Actively Secure PPML with Soft-Boundary Overflow Detection

Yuntian Chen, Tianpei Lu*, Keyu Ji*, Zhiyuan Ning, Zhanyong Tang, Cong Wang

Abstract

MPC-based privacy-preserving machine learning (PPML) typically operates over finite rings rather than floating-point arithmetic, making arithmetic overflow an inherent risk. Existing overflow-detection techniques generally require the evaluation of nonlinear functions, resulting in substantial computation and communication overhead. We propose Soft Boundary Detection, which reduces boundary verification to probabilistic functions over partitioned intervals, with a cost comparable to that of linear computation. Building on this abstraction, we develop an efficient mechanism for detecting arithmetic overflow in PPML workloads. Moreover, we show that Soft Boundary Detection naturally extends to the malicious-adversary setting. In ring-based computation, it further avoids the need to choose a ring space substantially larger than the actual data range to ensure soundness against malicious behavior. Our framework provides two key benefits: (1) highly efficient arithmetic overflow detection, and (2) active security without requiring ring extension or enlarging the underlying ring space. Experimental results demonstrate that our approach substantially strengthens security while retaining high efficiency, providing a practical solution for robust and actively secure PPML.

1 Introduction

In recent years, secure multi-party computation (MPC) [2, 23, 46] has emerged as a fundamental cryptographic paradigm for privacy-preserving computation, enabling multiple parties to jointly evaluate functions over their private inputs without revealing sensitive information. Driven by privacy-critical applications such as machine learning, data mining, and statistical analysis, a wide range of MPC systems [18, 37, 39, 40, 43] have been developed to support large-scale secure computation [10].

A fundamental challenge underlying these systems is that most efficient MPC protocols operate over finite rings, which are inherently discrete algebraic structures. In contrast, many real-world applications, particularly in privacy-preserving machine learning [7, 8, 37, 39, 43, 45, 51], rely on real-number arithmetic and continuous optimization. This mismatch creates a long-standing gap between application-level numerical computation and the underlying discrete MPC representation. To bridge this gap, real numbers are typically encoded using fixed-point representations over the ring $\mathbb{Z}_{2^\ell}$, where arithmetic is performed modulo 2^ℓ with appropriate scaling factors. Designing efficient and MPC-friendly methods to represent and manipulate fractional values within such discrete structures remains a fundamental research problem.

While fixed-point arithmetic enables practical deployment of machine learning models in MPC, it introduces a subtle but critical issue: all computations are performed modulo 2^ℓ, which leads to *silent modular wraparound*. In particular, when intermediate values exceed the representable range of $\mathbb{Z}_{2^\ell}$, the result is automatically reduced modulo 2^ℓ without any explicit error signal. Unlike conventional computing systems that detect or saturate overflow, computations over finite rings in MPC treat such wraparound as valid arithmetic. This behavior introduces discontinuities in numerical computation that do not exist in plaintext machine learning pipelines, and may significantly affect correctness and stability of intermediate computations.

To address this issue, existing MPC systems rely on *hard-bound overflow detection* mechanisms, which determine whether a computation exceeds the valid range through exact comparison or bit-level decomposition [4, 48]. However, these nonlinear computations are extremely expensive, often incurring costs more than two orders of magnitude higher than those of linear computations.

More broadly, exact thresholding is not always required by the application semantics. Many privacy-preserving workflows can tolerate an explicitly defined transition region in which either decision is admissible, provided that inputs sufficiently far from the boundary are handled deterministically. Examples include fuzzy membership queries [3], uncertainty sampling [30], and conformal prediction [42]. A soft-

* Co-Corresponding author

boundary primitive therefore offers a reusable abstraction for secure threshold decisions with a gray zone: it replaces an exact predicate at the cutoff with a controlled probabilistic decision while retaining deterministic behavior away from the transition region.

The problem becomes substantially more delicate against malicious adversaries. Even without considering overflow checks, achieving malicious security over rings is highly inefficient. Due to the prevalence of non-invertible elements in the ring, conventional multiplication-based computation verification techniques are unable to detect certain errors [13, 17, 31, 35]. One approach is to perform the computation over a ring whose size is far larger than the actual data range [13, 17, 31], thereby making the probability that an error escapes detection negligible. Another approach is to use a polynomial ring [35], which follows the same principle: enlarging the effective ring size to reduce the probability of undetected errors to a negligible level. As a result, in practical evaluations, maliciously secure protocols over rings are typically several times slower than their semi-honest counterparts, making it almost impractical to further incorporate overflow checks in the malicious setting.

Our results. In this work, we address both of the above challenges: overflow detection and malicious security over rings. We propose a new primitive called soft-boundary detection and instantiate it for fixed-point arithmetic overflow. Instead of treating a threshold crossing as a strictly binary event, soft-boundary detection introduces a transition interval around the boundary. Values outside the interval are classified deterministically, while values inside the interval are assigned outcomes according to a probabilistic rule. This creates a smooth transition between the two states, reducing reliance on exact comparison while preserving deterministic semantics outside the boundary region. Although our protocols and evaluation focus on overflow detection, the primitive also captures the approximate range, and fuzzy membership tasks described above whenever their semantics admit a transition region. Moreover, our soft-boundary checking scheme is built upon an extension from the ring to a field, which naturally overcomes the difficulty of correctness verification over rings. This enables efficient maliciously secure fixed-point arithmetic over rings and, in turn, supports efficient actively secure PPML.

Contributions. We summarize our contributions as follows:

- We introduce **soft-boundary detection**, a probabilistic alternative to hard comparison for secure threshold tasks that admit a transition region, and instantiate it as an efficient overflow-detection mechanism that preserves correctness outside that region while incurring only a cost comparable to that of linear computation.
- We design an **actively secure fixed-point MPC framework** over rings, combining batched ring-to-field consistency verification with checks of the auxiliary wraparound data. Unlike conventional maliciously secure fixed-point arithmetic, our framework supports efficient overflow detection without requiring a modulus far larger than the actual computation range.
- We implement our scheme and integrate it into an actively secure PPML framework. Compared with the state of the art, our system achieves better performance.

Performance. Our soft-boundary overflow-detection protocol substantially outperforms hard-boundary detection [11]. For $N = 10^4$ under WAN, it achieves an $11.44\times$ online speedup and reduces online communication by $9.50\times$. Building on this primitive, our actively secure fixed-point matrix-multiplication protocol achieves up to $7.49\times$ and $6.94\times$ speedups over MP-SPDZ [27] and Li *et al.* [31], respectively, while reducing communication by up to $3.42\times$ and $1.24\times$. For end-to-end LAN inference, our protocol achieves speedups of $8.70\times$ and $5.70\times$ over Li *et al.* on BERT-base and GPT-2, respectively, and $4.55\times$ and $4.51\times$ over Mosformer [11].

Paper Organization. The remainder of this paper is organized as follows. Section 2 reviews background on MPC and fixed-point arithmetic. Section 3 introduces the soft-boundary overflow detection primitive. Section 4 presents the actively secure fixed-point MPC framework, and Section 5 evaluates performance. Additional related work is discussed in Appendix A.

2 Preliminaries

Notation. Let $\mathcal{P} := \{P_0, P_1, P_2\}$ be the three parties. We denote the ℓ-bit coordinate integer ring as $\mathbb{Z}_{2^\ell} := \{0, \ldots, 2^\ell - 1\}$. For any ring element $x \in \mathbb{Z}_{2^\ell}$, its ith bit in big-endian serialization is denoted by $x_{(i)}$, for $i \in \{0, \ldots, \ell - 1\}$. We define the most and second-most significant-bit functions as $\mathsf{msb}(x) := x_{(0)}$ and $\mathsf{smsb}(x) := x_{(1)}$, respectively; in particular, $\mathsf{sign}(x) := \mathsf{msb}(x)$. For $q \geq 2$ and $a, b \in \mathbb{Z}_q$, we define the wrap-around bit of their integer sum by $\mathsf{wrap}^q(a,b) := \left\lfloor \frac{a+b}{q} \right\rfloor \in \{0,1\}$. When there is no ambiguity from the context, we omit the subscripts. Let $\Diamond \in \{+, \times\}$ represent a generic arithmetic operator executing element-wise addition, element-wise (Hadamard) multiplication. We use λ to denote the security parameter and let field $\mathbb{F}_p$ size $p > 2^\lambda$ with prime p. Let $\mathcal{R}(x) : \mathbb{Z}_{2^\ell} \to \{0,1\}$ be a random predicate that maps an algebraic input to a binary choice based on a context-dependent probability distribution. We denote by $\eta_{j,k}$ the symmetric cryptographic seed securely shared between P_j and P_k to anchor local pseudorandom generators (PRGs).

Secret Sharing Schemes. We build our multi-party pipelines upon two distinct secret sharing configurations over the ring $\mathbb{Z}_{2^\ell}$.

- Two-party additive secret share ($[\cdot]^{2^\ell}$). A secret value $x \in \mathbb{Z}_{2^\ell}$ is split into an additive tuple $[x] := ([x]_1^{2^\ell}, [x]_2^{2^\ell}) \in \mathbb{Z}_{2^\ell}^2$

such that $x = [x]_1^{2^\ell} + [x]_2^{2^\ell} \pmod{2^\ell}$. The additive shares $[x]_1^{2^\ell}$ and $[x]_2^{2^\ell}$ are held privately by P_1 and P_2, respectively.

- Three-party masked secret share ($\langle\cdot\rangle^{2^\ell}$). A value $x \in \mathbb{Z}_{2^\ell}$ is split into a masked secret share $\langle x\rangle := (m_x, [r_x])$, where $m_x = x - r_x \pmod{2^\ell}$. P_0 holds $([r_x]_1^{2^\ell}, [r_x]_2^{2^\ell})$. P_1 holds $(m_x, [r_x]_1^{2^\ell})$ and P_2 holds $(m_x, [r_x]_2^{2^\ell})$ respectively.

For a vector $\mathcal{L} = (l_0, \ldots, l_{n-1})$, we use $\langle\mathcal{L}\rangle^{2^\ell}$ (resp., $[\mathcal{L}]$) to denote component-wise secret sharing vectors $(\langle l_0\rangle^{2^\ell}, \ldots, \langle l_{n-1}\rangle^{2^\ell})$. For secrets x, y and a public coefficient c, the parties can locally evaluate $\langle x+y\rangle^{2^\ell} = \langle x\rangle^{2^\ell} + \langle y\rangle^{2^\ell}$, $\langle c \cdot x\rangle^{2^\ell} = c \cdot \langle x\rangle^{2^\ell}$, and the affine shift $\langle x + c\rangle^{2^\ell} = (\langle x\rangle_1^{2^\ell} + c, \langle x\rangle_2^{2^\ell})$ without interaction.

Share Generation. Let $\Pi_{\langle\cdot\rangle}$ (or $\Pi_{[\cdot]}$) characterize the sub-protocols designated for establishing the $\langle\cdot\rangle$ ($[\cdot]$) share instances. We use $\Pi_{\langle\cdot\rangle}(x)$ to indicate that a specific party acts as the input dealer for x, whereas an omit-parameter call $\Pi_{\langle\cdot\rangle}$ signifies the joint non-interactive sampling of a randomized shared value. To minimize network ingestion, seed $\eta_{j,k}$ held by P_j and P_k are exploited via local PRGs [47]. The underlying share routing logic is instantiated as follows:

- $[x]^{2^\ell} \leftarrow \Pi_{[\cdot]}^{2^\ell}$: Parties P_0 and P_1 generate $[x]_1 \in \mathbb{Z}_{2^\ell}$ through seed $\eta_{0,1}$, while P_0 and P_2 generate $[x]_2 \in \mathbb{Z}_{2^\ell}$ using seed $\eta_{0,2}$.
- $\langle x\rangle^{2^\ell} \leftarrow \Pi_{\langle\cdot\rangle}^{2^\ell}(x, P_k)$: If $k = 0$: P_0 and P_1 generate $[r_x]_1^{2^\ell}$ using $\eta_{0,1}$. P_0 and P_2 generate $[r_x]_2^{2^\ell}$ using $\eta_{0,2}$. P_0 then computes the masked scalar $m_x = x - [r_x]_1 - [r_x]_2 \pmod{2^\ell}$ and sends it directly to P_1 and P_2. If $k = 1$ or $k = 2$: P_k and P_0 generate $[r_x]_k^{2^\ell}$ using $\eta_{0,k}$. P_k and P_{2-k} generate m_x using $\eta_{k,2-k}$. P_k then computes $[r_x]_{2-k} = x - [r_x]_k - m_x \pmod{2^\ell}$ and sends it directly to P_0 and P_{2-k}.
- $\langle x\rangle^{2^\ell} \leftarrow \Pi_{\langle\cdot\rangle}^{2^\ell}$: All parties compute $[r_x]^{2^\ell} \leftarrow \Pi_{[\cdot]}^{2^\ell}$. P_1 and P_2 subsequently generate $m_x \in \mathbb{Z}_{2^\ell}$ together using their seed $\eta_{1,2}$.

Secure Three Party Computation. In this work, secure three-party computation over rings in the semi-honest setting is modeled by the functionality $\mathcal{F}_{\mathsf{3pc}}$, which takes as input $(\mathsf{input}, f, (\langle x_0\rangle, \ldots, \langle x_{n-1}\rangle))$ from $\mathcal{P}$ and output $\langle z\rangle$, where f is an n-input function to be evaluated, $\langle x_0\rangle, \ldots, \langle x_{n-1}\rangle$ are the secret-shared inputs, and $z = f(x_0, \ldots, x_{n-1})$. Thus, $\langle z\rangle$ is a secret sharing of evaluation result. We instantiate this functionality using the protocol of ASTRA [7]; further details are provided in Appendix C.

We use the functionality $\mathcal{F}_{\mathsf{active3pc}}$ to model actively secure three-party computation over a field. Similar to $\mathcal{F}_{\mathsf{3pc}}$, it takes as input $(\mathsf{input}, f, (\langle x_0\rangle, \ldots, \langle x_{n-1}\rangle))$ from $\mathcal{P}$ and outputs $\langle z\rangle$. In addition, $\mathcal{F}_{\mathsf{active3pc}}$ accepts an abort signal from the adversary; upon receiving this signal, the functionality terminates without producing an output. Since the computation is performed over a field, batched correctness verification against malicious adversaries incurs only negligible overhead. The complete ideal functionality is given in Figure 8 (Appendix B).

