where given a compression parameter λ (which is usually a small constant), we reduce the length of the inner-product relation from $4m$ to $4m/\lambda$. We first review the computation task in the first round of the recursion of the distributed zero-knowledge proof construction [6,7]. We refer the readers to Appendix C.1 for the overview of the whole technique in [6,7].

4.1 Computation Task of First Round of Recursion

An Overview of First Round of Recursion. Recall that we want to verify the inner-product $\langle \vec{u}, \vec{v} \rangle = -\frac{m}{2}$, where m is the number of AND gates, and $\vec{u}, \vec{v}$ are vectors of size $4m$. In particular, $\vec{u}$ is known by P_i, P_{i+1} and $\vec{v}$ is known by P_i, P_{i-1}

To reduce the length of the inner-product relation, we first separate $\vec{u}, \vec{v}$ into λ sub-vectors of size $4m/\lambda$:

$$\vec{u} = (\vec{u}_1, \ldots, \vec{u}_\lambda), \vec{v} = (\vec{v}_1, \ldots, \vec{v}_\lambda).$$

(Note that here the indices $\{1, \cdots, \lambda\}$ are pointed to the sub-vectors, rather than vector elements.) Then the prover P_i defines two vectors of degree-$(\lambda - 1)$ polynomials $\vec{p}(X), \vec{q}(X)$ such that $\vec{p}(i) = \vec{u}_i, \vec{q}(i) = \vec{v}_i$ for all $i \in [\lambda]$. Next P_i computes a degree-$2(\lambda - 1)$ polynomial $G(X) = \langle \vec{p}(X), \vec{q}(X) \rangle$. Note that $\langle \vec{u}, \vec{v} \rangle = \sum_{i=1}^{\lambda} G(i)$. Now P_i uses RSS to share the coefficients of $G(X)$ to the other two parties. Note that P_{i+1} can locally compute $\vec{p}(X)$ while P_{i-1} can locally compute $\vec{q}(X)$. So the verification check becomes to check:

- $G(X) = \langle \vec{p}(X), \vec{q}(X) \rangle$,
- $\sum_{i=1}^{\lambda} G(i) = -\frac{m}{2}$.

For the first item, according to the Schwartz–Zippel Lemma, it is sufficient to sample a random evaluation point r and check whether $G(r) = \langle \vec{p}(r), \vec{q}(r) \rangle$. Observe that this reduces the check to an inner-product relation of size $4m/\lambda$ since $\vec{p}(r)$ and $\vec{q}(r)$ are two vectors of size $4m/\lambda$.

For the second item, all parties can locally compute a RSS of $\sum_{i=1}^{\lambda} G(i) + \frac{m}{2}$ and then check whether it is a RSS of 0.

Computation Task of First Round of the Recursion. The most expensive task in Step 1 is to compute the coefficients of $G(X)$. A straightforward way would be first computing $\vec{p}(X), \vec{q}(X)$ and then computing $G(X) = \langle \vec{p}(X), \vec{q}(X) \rangle$. However, interpolating a degree-$(\lambda - 1)$ polynomial requires around λ^2 multiplication operations. The computation complexity of computing each of $\vec{p}(X), \vec{q}(X)$ is about $4m/\lambda \cdot \lambda^2 = 4m\lambda$. As a result, computing $G(X)$ would require $8m\lambda$ multiplication operations[3].

We note that a degree-$2(\lambda - 1)$ polynomial can be fully determined by $2\lambda - 1$ evaluation points. Thus, instead of sharing coefficients of $G(X)$, we change to share $\{G(i)\}_{i=1}^{2\lambda-1}$, which is sufficient for parties to compute $G(r)$ at a later point. For every $i \in [2\lambda - 1]$, $G(i) = \langle \vec{p}(i), \vec{q}(i) \rangle$. We note that by the

[3] We do not consider to use FFT for interpolation since (1) λ is usually very small and (2) our experiment uses $p = 2^{61} - 1$ for the field size and there is no suitable sub-group in $\mathbb{F}_{2^{61}-1}$ to use FFT.

property of polynomials, $\vec{p}(i)$ can be expressed as a linear combination of $\{\vec{p}(j) = \vec{u}_j\}_{j=1}^{\lambda}$. Similarly, $\vec{q}(i)$ can be expressed as a linear combination of $\{\vec{q}(j) = \vec{v}_j\}_{j=1}^{\lambda}$. Thus, $G(i)$ can be computed as a linear combination of

$$\{\langle \vec{p}(j_1), \vec{q}(j_2)\rangle\}_{j_1,j_2\in[\lambda]} = \{\langle \vec{u}_{j_1}, \vec{v}_{j_2}\rangle\}_{j_1,j_2\in[\lambda]}.$$

Here $j_1, j_2 \in [\lambda]$ are indices of the sub-vectors. The main observation is that, computing $\{\langle \vec{u}_{j_1}, \vec{v}_{j_2}\rangle\}_{j_1,j_2\in[\lambda]}$ only requires $4m\lambda$ multiplication operations, which saves the computation complexity by a factor of 2. Thus, our implementation uses this idea to compute $G(X)$. And our optimization highly relies on this observation.

In summary, the main computation task in the first round is to compute $\{\langle \vec{u}_{j_1}, \vec{v}_{j_2}\rangle\}_{j_1,j_2\in[\lambda]}$.

4.2 Main Optimizations

1. Computing $\langle \vec{u}_{j_1}, \vec{v}_{j_2}\rangle$ Using Binary Operations. A straightforward way of computing $\{\langle \vec{u}_{j_1}, \vec{v}_{j_2}\rangle\}_{j_1,j_2\in[\lambda]}$ is to transform the AND triples to the sub-vectors $\{\vec{u}_j, \vec{v}_j\}_{j\in[\lambda]}$ over $\mathbb{F}_p$ and compute each inner-product $\langle \vec{u}_{j_1}, \vec{v}_{j_2}\rangle$ of these sub-vectors. However, during the transformation, for each AND triple, we transform from checking $a \cdot b \oplus c \cdot d \oplus e \oplus f = 0$, where each value is a single bit, to checking $\sum_{k=1}^{4} g_k \cdot h_k + 1/2 = 0$, where each value is an element in $\mathbb{F}_p$. Given that we have to deal with a large amount of AND triples, the blowing up in the storage leads to a large overhead in the running time.

Our first improvement is to compute $\langle \vec{u}_{j_1}, \vec{v}_{j_2}\rangle$ using the original data without transforming them to $\vec{u}_{j_1}, \vec{v}_{j_2}$ over $\mathbb{F}_p$ explicitly. For simplicity, we set $\lambda = 4$. The following idea naturally extends to the case where λ is a multiple of 4. Now suppose the AND triples we need to verify are denoted by

$$\{[\![x^{(i)}]\!]^{\mathrm{B}}, [\![y^{(i)}]\!]^{\mathrm{B}}, [\![z^{(i)}]\!]^{\mathrm{B}}\}_{i=1}^{m}.$$

For all $i \in [m]$, we want to verify

$$a^{(i)} \cdot b^{(i)} \oplus c^{(i)} \cdot d^{(i)} \oplus e^{(i)} \oplus f^{(i)} = \sum_{k=1}^{4} g_k^{(i)} \cdot h_k^{(i)} + \frac{1}{2} = 0.$$

In the case of $\lambda = 4$, we set $\vec{u}_j = (g_j^{(i)})_{i=1}^m$ and $\vec{v}_j = (h_j^{(i)})_{i=1}^m$ for $j \in [4]$.

We observe that $\langle \vec{u}_{j_1}, \vec{v}_{j_2}\rangle = \sum_{i=1}^{m} g_{j_1}^{(i)} \cdot h_{j_2}^{(i)}$. Recall that each of $g_{j_1}^{(i)}, h_{j_2}^{(i)}$ is computed from $a^{(i)}, b^{(i)}, \ldots, f^{(i)}$. Thus, we may use the original data to compute each $g_{j_1}^{(i)} \cdot h_{j_2}^{(i)}$. This allows us to save the cost of transforming bit values to field elements.

2. Batching Computation for $g_{j_1}^{(i)} \cdot h_{j_2}^{(i)}$. Naturally, a batch of 64 AND triples are stored using the `uint64` datatype. To be more concrete, for every 64 AND triples, a value $\boldsymbol{a} \in \mathbb{Z}_{2^{64}}$ is used to store $a^{(1)}, \ldots, a^{(64)}$, and similar for $\boldsymbol{b}, \boldsymbol{c}, \boldsymbol{d}, \boldsymbol{e}, \boldsymbol{f}$.

Our second improvement is to batch the computation of $g_{j_1}^{(i)} \cdot h_{j_2}^{(i)}$ using $\boldsymbol{a}, \boldsymbol{b}, \ldots, \boldsymbol{f}$. We take $j_1 = j_2 = 1$ as an example. Other cases follow naturally. When $j_1 = j_2 = 1$, we have $g_1^{(i)} = -2a^{(i)}c^{(i)}(1 - 2e^{(i)})$ and $h_1^{(i)} = b^{(i)}d^{(i)}(1 - 2f^{(i)})$. Then

$$\begin{aligned} g_1^{(i)} \cdot h_1^{(i)} &= -2a^{(i)}b^{(i)}c^{(i)}d^{(i)} + 4a^{(i)}b^{(i)}c^{(i)}d^{(i)}e^{(i)} \\ &\quad + 4a^{(i)}b^{(i)}c^{(i)}d^{(i)}f^{(i)} - 8a^{(i)}b^{(i)}c^{(i)}d^{(i)}e^{(i)}f^{(i)}. \end{aligned}$$

Observe that we can compute $\boldsymbol{abcd}$ via bit-wise AND operations and then split the result bit by bit. Similarly, we first compute $\boldsymbol{abcde}, \boldsymbol{abcdf}, \boldsymbol{abcdef}$ and then split the results bit by bit. This improvement allows us to speed up the computation of each $g_{j_1}^{(i)} \cdot h_{j_2}^{(i)}$.

We note that a similar improvement also applies to [7] except that after splitting the result bit by bit, they also need to multiply each bit by a random coefficient in $\mathbb{F}_{2^{64}}$. Our experiment has implemented this improvement in [7].

3. Further Optimization with Carefully Chosen Coefficients. Recall that we do not have to multiply random coefficients when transforming the m inner-product relations to a single length-$4m$ inner-product relation. Instead, we simply sum up all m inner-product relations.

When computing $\langle \vec{u}_{j_1}, \vec{v}_{j_2}\rangle = \sum_{i=1}^{m} g_{j_1}^{(i)} \cdot h_{j_2}^{(i)}$, while we can compute $\{g_{j_1}^{(i)} \cdot h_{j_2}^{(i)}\}_{i=1}^m$ in a batch way, we have to split the result bit by bit and then add all bits together.

Our third improvement is to multiply a suitable constant with each of the m inner-product relation so that we can avoid bit splitting. Again, we take $j_1 = j_2 = 1$ as an example and focus on the first 32 AND triples. In this case, we want to compute $\sum_{i=1}^{32} g_1^{(i)} \cdot h_1^{(i)}$. This requires us to compute $\sum_{i=1}^{32} a^{(i)}b^{(i)}c^{(i)}d^{(i)}$. Then after computing $\boldsymbol{abcd}$, we have to compute the summation of the least-significant 32 bits of $\boldsymbol{abcd}$. We observe that, if our goal becomes to compute $\sum_{i=1}^{32} 2^{i-1}a^{(i)}b^{(i)}c^{(i)}d^{(i)}$, then we can simply take the least-significant 32 bit as the result!