Inner-Product Verification. We model inner-product verification over ring by the functionality $\mathcal{F}_{\mathsf{InnerVerify1}}$ in the semi-honest setting. It takes as input $(\mathsf{input}, (\langle\mathbf{X}\rangle, \langle\mathbf{Y}\rangle, \langle Z\rangle))$ from all parties in $\mathcal{P}$, where $\mathbf{X} = (X_1, \ldots, X_n)$ and $\mathbf{Y} = (Y_1, \ldots, Y_n)$ have the same dimension. The functionality computes $\Delta = \sum_{i=1}^n X_i Y_i - Z$ and sends a bit b to every party, where $b = 0$ if $\Delta = 0$ and $b = 1$ otherwise. Thus, a single invocation verifies the relation $\mathbf{X} \cdot \mathbf{Y} = Z \pmod{p}$. Linear verification terms may be incorporated into Z, allowing a batch containing both addition and multiplication relations to use the same interface.

We use $\mathcal{F}_{\mathsf{InnerVerify2}}$ for the corresponding actively secure verification (over field). As with $\mathcal{F}_{\mathsf{active3pc}}$, the adversary may issue an abort signal; if abort $= 1$, the functionality terminates without producing an output. The complete ideal functionality is given in Figure 11, and its concrete realization is described in Appendix F.

Threat Model. We consider three computing parties connected by pairwise secure channels, and a static, probabilistic polynomial-time (PPT) adversary that corrupts at most one party. Our initial soft-boundary detection protocol is designed for the *semi-honest* setting: a corrupted party follows the prescribed protocol but may attempt to infer additional information from its internal state and the messages it receives. Our enhanced protocol provides *active security with abort* in a one-bit-leakage model. Here, the corrupted party is malicious, who may deviate arbitrarily from the protocol. The ideal functionality explicitly permits the adversary to learn at most one leakage bit for an entire batch execution. We formulate and prove both guarantees in the Universal Composability (UC) framework [**?**]. Concretely, for every PPT real-world adversary $\mathcal{A}$ corrupting at most one party, there exists a PPT ideal-world simulator $\mathcal{S}$ such that no PPT environment $\mathcal{Z}$ can distinguish a real execution of the protocol from an ideal execution with the corresponding functionality.

3 Soft-boundary Overflow Detection

In this section, we propose a soft-boundary overflow detection protocol with the secure multiparty computation setting. The concept of a soft boundary can be viewed as a relaxation of the conventional hard boundary. A hard boundary employs a discrete threshold, where values exceeding the threshold are strictly classified as overflows, whereas those below are not. A soft boundary instead introduces a transition interval. Values above this interval are deterministically treated as overflows, while those below are considered safe. For values falling within the interval, the protocol outputs an overflow signal probabilistically, creating a smooth transition between the two states. The soft-boundary mechanism can be effectively applied to scenarios such as fuzzy comparisons and fuzzy membership queries. In the following, we provide the formal definition of soft-boundary detection as illustrated in Figure 1.

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

$\mathcal{F}_{\mathsf{detect}}$ interacts with $\mathcal{P} := \{P_0, P_1, P_2\}$ and adversary $\mathcal{S}$.

- Upon receiving $(\mathsf{Input}, \mathsf{sid}, x_i, y_i)$ from all parties $\mathcal{P}$, $\mathcal{F}_{\mathsf{detect}}$ calculates $x = \sum_{i=0}^{2} x_i, y = \sum_{i=0}^{2} y_i$ over ring $\mathbb{Z}_{2^\ell}$.
- If $x \Diamond y > 2^{\ell-1}$, $\mathcal{F}_{\mathsf{detect}}$ sets $b = 1$;
- If $x \Diamond y < 2^{\ell-2}$, $\mathcal{F}_{\mathsf{detect}}$ sets $b = 0$;
- If $2^{\ell-1} > x \Diamond y > 2^{\ell-2}$, $\mathcal{F}_{\mathsf{detect}}$ samples b from a Bernoulli distribution based on $x \Diamond y$: with probability $\frac{x \Diamond y - 2^{\ell-2}}{2^{\ell-2}}$, $b = 1$; and with probability $1 - \frac{x \Diamond y - 2^{\ell-2}}{2^{\ell-2}}$, $b = 0$;
- $\mathcal{F}_{\mathsf{detect}}$ sends $(\mathsf{output}, \mathsf{sid}, b)$ to all parties.

Figure 1: Soft-boundary Overflow Detection Functionality.

Definition 1. A soft-boundary detection $b \leftarrow \mathcal{G}(x, \mathbb{B})$ is employed to determine whether x exceeds the boundary $\mathbb{B}$, where $\mathbb{B}$ is defined by two thresholds $L < R$. The output is $b = 0$ if $x \le L$, and $b = 1$ if $x \ge R$. If $L < x < R$, the output is $b = \mathcal{R}(x)$, where $\mathcal{R}(x) \in \{0, 1\}$ is a random function.

The definition specifies the desired input–output behavior, but does not yet provide an efficient realization over secret-shared values. In particular, directly deciding which of the three regions contains x would require secure comparisons and would therefore forfeit the main efficiency benefit of a soft boundary. Our construction instead encodes the three-region behavior as a consistency test between a ring computation and a field computation. We develop this construction in three steps. First, we recall when cross-domain consistency exactly characterizes modular wrap-around for addition and multiplication. Second, we show how the required wrap-around information can be recovered from local share bits when the reconstructed value lies in the lower half of the ring. Finally, we deliberately apply this recovery rule beyond that range: the resulting equality test is always satisfied in the safe region, is satisfied with a value-dependent probability in the transition region, and always fails in the overflow region. These three cases realize Definition 1 without evaluating an explicit comparison.

We begin with the basic cross-domain consistency argument. Consider an MPC computation $y = f_{2^\ell}(x_1, \ldots, x_n)$ over $\mathbb{Z}_{2^\ell}$ and the corresponding computation $y' = f_K(x_1, \ldots, x_n)$ over a larger prime field $\mathbb{Z}_p$. For the two operations considered in this work, namely addition and multiplication, agreement between the two representations characterizes whether the integer computation remains below 2^ℓ, as stated next.

Theorem 1. *Let $x_1, x_2, x \in \mathbb{Z}_{2^\ell}$ and $x' \in \mathbb{Z}_p$ for prime $p > 2^\ell$, satisfy $x_1 \Diamond x_2 = x \pmod{2^\ell}$ for $\Diamond \in \{+, \times\}$ and $x_1 \Diamond x_2 = x' \pmod{K}$, then $x = x'$ if and only if $x_1 \Diamond x_2 < 2^\ell$.*

Proof. i. For $x_1 \Diamond x_2 < 2^\ell$, it follows that $x = x'$. It is trivial that the computation will not wrap around in either 2^ℓ or p, thus yielding identical results.

ii. To prove that $x = x'$ implies $x_1 \Diamond x_2 < 2^\ell$, we show that $x = x'$ cannot hold if $x_1 \Diamond x_2 \ge 2^\ell$. Given $x_1 \Diamond x_2 = x \pmod{2^\ell}$ and $x_1 \Diamond x_2 = x' \pmod{p}$, we have the equation $x + a2^\ell - x' - b \cdot p = 0$ for some positive integers a and b. Since $x_1 < 2^\ell$ and $x_2 < 2^\ell$, we have $b < a < 2^\ell$ (considering $\Diamond \in \{+, \times\}$, $x_1 \Diamond x_2 < 2^{2\ell}$). If $x = x'$, the equation simplifies to $a2^\ell = b \cdot p$. Since p is a prime number greater than 2^ℓ, $a2^\ell$ must be a multiple of p (i.e., $p \mid a2^\ell$). This implies that a must be a multiple of p, which contradicts the constraint $a < 2^\ell$. Therefore, there are no non-trivial integer solutions for a and b under the given constraints.

□

Theorem 1 establishes the algebraic basis of our detector: for a single addition or multiplication, overflow can be identified by comparing the ring result with its field counterpart. Realizing this test on secret-shared values, however, requires the parties to lift a ring sharing into the field without reconstructing its secret. The lift must therefore account for the wrap-around introduced when the shares are added modulo 2^ℓ.

The next theorem provides exactly this correction information. When the reconstructed value lies in the lower half of the ring, the wrap-around bit is determined by the most significant bits of the two local shares. This property, previously used by Dong *et al.* [21] and Liu *et al.* [33] in ring-extension protocols for privacy-preserving machine learning, allows the cross-domain test of Theorem 1 to be implemented over secret shares. We state the property formally below.

Theorem 2. *Let $x_1, x_2, x \in \mathbb{Z}_{2^\ell}$ satisfy $x_1 + x_2 = x \pmod{2^\ell}$. If $x < 2^{\ell-1}$, then the equation $x_{1,0} \vee x_{2,0} = b$ holds for $x_1 + x_2 = x + b \cdot 2^\ell$.*

Proof. The condition $x < 2^{\ell-1}$ implies that the most significant bit (MSB) of x is 0. We consider the two possible values of the wrap-around bit $b \in \{0, 1\}$:

- Case 1: $b = 0$. In this case, $x_1 + x_2 = x$. Since $x < 2^{\ell-1}$ and x_1, x_2 are non-negative integers, it must hold that $x_1 \le x < 2^{\ell-1}$ and $x_2 \le x < 2^{\ell-1}$. Therefore, the MSBs of both summands are zero ($x_{1,0} = 0$ and $x_{2,0} = 0$). It follows that $x_{1,0} \vee x_{2,0} = 0 \vee 0 = 0$, which matches b.
- Case 2: $b = 1$. In this case, $x_1 + x_2 = x + 2^\ell$. Since $x \ge 0$, we have $x_1 + x_2 \ge 2^\ell$. We prove $x_{1,0} \vee x_{2,0} = 1$ by contradiction. Suppose $x_{1,0} = 0$ and $x_{2,0} = 0$. This implies $x_1 < 2^{\ell-1}$ and $x_2 < 2^{\ell-1}$. Summing these inequalities yields $x_1 + x_2 < 2^\ell$, which contradicts $x_1 + x_2 \ge 2^\ell$. Thus, at least one of the MSBs must be 1, satisfying $x_{1,0} \vee x_{2,0} = 1$, which matches b.

Thus, the identity $x_{1,0} \vee x_{2,0} = b$ holds whenever $x < 2^{\ell-1}$.

□

Building on Theorem 2, we now instantiate the boundary used by our detector. For the (non-modular) integer result $v = x^{(1)} \Diamond x^{(2)}$, its lower and upper thresholds are $L = 2^{\ell-2}$

and $R = 2^{\ell-1}$, respectively. Hence, $v \le L$ is the *safe region*, $L < v < R$ is the *transition region*, and $v \ge R$ is the *overflow region*; equivalently, the soft-boundary interval is $\mathbb{B} = (L,R) = (2^{\ell-2}, 2^{\ell-1})$. Rather than first checking which region contains v, we apply the local-share correction $\mathsf{msb}(x_1) \vee \mathsf{msb}(x_2)$ unconditionally and compare the resulting field representation with a direct evaluation over $\mathbb{Z}_p$. In the safe region, the correction is exact and the two field values agree. In the transition region, they agree with probability $1-(v-L)/(R-L)$. In the overflow region, they disagree. Consequently, the field-equality bit realizes the soft-boundary output of Definition 1, as formalized in the following theorem.

Theorem 3. *Let $x^{(1)}, x^{(2)} \in \mathbb{Z}_{2^\ell}$ be two values such that $x^{(i)} < 2^{\ell-1}$ for $i \in \{1,2\}$, and let $[x^{(i)}] = (x_1^{(i)}, x_2^{(i)})$ be their arithmetic secret sharings. Let $[y] = (y_1, y_2)$ be the output of a secure protocol Π for the operation $y = x^{(1)} \lozenge x^{(2)} \pmod{2^\ell}$, where $\lozenge \in \{+, \times\}$. Define the transformed values $\hat{x}^{(i)}$ and $\hat{y}$ in the field $\mathbb{Z}_p$ (where p is a prime) as follows:*

$$\hat{x}^{(i)} = x_1^{(i)} + x_2^{(i)} - \mathsf{msb}(x_1^{(i)}) \vee \mathsf{msb}(x_2^{(i)}) \times 2^\ell \pmod{p}$$

$$\begin{aligned}\hat{y} = {} & y_1 \pmod{2^{\ell-1}} + y_2 \pmod{2^{\ell-1}} \\ & - \mathsf{smsb}(y_1) \vee \mathsf{smsb}(y_2) \times 2^{\ell-1} \pmod{p}\end{aligned}$$

where $\mathsf{smsb}(\cdot)$ denotes the second most significant bit.

Then, the following properties hold for the relationship between $\hat{y}$ and $\hat{x}^{(1)} \lozenge \hat{x}^{(2)} \pmod{p}$.