To make this idea work, instead of checking $\sum_{i=1}^{m}(\sum_{k=1}^{4} g_k^{(i)} \cdot h_k^{(i)} + 1/2) = 0$, we change to check

$$\sum_{i=1}^{m} 2^{(i-1) \bmod 32}\left(\sum_{k=1}^{4} g_k^{(i)} \cdot h_k^{(i)} + 1/2\right) = 0.$$

In addition, we have to ensure that the accumulated error is bounded by the field size p so that if $\sum_{k=1}^{4} g_k^{(i)} \cdot h_k^{(i)} + 1/2 \neq 0 \bmod p$ for some i, then $\sum_{i=1}^{m} 2^{(i-1) \bmod 32}(\sum_{k=1}^{4} g_k^{(i)} \cdot h_k^{(i)} + 1/2) \neq 0 \bmod p$. Recall that each $\sum_{k=1}^{4} g_k^{(i)} \cdot h_k^{(i)} + 1/2$ is either 0 or 1. Thus

$$\sum_{i=1}^{m} 2^{(i-1) \bmod 32}\left(\sum_{k=1}^{4} g_k^{(i)} \cdot h_k^{(i)} + 1/2\right) \leq \sum_{i=1}^{m} 2^{(i-1) \bmod 32} < 2^{27}m.$$

In our experiment, we choose $p = 2^{61} - 1$. It is sufficient to verify 2^{33} AND gates.

We remark that this improvement is exclusive for our approach and does not work for [7].

4.3 Other Optimizations

Using Mersenne Fields for Fast Arithmetic Computation. Our experiment chooses to use $\mathbb{F}_p$ where $p = 2^{61} - 1$, which is known as a Mersenne prime. The major bottleneck of doing arithmetic operations over $\mathbb{F}_p$ is to do modular operations. By using a Mersenne prime, we can speed up the modular operations by using left/right shifts and additions. In Section E.1, we show that multiplication operations over $\mathbb{F}_{2^{61}-1}$ are about 7.6× faster than those over $\mathbb{F}_{2^{64}}$.

Speeding Up Inner-product Operations. The distributed zero-knowledge proofs [6,7] require to do many inner-product operations locally. To speed up inner-product operations, we can first compute the integer result and then apply the modular operation of p once at the end. Similar tricks also apply when we need to compute addition of many values: we first compute the integer result and only apply once the modular operation of $p = 2^{61} - 1$.

A similar optimization also works for inner-product operations over $\mathbb{F}_{2^{64}}$. This is because an multiplication over $\mathbb{F}_{2^{64}}$ corresponds to multiplying two degree-63 polynomials and then applying a modular operation of a degree-64 irreducible polynomial. For inner-product operations, it suffices to only apply the modular operation at the end. Our experiment has taken this optimization for [7] into consideration.

Lookup Table for Computing $\vec{p}(r), \vec{q}(r)$. After resolving the computation of the coefficients of $G(X)$, the computation bottleneck becomes to computing $\vec{p}(r), \vec{q}(r)$. Recall that $\vec{p}(X), \vec{q}(X)$ are vectors of degree-$(\lambda - 1)$ polynomials and the vector size is $4m/\lambda$. To demonstrate our idea, we take $\lambda = 4$ as an example and the same idea generalizes to the case where λ is a multiple of 4. In this case, $\vec{p}(X), \vec{q}(X)$ are of size $4m/\lambda = m$.

Let $\vec{p}(X) = (p_1(X), \ldots, p_m(X))$. Then the goal is to compute $p_i(r)$ for all $i \in [m]$. For each $p_i(r)$, it can be expressed as a fixed linear combination of $\{p_i(k)\}_{k=1}^{4} = \{g_k^{(i)}\}_{k=1}^{4}$. We note that $\{g_k^{(i)}\}_{k=1}^{4}$ are determined by $\{a^{(i)}, c^{(i)}, e^{(i)}\}$. Then there exists a map $M : \{0,1\}^3 \to \mathbb{F}_p$ such that $M(a^{(i)}, b^{(i)}, c^{(i)}) = p_i(r)$ for all $i \in [m]$. Thus, we establish a lookup table of size $2^3 = 8$ for M and each time $p_i(r)$ is computed by checking the lookup table with input $a^{(i)}, b^{(i)}, c^{(i)}$.

In general, when $\lambda \geq 4$, the lookup table is of size $2^{3\lambda/4}$. In our experiment, we choose $\lambda = 32$ for the first round of the recursion (to maximally take advantage of the fast computation for the first round due to the above optimizations). In this case the lookup table would have size 2^{24}. To reduce the size of the lookup table, we construct two lookup tables, one for the linear combinations of $\{p_i(k)\}_{k=1}^{16}$, and one for the linear combinations of $\{p_i(k)\}_{k=17}^{32}$. In this way, each lookup table is of size 2^{12} and $p_i(r)$ can be computed by summing up the results from the two lookup tables.

A similar optimization also works for [7], for a fair comparison, our experiment has taken this optimization for [7] into consideration.

4.4 Parameter Setup

Instead of checking all AND triples at the end of the protocol, we divide the AND triples into batches and apply our verification protocol for each batch in parallel. We note that the batch size slightly affects the running time and we choose to use the batch size $\texttt{bs} = 640,000$ to achieve the best possible running time for our protocol.

Since the first round computation has been significantly improved, our protocol uses the compression parameter $\lambda = 32$ in the first round and $\lambda' = 8$ for the rest of rounds. In this case, by following the same argument as that for Theorem 3.2.1, the soundness error achieved in our protocol is

$$\frac{2\lambda}{p-\lambda} + \frac{2\lambda'(\log_{\lambda'}(\texttt{bs}/\lambda)+1)+1}{p-\lambda'-1} \leq 2^{-53},$$

which is sufficiently small for most of applications.

5 Application: Secure DNN Inference

Privacy-preserving machine learning (PPML) has been an increasingly popular line of research for guaranteeing privacy of sensitive data involved in machine learning tasks. Plenty of works [12, 16, 17, 32, 35, 36, 38, 43, 45, 46] have worked on building secure computation protocols over fixed-point arithmetic for machine learning primitives, including linear functions such as convolution, matrix multiplication, and non-linear functions such as ReLU, Max Pooling, Sigmoid, etc. These in turn are built on lower-level primitives, such as multiplication, dot product, truncation and comparison, etc.

A generic approach of supporting non-linear operations like truncation and comparison is to design specific protocols based on binary computation. Our protocol can be used as an efficient primitive to improve this part. In the following, we demonstrate why the non-linear operations are so costly in the current state-of-the-art. We only focus on the setting of 3-party computation with honest majority.

5.1 Fixed-point Arithmetic

At a high level, the idea of fixed-point arithmetic is to simulate the computation over real numbers on finite rings/fields. In the fixed-point representation, a k-bit signed fixed-point $\widetilde{x}$ is composed of two parts, an e-bit integer part and a d-bit decimal part s.t. $k = e + d$. We can map the fixed-point number $\widetilde{x}$ into an element x in ring $\mathbb{Z}_{2^k}$ by setting $x := 2^d \cdot \widetilde{x}$.

The addition and multiplication operations over fixed-point numbers can be mapped to the ones over ring elements. Note that after multiplying two fixed-point numbers, the length of the decimal part doubles. To continue the computation, the multiplication result requires a proper adjustment to the decimal part by truncating the least-significant d bits so that the decimal part remains d bits.

5.1.1 Comparison

Comparison is a basic primitive in machine learning tasks. The activation function ReLU and Max Pooling layers in neural networks all can be decomposed into comparison operations. As we will discuss below, exact truncation protocols also require secure comparisons.

A secure comparison protocol over ring $\mathbb{Z}_{2^k}$ takes as input two sharings $[\![x]\!]^{\mathrm{R}}$, $[\![y]\!]^{\mathrm{R}}$, and outputs $[\![z]\!]^{\mathrm{R}}$ where $z = 0$ when $x \geq y$ and $z = 1$ otherwise. In the literature, secure comparison is usually transformed to the MSB (Most Significant Bit) extraction problem. Shares are first transformed from the arithmetic domain to the binary field; then the parties securely compute some designed binary circuits (such as full adder or parallel prefix adder) to extract the MSB of the difference of the two secrets. In current state-of-the-art solutions [32, 35], a secure comparison of two elements in $\mathbb{Z}_{2^k}$ requires computing a binary circuit of $3k$ AND gates.

5.1.2 Truncation

A truncation protocol over rings takes as input a sharing $[\![x]\!]^{\mathrm{R}}$ of integer $x \in \mathbb{Z}_{2^k}$, and outputs a sharing of x' where $x' = [x/2^d]$, and d is bit length of the decimal part in the fixed-point representation.

Truncation operations are intensively used in a secure computation protocol for Machine Learning since it is invoked after *every* arithmetic multiplication operation or arithmetic dot product operation. There are two trends in designing protocols for truncation: achieving the exact truncation result or a probabilistic truncation result with one-bit error.

Exact Truncation. An exact truncation protocol always rounds $x/2^d$ down and abandons the least d significant bits of x, i.e., $x' = \lfloor x/2^d \rfloor$. Achieving the exact truncation requires to compute comparison [11, 18].

Probabilistic Truncation. A probabilistic truncation protocol rounds $x/2^d$ to $x' = \lfloor x/2^d \rfloor + w$ where $w = 1$ with probability $\gamma := \frac{x \bmod 2^d}{2^d}$ and $w = 0$ with probability $1 - \gamma$. Here γ is the distance between the decimal part of $x/2^d$ and 0. While the starting point of using probabilistic truncation is to avoid secure comparisons required for exact truncation, we notice that the known efficient protocols for probabilistic truncation based on [11] (including but not limited to protocols in [17, 18, 32, 35]) do *not* securely compute the desired functionality. As a result, we have to use the expensive exact truncation, and this greatly increasing the number of AND gates. In Table 1, we use the MP-SPDZ [28] framework to compile three neural networks with the option `trunc_pr` enabled and disabled respectively, to count the amount of binary AND gates needed when using probabilistic truncation and exact truncation in the two cases. As we can see, exact truncation requires much more AND gates than probabilistic truncation.

	# of AND Gates with Probabilistic Trunc.	# of AND Gates with Exact Trunc.
ResNet-50	1,220,520,760	4,339,682,635
DenseNet	1,986,520,005	5,511,885,885
SqeezeNet	615,161,565	1,026,065,325

Table 1: Number of AND gates in ResNet-50, DenseNet, and SqeezeNet, with probabilistic truncation and exact truncation.

In the following, we formally describe this security issue. We show that this security issue even appears in the semi-honest security.

5.2 On the Security Issue of Probabilistic Truncation Protocols

We first review the definition of multiparty computation for semi-honest security from [9].

Security Definition. Let Π be an n-party protocol and let $\mathtt{F} : \{0,1\}^{I_1} \times \ldots \times \{0,1\}^{I_n} \to \{0,1\}^{O_1} \times \ldots \times \{0,1\}^{O_n}$ be an n-ary function. We consider static semi-honest security against t corrupted parties. Let $\mathcal{C}$ denote the set of corrupted parties, and $\mathcal{H}$ the set of honest parties. Let $\vec{x} = (x_1, \ldots, x_n)$ denote the inputs of all parties.

We say a protocol Π is t-private for $\mathcal{F}$ with error ε if there exists an ideal adversary $\mathcal{S}$ s.t. for all set $\mathcal{C}$ of at most t corrupted parties and for all inputs $\vec{x}$, the following two distributions are statistically close with distance ε

$$(\mathtt{Output}^{\Pi}(\vec{x}), \mathtt{View}_{\mathcal{C}}(\vec{x})) \approx_{\varepsilon} (\mathcal{F}(\vec{x}), \mathcal{S}(\mathcal{C}, \vec{x}_{\mathcal{C}}, \mathcal{F}_{\mathcal{C}}(\vec{x}))).$$

Here $\mathtt{Output}^{\Pi}(\vec{x})$ denotes the protocol outputs of all parties when the inputs are $\vec{x}$, $\mathtt{View}_{\mathcal{C}}(\vec{x})$ denotes the views of corrupted parties when the inputs are $\vec{x}$, and $\vec{x}_{\mathcal{C}}$ denotes the sub-vector $(x_i)_{P_i \in \mathcal{C}}$, $\mathcal{F}_{\mathcal{C}}(\vec{x})$ denotes the function outputs of corrupted parties.

To simplify the expression, we adapt the hybrid model defined in [9]. We refer the readers to [9] for more details about this model.

Overview of the Approach in [11]. We give an overview of the approach by Catrina and Hoogh. We consider a general linear secret sharing scheme Σ over $\mathbb{Z}_{2^k}$ and use $[x]$ to denote a Σ-sharing of x. In the beginning, we assume all parties hold a secret sharing of x, denoted by $[x]$. We further assume that x is much smaller than 2^k in the sense that $x < 2^{k-\tau}$ where τ is the security parameter.

Suppose the goal is to truncate the least-significant d bits of x. Let x^d denote the value after truncating the least-significant d bits of x. Then the goal is to compute a secret sharing $[x^d]$. We describe the approach in [11] in the $\mathcal{F}_{\mathrm{truncPair}}$-hybrid model in Protocol 1. Concretely $\mathcal{F}_{\mathrm{truncPair}}$ samples a random r, generates $([r],[r^d])$, and distributes the shares to all parties.

Protocol 1: TRUNCPR

1. All parties hold $[x]$ with the guarantee that $x < 2^{k-\tau}$.
2. All parties invoke $\mathcal{F}_{\mathrm{truncPair}}$ and receive $([r],[r^d])$.
3. All parties locally compute $[x+r] = [x]+[r]$ and reconstruct $x+r$ to P_1.
4. P_1 computes and sends $(x+r)^d$ to all parties.
5. All parties locally compute $[x'] = (x+r)^d - [r^d]$.

Let x_d denote the least-significant d bits of x. In the case that $x+r < 2^k$, which happens with probability $x/2^k < 2^{-\tau}$, the resulting sharing of the above protocol satisfies that, with probability $x_d/2^d$, $x' = x^d+1$, and with probability $1-x_d/2^d$, $x' = x^d$. To see why this is the case, note that when $x+r < 2^k$,

- If $x_d + r_d < 2^d$, then $x+r = x_d+r_d+2^d(x^d+r^d)$. Thus $(x+r)^d = x^d+r^d$, and $x' = (x+r)^d - r^d = x^d$.
- If $x_d+r_d \geq 2^d$, then $x+r = (x_d+r_d-2^d)+2^d(x^d+r^d+1)$. Thus $(x+r)^d = x^d+r^d+1$, and $x' = (x+r)^d - r^d = x^d+1$.

At a first glance, the value that is revealed to P_1 is $x+r$, which is masked by a random value r. Thus, P_1 learns no information about the secret x. However, for a fixed input $[x]$, whether the output x' is rounding up or down is solely determined by r since this is the only randomness introduced in the protocol. This means that x' is fixed given the view of P_1. On the other hand, for semi-honest security, we would expect that x' is truncated by an ideal functionality which should be independent of the view of P_1.

Security Issue. To demonstrate the security issue of the approach in [11], we first need to define the ideal functionality for it. Consider $\mathcal{F}_{\mathrm{truncPr}}$ in Functionality 2.

Functionality 2: $\mathcal{F}_{\mathrm{truncPr}}$

1. $\mathcal{F}_{\mathrm{truncPr}}$ receives the shares of $[x]$ from all parties.
2. $\mathcal{F}_{\mathrm{truncPr}}$ reconstructs x and define x_d to be the least-significant d bits of x, and x^d to be the result after truncating the least-significant d bits of x. Then $\mathcal{F}_{\mathrm{truncPr}}$ randomly samples x' s.t.
   - With probability $x_d/2^d$, $x' = x^d+1$.
   - With probability $1-x_d/2^d$, $x' = x^d$.
3. $\mathcal{F}_{\mathrm{truncPr}}$ generates a random Σ-sharing of x' and distributes $[x']$ to all parties.

We show the following theorem.

Theorem 5.2.1 *Suppose Σ is a linear secret sharing scheme over $\mathbb{Z}_{2^k}$ s.t. any t shares reveal no information about the secret. Let τ denote the security parameter. For all $\varepsilon = \mathtt{negl}(\tau)$, the protocol* TRUNCPR *is* not *t-private for $\mathcal{F}_{truncPr}$ with statistical error ε according to the semi-honest security in [9].*

Proof 5.2.1 *For the sake of contradiction, let's assume that* TRUNCPR *is t-private for $\mathcal{F}_{truncPr}$. By definition, there exists an ideal adversary $\mathcal{S}$ s.t. for all input sharing $[x]$ s.t. $x < 2^{k-\tau}$,*

$$([\tilde{x}'], \mathtt{View}_{\mathcal{C}}([x])) \approx_{\varepsilon} ([x'], \mathcal{S}(\mathcal{C},[x]_{\mathcal{C}},[x']_{\mathcal{C}})),$$

where $[\tilde{x}']$ denotes the output sharing in the real world, $[x']$ denotes the output sharing in the ideal world, and $[x']_{\mathcal{C}}$ denotes the shares of $[x']$ of corrupted parties. This implies that

$$(\tilde{x}', \mathtt{View}_{\mathcal{C}}([x])) \approx_{\varepsilon} (x', \mathcal{S}(\mathcal{C},[x]_{\mathcal{C}},[x']_{\mathcal{C}})). \qquad (8)$$

Consider the case where $x = 2^{d-1}$ and P_1 is corrupted. We first claim that in the real world, $\tilde{x}'$ is determined by $\mathtt{View}_{\mathcal{C}}([x])$. Note that in the real world, $\tilde{x}' = (x+r)^d - r^d$. Thus $\tilde{x}'$ is determined by x and r. Since we have fixed the input sharing $[x]$ and $x+r$ is learnt by P_1, which is corrupted, $\tilde{x}'$ is fixed given the views of corrupted parties. As a result, for the distribution of $(\tilde{x}', \mathtt{View}_{\mathcal{C}}([x]))$, $\tilde{x}'$ is fixed given $\mathtt{View}_{\mathcal{C}}([x])$.

Now we analyze the distribution of $(x', \mathcal{S}(\mathcal{C},[x]_{\mathcal{C}},[x']_{\mathcal{C}}))$. Since Σ is a linear secret sharing scheme s.t. any t shares reveal no information about the secret, the shares of $[x']$ of corrupted parties are independent of the secret x'. Since $x = 2^{d-1}$, $\mathcal{F}_{truncPr}$ will set $x' = x^d = 0$ or $x' = x^d+1 = 1$ with probability $1/2$. And the randomness used to determine x' is independent of the shares of $[x]$ and $[x']$ of corrupted parties. Thus, given $\mathcal{S}(\mathcal{C},[x]_{\mathcal{C}},[x']_{\mathcal{C}})$, x' is a random bit. As a result, for the distribution $(x', \mathcal{S}(\mathcal{C},[x]_{\mathcal{C}},[x']_{\mathcal{C}}))$, given $\mathcal{S}(\mathcal{C},[x]_{\mathcal{C}},[x']_{\mathcal{C}})$, x' is a random bit.

This implies Equation 8 does not hold (as given the second part of the distribution, $\tilde{x}'$ in the LHS is fixed while x' in the RHS is a random bit). Thus, such an ideal adversary $\mathcal{S}$ cannot exist. The protocol TRUNCPR *is not t-private for $\mathcal{F}_{truncPr}$.*

6 Evaluation

In this section, we show the implementation of our protocol and its application to DNN inference, as well as benchmark results compared with the best of previous works. Our code is available at https://github.com/AntCPLab/malicious_3pc_binary.

Depth		Semi	Ours	BGIN19	FLNW17
1	LAN Time	0.12	1.15	3.42	0.95
	WAN Time	2.97	3.31	5.67	11.74
10	LAN Time	0.12	1.14	3.78	0.90
	WAN Time	2.18	2.58	4.61	11.41
100	LAN Time	0.12	1.14	3.84	0.91
	WAN Time	5.18	5.85	6.78	14.94
1000	LAN Time	0.18	1.16	3.87	0.96
	WAN Time	41.05	41.83	42.76	51.05
10000	LAN Time	0.70	1.36	4.05	1.50
	WAN Time	401.60	402.47	403.35	412.35
	Comm.	24.00	24.80	24.57	224.16

Table 2: Time (s), communication (MB) for computing circuits of 64 million AND gates with different depths.

	Semi	Ours	BGIN19	FLNW17
LAN Time	0.04	1.08	3.83	0.86
WAN Time	3.86	4.77	5.75	14.98
Comm.	24.00	24.81	24.57	224.39

Table 3: Time (s), communication (MB) for computing AES-128 circuits 10,000 times in parallel (with totally 64,000,000 AND gates).

6.1 Implementation

Our implementation is based on MP-SPDZ [28], a popular and versatile framework for a variety of MPC protocols. We implemented the 61-bit Mersenne field as the underlying prime field, and adopted the Fiat-Shamir transformation [20] to obtain a constant-round proof system.

To comprehensively evaluate our protocol, we additionally implement the FLIOP based verification protocol BGIN19 in [7] using the binary extension field $\mathbb{F}_{2^{64}}$ in MP-SPDZ, including all the optimization techniques[4] described in Section 4.3 (as well as fast Intel instructionsfor multiplications over binary extension fields). We also compare with the cut-and-choose based protocol FLNW17 [21] that has been implemented in MP-SPDZ. (As we mentioned above, we choose to compare with [21] rather than the optimized version [2] because in MP-SPDZ, the implementation of [21] is faster than that of [2]. And the work [2] does not open the source of their implementation.) In addition, we also test the best-known semi-honest protocol Semi [3] to understand the cost of achieving malicious security for binary computation.

To support DNN inference which involves mixed-circuit computations, we integrate our protocol with the state-of-the-art RSS based maliciously secure 3PC protocol, SpdzWise [1] (denoted by SW in the following), for arithmetic circuits, and dabits techniques [42] for transforming between arithmetic and binary worlds. We remark that, our experiment is mainly to understand the advantage of using our protocol for the binary computation in DNN compared with using [7, 21] rather than building a new protocol for the DNN reference. In Appendix D, we compare our choice for DNN inference and the state-of-the-art 3PC for DNN [17, 32].