- *Deterministic Inequality: If $\hat{x}^{(1)} \lozenge \hat{x}^{(2)} \ge 2^{\ell-1}$, then $\hat{y} \neq \hat{x}^{(1)} \lozenge \hat{x}^{(2)} \pmod{p}$;*
- *Probabilistic Equality: If $2^{\ell-1} > \hat{x}^{(1)} \lozenge \hat{x}^{(2)} \ge 2^{\ell-2}$, then $\hat{y} = \hat{x}^{(1)} \lozenge \hat{x}^{(2)} \pmod{p}$ holds with probability $1 - (\hat{x}^{(1)} \lozenge \hat{x}^{(2)} - 2^{\ell-2} - 1)/2^{\ell-2}$;*
- *Deterministic Equality: If $\hat{x}^{(1)} \lozenge \hat{x}^{(2)} < 2^{\ell-2}$, then $\hat{y} = \hat{x}^{(1)} \lozenge \hat{x}^{(2)} \pmod{p}$.*

Proof. We consider the following three cases:

- For $\hat{x}^{(1)} \lozenge \hat{x}^{(2)} < 2^{\ell-2}$, the equality $\hat{y} = \hat{x}^{(1)} \lozenge \hat{x}^{(2)} \pmod{p}$ holds evidently according to Theorem 2;
- For the case where $\hat{x}^{(1)} \lozenge \hat{x}^{(2)} > 2^{\ell-1}$, the actual wrap-around bit should be computed as $\mathsf{smsb}(y_1) \wedge \mathsf{smsb}(y_2)$ (a conclusion that can be drawn through an analysis similar to Theorem 2). When the values of $\mathsf{smsb}(y_1) \wedge \mathsf{smsb}(y_2)$ and $\mathsf{smsb}(y_1) \vee \mathsf{smsb}(y_2)$ coincide, the wrap-around bit is computed correctly. In this case, since $\hat{y}$ discards the leading bit of $\hat{x}^{(1)} \lozenge \hat{x}^{(2)}$, it follows evidently that $\hat{y} \neq \hat{x}^{(1)} \lozenge \hat{x}^{(2)} \pmod{p}$. Conversely, the expressions $\mathsf{msb}(y_1) \wedge \mathsf{msb}(y_2)$ and $\mathsf{msb}(y_1) \vee \mathsf{msb}(y_2)$ diverge only when the pair $(\mathsf{msb}(y_1), \mathsf{msb}(y_2))$ equals $(0,1)$ or $(1,0)$. In these instances, the term originally intended to subtract $0 \cdot 2^\ell$ becomes a subtraction of $1 \cdot 2^\ell$. This discrepancy leads to
$$\hat{y} = \hat{x}^{(1)} \lozenge \hat{x}^{(2)} - 2^{\ell-1} - 2^\ell \pmod{p}$$
For the expression to equal $\hat{x}^{(1)} \lozenge \hat{x}^{(2)}$, it is necessary and sufficient that $-2^{\ell-1} - 2^\ell \equiv 0 \pmod{p}$. However, this contradicts the given conditions. Consequently, it holds that $\hat{y} \neq \hat{x}^{(1)} \lozenge \hat{x}^{(2)} \pmod{p}$;
- For the case where $2^{\ell-1} > \hat{x}^{(1)} \lozenge \hat{x}^{(2)} > 2^{\ell-2}$, the accuracy of the wrap-around bit calculation depends on the consistency of the MSB operations. If $\mathsf{smsb}(y_1) \wedge \mathsf{smsb}(y_2)$ equals $\mathsf{smsb}(y_1) \vee \mathsf{smsb}(y_2)$, the wrap-around bit is computed correctly, yielding $\hat{y} = \hat{x}^{(1)} \lozenge \hat{x}^{(2)} \pmod{p}$. Conversely, if $\mathsf{smsb}(y_1) \wedge \mathsf{smsb}(y_2)$ and $\mathsf{smsb}(y_1) \vee \mathsf{smsb}(y_2)$ diverge, an analysis analogous to the previous case shows that $\hat{y} = \hat{x}^{(1)} \lozenge \hat{x}^{(2)} - 2^\ell \pmod{p}$. Since $-2^\ell \not\equiv 0 \pmod{p}$, it follows that $\hat{y} \neq \hat{x}^{(1)} \lozenge \hat{x}^{(2)} \pmod{p}$. It remains to compute the probability of the two cases above. Since $y_1 \bmod 2^{\ell-1}$ is uniformly distributed, $\hat{y} = \hat{x}^{(1)} \lozenge \hat{x}^{(2)}$ holds iff $\mathsf{smsb}(y_1) = \mathsf{smsb}(y_2)$. For $2^{\ell-2} \le \hat{x}^{(1)} \lozenge \hat{x}^{(2)} < 2^{\ell-1}$, this occurs when
$$\hat{x}^{(1)} \lozenge \hat{x}^{(2)} - 2^{\ell-2} < y_1 \bmod 2^{\ell-1} < 2^{\ell-2}$$
or
$$\hat{x}^{(1)} \lozenge \hat{x}^{(2)} < y_1 \bmod 2^{\ell-1} < 2^{\ell-1}.$$
Each interval contains $2^{\ell-1} - v - 1$ valid values. Therefore,
$$\Pr[\hat{y} = v] = 1 - \frac{v - 2^{\ell-2} + 1}{2^{\ell-2}}.$$

□

Theorem 3 reduces soft-boundary detection to an equality test over $\mathbb{Z}_p$. Figure 2 instantiates this reduction as the protocol Π_{detect}. The parties first evaluate $z = x \lozenge y$ over $\mathbb{Z}_{2^\ell}$. They then derive, for each of x, y, and z, three correction bits: $b^{(1)}$ is secret-shared by P_0 from the relevant bit of the mask, $b^{(2)}$ is obtained locally from the corresponding bit of the public masked value, and $b^{(3)}$ accounts for wrap-around between the mask shares. Applying
$$f(a, b^{(1)}, b^{(2)}, b^{(3)}) = a - 2^\ell (b^{(1)} + b^{(2)} - b^{(1)} b^{(2)} + b^{(3)})$$
to these values lifts the ring sharings to field sharings $\langle \hat{x} \rangle^p$, $\langle \hat{y} \rangle^p$, and $\langle \hat{z} \rangle^p$. In parallel, the parties compute $\langle z' \rangle^p = \langle \hat{x} \rangle^p \lozenge \langle \hat{y} \rangle^p$ directly in the field. Finally, they invoke $\mathcal{F}_{\mathsf{eq}}[K]$ on $\langle z' \rangle^p$ and $\langle \hat{z} \rangle^p$. Equality yields $b = 0$, whereas inequality yields $b = 1$; by Theorem 3, this bit has exactly the soft-boundary distribution of Definition 1. The same construction extends to vector operations, including multidimensional inner products, as described in Appendix G.

We show the security of Π_{detect} with the following theorem.

Theorem 4. *The protocol Π_{detect} in Fig. 2 UC-realizes $\mathcal{F}_{\mathsf{detect}}$ in Fig. 1 in $\{\mathcal{F}_{\mathsf{3pc}}, \mathcal{F}_{\mathsf{eq}}\}$-hybrid model against a semi-honest adversary who can statically corrupted up to 1 party.*

The proof of Theorem 4 is provided in Appendix M.

Handling Computations between Secret Shares and Plaintexts. We can generalize the soft-boundary check to mixed

Protocol $\Pi_{\mathsf{detect}}(\Diamond, \langle x\rangle^{2^\ell}, \langle y\rangle^{2^\ell})$

Input : $\langle x\rangle^{2^\ell}$, $\langle y\rangle^{2^\ell}$;
Output : a soft-boundary interval detection result b, which holds $b \leftarrow \mathcal{G}(x \Diamond y, \mathbb{B} = (2^{\ell-2}, 2^{\ell-1}))$.
Execution:

- All parties send $(\mathsf{input}, \Diamond, (\langle x\rangle^{2^\ell}, \langle y\rangle^{2^\ell}))$ to $\mathcal{F}_{\mathsf{3pc}}$ and receive $\langle z\rangle^{2^\ell}$.
- All parties invoke $\langle b_x^{(1)}\rangle^p \leftarrow \Pi^p_{\langle\cdot\rangle}(\mathsf{msb}(r_x), P_0)$, $\langle b_y^{(1)}\rangle^p \leftarrow \Pi^p_{\langle\cdot\rangle}(\mathsf{msb}(r_y), P_0)$ and $\langle b_z^{(1)}\rangle^p \leftarrow \Pi^p_{\langle\cdot\rangle}(\mathsf{smsb}(r_z), P_0)$;
- All parties invoke $\langle b_x^{(3)}\rangle^p \leftarrow \Pi^p_{\langle\cdot\rangle}(\mathsf{wrap}^{2^\ell}([r_x]_1, [r_x]_2), P_0)$, $\langle b_y^{(3)}\rangle^p \leftarrow \Pi^p_{\langle\cdot\rangle}(\mathsf{wrap}^{2^\ell}([r_y]_1, [r_y]_2), P_0)$ and $\langle b_z^{(3)}\rangle^p \leftarrow \Pi^p_{\langle\cdot\rangle}(\mathsf{wrap}^{2^{\ell-1}}([r_z]_1 \mod 2^{\ell-1}, [r_z]_2 \mod 2^{\ell-1}), P_0)$;
- All parties set $\langle b_x^{(2)}\rangle^p := (\mathsf{msb}(m_x), [0] := (0,0))$, $\langle b_y^{(2)}\rangle^p := (\mathsf{msb}(m_y), [0] := (0,0))$ and $\langle b_z^{(2)}\rangle^p := (\mathsf{smsb}(m_z), [0] := (0,0))$;
- All parties set $\langle \tilde{z}\rangle_j^p = \langle z\rangle_j^{2^\ell} \pmod{2^{\ell-1}}$, $\langle \tilde{x}\rangle_j^p = \langle x\rangle_j^{2^\ell}$ and $\langle \tilde{y}\rangle_j^p = \langle y\rangle_j^{2^\ell}$;
- All parties set $f(a,b,c,d) = a - 2^\ell \times (b + c - b \times c + d)$;
- All parties send $(\mathsf{input}, f, (\langle \tilde{x}\rangle^p, \langle b_x^{(1)}\rangle, \langle b_x^{(2)}\rangle, \langle b_x^{(3)}\rangle))$ to $\mathcal{F}_{\mathsf{3pc}}$, and receive $\langle \hat{x}\rangle^p$;
- All parties send $(\mathsf{input}, f, (\langle \tilde{y}\rangle^p, \langle b_y^{(1)}\rangle, \langle b_y^{(2)}\rangle, \langle b_y^{(3)}\rangle))$ to $\mathcal{F}_{\mathsf{3pc}}$, and receive $\langle \hat{y}\rangle^p$;
- All parties send $(\mathsf{input}, f, (\langle \tilde{z}\rangle^p, \langle b_z^{(1)}\rangle, \langle b_z^{(2)}\rangle, \langle b_z^{(3)}\rangle))$ to $\mathcal{F}_{\mathsf{3pc}}$, and receive $\langle \hat{z}\rangle^p$;
- All parties send $(\mathsf{input}, \Diamond, (\langle \hat{x}\rangle^p, \langle \hat{y}\rangle^p))$ to $\mathcal{F}_{\mathsf{3pc}}$, and receive $\langle z'\rangle^p$;
- All parties send $\langle z'\rangle^p$ and $\langle \hat{z}\rangle^p$ to $\mathcal{F}_{\mathsf{eq}}[K]$ to verify $z' = \hat{z}$.

Figure 2: 3PC Soft-boundary Overflow Detection Protocol.

share-plaintext computations by substituting one of the secret-shared variables in the aforementioned protocol with a public plaintext. It is straightforward to verify that this modification does not compromise the correctness of the protocol. This follows from the fact that, in Theorem 3, when the plaintext value of $\hat{x}$ is directly involved in the computation, the need for eliminating the wrap-around bit is bypassed, yet the same mathematical properties are preserved. By an abuse of notation, we reuse $\Pi_{\mathsf{detect}}(\Diamond, \langle x\rangle^{2^\ell}, y)$ to denote the aforementioned procedure for mixed computations of secret shares and plaintexts.

Two-Party Variant. The construction above is instantiated directly in the 3PC replicated-secret-sharing setting. For completeness, we also provide a 2PC variant based on additive secret sharing in Appendix D; its full specification appears in Figure 9. The two variants implement the same ring-to-field consistency test from Theorem 3, but differ in how they derive the wrap-around correction required to lift a ring sharing into $\mathbb{Z}_p$.

3.1 A Soft-boundary Batch Detection Scheme

We observe that the Soft-boundary Overflow Detection requires the input variable to satisfy $x^{(i)} < 2^{\ell-1}$. Although this condition may be violated during computation, it can be addressed by performing soft-boundary detection at each intermediate step. Specifically, since the protocol deterministically outputs 1 whenever $x^{(1)} \Diamond x^{(2)} \geq 2^{\ell-1}$, using this output as an abort condition for overflows ensures the soundness of the overall overflow detection mechanism. For the entire computational circuit, soft-boundary overflow detection for all gates can be performed in batches.

Theorem 5 (Probability of Batch Soft-boundary Overflow Detection)**.** *Consider N arithmetic gates indexed by $i \in \{1,\ldots,N\}$. Gate i has inputs $x_i, y_i < 2^{\ell-1}$, operator $\Diamond_i \in \{+,\times\}$, and non-modular integer result $z_i = x_i \Diamond_i y_i$. Let B_i be the output of an invocation of Π_{detect} on gate i; equivalently, $B_i \leftarrow \mathcal{G}(z_i, \mathbb{B})$ for $\mathbb{B} = (2^{\ell-2}, 2^{\ell-1})$ with the transition probability established in Theorem 3. Suppose that the random choices used to generate the bits $\{B_i\}_{i=1}^N$ are mutually independent. The gates are partitioned into three disjoint sets based on their intermediate results:*

- ***Safe Set*** *$\mathcal{S}$: Contains a gates where $z_i \leq 2^{\ell-2}$.*
- ***Probabilistic Set*** *$\mathcal{P}$: Contains b gates where $z_i \in (2^{\ell-2}, 2^{\ell-1})$.*
- ***Overflow Set*** *O: Contains c gates where $z_i \geq 2^{\ell-1}$.*

Let $B_{\mathsf{batch}} = \bigvee_{i=1}^N B_i$. Then

$$\Pr[B_{\mathsf{batch}} = 1] = \begin{cases} 1, & c > 0, \\ 1 - \prod_{i\in\mathcal{P}} \dfrac{2^{\ell-1} - z_i + 1}{2^{\ell-2}}, & c = 0 \text{ and } b > 0, \\ 0, & b = c = 0. \end{cases} \tag{1}$$

Proof. The event $B_{\mathsf{batch}} = 0$ occurs if and only if $B_i = 0$ for every gate. By the independence assumption,

$$\Pr[B_{\mathsf{batch}} = 0] = \prod_{i=1}^{N} \Pr[B_i = 0]. \tag{2}$$

By Definition 1 and Theorem 3, $\Pr[B_i = 0] = 1$ for $i \in \mathcal{S}$, $\Pr[B_i = 0] = 0$ for $i \in O$, and

$$\Pr[B_i = 0] = 1 - \frac{z_i - 2^{\ell-2} - 1}{2^{\ell-2}} = \frac{2^{\ell-1} - z_i + 1}{2^{\ell-2}} \tag{3}$$

for $i \in \mathcal{P}$. If $c > 0$, the product therefore contains a zero factor and $\Pr[B_{\mathsf{batch}} = 1] = 1$. If no overflow index exists, the gates in $\mathcal{S}$ contribute only unit factors, leaving the product over $\mathcal{P}$. Taking its complement gives the second case. If $b = c = 0$, every gate belongs to $\mathcal{S}$ and the aggregate output is zero. □

Theorem 5 characterizes the target distribution obtained by taking the logical OR of the per-gate detection bits. Figure 10 (in Appendix E) presents the corresponding batch protocol $\Pi_{\text{batch-detect}}$, which takes as input N tuples $\{(\diamondsuit_i, \langle x_i \rangle^{2^\ell}, \langle y_i \rangle^{2^\ell})\}_{i=1}^N$. Rather than invoking a separate field-equality test for each tuple, the protocol aggregates the resulting field relations into a single randomized verification instance.

Batch Verification via Inner Product $\mathcal{F}_{\text{InnerVerify1}}$. Under the batch verification framework, each tuple $\{(\diamond_i, x_i, y_i, z_i)\}_{i=1}^n$ undergoes a field extension to $\mathbb{F}_p$ as defined in Π_{detect}. To optimize performance, rather than invoking $\mathcal{F}_{\text{3pc}}$ and $\mathcal{F}_{\text{eq}}$ for each tuple individually, we aggregate all verification instances into a single inner product identity.