We test four variants of DNN inference protocols by using Semi, our protocol, BGIN19, and FLNW17 for binary computation respectively. (The corresponding integrated protocols are denoted by SW + Semi, SW + Ours, SW + BGIN19 and SW + FLNW17.) We also report the cost of running the entirely semi-honest protocol (denoted by Semi + Semi) on DNN inference.

6.2 Experimental Setup

To thoroughly evaluate our protocol, we conduct three kinds of experiments: (1) microbenchmarks for arithmetic operations over the binary extension field $\mathbb{F}_{2^{64}}$ and the prime field $\mathbb{F}_p$, where $p = 2^{61} - 1$ is the 61-bit Mersenne prime, (2) benchmarks for running pure binary circuits, including large binary circuits of varied depths and AES circuits, and (3) secure inference for three large-scale DNN models.

All the experiments were run on three Alibaba Cloud g7.8xlarge instances running Ubuntu 20.04, each equipped with 32-core Intel(R) Xeon(R) Platinum 8369B CPU @2.70GHz and 128GB of RAM. The machines are in a LAN with about 23Gbps bandwidth and $30\mu s$ (one-way) latency. As for WAN setting, we use the linux *tc* command to set the bandwidth at 80Mbps and latency at 40ms, which could simulate an actual network condition between two very distant machines. We run all programs with a single thread, but run the NN inference under WAN with 32 threads because single thread might be too slow for that setting.

Due to page limit, the microbenchmarks for arithmetic operations over different fields are moved to Appendix E.1.

6.3 Benchmarks for Binary Circuits

Binary Circuits with Different Depths. We construct pure binary circuits of different depths ranging from 1 to 10,000, each of which computes 64 million AND gates with random inputs. We report the running time and the (global) communication cost of the protocols in Table 2.

For the communication complexity, we can see that both our protocol and BGIN19 only add a sub-linear cost to Semi (which communicates 1 bit per AND gate per party), while the implementation of FLNW17 in MP-SPDZ communicates around 9 bit per AND gate per party.

[4] We also implement the non-recursive version of the verification protocol in [7]; however it performs much worse than the recursive version, so we choose the recursive version of [7] as our main baseline.

Model	# of Threads		Semi + Semi	SW + Semi[1]	SW + Ours	SW + BGIN19	SW + FLNW17
ResNet-50	1	LAN Time	89.68	336.71	372.97	582.55	391.23
	32	LAN Time	18.66	56.64	63.33	81.48	87.48
	32	WAN Time	544.15	1969.48	2048.89	2096.17	2786.22
		Comm.	7537.86	27791.9	27846.10	27830.40	41114.30
DenseNet	1	LAN Time	63.42	305.89	375.72	622.83	371.124
	32	LAN Time	12.98	57.07	66.13	84.60	94.39
	32	WAN Time	713.42	1994.98	2070.69	2096.17	2842.19
		Comm.	8919.85	31924.50	31993.40	31973.5	48709.60
SqueezeNet	1	LAN Time	13.61	49.13	58.89	106.20	63.09
	32	LAN Time	2.23	9.93	11.28	14.17	15.70
	32	WAN Time	200.19	432.05	448.96	455.29	674.26
		Comm.	1403.22	4803.36	4816.58	4812.76	8047.35

[1] The setting SW + Semi does not correspond to a meaningful security notion. Our intention is to report the best possibility one can achieve by only improving the binary computation part.

Table 4: Time (s), communication (MB) for secure inference on different neural networks.

For the running time under LAN, our protocol is 3 ∼ 3.4× faster than BGIN19 and is close to FLNW17 (with 1 ∼ 1.3× slow down). This is as expected given that arithmetic operations over prime fields are more efficient than those over binary extension fields as demonstrated in Appendix E.1, and given the optimizations we introduced in Section 4.3.

In the WAN setting, our protocol is 1 ∼ 1.8× faster than BGIN19. The gap is smaller than LAN, because in the WAN setting, the communication complexity becomes the major bottleneck and the communication complexity of ours and BGIN19 are very close. Our protocol is 2.6 ∼ 4.2× faster than FLNW17 under depths of 1 ∼ 100 due to the saving in the communication complexity. For the circuits of depths 1000 ∼ 10000, all the protocols perform close because the network latency cost becomes dominating.

AES Circuit. We utilize the AES-128 circuit file provided by MP-SPDZ and run 10,000 times in parallel. Each AES-128 circuit consists of 6,400 AND gates, 28,176 XOR gates and 2,087 INV gates with a depth of 60. The result is shown in Table 3.

In the LAN setting, our protocol is 3.5× faster than BGIN19, and 1.25× slower than FLNW17. In the WAN setting, our protocol is about 1.2× faster than BGIN19, and 3.1× faster than FLNW17. These experiment results are consistent with Table 2. As a conclusion, our protocol is 3× faster than BGIN19 and is close to FLNW17 under LAN, and our protocol achieves 9× saving in the communication compared with FLNW17 (as BGIN19 does).

6.4 Maliciously Secure DNN Inference

We run three deep neural networks: ResNet-50, DenseNet, and SqeezeNet which are commonly used in industry.

For the total communication, both SW + Ours and SW + BGIN19 are similar to the baseline SW + Semi as expected. On the other hand, SW + FLNW17 needs 1.5 ∼ 1.7× more communication than SW + Ours and SW + BGIN19. The additional communication cost of SW + FLNW17 is due to the protocol FLNW17 for the binary part.

In the LAN setting, SW + Ours is 1.3 ∼ 1.8× faster than SW + BGIN19, while 1 ∼ 1.4× faster than SW + FLNW17. In the WAN setting, SW + Ours has almost the same speed with SW + BGIN19 (which is as expected as the communication cost becomes the bottleneck in WAN), while 1.4 ∼ 1.5× faster than SW + FLNW17.

The end-to-end improvement is not as large as in 6.3, because all the protocols use the same protocol SpdzWise [1] to compute the arithmetic part of NNs. However, the difference of SW + Ours (w.r.t. SW + BGIN19 or SW + FLNW17) and the SW + Semi can be viewed as the cost of achieving malicious security for the binary computation. We note that our such cost is only 7% ∼ 10% of SW + FLNW17's under WAN and 15% ∼ 33% of SW + BGIN19's under LAN.

Comparison With the Fully Semi-honest Protocol. We also report the communication and runtime of the fully semi-honest protocol for the DNN inference in Table 4. Our purpose is to understand the real overhead of achieving malicious security for the DNN inference.

For the communication complexity, compared with the fully semi-honest protocol Semi + Semi, both SW + Ours and SW + BGIN19 cost 3.4 ∼ 3.7× more communication for achieving malicious security, while SW + FLNW17 costs 6.1 ∼ 6.5× more communication. From the comparison with SW + Semi, we can see that the main additional communication cost of SW + Ours and SW + BGIN19 comes from achieving malicious security for computation other than the binary part in the DNN inference, which includes the use of SW for arithmetic computation, the use of dabits techniques [42] for transforming between arithmetic and binary worlds and so on. On the other hand, for SW + FLNW17, the additional cost also comes from the binary part due to the use of FLNW17 for binary computation.

For the running time, compared with the fully semi-honest protocol, in the LAN setting SW + Ours is $3.4 \sim 5.9\times$ slower, while SW + BGIN19 and SW + FLNW17 are $4.2 \sim 9.8\times$ and $3.7 \sim 6.2\times$ slower respectively. In the WAN setting, the gaps become smaller: SW + Ours is $2.2 \sim 3.8\times$ slower, SW + BGIN19 and SW + FLNW17 are $2.3 \sim 3.9\times$ and $4.2 \sim 5.4\times$ slower. This mainly arises from the network latency cost in the WAN setting.

As we can see, with our efficient protocol for binary computation, the major bottleneck becomes to achieve malicious security for arithmetic computation, which we believe is an interesting research question.

7 Conclusion

In this work we studied maliciously secure 3PC protocol for binary circuits in an honest majority. Following previous work [7], we use distributed zero-knowledge proofs to verify semi-honest computations conducted over the binary field $\mathbb{F}_2$. But differently, we transform the verification of the semi-honest computations over $\mathbb{F}_2$ into prime fields, and take advantage of the algebraic structure of prime fields to accelerate computations. As a result, we can achieve sub-linear additional communication cost at much lower computational cost than the extension field based approach in [7].

We did experiments and proved that our protocol can achieve more than 3× speed-up over [7] in LAN and more than 4× speed-up over [21] in WAN for computing binary circuits. When applied to secure DNN inference, we could reduce the overhead for obtaining malicious security in binary part by more than 67%.

Acknowledgments

We sincerely thank all the anonymous reviewers and the shepherd for their valuable comments and suggestions to help us improve this work. The authors Yun Li, Yufei Duan, and Chao Zhang are supported in part by the National Key Research and Development Program of China (2021YFB2701000) and National Natural Science Foundation of China (61972224).

References

[1] Mark Abspoel, Anders Dalskov, Daniel Escudero, and Ariel Nof. An efficient passive-to-active compiler for honest-majority mpc over rings. In *International Conference on Applied Cryptography and Network Security*, pages 122–152. Springer, 2021.

[2] Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, and Or Weinstein. Optimized honest-majority mpc for malicious adversaries — breaking the 1 billion-gate per second barrier. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 843–862, 2017.

[3] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 805–817. ACM, 2016.

[4] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *Annual International Cryptology Conference*, pages 420–432. Springer, 1991.

[5] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In Harriet Ortiz, editor, *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*, pages 503–513. ACM, 1990.

[6] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear pcps. In *Annual International Cryptology Conference*, pages 67–97. Springer, 2019.

[7] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Practical fully secure three-party computation via sublinear distributed zero-knowledge proofs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 869–886, 2019.

[8] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Efficient fully secure computation via distributed zero-knowledge proofs. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 244–276. Springer, 2020.

[9] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of CRYPTOLOGY*, 13(1):143–202, 2000.

[10] Ignacio Cascudo, Ronald Cramer, Chaoping Xing, and Chen Yuan. Amortized complexity of information-theoretically secure MPC revisited. In Hovav Shacham

and Alexandra Boldyreva, editors, *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part III*, volume 10993 of *Lecture Notes in Computer Science*, pages 395–426. Springer, 2018.

[11] Octavian Catrina and Sebastiaan de Hoogh. Improved primitives for secure multiparty integer computation. In *International Conference on Security and Cryptography for Networks*, pages 182–199. Springer, 2010.

[12] Harsh Chaudhari, Rahul Rachuri, and Ajith Suresh. Trident: Efficient 4pc framework for privacy preserving machine learning. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

[13] David Chaum. Blind signature system. In *Advances in cryptology*, pages 153–153. Springer, 1984.

[14] David Chaum, Claude Crépeau, and Ivan Damgard. Multiparty unconditionally secure protocols. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 11–19, 1988.

[15] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority MPC for malicious adversaries. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part III*, volume 10993 of *Lecture Notes in Computer Science*, pages 34–64. Springer, 2018.

[16] Anders Dalskov, Daniel Escudero, and Marcel Keller. Secure evaluation of quantized neural networks. *Proceedings on Privacy Enhancing Technologies*, 4:355–375, 2020.