The verification is constructed by initializing two vectors $\mathbf{X} = \emptyset, \mathbf{Y} = \emptyset$ as empty sets and a scalar $Z = 0 \in \mathbb{F}_p$. For each instance $i \in [n]$, we sample a random weight $r_i \xleftarrow{\$} \mathbb{F}_p$ and update the verification parameters based on the operation type: 1) Addition ($\diamond = +$): We accumulate the weighted difference into the scalar, $Z \leftarrow Z + r_i(z_i - x_i - y_i) \pmod p$; 2) Multiplication ($\diamond = \times$): We append $r_i \cdot x_i$ to $\mathbf{X}$ and y_i to $\mathbf{Y}$, and update the scalar $Z \leftarrow Z + r_i \cdot z_i \pmod p$. Finally, the entire batch is verified by checking the inner product relationship using $\mathcal{F}_{\text{InnerVerify1}}$: $\mathbf{X} \cdot \mathbf{Y} = Z \pmod p$. This approach reduces the overhead of n individual checks to a single inner-product verification.

We can directly apply the MSS inner product protocol [7] over $\mathbb{Z}_p$ to realize $\mathcal{F}_{\text{InnerVerify1}}$ (See Appendix. C for detail). It verifies the identity $\mathbf{X} \cdot \mathbf{Y} - Z = 0$. The communication complexity of this verification is highly efficient and remains constant, regardless of the dimension of the inner product. It requires only $3 \log K$ bits for the inner product computation and an additional $6 \log K$ bits for the zero-opening (Open-to-Zero) procedure.

3.2 Soft-boundary Detection Scheme against Malicious Adversary

To achieve security against malicious adversaries, we formally define the security goal of our soft-boundary overflow detection via an ideal functionality $\mathcal{F}_{\text{active-batch-detect}}$ with abort (Figure 3), which is instantiated by transforming the detection verification into multiplication verification over a finite field $\mathbb{Z}_p$. Under this framework, the protocol takes secret-shared triples as input, outputting a valid multiplication triple if and only if no overflow occurs, thereby allowing any invalid triples caused by overflows to be intercepted.

In particular, the interactive workflow of our semi-honest secure soft-boundary protocol is partitioned into two distinct phases. In the first phase, P_0 secret-shares the auxiliary materials required for computing wrap-around values. In the second phase, the parties perform multiplication to derive intermediate values such as $\hat{x}$ and $\hat{y}$, followed by a verification

Functionality $\mathcal{F}_{\text{active-batch-detect}}$

$\mathcal{F}_{\text{active-batch-detect}}$ interacts with $\mathcal{P} := \{P_0, P_1, P_2\}$ and adversary $\mathcal{S}$.

- Upon receiving $(\mathsf{Input}, \mathsf{sid}, \{(\diamondsuit_i, x_{i,j}, y_{i,j})\}_{i=1}^N)$ from every party P_j, $\mathcal{F}_{\text{active-batch-detect}}$ reconstructs x_i and y_i over $\mathbb{Z}_{2^\ell}$ and sets $v_i = x_i \diamondsuit_i y_i$ for every $i \in \{1, \ldots, N\}$.
- Upon receiving $(\mathsf{leak}, \mathsf{sid}, f)$ from adversary $\mathcal{S}$, $\mathcal{F}_{\text{active-batch-detect}}$ sends $(\mathsf{abort}, \mathsf{sid})$ to all parties and halts if $f(\{(\diamondsuit_i, x_i, y_i)\}_{i=1}^N) = 1$.
- For each $i \in \{1, \ldots, N\}$, $\mathcal{F}_{\text{active-batch-detect}}$ sets $b_i = 0$ if $v_i \le 2^{\ell-2}$, sets $b_i = 1$ if $v_i \ge 2^{\ell-1}$, and otherwise samples b_i independently with $\Pr[b_i = 1] = (v_i - 2^{\ell-2})/2^{\ell-2}$.
- $\mathcal{F}_{\text{active-batch-detect}}$ sets $b = \bigvee_{i=1}^N b_i$.
- $\mathcal{F}_{\text{active-batch-detect}}$ sends $(\mathsf{output}, \mathsf{sid}, b)$ to all parties.

Figure 3: Batch Soft-boundary Overflow Detection Functionality with Abort.

process over $\mathbb{F}_p$ to ensure the multiplication triples satisfy the required algebraic relationships. For the second phase, any standard maliciously secure multiplication verification protocol can be directly applied. Regarding the first phase, a malicious P_0 could distribute incorrect shares to compromise the protocol's correctness. To mitigate this threat, we introduce a dual execution mechanism. Specifically, we require P_1 to replicate the role of P_0 by generating an independent set of auxiliary materials. The protocol is then executed redundantly using both sets of materials.

Specifically, let $m_{x^{(i)}}$ be the masked value and $[r_{x^{(i)}}]_1, [r_{x^{(i)}}]_2$ be the secret shares of the mask generated by the i-th dealer. For a semi-honest execution, the following algebraic identity must hold regardless of the distribution of shares:

$$\begin{aligned} &\mathsf{wrap}(m_{x^{(i)}}, [r_{x^{(i)}}]_1 + [r_{x^{(i)}}]_2) + \mathsf{wrap}([r_{x^{(i)}}]_1, [r_{x^{(i)}}]_2) \\ &= \mathsf{wrap}(m_{x^{(i)}}, [r_{x^{(i)}}]_1) + \mathsf{wrap}(m_{x^{(i)}} + [r_{x^{(i)}}]_1, [r_{x^{(i)}}]_2) \end{aligned} \tag{4}$$

This identity stems from the fact that the total number of overflows in a three-operand addition $(m + r_1 + r_2)$ is invariant to the order of grouping.

Our proposed actively secure protocol is illustrated in Figures 4. Due to the integration of the dual execution mechanism, an adversary may launch a selective-failure attack, potentially resulting in a one-bit leakage. For instance, in a selective-failure attack, a malicious P_0 might provide fraudulent shares of $\mathsf{msb}(r_x)$. The adversary then leaks information about $\mathsf{msb}(m_x)$ by monitoring whether the inconsistency leads to a protocol abort during the verification phase. However, by employing the aforementioned batch verification, this leakage is constrained to a single bit for the entire batch.

Actively Secure Inner-Product Verification. We use the functionality $\mathcal{F}_{\text{InnerVerify2}}$ defined in the preliminaries to verify the field relations generated by the protocol. Specifically, one invocation checks the auxiliary correction terms used

Protocol $\Pi^{N}_{\mathsf{active\text{-}batch\text{-}detect}}(\langle\Diamond\rangle, \langle x_i\rangle^{2^\ell}, \langle y_i\rangle^{2^\ell})$

Input : N tuples $\{\langle x_i\rangle^{2^\ell}, \langle y_i\rangle^{2^\ell}\}_{i\in[N]}$

Output : a soft-boundary overflow detection result b, which holds $b \leftarrow \bigvee_{i=0}^{N-1} \mathcal{G}(x_i \Diamond y_i, \mathbb{B} = (2^{\ell-2}, 2^{\ell-1}))$.

Execution:

- All parties set $\mathbf{A} = \langle\rangle$, $\mathbf{B} = \langle\rangle$ and $C = \langle 0\rangle$;
- For each $i \in [N]$, all parties do
  - All parties send $(\mathsf{input}, \Diamond, (\langle x_i\rangle^{2^\ell}, \langle y_i\rangle^{2^\ell}))$ to $\mathcal{F}_{\mathsf{3pc}}$ and receive $\langle z_i\rangle^{2^l}$.
  - P_0 invokes $\langle b^{(1)}_{x_i}\rangle^p \leftarrow \Pi^p_{\langle\cdot\rangle}(\mathsf{msb}(r_{x_i}))$, $\langle b^{(1)}_{y_i}\rangle^p \leftarrow \Pi^p_{\langle\cdot\rangle}(\mathsf{msb}(r_{y_i}))$ and $\langle b^{(1)}_{z_i}\rangle^p \leftarrow \Pi^p_{\langle\cdot\rangle}(\mathsf{smsb}(r_{z_i}))$;
  - P_0 invokes $\langle b^{(3)}_{x_i}\rangle^p \leftarrow \Pi^p_{\langle\cdot\rangle}(\mathsf{wrap}([r_{x_i}]_1, [r_{x_i}]_2))$, $\langle b^{(3)}_{y_i}\rangle^p \leftarrow \Pi^p_{\langle\cdot\rangle}(\mathsf{wrap}([r_{y_i}]_1, [r_{y_i}]_2))$ and $\langle b^{(3)}_{z_i}\rangle^p \leftarrow \Pi^p_{\langle\cdot\rangle}(\mathsf{wrap}^{2^{\ell-1}}([r_{z_i}]_1, [r_{z_i}]_2))$;
  - P_1 invokes $\langle c^{(1)}_{x_i}\rangle^p \leftarrow \Pi^p_{\langle\cdot\rangle}(\mathsf{msb}([r_{x_i}]_1 + m_{x_i}))$, $\langle c^{(1)}_{y_i}\rangle^p \leftarrow \Pi^p_{\langle\cdot\rangle}(\mathsf{msb}([r_{y_i}]_1 + m_{y_i}))$ and $\langle c^{(1)}_{z_i}\rangle^p \leftarrow \Pi^p_{\langle\cdot\rangle}(\mathsf{smsb}([r_{z_i}]_1 + m_{z_i}))$;
  - P_1 invokes $\langle c^{(3)}_{x_i}\rangle^p \leftarrow \Pi^p_{\langle\cdot\rangle}(\mathsf{wrap}([r_{x_i}]_1, m_{x_i}))$, $\langle c^{(3)}_{y_i}\rangle^p \leftarrow \Pi^p_{\langle\cdot\rangle}(\mathsf{wrap}([r_{y_i}]_1, m_{y_i}))$ and $\langle c^{(3)}_{z_i}\rangle^p \leftarrow \Pi^p_{\langle\cdot\rangle}(\mathsf{wrap}^{2^{\ell-1}}([r_{z_i}]_1, m_{z_i}))$;
  - All parties set $\langle b^{(2)}_{x_i}\rangle^p := (\mathsf{msb}(m_{x_i}), [0] := (0,0))$, $\langle b^{(2)}_{y_i}\rangle^p := (\mathsf{msb}(m_{y_i}), [0] := (0,0))$ and $\langle b^{(2)}_{z_i}\rangle^p := (\mathsf{smsb}(m_{z_i}), [0] := (0,0))$;
  - All parties set $\langle c^{(2)}_{x_i}\rangle^p := (0, [\mathsf{msb}([r_{x_i}]_1)] := (\mathsf{msb}([r_{x_i}]_1), 0))$, $\langle c^{(2)}_{y_i}\rangle^p := (0, [\mathsf{msb}([r_{y_i}]_1)] := (\mathsf{msb}([r_{y_i}]_1), 0))$ and $\langle c^{(2)}_{z_i}\rangle^p := (0, [\mathsf{smsb}([r_{z_i}]_1)] := (\mathsf{smsb}([r_{z_i}]_1, 0))$;
  - All parties set $\langle \hat{z}_i\rangle^p_j = \langle z_i\rangle^{2^\ell}_j \pmod{2^{\ell-1}}$, $\langle \hat{x}_i\rangle^p_j = \langle x_i\rangle^{2^\ell}_j$ and $\langle \hat{y}_i\rangle^p_j = \langle y_i\rangle^{2^\ell}_j$;
  - All parties set $f(a,b,c,d) = a - 2^\ell \times (b + c - b \times c + d)$;
  - All parties send $(\mathsf{input}, f, \langle \hat{x}_i\rangle^p, \langle b^{(1)}_{x_i}\rangle, \langle b^{(2)}_{x_i}\rangle, \langle b^{(3)}_{x_i}\rangle)$, $(\mathsf{input}, f, \langle \hat{y}_i\rangle^p, \langle b^{(1)}_{y_i}\rangle, \langle b^{(2)}_{y_i}\rangle, \langle b^{(3)}_{y_i}\rangle)$, and $(\mathsf{input}, f, \langle \hat{z}_i\rangle^p, \langle b^{(1)}_{z_i}\rangle, \langle b^{(2)}_{z_i}\rangle, \langle b^{(3)}_{z_i}\rangle)$ to $\mathcal{F}_{\mathsf{active3pc}}$, and reveive $\langle \hat{x}_i\rangle^p$, $\langle \hat{y}_i\rangle^p$, and $\langle \hat{z}_i\rangle^p$, respectively.
- All parties compute $\langle r\rangle \leftarrow \Pi_{\langle\cdot\rangle}$ and open r.
- All parties set $\langle Z\rangle = \langle 0\rangle$, $\mathbf{X} = (), \mathbf{Y} = ()$;
- For each $i \in [N]$, all parties do
  - If $\Diamond = +$, set $\langle Z\rangle \leftarrow \langle Z\rangle^p + r^i(\langle z_i\rangle^p - \langle x_i\rangle^p - \langle y_i\rangle^p)$;
  - If $\Diamond = \times$, set $\mathbf{X} \leftarrow \mathbf{X}||(r^i \cdot \langle x_i\rangle^p)$, $\mathbf{Y} \leftarrow \mathbf{Y}||(\langle y_i\rangle)$, $\langle Z\rangle^p \leftarrow \langle Z\rangle^p + r^i \cdot \langle \hat{z}_i\rangle^p$;
  - $\mathbf{A} \leftarrow \mathbf{A}||(r^{N+i} \cdot \langle b^{(1)}_{x_i}\rangle^p, r^{N+i} \cdot \langle c^{(1)}_{x_i}\rangle^p, r^{2N+i} \cdot \langle b^{(1)}_{y_i}\rangle^p, r^{2N+i} \cdot \langle c^{(1)}_{y_i}\rangle^p, r^{3N+i} \cdot \langle b^{(1)}_{z_i}\rangle^p, r^{3N+i} \cdot \langle c^{(1)}_{z_i}\rangle^p)$;
  - $\mathbf{B} \leftarrow \mathbf{B}||(\langle b^{(2)}_{x_i}\rangle^p, -\langle c^{(2)}_{x_i}\rangle^p, \langle b^{(2)}_{y_i}\rangle^p, -\langle c^{(2)}_{y_i}\rangle^p, \langle b^{(2)}_{z_i}\rangle^p, -\langle c^{(2)}_{z_i}\rangle^p)$;
  - $\langle C\rangle^p \leftarrow \langle C\rangle^p + r^{N+i} \cdot \big(\langle b^{(1)}_{x_i}\rangle^p + \langle b^{(2)}_{x_i}\rangle^p + \langle b^{(3)}_{x_i}\rangle^p - \langle c^{(1)}_{x_i}\rangle^p - \langle c^{(2)}_{x_i}\rangle^p - \langle c^{(3)}_{x_i}\rangle^p\big) + r^{2N+i} \cdot \big(\langle b^{(1)}_{y_i}\rangle^p + \langle b^{(2)}_{y_i}\rangle^p + \langle b^{(3)}_{y_i}\rangle^p - \langle c^{(1)}_{y_i}\rangle^p - \langle c^{(2)}_{y_i}\rangle^p - \langle c^{(3)}_{y_i}\rangle^p\big) + r^{3N+i} \cdot \big(\langle b^{(1)}_{z_i}\rangle^p + \langle b^{(2)}_{z_i}\rangle^p + \langle b^{(3)}_{z_i}\rangle^p - \langle c^{(1)}_{z_i}\rangle^p - \langle c^{(2)}_{z_i}\rangle^p - \langle c^{(3)}_{z_i}\rangle^p\big)$
- All parties send $\mathbf{A}, \mathbf{B}, C$ to $\mathcal{F}_{\mathsf{InnerVerify2}}$ and receive $d = \{0,1\}$.
- All parties abort if $d = 1$;
- All parties send $\mathbf{X}, \mathbf{Y}, Z$ to $\mathcal{F}_{\mathsf{InnerVerify2}}$ and receive $b = \{0,1\}$;

Figure 4: Actively Secure Soft-boundary Overflow Detection Protocol.

in the ring-to-field lift, while a second checks the aggregated arithmetic relations. Any inconsistency causes an abort. A concrete realization of $\mathcal{F}_{\mathsf{InnerVerify2}}$ is given in Appendix F.