[17] Anders P. K. Dalskov, Daniel Escudero, and Marcel Keller. Fantastic four: Honest-majority four-party secure computation with malicious security. In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 2183–2200. USENIX Association, 2021.

[18] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for mpc over mixed arithmetic-binary circuits. In *Annual International Cryptology conference*, pages 823–852. Springer, 2020.

[19] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for mpc over mixed arithmetic-binary circuits. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020*, pages 823–852, Cham, 2020. Springer International Publishing.

[20] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 186–194. Springer, 1986.

[21] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 225–255. Springer, 2017.

[22] O Goldreich, S Micali, and A Wigderson. How to play any mental game. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 218–229, 1987.

[23] Oded Goldreich. *The Foundations of Cryptography - Volume 2: Basic Applications*. Cambridge University Press, 2004.

[24] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM (JACM)*, 33(4):792–807, 1986.

[25] Vipul Goyal and Yifan Song. Malicious security comes free in honest-majority MPC. *IACR Cryptol. ePrint Arch.*, page 134, 2020.

[26] Vipul Goyal, Yifan Song, and Chenzhi Zhu. Guaranteed output delivery comes free in honest majority MPC. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part II*, volume 12171 of *Lecture Notes in Computer Science*, pages 618–646. Springer, 2020.

[27] Mitsuru Ito, Akira Saito, and Takao Nishizeki. Secret sharing scheme realizing general access structure. *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)*, 72(9):56–64, 1989.

[28] Marcel Keller. Mp-spdz: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1575–1590, New York, NY, USA, 2020. Association for Computing Machinery.

[29] Ryo Kikuchi, Nuttapong Attrapadung, Koki Hamada, Dai Ikarashi, Ai Ishida, Takahiro Matsuda, Yusuke Sakai,

and Jacob C. N. Schuldt. Field extension in secret-shared form and its applications to efficient secure computation. In Julian Jang-Jaccard and Fuchun Guo, editors, *Information Security and Privacy - 24th Australasian Conference, ACISP 2019, Christchurch, New Zealand, July 3-5, 2019, Proceedings*, volume 11547 of *Lecture Notes in Computer Science*, pages 343–361. Springer, 2019.

[30] Mehmet Kiraz and Berry Schoenmakers. A protocol issue for the malicious case of yao's garbled circuit construction. In *27th Symposium on Information Theory in the Benelux*, volume 39, 2006.

[31] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free xor gates and applications. In *International Colloquium on Automata, Languages, and Programming*, pages 486–498. Springer, 2008.

[32] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. {SWIFT}: Super-fast and robust {Privacy-Preserving} machine learning. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2651–2668, 2021.

[33] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 52–78. Springer, 2007.

[34] Payman Mohassel and Matthew Franklin. Efficiency tradeoffs for malicious two-party computation. In *International Workshop on Public Key Cryptography*, pages 458–473. Springer, 2006.

[35] Payman Mohassel and Peter Rindal. Aby^3: A mixed protocol framework for machine learning. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 35–52. ACM, 2018.

[36] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 19–38. IEEE Computer Society, 2017.

[37] Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In *Proceedings of the 1st ACM Conference on Electronic Commerce*, pages 129–139, 1999.

[38] Arpita Patra and Ajith Suresh. BLAZE: blazing fast privacy-preserving machine learning. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

[39] Benny Pinkas, Thomas Schneider, Nigel P Smart, and Stephen C Williams. Secure two-party computation is practical. In *International conference on the theory and application of cryptology and information security*, pages 250–267. Springer, 2009.

[40] Antigoni Polychroniadou and Yifan Song. Constant-overhead unconditionally secure multiparty computation over binary fields. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part II*, volume 12697 of *Lecture Notes in Computer Science*, pages 812–841. Springer, 2021.

[41] Mike Rosulek and Lawrence Roy. Three halves make a whole? beating the half-gates lower bound for garbled circuits. In *Annual International Cryptology Conference*, pages 94–124. Springer, 2021.

[42] Dragos Rotaru and Tim Wood. Marbled circuits: Mixing arithmetic and boolean circuits with active security. In Feng Hao, Sushmita Ruj, and Sourav Sen Gupta, editors, *Progress in Cryptology – INDOCRYPT 2019*, pages 227–249, Cham, 2019. Springer International Publishing.

[43] Bita Darvish Rouhani, M. Sadegh Riazi, and Farinaz Koushanfar. Deepsecure: scalable provably-secure deep learning. In *Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018*, pages 2:1–2:6. ACM, 2018.

[44] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[45] Sameer Wagh, Divya Gupta, and Nishanth Chandran. Securenn: 3-party secure computation for neural network training. *Proceedings on Privacy Enhancing Technologies*, 3:26–49, 2019.

[46] 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.

[47] Avi Wigderson, MB Or, and S Goldwasser. Completeness theorems for noncryptographic fault-tolerant distributed computations. In *Proceedings of the 20th Annual Symposium on the Theory of Computing (STOC'88)*, pages 1–10, 1988.

[48] Andrew C Yao. Protocols for secure computations. In *23rd annual symposium on foundations of computer science (sfcs 1982)*, pages 160–164. IEEE, 1982.

[49] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 162–167. IEEE, 1986.

[50] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 220–250. Springer, 2015.

[51] Richard Zippel. Probabilistic algorithms for sparse polynomials. In *International symposium on symbolic and algebraic manipulation*, pages 216–226. Springer, 1979.

A Building Block Functionalities

A.1 $\mathcal{F}_{\text{rand}}$ - Generating Random Shares.

We define a functionality $\mathcal{F}_{\text{rand}}$ to generate a replicated sharing of a random secret value $r \in R$ where R denotes $\mathbb{Z}_{2^k}, \mathbb{F}_2$ or $\mathbb{F}_p$.

FUNCTIONALITY A.1.1 *($\mathcal{F}_{rand}$ - Generating Random Shares).*

Let $\mathcal{S}$ be the ideal world adversary and P_i be the corrupted party controlled by $\mathcal{S}$.

- $\mathcal{F}_{\text{rand}}$ receives r_i, r_{i-1} from $\mathcal{S}$.
- $\mathcal{F}_{\text{rand}}$ picks a random $r \in R$, sets $r_{i+1} := r - r_i - r_{i-1}$.
- $\mathcal{F}_{\text{rand}}$ hands the honest parties P_{i+1}, P_{i-1} the shares (r_{i+1}, r_i) and (r_{i-1}, r_{i+1}) respectively.

We follow previous work [7] to securely instantiate functionality $\mathcal{F}_{\text{rand}}$ in the following. It relies on a pseudo-random function to generate a sharing of randomness $r \in R$ where R denotes $\mathbb{Z}_{2^k}, \mathbb{F}_2$ or $\mathbb{F}_p$.

- Each party P_i picks a random key $\mathrm{K}_i \in \{0,1\}^\kappa$ and sends it to P_{i+1}.
- Each party P_i computes $r_i := \mathrm{F}_{\mathrm{K}_i}(\mathsf{cnt})$ and $r_{i-1} := \mathrm{F}_{\mathrm{K}_{i-1}}(\mathsf{cnt})$ using an agreed pseudo-random function F from the family $\mathcal{F} = \{\mathrm{F}_\mathrm{K} | \mathrm{K} \in \{0,1\}^\kappa, \mathrm{F}_\mathrm{K} : \{0,1\}^\kappa \to R\}$ and cnt is an agreed public counter that increments each time.
- Each party P_i sets (r_i, r_{i-1}) as its share of r.

A.2 $\mathcal{F}_{\text{coin}}$ - Generating Random Coins.

We define a functionality $\mathcal{F}_{\text{coin}}$ for generating a random value $r \in R$ and handing it to all parties. It can be simply instantiated by calling $\mathcal{F}_{\text{rand}}$ to generate a sharing of a random secret value $r \in R$ then opening r to all parties.

A.3 $\mathcal{F}_{\text{input}}$ - Secure Sharing of Inputs.

We define a functionality $\mathcal{F}_{\text{input}}$ for securely share the parties' inputs over $\mathbb{F}_2$ in Functionality A.3.1, and present a protocol in previous works [7, 15] for securely instantiate $\mathcal{F}_{\text{input}}$ in the following.

FUNCTIONALITY A.3.1 *($\mathcal{F}_{input}$ - Secure Sharing of Inputs).*

Let $\mathcal{S}$ be the ideal world adversary and P_i the corrupted party controlled by $\mathcal{S}$.

- $\mathcal{F}_{\text{input}}$ receives inputs $x^{(1)}, \cdots, x^{(q)} \in \mathbb{F}_2$ from the parties, and the shares $(x_i^{(\ell)}, x_{i-1}^{(\ell)})$ of the corrupted party from $\mathcal{S}$ for $\ell \in [q]$.
- $\mathcal{F}_{\text{input}}$ computes $x_{i+1}^{(\ell)} := x^{(\ell)} \oplus x_i^{(\ell)} \oplus x_{i-1}^{(\ell)}$ for $\ell \in [q]$.
- $\mathcal{F}_{\text{input}}$ hands the honest parties P_{i+1}, P_{i-1} the shares $(x_{i+1}^{(\ell)}, x_i^{(\ell)})$ and $(x_{i-1}^{(\ell)}, x_{i+1}^{(\ell)})$ for $\ell \in [q]$ respectively.

We now review the protocol in [7, 15] for securely realizing functionality $\mathcal{F}_{\text{input}}$ in Protocol A.3.1. At a high level, for each input $x^{(\ell)}$ held by some party P_i, the parties call $\mathcal{F}_{\text{rand}}$ to obtain a sharing $[\![r^{(\ell)}]\!]^{\mathsf{B}}$ where $r^{(\ell)}$ will be used as a random mask of $x^{(\ell)}$. Next the parties run $\mathsf{reconstruct}([\![r^{(\ell)}]\!]^{\mathsf{B}}, i)$ to reveal $r^{(\ell)}$ to the input owner P_i. P_i then computes the masked value $v^{(\ell)} := x^{(\ell)} \oplus r^{(\ell)}$, and echo-broadcasts $v^{(\ell)}$ to the other parties. The parties can obtain a sharing of $x^{(\ell)}$ by locally computing $[\![x^{(\ell)}]\!]^{\mathsf{B}} := [\![r^{(\ell)}]\!]^{\mathsf{B}} \oplus v^{(\ell)}$.

B Full Protocol for Computing a Binary Circuit

We describe the full protocol from [7] for securely computing a binary circuit in Protocol B.0.1.