Fig. 4 depicts our actively secure protocol. For each checked value $t \in \{x_i, y_i, z_i\}$, the dual executions produce correction triples $(b^{(1)}_t, b^{(2)}_t, b^{(3)}_t)$ and $(c^{(1)}_t, c^{(2)}_t, c^{(3)}_t)$. Their consistency is equivalent to

$$b^{(1)}_t + b^{(2)}_t - b^{(1)}_t b^{(2)}_t + b^{(3)}_t = c^{(1)}_t + c^{(2)}_t - c^{(1)}_t c^{(2)}_t + c^{(3)}_t. \quad (5)$$

They verify $X = (b^{(1)}_t, c^{(1)}_t)$, $Y = (b^{(2)}_t, c^{(2)}_t)$ and $Z = b^{(1)}_t + b^{(2)}_t + b^{(3)}_t - c^{(1)}_t - c^{(2)}_t - c^{(3)}_t$ as inner product verification $X \times Y - Z = 0$. The coordinates for all $t \in \{x_i, y_i, z_i\}$ and all $i \in [N]$ are concatenated, using independent random coefficients, into one batched inner-product statement. Thus, a single inner-product verification simultaneously checks the consistency of all dual-execution triples, except with the soundness error of the random linear combination.

We show the security of $\Pi_{\mathsf{active\text{-}batch\text{-}detect}}$ with the following theorem.

Theorem 6. *Let λ be a security parameter, p be a prime number and $p > 2^\lambda$. Let the PRG used in the sub-protocol $\Pi_{[\cdot]}$ (c.f. Sec. 2) in the protocol $\Pi_{\mathsf{batch\text{-}active\text{-}detect}}$ be a secure peseudo-random geneartor with advantage $\mathsf{Adv}_{\mathsf{PRG}}$. The protocol $\Pi_{\mathsf{active\text{-}batch\text{-}detect}}$ in Fig. 4 UC-realizes $\mathcal{F}_{\mathsf{active\text{-}batch\text{-}detect}}$ in Fig. 1 in $\{\mathcal{F}_{\mathsf{3pc}}, \mathcal{F}_{\mathsf{active3pc}}, \mathcal{F}_{\mathsf{InnerVerify2}}\}$-hybrid model against a malicious adversary who can statically corrupted up to 1 party.*

The proof of Theorem 6 is provided in Appendix N.

4 Actively Secure PPML via Soft-Boundary Detection

In this section, we implement actively secure PPML inference based on our soft-boundary detection framework, providing security with abort.

To motivate this framework, we first highlight the inherent vulnerabilities of current fixed-point MPC: when processing expansive architectures like Transformers, modern inference

frameworks (such as CryptoFlow2 [41] and Mosformer [11]) represent real-valued parameters and activations as scaled fixed-point integers mapped onto a finite ring $\mathbb{Z}_{2^\ell}$ (typically $\mathbb{Z}_{2^{64}}$). Specifically, high-dimensional layers with dense inner products routinely overshoot the ring capacity; unlike standard CPU execution, the underlying MPC protocol throws no runtime flags or exceptions during this process. This *silent modular wraparound* [9, 49] destroys the algebraic linearity between the real and finite ring domains, causing massive, discontinuous jumps that severely degrade model utility. Beyond correctness errors, traditional defenses deploy deterministic hard-boundary checks ($\mathbf{1}_{x\geq 2^{\ell-1}}$) to trigger an immediate abort upon overflow. However, this inadvertently grants an active adversary $\mathcal{A}$ a noiseless probing oracle: by injecting fractional errors to induce targeted modular wraparounds, $\mathcal{A}$ adaptively monitors the termination status (Abort vs. Success), extracting exactly 1 bit of plaintext weight per query across smooth C^∞ activation fields (e.g., GELU, Swish).

Bridging Paradigm: Achieving Security with Abort via Verification. To mitigate these systemic vulnerabilities without the heavy online overhead of cryptographic commitments (such as IT-MACs in SPDZ [19]), we leverage an honest-majority post-execution verification paradigm. The feasibility of upgrading to malicious security through verification fundamentally depends on whether the secret-sharing scheme possesses "tamper-evident" properties [32]. Schemes like RSS under an honest majority possess an "immutable shape" due to algebraic redundancy: any unauthorized adversarial tampering automatically invalidates the sharing state, enabling lightweight verification post-computation.

Encoding Negative Fixed-Point Values. Our ring-to-field extension reserves the two most significant bits and therefore requires both of them to be zero. To accommodate negative fixed-point values under this requirement, let $a \in [-2^{\ell-3}, 2^{\ell-3})$ denote a signed scaled integer, whose conventional ring encoding is $a \bmod 2^\ell$. Before applying the extension, we use the offset encoding $a^+ = (a + 2^{\ell-3}) \bmod 2^\ell$. This public shift maps the entire signed interval to $[0, 2^{\ell-2})$, so that $\mathsf{msb}(a^+) = \mathsf{smsb}(a^+) = 0$, including when $a < 0$. Because the offset is public, it can be added locally to the secret shares without communication. When constructing the field arithmetic relation, the parties compensate for the same offset (equivalently, decode via $a = a^+ - 2^{\ell-3}$), so the shift changes only the representation and not the underlying fixed-point computation.

Challenges in Active Ring Verification and Overflow Detection. Translating this verification paradigm into high-throughput systems, however, encounters two fundamental bottlenecks:

1. Ring-Based Cryptographic Verification. While verification over finite fields is efficient via batching, ring verification ($\mathbb{Z}_{2^k}$) is severely bottlenecked by zero divisors, allowing adversaries to hide errors. Existing countermeasures—such as global precision padding [14] or quadratic extension rings [5]—incur prohibitive computational and communication overheads.

2. High-Overhead and Leakage-Prone Overflow Detection. Implementing active overflow checks hits a usability wall. Conventional circuit-based boolean comparisons are prohibitively expensive, while deterministic hard-boundary checks create catastrophic security loopholes by acting as noiseless oracles (yielding deterministic 1-bit leaks per query). Consequently, an ideal solution must simultaneously bypass expensive boolean circuits and deterministic leakage channels.

4.1 Our Solution: Active Security via Inner Product Verification

To address the dual bottlenecks of ring verification and overflow detection identified in Section 4, we propose an actively secure framework that encapsulates both arithmetic correctness and overflow enforcement into an ideal functionality $\mathcal{F}_{\mathsf{fixed\text{-}arithm}}$.

Importantly, actively secure soft-boundary overflow detection already provides the verification machinery required for actively secure fixed-point arithmetic. Detecting overflow requires lifting the ring operands and result to the field and checking that the lifted values are consistent with both the fixed-point arithmetic relation and the auxiliary sign and wraparound information. These are precisely the checks needed to prevent a malicious party from altering a fixed-point computation. Thus, active soft-boundary overflow detection naturally realizes active fixed-point arithmetic with abort; no additional cryptographic mechanism is required. The only implementation-level distinction is whether the two families of constraints are submitted to two separate inner-product verifications—one for malicious-deviation detection and one for overflow detection—or concatenated into a single inner-product relation and verified together. Our construction adopts the latter, unified formulation, thereby obtaining arithmetic correctness and soft-boundary overflow enforcement from one invocation of the inner-product verification.

As formalized in Figure 5, $\mathcal{F}_{\mathsf{fixed\text{-}arithm}}$ simultaneously governs ring-level arithmetic correctness and soft-boundary overflow enforcement. Specifically, for inputs x, y and operation $\Diamond$, the verification mechanism operates in two dimensions. First, to guarantee arithmetic correctness without heavy padding, we lift the verification to an extended field $\mathbb{F}_p$ and aggregate multiple individual checks into a batched inner product, such that the correctness of $x \Diamond y \bmod 2^\ell = z$ can be efficiently verified through the corresponding relation over $\mathbb{F}_p$. Second, rather than relying on deterministic hard boundaries that may leak information, the functionality implements a probabilistic soft-boundary policy: values below the lower threshold are accepted ($b = 0$), values above the upper threshold trigger an abort ($b = 1$), and values within the transition region are assigned an abort probability that increases

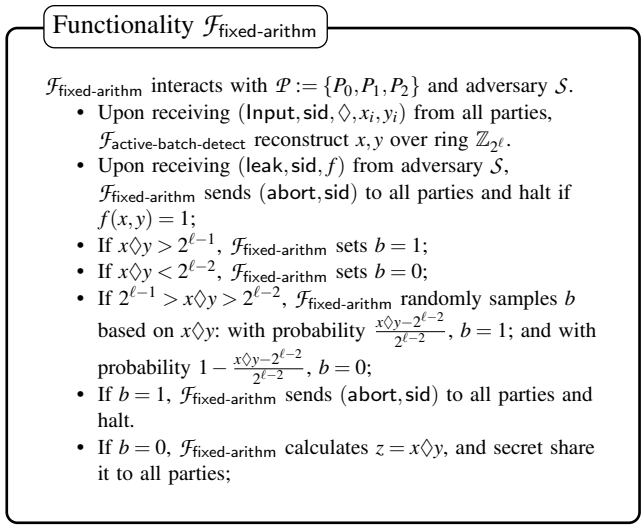

Figure 5: Secure Fixed-Point Arithmetic Functionality with Abort.

smoothly with the magnitude of $x \Diamond y$. Because the arithmetic-consistency and overflow constraints are aggregated into the same inner-product instance, $\mathcal{F}_{\text{fixed-arithm}}$ provides a direct realization of actively secure fixed-point arithmetic while mitigating parameter-extraction attacks.

The formal abstraction of this security boundary and arithmetic policy is therefore captured by Functionality $\mathcal{F}_{\text{fixed-arithm}}$, which establishes the cryptographic interface for secure fixed-point operations with abort.

Effectiveness and Significance. The soft-boundary mechanism ensures that if no overflow alert is triggered ($b = 0$), then $x \Diamond y < 2^{\ell-1}$, and the computation result remains within the valid arithmetic range. If an alert is triggered ($b = 1$), then $x \Diamond y \geq 2^{\ell-2}$, indicating that the operation either lies in the critical boundary region or has exceeded the safe range. Crucially, this mechanism mitigates active parameter-extraction attacks. Instead of using a deterministic hard boundary that exposes an exact one-bit comparison result, the soft boundary introduces probabilistic behavior into the adversary's probing observations, thereby preventing the boundary response from directly revealing precise information about the underlying value. At the same time, it avoids the substantial overhead of conventional Boolean-circuit-based comparisons.

Secure Fixed-Point Arithmetic over Soft-Boundary Ring. Building upon $\mathcal{F}_{\text{fixed-arithm}}$, we consider secure fixed-point computation over the ring $\mathbb{Z}_{2^\ell}$ with a soft boundary defined over the interval $(2^{\ell-2}, 2^{\ell-1})$. In addition to enforcing the correctness of fixed-point operations, $\mathcal{F}_{\text{fixed-arithm}}$ allows the adversary to issue a one-bit leakage query that may trigger an abort. Moreover, an overflow event governed by the soft-boundary policy may also lead to an abort. As a result, the adversary cannot deterministically distinguish whether an observed abort originates from its leakage query or from the probabilistic soft-boundary enforcement.

To concretely instantiate this functionality in the 3PC setting, we propose an actively secure fixed-point overflow detection protocol $\Pi_{\text{fix-arithm}}$, as formally detailed in Figure 14 (Appendix H). Importantly, the protocol begins by invoking the semi-honest functionality $\mathcal{F}_{\text{3pc}}$ to evaluate the fixed-point operations over $\mathbb{Z}_{2^\ell}$; this initial invocation alone does not provide active security. The subsequent verification phase lifts the recorded arithmetic relations and the auxiliary overflow constraints to the field and submits them jointly to $\mathcal{F}_{\text{InnerVerify2}}$.

4.2 Application to PPML Private Inference

We apply our actively secure fixed-point arithmetic with soft-boundary overflow detection to the linear layers of private Transformer inference, including the QKV projections, attention matrix multiplications, output projections, and MLP layers. These high-dimensional linear operations contain the principal accumulation paths in which fixed-point overflow may occur, and their arithmetic and overflow constraints are verified in batches using the construction from Section 4.

For the non-linear layers, such as Softmax and GELU, we directly use the corresponding protocols from Mosformer [11]. Thus, our PPML implementation changes only the linear layers by equipping them with active verification and soft-boundary overflow detection, while retaining Mosformer's non-linear-layer design. The resulting inference pipeline is summarized in Protocol $\Pi_{\text{Active-Transformer}}$, shown in Figure 15 (Appendix J).

5 Benchmarks

5.1 Benchmark setting

We implement our protocols in C++. All the benchmark code [1] can be found on the anonymous GitHub repository. For our experiments, the benchmark is performed in a server that runs Ubuntu 18.04.2 LTS with Intel(R) Xeon(R) Silver 4214 CPU @ 2.20GHz, 48 CPUs, 128 GB Memory. Our experiments are performed in a local area network, using traffic control in Linux to simulate three network settings: (1) Local-area settings (LAN): 2Gbps bandwidth with 1ms round-trip latency (RTT). (2)metropolitan-area settings (MAN): 256Mbps bandwidth with 1ms RTT. (2) wide-area setting (WAN): 100Mbps bandwidth with 10ms RTT.

5.2 Theoretical Bounds and Empirical Validation of Fixed-Point Overflow

Analysis of Fixed-Point Overflow. In native-ring MPC over $\mathbb{Z}_{2^\ell}$, intermediate fixed-point overflow during high-dimensional inner products causes *silent modular wraparounds* $(\text{mod } 2^\ell)$, permanently corrupting secret

shares without runtime exceptions. Consider a d-dimensional inner product $\sum_{i=1}^{d} \hat{x}_i \cdot \hat{y}_i$ with scale factor ℓ_f and input absolute bound $|x_i|, |y_i| < M$. To prevent the un-truncated sum $d \cdot M^2 \cdot 2^{2\ell_f}$ from encroaching upon the MSB (sign bit), the aggregate must reside within $[0, 2^{\ell-1})$. Thus, the strict worst-case safe dimension ceiling is:

$$d_{\max}^{\text{worst}} = \left\lfloor \frac{2^{\ell-1-2\ell_f}}{M^2} \right\rfloor \tag{6}$$

Implications on PPML Workloads. Equation 6 reveals a fundamental vulnerability in standard privacy-preserving machine learning (PPML) pipelines.

For inputs explicitly bounded by $|x_i|, |y_i| \leq 1$ ($M = 1$), a 32-bit ring with $\ell_f = 12$ gives $d_{\max}^{\text{worst}} = \lfloor 2^7 \rfloor = 128$. For the more conservative bound $M = 2$ used in Table 1, the corresponding bound is $d_{\max}^{\text{worst}} = 32$. This introduces an immediate execution barrier for modern deep learning architectures. Taking Transformers as a prominent example, their massive hidden dimensions (e.g., $d = 768$ or 1024) inevitably trigger silent state corruption. Crucially, even 64-bit rings are not unconditionally immune: activation outliers ($M > 1$) and dimension scaling—such as long-context processing in large language models—rapidly compress the integer headroom, necessitating proactive overflow mitigation.

Comprehensive Overflow. Table 1 populates the safe dimension boundaries across common ring bit-lengths ($\ell \in \{32, 64\}$) and scale factors ℓ_f, mapping theoretical ceilings directly against standard neural network layer shapes.

Additional details and statistical assumptions are provided in Appendix I.

Overflow Vulnerabilities in Conventional Frameworks. To investigate the robustness of fixed-point representations in frameworks such as ABY, ABY3, CryptFlow2, and SIRNN, we benchmark the secure inner product (IP) under a 37-bit bitwidth and 8-bit scale ($\ell = 37, \ell_f = 8$). We evaluate arithmetic accuracy across representative dimensions: $d \in \{10^3, 10^4, 10^5\}$.

As shown in Table 2, computations are accurate for $d \leq 10^4$ but fail at $d = 10^5$ due to a hardcoded bit-mask (`mask = (1ULL << 14) - 1`) in the EzPC implementation. This constraint limits input elements to $[-8192, 8192]$, meaning that at $d = 10^5$, the reported accumulated value 4.10×10^8 must be compared with the ring boundary in the same encoded domain. Under the stated values, 4.10×10^8 is below $2^{36} \approx 6.87 \times 10^{10}$ and therefore does not by itself demonstrate modular overflow.

5.3 Performance Analysis of Soft-Boundary Overflow Detection

Performance Breakdown of Π_{detect}. To evaluate the efficiency of 3PC-Π_{detect}, we benchmark its preprocessing and online phases under LAN, MAN, and WAN settings. All experiments are conducted over $\mathbb{Z}_{2^{32}}$ with $N \in \{10^2, 10^3, 10^4\}$, where N denotes the number of addition or multiplication gates jointly verified in a single detection batch. As shown in Table 3, the communication cost scales linearly with N. At $N = 10^4$, detecting addition gates incurs 585 KB and 741 KB of communication in the preprocessing and online phases, respectively, while detecting multiplication gates requires 663 KB and 897 KB. Under LAN, the online phase verifies 10^4 addition and multiplication gates in 62.81 ms and 67.82 ms, respectively. Owing to the interactive nature of the online phase, its runtime increases under MAN and WAN, while the communication volume remains unchanged. Overall, Π_{detect} achieves linear communication scalability and efficient batch verification, with multiplication detection introducing only moderate overhead compared with addition detection.

Performance Breakdown of $\Pi_{\mathsf{batch\text{-}detect}}$. Table 4 presents the performance of the batch detection protocol. Unlike Π_{detect}, which verifies each detection instance individually, $\Pi_{\mathsf{batch\text{-}detect}}$ aggregates the N Π_{detect} and validates the resulting relation through a single invocation of $\mathcal{F}_{\mathsf{InnerVerify1}}$. Although the per-gate field extension and share conversion cause the total communication to remain linear in N, batching significantly reduces the online overhead. At $N = 10^4$, the online communication costs for addition and multiplication detection are 390.73 KB and 468.86 KB, respectively, representing reductions of 47.27% and 47.73% over Π_{detect}. Under WAN, their online runtimes decrease from 11682.2 ms and 13653.0 ms to 95.95 ms and 116.27 ms, yielding speedups of 121.75× and 117.42×, respectively. Although fixed aggregation overhead affects small batches under LAN, this overhead is amortized as the batch size increases. Therefore, $\Pi_{\mathsf{batch\text{-}detect}}$ is particularly well suited for large-scale circuits and high-latency network environments.

In addition, we implemented hard-boundary detection based on Mosformer [11], a state-of-the-art maliciously secure private-inference framework, and compared it with our soft-boundary detection method. Specifically, hard-boundary detection first invokes Mosformer's [11] multiplication subprotocol to compute the product and then compares the result with a predefined threshold to determine whether an overflow has occurred. This is also a standard approach for overflow detection. Since Mosformer already provides both required subprotocols, the corresponding hard-boundary detector can be constructed straightforwardly. As shown in Table 4, our $\Pi_{\mathsf{batch\text{-}detect}}$ achieves a significant improvement in execution speed while substantially reducing communication overhead. The detailed speedup results are presented in Figure 6.

We also evaluate the efficiency and communication overhead of $\Pi^N_{\mathsf{batch\text{-}active\text{-}detect}}$, with detailed experimental results provided in Appendix K. For $N = 10^4$, $\Pi^N_{\mathsf{batch\text{-}active\text{-}detect}}$ completes batched actively secure overflow detection for addition and multiplication in only 0.365 s and 0.696 s under LAN, with total communication costs of 2.21 MB and 2.72 MB, respectively. Even under WAN, the corresponding runtimes are only 0.864 s and 1.454 s

Table 1: Theoretical Upper Bounds of Safe Inner Product Dimension ($d_{\max}$) Across Diverse Bitwidth and Scale Configurations ($M = 2$)

Total Bitwidth (ℓ)	Scale Factor (ℓ_f)	High-order Bits ($\ell - \ell_f$)	Worst-case Threshold	Safe Dimension Bound	Representative Target Scenario
16-bit	4	12	2^{16-3-8}	32	Low-precision Edge AI Inference
16-bit	6	10	$2^{16-3-12}$	2	Highly Unstable (Immediate Overflow)
20-bit	6	14	$2^{20-3-12}$	32	Lightweight Custom Accelerators
24-bit	8	16	$2^{24-3-16}$	32	Medium-precision ASIC Architectures
28-bit	10	18	$2^{28-3-20}$	32	Layer-wise Adaptive Fixed-Point
32-bit	8	24	$2^{32-3-16}$	8192	Classical ABY3-style Pipelines
32-bit	12	20	$2^{32-3-24}$	32	High-risk Zone (Fails Modern DNNs)
37-bit	8	29	$2^{37-3-16}$	262144	**Our tested EzPC config**
37-bit	12	25	$2^{37-3-24}$	1024	Baseline EzPC (Highly Vulnerable)
42-bit	12	30	$2^{42-3-24}$	32768	Proprietary Transformer Private Inference
54-bit	16	38	$2^{54-3-32}$	524288	High-precision Specialized Security Rings
64-bit	12	52	$2^{64-3-24}$	$\sim 1.37 \times 10^{11}$	Robust but Bandwidth-wasteful
64-bit	16	48	$2^{64-3-32}$	$\sim 5.37 \times 10^{8}$	Standard CryptFlow2 / Falcon
64-bit	20	44	$2^{64-3-40}$	$\sim 2.10 \times 10^{6}$	High-precision LLM Hidden Layers
64-bit	24	40	$2^{64-3-48}$	8192	Critical LLM Long-context Failure Point

Table 2: Bitwidth Overflow Benchmarking on EzPC Secure Fixed-Point Inner Product ($\ell = 37, \ell_f = 8$)

Dim. (d)	True IP	Computed IP	Rel. Err.	Status
10^3	4.07×10^6	4.07×10^6	4.6×10^{-6}	✓(Safe)
10^4	4.10×10^7	4.10×10^7	4.7×10^{-6}	✓(Safe)
10^5	4.10×10^8	-1.27×10^8	1.31	**Overflow**

Table 3: Performance of 3PC-$\Pi_{\mathsf{detect}}(\Diamond = +, \times)$ under different network settings ($\ell = 32$). The reported results are averaged over five independent runs.

Protocol	N	Offline (ms)			Online (ms)			Offline Comm.	Online Comm.
Π_{detect}		LAN	MAN	WAN	LAN	MAN	WAN	(KB)	(KB)
+	10^2	15.66	29.78	29.84	22.88	43.73	169.38	5.86	7.42
+	10^3	12.49	36.89	251.56	23.65	247.60	1846.98	58.01	73.48
+	10^4	23.09	39.71	98.35	62.81	1366.74	11682.2	585.00	741.00
×	10^2	18.62	22.95	14.90	19.65	33.24	167.43	6.64	8.98
×	10^3	11.90	27.74	53.11	25.73	254.20	2147.89	65.74	88.95
×	10^4	23.09	46.66	105.92	67.82	1606.45	13653.0	663.00	897.00

5.4 Performance of Actively Secure Fixed-Point Matrix Multiplication

In Transformer models such as BERT, they underpin attention and feed-forward layers. Having established the efficiency of the soft-boundary overflow-detection protocol, we next evaluate the actively secure fixed-point arithmetic protocols built upon it for matrix operations. Table 5 compares our protocol with MP-SPDZ [27] and protocols from Li *et al.* [31].

Compared with MP-SPDZ, our protocol achieves speedups ranging from 1.69× to 7.49× across all workloads and network settings. At $N = 10^5$, the speedups over MP-SPDZ [27] reach 7.49×, 6.04×, and 4.52× under LAN, MAN, and WAN, respectively. Compared with Li *et al.* [31], our protocol achieves speedups ranging from 1.09× to 6.94×, reaching 6.94×, 3.63×, and 1.81× at $N = 10^5$. In terms of total communication, our protocol uses less communication than MP-SPDZ by factors of 2.60×, 3.05×, and 3.42× for $N = 10^3$, 10^4, and 10^5, respectively. Compared with Li *et al.*, communication is slightly higher at $N = 10^3$ (0.94×) but lower by factors of 1.10× and 1.24× at $N = 10^4$ and 10^5. These results demonstrate that the proposed protocol provides increasingly larger runtime advantages as the matrix workload grows, while maintaining competitive communication costs. The detailed pairwise comparisons are shown in Figure 7.

Table 4: Performance comparison of 3PC-$\Pi_{\mathsf{batch\text{-}detect}}$ and Hard-boundary-detect [11] under different network settings ($\ell = 32$). The reported results are averaged over five independent runs.

Protocol	N	Offline (ms)			Online (ms)			Offline Comm.	Online Comm.
$\Pi^{\mathsf{batch}}_{\mathsf{detect}}$		LAN	MAN	WAN	LAN	MAN	WAN	(KB)	(KB)
+	10^2	0.32	0.38	0.48	0.54	4.76	4.91	5.86	4.02
+	10^3	1.83	3.24	3.36	2.33	6.80	6.87	58.59	39.17
+	10^4	17.34	28.29	30.15	12.02	25.72	26.57	585.94	390.73
× (Ours)	$\mathbf{10^2}$	**0.35**	**0.50**	**0.56**	**0.84**	**6.81**	**7.25**	**6.25**	**4.80**
× Hard-detect	10^2	–	–	–	2.15	56.62	265.13	–	44.53
× (Ours)	$\mathbf{10^3}$	**1.88**	**3.40**	**3.49**	**3.59**	**9.55**	**10.43**	**62.50**	**46.98**
× Hard-detect	10^3	–	–	–	6.85	35.06	280.61	–	445.31
× (Ours)	$\mathbf{10^4}$	**22.01**	**31.21**	**33.09**	**24.07**	**39.53**	**44.08**	**625.00**	**468.86**
× Hard-detect	10^4	–	–	–	49.03	153.63	504.16	–	4453.13

5.5 Performance of End-to-End Actively Secure PPML Inference

Building on the preceding evaluations of overflow detection and matrix multiplication, we next evaluate end-to-end actively secure PPML inference on BERT-base and GPT-2 under LAN, MAN, and WAN settings. We use our protocol for linear matrix-multiplication layers and adopt the nonlinear-operation protocols from Mosformer [11], which is also based on replicated secret sharing. We report the resulting end-to-end runtime and communication costs.

As shown in Table 6, our protocol achieves end-to-end runtimes of 128.85 s for BERT-base and 98.25 s for GPT-2. Compared with Li *et al.*, our protocol achieves runtime speedups of 8.70× and 5.70×, and communication reduc-

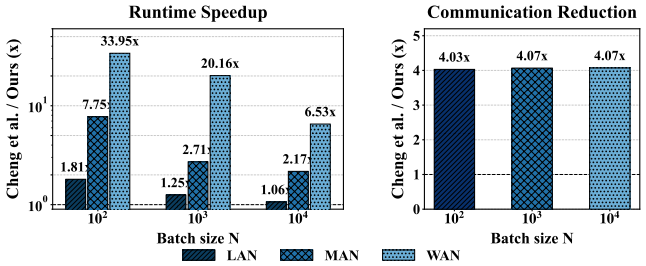

Figure 6: Runtime speedup and communication reduction compared with Hard-boundary overflow detection [11].