PROTOCOL A.3.1 *(Secure Sharing of Inputs).*

- **Inputs:** The parties hold the inputs $x^{(1)}, \cdots, x^{(q)} \in \mathbb{F}_2$; each $x^{(\ell)}$ is held by some party P_i.
- **Protocol:**
    - The parties call $\mathcal{F}_{\text{rand}}$ q times on $\mathbb{F}_2$ and obtain sharings $\llbracket r^{(1)} \rrbracket^{\mathsf{B}}, \cdots, \llbracket r^{(q)} \rrbracket^{\mathsf{B}}$.
    - For $\ell \in [q]$ where input $x^{(\ell)}$ is owned by some party P_i, the parties run $\mathsf{reconstruct}(\llbracket r^{(\ell)} \rrbracket^{\mathsf{B}}, i)$ to reveal $r^{(\ell)}$ to party P_i. If P_i detects inconsistency in the reconstruction procedure, it sends abort to the other parties and halts.
    - For each input $x^{(\ell)}$, the input owner P_i computes $v^{(\ell)} := x^{(\ell)} \oplus r^{(\ell)}$, and sends it to the other parties.
    - Each party sends the vector $(v^{(1)}, \cdots, v^{(q)})$ to the other parties, and checks the consistency between the vectors it received and its own. If inconsistency happens, it outputs abort and halts.
    - For $\ell \in [q]$, the parties locally compute input sharings $\llbracket x^{(\ell)} \rrbracket^{\mathsf{B}} := \llbracket r^{(\ell)} \rrbracket^{\mathsf{B}} \oplus v^{(\ell)}$.
- **Output:** The parties output $\llbracket x^{(1)} \rrbracket^{\mathsf{B}}, \cdots, \llbracket x^{(q)} \rrbracket^{\mathsf{B}}$.

C Our Verification Protocol and Security Proof

C.1 Step 4 of Our Verification Protocol: Applying Distributed Zero-knowledge Proofs

For completeness, here we elaborate how to apply the distributed zero-knowledge proofs [6] to the length-$4m$ inner-product relation described in Equation 6. We generalize the distributed zero-knowledge proof system based on the FLIOP construction [6] by introducing a compression parameter λ for recursion.

At a high level, the inner-product relation is transformed into polynomial equations via interpolation; then in a recursive way, the size of the proving task is shrunken in each round by a factor of λ until to the minimum size, and the verifiers conduct a simple final check to decide the proof.

(Step 4.1) Transform to Polynomials. Recall that our proving task is $\sum_{k=1}^{4m} u_k \cdot v_k = -\frac{m}{2}$. Let $m' := 4m$. We now define a circuit C computing the length-m' inner-product, which includes m' multiplication gates and sums their outputs up. We define $\mathsf{out} := -\frac{m}{2}$, denoting the output of the circuit taking as input $\vec{u}$ and $\vec{v}$, i.e.,

$$\sum_{k=1}^{m'} u_k v_k = \mathsf{out}. \tag{9}$$

Now we turn to use polynomials to express the relation. We arrange the m' multiplication gates into s groups, each of which consists of λ multiplication gates. The k-th multiplication gates in the t-th group where $k \in [\lambda], t \in [s]$ takes as left input $u_{(t-1)\lambda+k}$ and right input $v_{(t-1)\lambda+k}$. For each group t, we interpolate the values on the input wires into a polynomial. In specific, we define polynomials $p_1(X), \cdots, p_s(X)$ corresponding to the values on the left input wires, and $q_1(X), \cdots, q_s(X)$ corresponding to the values on the right input wires, s.t. for $t \in [s], k \in [\lambda]$,

$$p_t(k) = u_{(t-1)\lambda+k}, \; q_t(k) = v_{(t-1)\lambda+k}. \tag{10}$$

Each polynomial is of degree $\lambda - 1$.

Then we can encode the output of the circuit by defining a polynomial $G(X)$ s.t.

$$G(X) = \sum_{t=1}^{s} p_t(X) \cdot q_t(X). \tag{11}$$

Clearly we have

$$\sum_{k=1}^{\lambda} G(k) = \sum_{k=1}^{\lambda} \sum_{t=1}^{s} p_t(k) \cdot q_t(k) = \sum_{k=1}^{m'} u_k v_k = \mathsf{out}. \tag{12}$$

If P_i honestly conducts all AND gates (i.e., Equation 9 holds), it implies Equation 12. Therefore the verification is reduced to checking the following two conditions:

1. whether Equation 12 holds, and
2. whether $G(X)$ is correctly constructed as in Equation 11.

P_i first shares the coefficients of $G(X)$ by using RSS. We next show how to check the above two conditions.

(Step 4.2) Check Polynomial Equations. Note that for all $k \in [\lambda]$, $G(k)$ is a linear combination of the coefficients of $G(X)$. Since all parties hold RSS of the coefficients of $G(X)$, they locally compute a RSS $\llbracket b \rrbracket^{\mathsf{F}} = \llbracket \sum_{k=1}^{\lambda} G(k) - \mathsf{out} \rrbracket^{\mathsf{F}}$. Then to check the first condition, it is sufficient to reconstruct $\llbracket b \rrbracket^{\mathsf{F}}$ and check whether $b = 0$.

As for the second condition, recall that the two verifiers distributively hold the prover's data, and they need to use their partial data to verify if polynomial $G(X)$ is constructed correctly. Specifically, note that the two verifiers hold $\vec{u}, \vec{v}$ respectively, and thus they can compute $p_t(X)$ and $q_t(X)$ for $t \in [s]$ separately on their own. To check if the polynomial equation in Equation 11 holds, the parties can choose a random point r from $\mathbb{F}_p$ and check if the the polynomial equation holds at this random point. If the equation does hold at r,

PROTOCOL B.0.1 *(Securely Computing a Binary Circuit).*

- **Inputs:** Each party P_i holds an input vector $\vec{w}^{(i)} \in \mathbb{F}_2^{I_i}$, where I_i denotes the input length of P_i.
- **Auxiliary Inputs:** The parties hold a description of a binary circuit $\mathcal{C}$ that computes $f : \mathbb{F}_2^{I_1} \times \mathbb{F}_2^{I_2} \times \mathbb{F}_2^{I_3} \to \mathbb{F}_2^{O_1} \times \mathbb{F}_2^{O_2} \times \mathbb{F}_2^{O_3}$. Let m be the number of AND gates in $\mathcal{C}$.
- **Setup:**
  1. Each party P_i chooses a random key $\mathrm{K}_i \in \{0,1\}^\kappa$ and sends it to P_{i+1}.
  2. For every AND gate, each party P_i computes $\rho_i := \mathrm{F}_{\mathrm{K}_i}(\mathsf{cnt}), \rho_{i-1} := \mathrm{F}_{\mathrm{K}_{i-1}}(\mathsf{cnt})$, and $\alpha_i := \rho_i \oplus \rho_{i-1}$, where F is an agreed pseudo-random function from the family $\mathcal{F} = \{\mathrm{F}_\mathrm{K} | \mathrm{K} \in \{0,1\}^\kappa, \mathrm{F}_\mathrm{K} : \{0,1\}^\kappa \to \mathbb{F}_2\}$, and cnt is an agreed public counter that increments each time.
- **Protocol:**
  1. ***Input Sharing***
     (a) Each party P_i sends its input vector $\vec{w}^{(i)}$ to $\mathcal{F}_{\text{input}}$.
     (b) Each party receives its shares of all inputs from $\mathcal{F}_{\text{input}}$. If any party receives abort from $\mathcal{F}_{\text{input}}$, then its sends abort to the other parties, outputs abort and halts.
  2. ***Circuit Evaluation***
     Let $G_1, G_2, \cdots, G_N$ be a predetermined topological ordering of the gates of the circuit $\mathcal{C}$. For $n \in [N]$, the parties work as follows:
     (a) If G_n is an XOR gate: given shares $(x_i, x_{i-1}), (y_i, y_{i-1})$ of x, y on the input wires, each party P_i locally computes $(x_i \oplus y_i, x_{i-1} \oplus y_{i-1})$ as its share of $x \oplus y$.
     (b) If G_n is an AND-by-a-constant gate: given shares (x_i, x_{i-1}) of x on the input wire, and a public constant $c \in \mathbb{F}_2$, each party P_i locally computes $(c \cdot x_i, c \cdot x_{i-1})$ as its share of $c \cdot x$.
     (c) If G_n is an AND gate: given shares $(x_i, x_{i-1}), (y_i, y_{i-1})$ of x, y on the input wires, each party P_i computes $z_i := x_i y_i \oplus x_i y_{i-1} \oplus x_{i-1} y_i \oplus \alpha_i$, sends z_i to party P_{i+1}, and sets (z_i, z_{i-1}) as its share of $x \cdot y$.
  3. ***Verification***
     (a) Each party P_i sends i, m, its shares $(x_i, x_{i-1}), (y_i, y_{i-1})$ on the input wires of each AND gate, (ρ_i, ρ_{i-1}) for computing correlated randomnesses for each AND gate, and the message z_i it sent out for each AND gate to $\mathcal{F}_{\text{vrfy}}$.
     (b) If any party receives abort from $\mathcal{F}_{\text{vrfy}}$, then its sends abort to the other parties, outputs abort and halts.
  4. If any party received abort in any of the previous steps, then it outputs abort and halts.
  5. ***Output Reconstruction***
     (a) For each output wire of the circuit $\mathcal{C}$, the parties run $\mathsf{reconstruct}([\![z]\!]^\mathrm{B}, i)$ where $[\![z]\!]^\mathrm{B}$ is the sharing of the value z on the output wire that should be revealed to P_i.
     (b) If any party receives abort in the reconstruct procedure, then its sends abort to the other parties, outputs abort and halts.
- **Output:** If a party didn't output abort, then it outputs the values it reconstructed on its output wires.

then provided that the field size is large enough, by Schwartz-Zipple lemma [51], Equation 11 holds with overwhelming probability.

Thus after the verifiers get replicated secret shares of coefficients $\vec{c}$ of polynomial $G(X)$, since $G(r)$ is a linear combination of the coefficients of $G(X)$, they can compute a RSS of $G(r)$. Recall that they can compute $p_t(X)$ and $q_t(X)$ themselves; thus the verifiers can reveal the shares of $G(r)$ and their evaluations $p_t(r)$ and $q_t(r)$ to check the equation

$$\sum_{t=1}^{s} p_t(r) \cdot q_t(r) = G(r). \tag{13}$$

(Step 4.3) Recursive Delegation. As shown above, checking the second condition requires the verifiers to communicate $\{p_t(r), q_t(r)\}_{t\in[s]}$, and evaluate a length-$s$ inner product. To further reduce their communication and computation complexity, the parties can utilize a recursive paradigm to delegate the evaluation of the inner-product to the prover again.

Specifically, we re-define $m' := s$ and vectors $\vec{v} := (p_1(r), \cdots, p_s(r)), \vec{u} := (q_1(r), \cdots, q_s(r))$ of length m', and also $\mathsf{out} := G(r)$. Now the parties can start a new proving task of a length-m' inner-product relation $\sum_{k=1}^{m'} u_k v_k = \mathsf{out}$ exactly as the form of Equation 9 in the first step.

Note that the length of the inner-product to prove is reduced by a factor of λ in each round of recursion. After $\log_\lambda m$ rounds of recursion, the length is reduced to 1, i.e., a single multiplication relation. At that time, the verifiers can directly evaluate the multiplication gate by revealing their local data $p_1(r)$ and $q_1(r)$ respectively, and then check the multiplication relation to finish the verification.

An Omitted Security Issue. An issue of the above approach is that revealing $p_1(r)$ and $q_1(r)$ is not secure. This is because $p_1(r)$ and $q_1(r)$ are related to the private inputs of the two verifiers. To protect $p_1(r)$ and $q_1(r)$, we follow the previous work [6] by setting $p_1(0), q_1(0)$ to be random values (and thus $p_1(X)$ and $q_1(X)$ now are of degree λ). These two random values are used as random masks to protect the private inputs of the two verifiers.