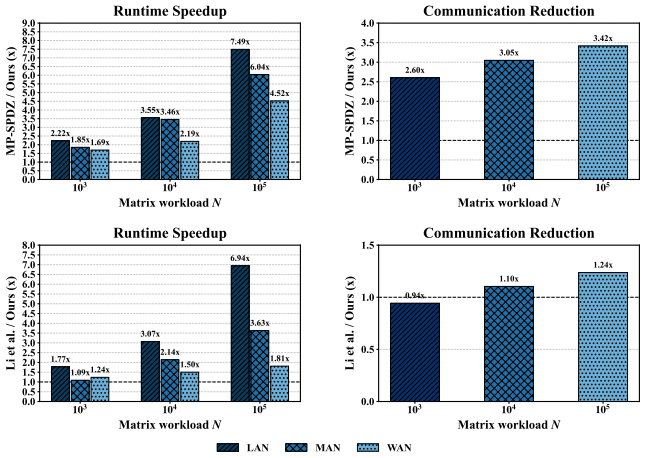

Figure 7: Runtime speedup and communication reduction compared with MP-SPDZ [27] and Li *et al.* [31].

Table 5: Performance comparison of 3PC-$\Pi_{\text{fix-arithm}}(\Diamond = \otimes)$, MP-SPDZ [27], and Li *et al.* [31] for matrix multiplication under different network settings ($\ell = 32$). Here, (d_1, d_2, d_3) denotes multiplication of a $d_1 \times d_2$ matrix and a $d_2 \times d_3$ matrix, and $N = d_1 \times d_2 \times d_3$.

Protocol	N	Offline (ms)			Online (ms)			Offline Comm.	Online Comm.
$\otimes$		LAN	MAN	WAN	LAN	MAN	WAN	(KB)	(KB)
MP-SPDZ [27]	10^3	-	-	-	20.07	41.16	271.77	-	228.39
Li *et al.* [31]	10^3	-	-	-	16.03	24.20	198.59	-	82.82
Ours	10^3	2.60	3.91	4.17	6.44	18.32	156.40	22.07	65.73
MP-SPDZ [27]	10^4	-	-	-	102.20	203.90	494.52	-	2265.87
Li *et al.* [31]	10^4	-	-	-	88.40	125.99	338.20	-	820.84
Ours	10^4	10.34	12.50	20.05	18.46	46.50	205.37	199.61	543.86
MP-SPDZ [27]	10^5	-	-	-	836.27	1633.87	3466.77	-	22645.04
Li *et al.* [31]	10^5	-	-	-	775.00	981.55	1388.90	-	8201.01
Ours	10^5	30.25	81.04	176.39	81.42	189.66	590.80	1978.52	4652.26

Table 6: Runtime and communication costs of our protocol under the LAN setting.

Model	Framework	Runtime (s)		Communication Cost (GB)	
		Online	Offline	Online	Offline
BERT-base	MP-SPDZ [27]	3644.97	69321.32	669.34	4448.38
	Li *et al.* [31]	401.84	718.77	3.06	82.30
	Cheng *et al.* [11]	586.73	-	4.69	63.67
	Ours	**108.93**	**19.92**	**13.94**	**2.38**
GPT-2	MP-SPDZ [27]	1702.66	31104.61	325.23	2010.22
	Li *et al.* [31]	192.83	367.15	1.51	40.47
	Cheng *et al.* [11]	442.99	-	3.45	33.67
	Ours	**84.52**	**13.73**	**8.38**	**2.01**

Table 7: Runtime and communication costs of our protocol and Li *et al.* [31] under MAN and WAN settings.

Model	Framework	MAN Runtime (s)		WAN Runtime (s)		Comm. (GB)	
		Online	Offline	Online	Offline	Online	Offline
BERT-base	Li *et al.* [31]	1455.274	4205.732	7359.577	15257.721	3.406	91.975
	Ours	537.523	88.722	1399.488	205.930	13.938	2.378
GPT-2	Li *et al.* [31]	619.067	1829.835	3364.080	6909.739	1.590	42.918
	Ours	340.236	70.619	943.322	178.593	8.380	2.010

tions of 5.23× and 4.04×. Compared with Mosformer, our protocol achieves runtime speedups of 4.55× and 4.51× for BERT-base and GPT-2, respectively. Beyond the LAN results discussed above, we further evaluate end-to-end inference under MAN and WAN settings, as summarized in Table 7. Compared with Li *et al.* [31], our protocol reduces the total runtime for BERT-base from 5661.01 s to 626.25 s under MAN and from 22617.30 s to 1605.42 s under WAN, yielding speedups of 9.04× and 14.09×, respectively. For GPT-2, it achieves speedups of 5.96× and 9.16× under MAN and WAN. In addition, communication is reduced by 5.85× for BERT-base and 4.28× for GPT-2, demonstrating the effectiveness of our protocol in high-latency network environments.

We additionally evaluate the operation-level performance breakdowns of the comparison protocols under LAN, MAN, and WAN settings. Tables 9 and 10 report the LAN results for BERT-base and GPT-2, respectively, while Tables 11 and 12 present the corresponding MAN and WAN results. The performance breakdown of our protocol is provided in Table 13. All of these results are reported in Appendix L.

6 Conclusion

We propose a soft boundary overflow detection method, which enables efficient overflow detection without costly exact comparisons or bit-level decomposition. Building on this method, we construct an actively secure fixed-point MPC framework over rings and integrate it into actively secure PPML inference. Experiments on basic operations, matrix multiplication, and end-to-end transformer inference show that our approach substantially improves efficiency over maliciously secure baselines while retaining overflow-detection capability.

Ethics Considerations

This work aims to improve the security and efficiency of privacy-preserving machine learning by designing efficient overflow detection for actively secure fixed-point computation. It can support privacy-sensitive applications, such as healthcare and finance, by protecting confidential data and reducing computational errors caused by silent numerical overflow. We acknowledge that privacy-enhancing technologies could theoretically be misused to conceal malicious activities. However, our contribution is a general cryptographic mechanism intended to protect legitimate data and improve the reliability of secure computation.

Open Science

Code Availability and Reproducibility. We make the C++ implementation of our actively secure soft-boundary overflow-detection protocols available at [1]. The repository includes the required EMP-toolkit dependencies, the Mosformer component, and scripts for reproducing the experiments. Users first run `setup_env_and_build.sh`, and then build the project with CMake. The script `run_multiparty_test.sh` automatically launches the three MPC parties and manages execution logs. The script `network_simulation.sh` reproduces experiments under the LAN, MAN, and WAN settings described in Section 5; it requires `sudo` privileges for Linux traffic control.

Data. Our microbenchmarks use synthetically generated inputs according to the dimensions and fixed-point configurations of each benchmark. The end-to-end evaluations execute BERT-base and GPT-2 layer workloads and do not require private, proprietary, or personally identifiable data. All benchmark configurations are specified in Section 5.

References

[1] Our implementation code. https://anonymous.4open.science/r/activedetect-0825/.

[2] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, 1991.

[3] Patrick Bosc and Olivier Pivert. Sqlf: a relational database language for fuzzy querying. *IEEE transactions on Fuzzy Systems*, 3(1):1–17, 1995.

[4] Elette Boyle, Nishanth Chandran, Niv Gilboa, Divya Gupta, Yuval Ishai, Nishant Kumar, and Mayank Rathee. Function secret sharing for mixed-mode and fixed-point secure computation. Cryptology ePrint Archive, Report 2020/1392, 2020.

[5] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Practical fully secure three-party computation via sublinear distributed zero-knowledge proofs. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019: 26th Conference on Computer and Communications Security*, pages 869–886, London, UK, November 11–15, 2019. ACM Press.

[6] Megha Byali, Harsh Chaudhari, Arpita Patra, and Ajith Suresh. Flash: Fast and robust framework for privacy-preserving machine learning. In *PoPETs*, 2020.

[7] Harsh Chaudhari, Ashish Choudhury, Arpita Patra, and Ajith Suresh. Astra: High throughput 3pc over rings with application to secure prediction. In *CCSW*, 2019.

[8] Harsh Chaudhari, Rahul Rachuri, and Ajith Suresh. Trident: Efficient 4pc framework for privacy preserving machine learning. In *NDSS*, 2020.

[9] Wei Chen, Xiapu Luo, Chengyu Yin, Bin Xiao, Man Ho Au, and Yajuan Tang. MUSE: Towards robust and stealthy mobile botnets via multiple message push services. In Joseph K. Liu and Ron Steinfeld, editors, *ACISP 16: 21st Australasian Conference on Information Security and Privacy, Part I*, volume 9722 of *Lecture Notes in Computer Science*, pages 20–39, Melbourne, VIC, Australia, July 4–6, 2016. Springer, Cham, Switzerland.

[10] Yuntian Chen, Tianpei Lu, Zhanyong Tang, Bingsheng Zhang, Zhiying Shi, Yuxiang Luan, and Zhuzhu Wang. Sok: Private transformer-based model inference. In *35th USENIX Security Symposium (USENIX Security 26)*, Baltimore, MD, 2026. USENIX Association.

[11] Ke Cheng, Yuheng Xia, Anxiao Song, Jiaxuan Fu, Wenjie Qu, Yulong Shen, and Jiaheng Zhang. Mosformer: Maliciously secure three-party inference framework for large transformers. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, pages 4124–4138, 2025.

[12] 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 *CRYPTO*, 2018.

[13] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. SPDZ2k: Efficient MPC mod 2^k for dishonest majority. Cryptology ePrint Archive, Report 2018/482, 2018.

[14] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. Spdz2k: Efficient MPC mod 2k for dishonest majority. In *CRYPTO*, 2018.

[15] Anders Dalskov, Daniel Escudero, and Ariel Nof. Fast fully secure multi-party computation over any ring with two-thirds honest majority. In *CCS*, 2022.

[16] Anders Dalskov, Daniel E. Escudero, and Marcel Keller. Fantastic four: Honest-majority four-party secure computation with malicious security. *IACR Cryptol. ePrint Arch.*, 2020.

[17] Ivan Damgård, Claudio Orlandi, and Mark Simkin. Yet another compiler for active security or: Efficient MPC over arbitrary rings. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part II*, volume 10992 of *Lecture Notes in Computer Science*, pages 799–829, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Cham, Switzerland.

[18] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, 2012.

[19] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662, Santa Barbara, CA, USA, August 19–23, 2012. Springer Berlin Heidelberg, Germany.

[20] Ivan Damgård, Claudio Orlandi, and Mark Simkin. Yet another compiler for active security or: Efficient mpc over arbitrary rings. In *CRYPTO*, 2018.

[21] Ye Dong, Wen jie Lu, Xiaoyao Hou, Kang Yang, and Jian Liu. M&m: Secure two-party machine learning through efficient modulus conversion and mixed-mode protocols. Cryptology ePrint Archive, Paper 2025/1551, 2025.

[22] Daniel Escudero and Vipul Goyal. Turbopack: Honest majority mpc with constant online communication. In *CCS*, 2022.

[23] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *STOC*, 1987.

[24] Vipul Goyal, Hanjun Li, Rafail Ostrovsky, and Antigoni Polychroniadou. Atlas: Efficient and scalable mpc in the honest majority setting. In *CRYPTO*, 2021.

[25] Christopher Harth-Kitzerow, Ajith Suresh, and Georg Carle. Sok: Truncation untangled: Scaling fixed-point arithmetic for privacy-preserving machine learning to large models and datasets. *Proc. Priv. Enhancing Technol.*, 2025(4):369–391, 2025.

[26] Eerikson Hendrik, Keller Marcel, Orlandi Claudio, Pullonen Pille, Puura Joonas, and Simkin Mark. Use your brain! arithmetic 3pc for any modulus with active security. In *ITC*, 2020.

[27] Marcel Keller. Mp-spdz: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, pages 1575–1590, 2020.

[28] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. Swift: Super-fast and robust privacy-preserving machine learning. In *USENIX*, 2021.

[29] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. Cryptflow: Secure tensorflow inference. In *S&P*, 2020.

[30] David D Lewis and Jason Catlett. Heterogeneous uncertainty sampling for supervised learning. In *Machine learning proceedings 1994*, pages 148–156. Elsevier, 1994.

[31] Yun Li, Daniel Escudero, Yufei Duan, Zhicong Huang, Cheng Hong, Chao Zhang, and Yifan Song. Sublinear distributed product checks on replicated secret-shared data over Z_{2^k} without ring extensions. In Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie, editors, *ACM CCS 2024: 31st Conference on Computer and Communications Security*, pages 825–839, Salt Lake City, UT, USA, October 14–18, 2024. ACM Press.

[32] Yehuda Lindell and Ariel Nof. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 259–276, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.

[33] Fengrun Liu, Xiang Xie, and Yu Yu. Scalable multiparty computation protocols for machine learning in the honest-majority setting. In *Proceedings of the 33rd USENIX Conference on Security Symposium*, SEC '24, USA, 2024. USENIX Association.

[34] Tianpei Lu, Xin Kang, Bingsheng Zhang, Zhuo Ma, Xiaoyuan Zhang, Yang Liu, Kui Ren, and Chun Chen. Efficient 2pc for constant round secure equality testing and comparison. 2025.

[35] Tianpei Lu, Bingsheng Zhang, Lichun Li, and Kui Ren. The communication-friendly privacy-preserving machine learning against malicious adversaries. *IEEE Transactions on Dependable and Secure Computing*, 2025.

[36] Tianpei Lu, Bingsheng Zhang, Lichun Li, Yuzhou Zhao, and Kui Ren. Lightning fast secure comparison for 3pc ppml. *Cryptology ePrint Archive*, 2023.

[37] Payman Mohassel and Peter Rindal. Aby3: A mixed protocol framework for machine learning. In *CCS*, 2018.

[38] Arpita Patra and Ajith Suresh. BLAZE: blazing fast privacy-preserving machine learning. In *NDSS*, 2020.

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

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

[41] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. Cryptflow2: Practical 2-party secure inference. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, pages 325–342, 2020.

[42] Glenn Shafer and Vladimir Vovk. A tutorial on conformal prediction. *Journal of machine learning research*, 9(3), 2008.

[43] Sameer Wagh, Divya Gupta, and Nishanth Chandran. Securenn: 3-party secure computation for neural network training. In *PoPETs*, 2019.

[44] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. Falcon: Honest-majority maliciously secure framework for private deep learning. In *PoPETs*, 2021.

[45] Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa. Piranha: A gpu platform for secure computation, 2022.

[46] Andrew C. Yao. Protocols for secure computations. In *SFCS*, 1982.

[47] Lindell Yehuda and Nof Ariel. A framework for constructing fast mpc over arithmetic circuits with malicious adversaries and an honest-majority. In *CCS*, 2017.