C.2 Full Description of Our Verification Protocol

We describe the full protocol for verification in Protocol C.2.1.

C.3 Proof of Theorem 3.2.1

Proof C.3.1 *Let $\mathcal{S}$ be the ideal world simulator controlling a corrupted party P_i and $\mathcal{A}$ the real world adversary. The simulator $\mathcal{S}$ is invoked by $\mathcal{F}_{vrfy}$ handing it an index of the prover party j, the input of the corrupted party P_i. We consider the following two different cases.*

Case 1: the prover party P_j is honest. *In this case, the simulator $\mathcal{S}$ receives from $\mathcal{F}_{vrfy}$ the input of the corrupted verifier P_i. Without loss of generality, we assume that $i = j+1$. $\mathcal{S}$ needs to simulate the role of the honest prover P_j and the other honest verifier P_{j-1} without knowing their inputs.*

- *$\mathcal{S}$ runs the setup step. It invokes the adversary $\mathcal{A}$ receiving the key K_{j+1} used by the corrupted verifier P_{j+1}, and thus can compute the randomness η used by P_{j+1}.*
- *$\mathcal{S}$ simulates the recursion step. It initializes the variables $\widehat{\mathsf{out}}_{j+1} := -\frac{m}{2}$ and m', s as defined in the protocol. In each round of iteration, it picks a random vector $\hat{\vec{c}}_{j+1}$ of length $2\lambda - 1$ (or $2\lambda + 1$ if $s = 1$), and simulates the honest prover P_j handing the coefficient share $\hat{\vec{c}}_{j+1}$ to $\mathcal{A}$. Then $\mathcal{S}$ simulates $\mathcal{F}_{coin}$ handing a random point r to $\mathcal{A}$. Recall that $\mathcal{S}$ knows the coefficient share $\hat{\vec{c}}_{j+1}$ and out_{j+1}, it can compute $\widehat{\mathsf{sum}}_{j+1} := \sum_{k=1}^{\lambda} \hat{G}_{j+1}(k)$ as well as the message $\hat{b}_{j+1} := \widehat{\mathsf{sum}}_{j+1} - \widehat{\mathsf{out}}_{j+1}$ that should be revealed to the other honest verifier P_{j-1}, and thus knows exactly if P_{j+1} has cheated upon receiving $\mathcal{A}$'s real message. To simulate the honest verifier P_{j-1}, $\mathcal{S}$ sets $\hat{b}_{j-1} := -\hat{b}_{j+1}$, and hands $\hat{b}_{j-1}$ to $\mathcal{A}$. Recall that $\mathcal{S}$ knows the corrupted verifier P_{j+1}'s input and the randomnesses η, it can compute polynomials $p_t(X)$ for $t \in [s]$ exactly as P_{j+1} does. If $s > 1$, $\mathcal{S}$ defines $\hat{u}_t := \hat{p}_t(X)$ for $t \in [s]$, $\mathsf{out}_{j+1} := \hat{G}_{j+1}(r)$ using the coefficient share $\hat{\vec{c}}_{j+1}$, as well as m', s as described in the protocol, and goes to the next iteration. Otherwise it proceeds to the next step. If $\mathcal{S}$ receives any incorrect messages from $\mathcal{A}$ in this process, it simulates the honest verifier aborting in the real execution, and outputs whatever $\mathcal{A}$ outputs.*
- *$\mathcal{S}$ simulates the final check step. Recall that $\mathcal{S}$ knows exactly P_{j+1}'s local data, and thus can compute $\hat{\mathsf{out}}_{j+1} := G_{j+1}(r), \hat{u} := \hat{p}_1(r)$. Then $\mathcal{S}$ picks a random value $\hat{v} \in \mathbb{F}_p$ as the message sent by the honest verifier P_{j-1}, computes $\hat{\mathsf{out}}_{j-1} := \hat{u} \cdot \hat{v} - \hat{\mathsf{out}}_{j+1}$. $\mathcal{S}$ then simulates P_{j-1} handing $\hat{\mathsf{out}}_{j-1}, \hat{v}$ to $\mathcal{A}$. $\mathcal{S}$ also checks if the messages received from $\mathcal{A}$ are consistent with the values $\mathcal{A}$ should have sent. If any message is inconsistent, then $\mathcal{S}$ simulates the honest verifier aborting, and outputs whatever $\mathcal{A}$ outputs.*

We claim that the distribution of the adversary's view in the simulation is indistinguishable from that in the real execution. Observe that the adversary's view consists of the share $\vec{c}_{j+1}$ of the coefficients of polynomial $G(X)$ from the prover P_j, share b_{j-1} from the honest verifier P_{j-1} in each round of recursion, and evaluation $\mathsf{out}_{j-1} := G_{j-1}(r), v := q_1(r)$ also from P_{j-1} in the final check. First, the distribution of $\vec{c}_{j+1}$ in the real execution and the simulation is identical since they are both uniformly random over $\mathbb{F}_p$ due to perfect secrecy of additive sharing. Next, as the coefficients of polynomial $G_{j-1}(X)$ are random, $b_{j-1} := \sum_{k=1}^{\lambda} G_{j-1}(k) - \mathsf{out}_{j-1}$ is also random

PROTOCOL C.2.1 *(Securely Computing $\mathcal{F}_{vrfy}$).*

- **Inputs:** Prover P_i holds input $(x_i^{(\ell)}, x_{i-1}^{(\ell)}, y_i^{(\ell)}, y_{i-1}^{(\ell)}, \rho_i^{(\ell)}, \rho_{i-1}^{(\ell)}, z_i)$ for $\ell \in [m]$.
  Verifiers P_{i+1}, P_{i-1} hold input $(x_i^{(\ell)}, y_i^{(\ell)}, \rho_i^{(\ell)}, z_i)$ and $(x_{i-1}^{(\ell)}, y_{i-1}^{(\ell)}, \rho_{i-1}^{(\ell)})$ for $\ell \in [m]$ respectively.
- **Auxiliary Inputs:** The parties hold a public parameter λ.
- **Setup:**
  1. Each party P_j for $j \in \{0,1,2\}$ chooses a random key $\mathrm{K}_j \in \{0,1\}^\kappa$ and sends it to P_{j+1}.
  2. P_i, P_{i+1} compute $\eta := \mathrm{F}_{\mathrm{K}_i}(\mathsf{cnt})$, and P_i, P_{i-1} compute $\tau := \mathrm{F}_{\mathrm{K}_{i-1}}(\mathsf{cnt})$ where F is an agreed pseudo-random function F from the family $\mathcal{F} = \{\mathrm{F}_\mathrm{K} | \mathrm{K} \in \{0,1\}^\kappa, \mathrm{F}_\mathrm{K} : \{0,1\}^\kappa \to \mathbb{F}_p\}$, and cnt is an agreed public counter that increments each time.
- **Protocol:**
  1. ***Preparation.***
     (a) For $\ell \in [m]$, P_i sets $a^{(\ell)} := x_i^{(\ell)}, c^{(\ell)} := y_i^{(\ell)}, e^{(\ell)} := x_i^{(\ell)} \cdot y_i^{(\ell)} \oplus z_i^{(\ell)} \oplus \rho_i^{(\ell)}, b^{(\ell)} := y_{i-1}^{(\ell)}, d^{(\ell)} := x_{i-1}^{(\ell)}, f^{(\ell)} := \rho_{i-1}^{(\ell)}$, $g_1^{(\ell)} := -2a^{(\ell)} \cdot c^{(\ell)} \cdot (1-2e^{(\ell)}), g_2^{(\ell)} := c^{(\ell)} \cdot (1-2e^{(\ell)}), g_3^{(\ell)} := a^{(\ell)} \cdot (1-2e^{(\ell)}), g_4^{(\ell)} := -(1-2e^{(\ell)})/2, h_1^{(\ell)} :=$ $b^{(\ell)} \cdot d^{(\ell)} \cdot (1-2f^{(\ell)}), h_2^{(\ell)} := d^{(\ell)} \cdot (1-2f^{(\ell)}), h_3^{(\ell)} := b^{(\ell)} \cdot (1-2f^{(\ell)}), h_4^{(\ell)} := 1-2f^{(\ell)}$, and defines vectors $\vec{u} := (g_1^{(1)}, \cdots, g_4^{(1)}, \cdots\cdots, g_1^{(m)}, \cdots, g_4^{(m)}), \vec{v} := (h_1^{(1)}, \cdots, h_4^{(1)}, \cdots\cdots, h_1^{(m)}, \cdots, h_4^{(m)})$ of length $4m$.
     (b) P_{i+1}, P_{i-1} compute vector $\vec{u}$ and $\vec{v}$ respectively as P_i does.
  2. ***Recursion.***
     Let $\mathsf{out} := -\frac{m}{2}, m' := 4m, s := \lceil m'/\lambda \rceil$. P_{i+1}, P_{i-1} define $\mathsf{out}_{i+1} := \mathsf{out}$ and $\mathsf{out}_{i-1} := 0$ respectively.
     (a) For $k \in [\lambda]$, if $(s-1)\lambda + k > m'$, the parties define $u_{(s-1)\lambda+k} := 0, v_{(s-1)\lambda+k} := 0$.
     (b) P_i defines polynomials $p_1(X), \cdots, p_s(X)$ and $q_1(X), \cdots, q_s(X)$ of degree $\lambda - 1$ s.t. for $t \in [s], k \in [\lambda]$, $p_t(k) := u_{(t-1)\lambda+k}, q_t(k) := v_{(t-1)\lambda+k}$. If $s = 1$, P_i additionally sets $p_1(0) := \eta, q_1(0) := \tau$, and the degree of $p_1(X), q_1(X)$ turns to λ.
     (c) P_{i+1} computes $p_1(X), \cdots, p_s(X)$ and P_{i-1} computes $q_1(X), \cdots, q_s(X)$ respectively as P_i does.
     (d) P_i computes polynomial $G(X)$ s.t. $G(X) = \sum_{t=1}^{s} p_t(X) \cdot q_t(X)$.
     (e) Let $\vec{c}$ be the coefficients of polynomial $G(X)$. P_i additively shares $\vec{c}$ by choosing a random vector $\vec{c}_{i+1}$ of the same length with $\vec{c}$ and sets $\vec{c}_{i-1} := \vec{c} - \vec{c}_{i+1}$. P_i hands shares $\vec{c}_{i-1}, \vec{c}_{i+1}$ to P_{i+1}, P_{i-1} respectively.
     (f) The parties call $\mathcal{F}_{\mathrm{coin}}$ to receive a random $r \in \mathbb{F}_p \backslash [\lambda]$. If $s = 1$, the parties call $\mathcal{F}_{\mathrm{coin}}$ to pick r from $\mathbb{F}_p \backslash [0, \lambda]$.
     (g) P_{i+1}, P_{i-1} evaluate $G_{i+1}(r), G_{i-1}(r)$ where $G_{i+1}(X), G_{i-1}(X)$ are polynomials defined by coefficient shares $\vec{c}_{i-1}, \vec{c}_{i+1}$, and compute $\mathsf{sum}_{i+1} := \sum_{k=1}^{\lambda} G_{i+1}(k), \mathsf{sum}_{i-1} := \sum_{k=1}^{\lambda} G_{i-1}(k)$ respectively.
     (h) P_{i+1}, P_{i-1} compute $b_{i+1} := \mathsf{sum}_{i+1} - \mathsf{out}_{i+1}$ and $b_{i-1} := \mathsf{sum}_{i-1} - \mathsf{out}_{i-1}$ respectively.
     (i) P_{i+1}, P_{i-1} reveal b_{i+1}, b_{i-1} to each other, reconstruct $b := b_{i+1} + b_{i-1}$, and check if $b = 0$. If the check doesn't pass, P_{i+1}, P_{i-1} send abort to the other parties and halts.
     (j) If $s = 1$, the parties go to Step 3; otherwise for $t \in [s]$, the parties define $u_t := p_t(r), v_t := q_t(r), m' := s, s :=$ $\lceil m'/\lambda \rceil$, and P_{i+1}, P_{i-1} define $\mathsf{out}_{i+1} := G_{i+1}(r)$, $\mathsf{out}_{i-1} := G_{i-1}(r)$, and then they go back to Step 2a.
  3. ***Final Check.***
     (a) P_{i+1}, P_{i-1} compute $\mathsf{out}_{i+1} := G_{i+1}(r), u := p_1(r)$ and $\mathsf{out}_{i-1} := G_{i-1}(r), v := q_1(r)$ respectively, and reveal out_{i+1}, u and out_{i-1}, v to each other.
     (b) P_{i+1}, P_{i-1} reconstruct $\mathsf{out} := \mathsf{out}_{i+1} + \mathsf{out}_{i-1}$, and check if $u \cdot v = \mathsf{out}$. If the check doesn't pass, P_{i+1}, P_{i-1} send abort to the other parties and halts.
  4. If a party received abort in any of the previous steps, then it outputs abort and halts.
- **Output:** If a party didn't output abort, then it outputs accept.

under the constraint that $b_{j-1} + b_{j+1} = 0$. Finally, consider the final evaluations $\mathsf{out}_{j-1} = G_{j-1}(r), v = q_1(r)$. Recall that when $s = 1$, the evaluation $q_1(0)$ is set to be a pseudo-random value τ generated by a pseudo-random function. Due to the fact that $q_1(r)$ can be expressed as a linear combination of $q_1(0), \cdots, q_1(\lambda)$, and the assumption that the pseudo-random function is (computationally) indistinguishable from a real random function, it is clear that $v = q_1^{(r)}$ is also (computationally) indistinguishable with the random value $\hat{v}$ in the simulation. Similarly, since $G(X) := p_1(X) \cdot q_1(X)$ when $s = 1$, then the evaluation $G(r)$ in the last round is also pseudo-random, and the evaluation share $\mathsf{out}_{j-1} = G_{j-1}(r)$ is also indistinguishably random. Thus, the messages out_{j-1}, v are indistinguishably random in both executions under the constraint that $\mathsf{out}_{j-1} + \mathsf{out}_{j+1} = u \cdot v$.

Case 2: the prover party P_j is corrupted $(i.e., i = j)$. *In this case, the simulator $\mathcal{S}$ receives from $\mathcal{F}_{vrfy}$ the input of the corrupted prover P_i, i.e., $(x_i^{(\ell)}, x_{i-1}^{(\ell)}, y_i^{(\ell)}, y_{i-1}^{(\ell)}, \rho_i^{(\ell)}, \rho_{i-1}^{(\ell)}, z_i)$ for $\ell \in [m]$. Now $\mathcal{S}$ knows exactly the inputs of the two honest verifiers, and thus can exactly simulate the role of the honest parties by following the protocol.*

- *$\mathcal{S}$ simulates the setup step, preparation step by following the instructions of the honest verifiers in the protocol.*
- *$\mathcal{S}$ simulates the recursion step and the final check. In each round of recursion, $\mathcal{S}$ receives coefficient shares from $\mathcal{A}$, simulates $\mathcal{F}_{coin}$ handing $\mathcal{A}$ a random point r, and follows the protocol to perform the check. $\mathcal{S}$ also follows the instructions to perform the final check. If any check doesn't pass, $\mathcal{S}$ simulates the honest verifiers aborting in the real execution. Otherwise, $\mathcal{S}$ sends* accept *to $\mathcal{F}_{vrfy}$.*

Observe that the only difference between the simulation and the real execution is the event that $\mathcal{F}_{vrfy}$ outputs abort *to the parties but the honest parties simulated by $\mathcal{S}$ output* accept*, i.e., the prover successfully cheated without being caught. Given the assumption that the field $\mathbb{F}_p$ is sufficiently large, we have $m < p$, and thus the above event happens iff the parties pick a bad random point r s.t. $G(X) \neq \sum_{t=1}^{s} p_t(X) \cdot q_t(X)$ but $G(r) = \sum_{t=1}^{s} p_t(r) \cdot q_t(r)$. In each round of iterations except the last one, $G(X)$ is of degree $2\lambda - 1$. By Schwartz-Zipple lemma, the probability of the above event is bounded by $\frac{2\lambda-1}{p-\lambda}$. In the last round of iteration, $G(X)$ is of degree $2\lambda + 1$, and the the probability turns to $\frac{2\lambda+1}{p-\lambda-1}$. For the prover to successfully cheat, this event should happen in one of the iterations of the protocol. There are at most $\lceil \log_\lambda m \rceil$ rounds of iteration, and thus the overall probability for the prover to successfully cheat is bounded by*

$$\frac{(\lceil \log_\lambda m \rceil - 1)(2\lambda - 1)}{p - \lambda} + \frac{2\lambda + 1}{p - \lambda - 1} < \frac{2\lambda(\log_\lambda m + 1) + 1}{p - \lambda - 1}.$$

This is exactly the statistical error allowed in the theorem. This concludes our proof.

D Implementation of secure NN in MP-SPDZ and Choices of Protocols for NN Inference

D.1 Implementation of secure NN Inference

As our contribution is a practically efficient 3PC for binary circuits, together with existing techniques for arithmetic computation and A2B, B2A techniques, we can evaluate mixed circuits that contain both arithmetic computation and binary computation. We first show how secure NN inference is implemented in MP-SPDZ framework.

The MP-SPDZ framework evaluates secure NN in two steps. First, it decomposes the circuit that contains truncation/comparison to mixed circuits with arithmetic operations, binary operations, and A2B, B2A operations. Then these operations are evaluated by suitable protocols. The framework provides several options for both steps. For the first step, we use the option from ABY^3 [35] for exact truncation and comparison. These are known to be the best choices to the best of our knowledge. (E.g., the same comparison protocol is used in SWIFT [32].) For the second step, we also choose the most practical protocol for arithmetic operations [1] and A2B, B2A operations [19, 42]. Note that our improvement is in the computation of binary gates. Using other protocols for arithmetic operations or A2B, B2A operations does not affect our improvement.

D.2 Comparison with Other Protocols for NN Inference

Comparison with SWIFT [32]. In the line of research on secure DNN inference, the state-of-the-art 3PC protocols SWIFT [32] uses probabilistic truncation based on [11] and ABY^3 [35]-style comparison, and apply the distributed zero-knowledge proofs [7] to both arithmetic and binary operations. As we argued in Section 5.2, the probabilistic truncation protocol is insecure. Also, the recent work [17] has argued that using [7] to verify arithmetic operations over rings are not practical. Besides, the implementation of [32] is not public. Due to these reasons, we do not compare with SWIFT.

Comparison with Fantastic Four [17]. The 3PC protocol of [17] also uses probabilistic truncation based on [11], and applies [1] for arithmetic operations as we do and uses the follow-up work [2] of [21] for binary operations. Therefore, our baseline `SW+FLNW17` is very similar to the 3PC protocol of [17] except that we change to use exact truncation protocols and we use [21] rather than [2] for binary computation. As we mentioned above, we notice that in the MP-SPDZ framework, the implementation of [21] is faster than the implementation of [2]. Thus, we choose [21] as our baseline rather than [2].

Field	Operation	Time
$\mathbb{F}_{2^{64}}$	Multiplication	0.76
$\mathbb{F}_p$	Multiplication	0.10
$\mathbb{F}_{2^{64}}$	Inner-product (optimized)	0.09
$\mathbb{F}_p$	Inner-product (optimized)	0.08

Table 5: Time (s) for multiplication and inner-product operations over $\mathbb{F}_{2^{64}}$ and $\mathbb{F}_p$ where $p = 2^{61} - 1$. Multiplications are run 10^8 times and inner-product is of length 10^8.

E Microbenchmarks and Experimental Parameter Setup

E.1 Microbenchmarks for Arithmetic Operations over Different Fields

In the microbenchmark, we test the running time of arithmetic operations over the binary extension field $\mathbb{F}_{2^{64}}$ and the 61-bit Mersenne prime field $\mathbb{F}_p$. We mainly test two kinds of operations: the multiplication operation and the inner-product operation with the optimization described in Section 4.3). We note that the implementation of the operations over $\mathbb{F}_{2^{64}}$ in MP-SPDZ uses the fast multiplication Intel instructions over binary extension fields. Multiplications are run 10^8 times and the inner-product operation is run with length 10^8 as well. Both operations are tested with one single thread.

We show the results in Table 5. As we can see, the multiplication operation over $\mathbb{F}_p$ is about 7.6× faster than that over $\mathbb{F}_{2^{64}}$. Considering the inner-product operation, due to the optimization techniques we adopted, the running time of it over the two fields is indeed very close. Whereas, it still justifies our claim that doing distributed zero-knowledge proofs over the binary extension field $\mathbb{F}_{2^{64}}$ is more expensive than doing them over the Mersenne prime field $\mathbb{F}_p$, as the protocols still require lots of single-multiplication operations other than inner-products.

E.2 Parameter Setup

Parameter Setup for Our Protocol. We follow the analysis in Section 4.4 and choose the parameters as follows:

- The prime field is of size $p = 2^{61} - 1$.
- The batch size is set to be $\texttt{bs} = 640,000$. It means that once all parties have computed bs AND gates, they start to use our verification protocol to check the correctness of this batch of AND gates.
- The compression parameter is set to be $\lambda = 32$ in the first round and $\lambda = 8$ in the following rounds for the optimal performance.

Parameter Setup for `BGIN19`. We choose the parameters for `BGIN19` as follows:

- The extension field is of size 2^{64}.
- The batch size is set to be $\texttt{bs} = 640,000$ as in ours.
- The compression parameter is set to be $\lambda = 8$ in each round, which we found in our experiments that is the optimal parameter setting.

Parameter Setup for `FLNW17`. In the implementation of the protocol `FLNW17` in MP-SPDZ, we may also setup two parameters B and $\texttt{bs}'$ which indicate the bucket size for generating Beaver triples using the cut-and-choose method and how many triples are prepared each time. To obtain the desired efficiency as stated in [21], the implementation requires to set $\texttt{bs}' \geq 10^6$ and it will generate $64 \cdot \texttt{bs}'$ AND triples each time. We choose parameters for `FLNW17` as follows.

- For binary computation, we choose the bucket size $\texttt{B} = 3$.
- Since the circuit size is 6.4×10^7, we choose $\texttt{bs}' = (6.4 \times 10^7)/64 = 10^6$. In this way, the protocol `FLNW17` can prepare all AND triples in one invocation of the preparation phase.