[48] Chenkai Zeng, Qi Feng, Debiao He, and Min Luo. LightShark: Actively secure machine-learning inference based on lightweight authenticated distributed comparison function. Cryptology ePrint Archive, Paper 2026/1487, 2026.

[49] Wen Zhang, Bingsheng Zhang, Tianpei Lu, and Kui Ren. PPML is more vulnerable to cryptanalytic extraction attacks. Cryptology ePrint Archive, Paper 2026/848, 2026.

[50] Lijing Zhou, Ziyu Wang, Hongrui Cui, Qingrui Song, and Yu Yu. Bicoptor: Two-round secure three-party nonlinear computation without preprocessing for privacy-preserving machine learning. In *S&P*, 2023.

[51] Lijing Zhou, Bingsheng Zhang, Ziyu Wang, Tianpei Lu, Qingrui Song, Su Zhang, Hongrui Cui, and Yu Yu. On probabilistic truncation in privacy-preserving machine learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(21):22955–22964, Apr. 2025.

A Related Work

Overflow detection. Detecting arithmetic overflow on secret-shared values is closely related to secure comparison and most-significant-bit extraction. Generic comparison protocols evaluate a bit decomposition or convert arithmetic shares to Boolean shares, which makes their cost depend on the bit length and often introduces multiple interaction rounds. SecureNN and CrypTFlow reduce sign extraction to specialized arithmetic primitives [29, 43], while Bicoptor obtains a two-round comparison by combining local truncations with a zero-test performed by the third party [50]. Subsequent work further develops constant-round comparison [34, 36] and studies the correctness–efficiency tradeoff of probabilistic truncation in PPML [25, 51]. Modulus-conversion and ring-extension techniques provide another route: they lift a sharing to a larger domain and correct the wrap-around introduced by the lift [21]. These techniques are designed primarily to recover an exact comparison bit, sign bit, or converted value. In contrast, our soft-boundary detector asks only whether a value lies in a prescribed safe, transition, or overflow interval. This relaxation turns overflow detection into a probabilistic verification problem and avoids invoking a full comparison for every arithmetic result.

Actively secure computation over rings. Arithmetic rings, and in particular $\mathbb{Z}_{2^\ell}$, are a natural domain for fixed-point computation because machine-word addition and multiplication are represented directly. Achieving active security over rings is nevertheless more involved than semi-honest evaluation: the protocol must detect malformed shares and inconsistent multiplications despite the presence of zero divisors. SPDZ$_{2^k}$ develops authenticated dishonest-majority MPC specialized to rings of characteristic 2^k [14]. In the honest-majority setting, compiler-based and replicated-sharing approaches establish active security over arbitrary rings [12, 20, 47], and later systems improve online communication, batching, or robustness [15, 22, 24]. Related three- and four-party constructions optimize arithmetic MPC for arbitrary moduli or obtain malicious security with an honest majority [16, 26].

These protocols protect the correctness of the modular computation itself: if the specified circuit evaluates in $\mathbb{Z}_{2^\ell}$, active security guarantees that an adversary cannot undetectably change that modular result. It does not, by itself, establish

that the corresponding integer computation stayed within its intended numerical range. A common solution is therefore to enlarge the ring and reserve statistical headroom, which increases the cost of all downstream arithmetic. Our work instead couples active verification with soft-boundary overflow detection, aiming to detect both malicious deviations and unintended numerical wrap-around without systematically moving the PPML workload to a substantially larger ring.

Actively secure PPML. Early MPC-based PPML systems demonstrated practical secure training and inference using arithmetic sharing and mixed-protocol evaluation [29, 37, 39, 43]. Their efficient linear layers are typically evaluated over rings, whereas nonlinear layers such as ReLU, comparison, and max pooling require Boolean sharing, garbled circuits, or specialized comparison protocols. ASTRA and BLAZE optimize three-party ring computation and secure prediction [7, 38]; FLASH and SWIFT additionally target robustness in practical PPML pipelines [6, 28]. FALCON provides an honest-majority maliciously secure framework for private deep learning [44], while Trident explores an efficient four-party design [8]. More recent work improves communication-efficient malicious PPML and extends actively secure inference to modern workloads [35, 48].

Most actively secure PPML frameworks focus on enforcing faithful execution of arithmetic and nonlinear primitives. Numerical overflow is generally prevented through conservative range analysis, wider rings, or extra comparison/truncation operations rather than treated as a first-class verified event. This distinction is important for fixed-point ML: a correctly authenticated multiplication may still be numerically invalid after modular wrap-around. Our construction complements maliciously secure PPML by integrating efficient overflow detection into the verification path, so that integrity of protocol execution and validity of the represented numerical computation can be checked together.

B Actively Secure Three-Party Computation Functionality

The ideal functionality below formalizes actively secure three-party computation (see Figure 8) over the field $\mathbb{Z}_p$ with abort. It evaluates a public function on the values encoded by the parties' input shares and releases only fresh shares of the result. A maliciously corrupted party may prevent delivery by instructing the functionality to abort, but cannot modify an accepted output.

C Concrete Arithmetic Protocols for Semi-Honest 3PC

This appendix presents the concrete instantiation of $\mathcal{F}_{\mathsf{3pc}}$ over $\mathbb{Z}_{2^\ell}$ following ASTRA [7]. Using the masked sharing

Functionality $\mathcal{F}_{\mathsf{active3pc}}$

$\mathcal{F}_{\mathsf{active3pc}}$ interacts with parties $\mathcal{P} := \{P_0, P_1, P_2\}$ and an adversary $\mathcal{S}$.

- Upon receiving $(\mathsf{input}, \mathsf{sid}, f, (([r_{x_j}]_1, [r_{x_j}]_2))_{j\in[n]})$ from P_0, $(\mathsf{input}, \mathsf{sid}, f, ((m_{x_j}, [r_{x_j}]_1))_{j\in[n]})$ from P_1, and $(\mathsf{input}, \mathsf{sid}, f, ((m_{x_j}, [r_{x_j}]_2))_{j\in[n]})$ from P_2, $\mathcal{F}_{\mathsf{active3pc}}$ check all the input are identical. If any share-consistency check fails, it sends $(\mathsf{abort}, \mathsf{sid})$ to all parties and halts.
- It reconstructs $x_j = m_{x_j} + [r_{x_j}]_1 + [r_{x_j}]_2 \pmod p$ for every $j \in [n]$ and computes $z = f(x_0, \ldots, x_{n-1})$ over $\mathbb{Z}_p$.
- It generates a fresh masked sharing $\langle z \rangle := (m_z, [r_z])$ of z and notifies $\mathcal{S}$ that the output is ready, without revealing z.
- Upon receiving $(\mathsf{abort}, \mathsf{sid})$ from $\mathcal{S}$, it sends $(\mathsf{abort}, \mathsf{sid})$ to all parties and halts. Otherwise, it sends $(\mathsf{output}, \mathsf{sid}, \langle z \rangle)$ to each party and halts.

Figure 8: Actively Secure Three-Party Computation Functionality with Abort.

$\langle x \rangle^{2^\ell} = (m_x, [r_x]^{2^\ell})$ defined above, addition is evaluated locally, whereas multiplication uses an offline sharing of the product of the input masks and an online exchange between P_1 and P_2. The corresponding addition and multiplication protocols are given below.

Addition. Given $\langle x \rangle^{2^\ell} = (m_x, [r_x]^{2^\ell})$ and $\langle y \rangle^{2^\ell} = (m_y, [r_y]^{2^\ell})$, the parties locally compute

$$[r_z]_i = [r_x]_i + [r_y]_i, \quad i \in \{1,2\}, \qquad m_z = m_x + m_y.$$

They obtain $\langle z \rangle^{2^\ell} = (m_z, [r_z]^{2^\ell})$, which encodes $z = x + y \pmod{2^\ell}$.

Multiplication. For $\langle z \rangle^{2^\ell} \leftarrow \Pi_{\mathsf{mul}}(\langle x \rangle^{2^\ell}, \langle y \rangle^{2^\ell})$, P_0 and P_1 jointly sample $[r_z]_1$ and $[\gamma_{xy}]_1$, while P_0 and P_2 jointly sample $[r_z]_2$. In the offline phase, P_0 reconstructs $r_x = [r_x]_1 + [r_x]_2$ and $r_y = [r_y]_1 + [r_y]_2$, computes $\gamma_{xy} = r_x r_y$, and sends $[\gamma_{xy}]_2 = \gamma_{xy} - [\gamma_{xy}]_1$ to P_2.

In the online phase, each P_i, for $i \in \{1,2\}$, locally computes

$$[m_z]_i = (i-1) m_x m_y - m_x [r_y]_i - m_y [r_x]_i + [r_z]_i + [\gamma_{xy}]_i.$$

The parties P_1 and P_2 exchange their shares and reconstruct $m_z = [m_z]_1 + [m_z]_2$. The resulting $\langle z \rangle^{2^\ell} = (m_z, [r_z]^{2^\ell})$ is a valid masked sharing of $z = xy \pmod{2^\ell}$.

Verification of Inner Product. For an inner-product triple $((\langle x_i \rangle^{2^\ell}, \langle y_i \rangle^{2^\ell})_{i=0}^{N-1}, \langle z \rangle^{2^\ell})$, P_0 computes $\gamma = \sum_{i=0}^{N-1} r_{x_i} r_{y_i} \pmod{2^\ell}$ and distributes an additive sharing $[\gamma] = ([\gamma]_1, [\gamma]_2)$ to P_1 and P_2.

Each P_j, for $j \in \{1,2\}$, locally computes

$$[V]_j = \sum_{i=0}^{N-1} \big((j-1) m_{x_i} m_{y_i} + m_{x_i} [r_{y_i}]_j + m_{y_i} [r_{x_i}]_j \big) + [\gamma]_j - (j-1) m_z - [r_z]_j \pmod{2^\ell}.$$

The resulting $[V] = ([V]_1, [V]_2)$ is an additive sharing of $V = \sum_{i=0}^{N-1} x_i y_i - z \pmod{2^\ell}$. The parties send $[V]$ to $\mathcal{F}_{\mathsf{eq}}[2^\ell]$ and accept if and only if $V = 0$.

D 2PC Protocol

We now instantiate the soft-boundary reduction from Theorem 3 in the two-party setting. Parties P_1 and P_2 hold additive shares $[x]^{2^\ell} = ([x]_1, [x]_2)$ and $[y]^{2^\ell} = ([y]_1, [y]_2)$ over $\mathbb{Z}_{2^\ell}$. They first invoke $\mathcal{F}_{\mathsf{2pc}}$ to evaluate $z = x \diamondsuit y$ over the ring. Each party then secret-shares the relevant most-significant bits of its local shares. These bits determine the wraparound corrections required to lift x, y, and z from $\mathbb{Z}_{2^\ell}$ to $\mathbb{Z}_p$.

Using these corrections, the parties obtain the lifted values $[\hat{x}]^p$, $[\hat{y}]^p$, and $[\hat{z}]^p$. In parallel, they evaluate the same operation over $\mathbb{Z}_p$ to obtain $[z']^p = [\hat{x} \diamondsuit \hat{y}]^p$. The final equality test checks whether $z' = \hat{z}$. Equality indicates that the ring result is consistent with its field evaluation, whereas inequality produces the soft-boundary overflow signal. The complete protocol is given in Figure 9.

Protocol $\Pi_{\mathsf{detect}}(\diamondsuit, [x]^{2^\ell}, [y]^{2^\ell})$

Input : $[x]^{2^\ell}$, $[y]^{2^\ell}$;
Output : a soft-boundary interval detection result b, which holds $b \leftarrow \mathcal{G}(x^{(1)} \diamondsuit x^{(2)}, \mathbb{B} = (2^{\ell-2}, 2^{\ell-1}))$.
Execution:

- All parties send $(\mathsf{input}, \diamondsuit, [x]^{2^\ell}, [y]^{2^\ell})$ to $\mathcal{F}_{\mathsf{2pc}}$ and receive $[z]^{2^\ell}$.
- P_j invokes $[b_x^{(j)}]^p \leftarrow \Pi^K_{[\cdot]}(\mathsf{msb}([x]_j))$, $[b_y^{(j)}]^p \leftarrow \Pi^K_{[\cdot]}(\mathsf{msb}([y]_j))$ and $[b_z^{(j)}]^p \leftarrow \Pi^K_{[\cdot]}(\mathsf{smsb}([z]_j))$ for $j \in \{1,2\}$;
- P_j set $[\hat{z}]^p_j = [z]^{2^\ell}_j \pmod{2^{\ell-1}}$, $[\hat{x}]^p_j = [x]^{2^\ell}_j$ and $[\hat{y}]^p_j = [y]^{2^\ell}_j$;
- All parties set $f(a,b,c) = a - 2^\ell \times (b + c - b \times c)$
- All parties send $(\mathsf{input}, f, [x]^p, [b_x^{(1)}], [b_x^{(2)}])$, $(\mathsf{input}, f, [y]^p, [b_y^{(1)}], [b_y^{(2)}])$, $(\mathsf{input}, f, [z]^p, [b_z^{(1)}], [b_z^{(2)}])$, and $(\mathsf{input}, \diamondsuit, [\hat{x}]^p, [\hat{y}]^p)$ to $\mathcal{F}_{\mathsf{2pc}}$ receive $[\hat{x}]^p$, $[\hat{y}]^p$, $[\hat{z}]^p$, and $[z']^p$, respectively.
- All parties send $[z']^p$ and $[\hat{z}]^p$ to $\mathcal{F}_{\mathsf{eq}}[K]$ to verify $z' = \hat{z}$.

Figure 9: 2PC Soft-boundary Overflow Detection Protocol.

E 3PC Batch Soft-boundary Overflow Detection Protocol

The 3PC batch soft-boundary overflow-detection protocol is presented in Figure 10.

Protocol $\Pi_{\mathsf{batch\text{-}detect}}(\{\diamondsuit_i, \langle x_i \rangle^{2^\ell}, \langle y_i \rangle^{2^\ell}\}_{i=1}^N)$

Input : N tuples $\{\diamondsuit_i, \langle x_i \rangle^{2^\ell}, \langle y_i \rangle^{2^\ell}\}_{i=1}^N$; Output : a soft-boundary overflow detection result b, where $b \leftarrow \bigvee_{i=1}^N \mathcal{G}(x_i \diamondsuit_i y_i, \mathbb{B} = (2^{\ell-2}, 2^{\ell-1}))$.
Execution:

- For each $i \in \{1, \ldots, N\}$, all parties do:
  - All parties send $(\mathsf{input}, \diamondsuit_i, (\langle x_i \rangle^{2^\ell}, \langle y_i \rangle^{2^\ell}))$ to $\mathcal{F}_{\mathsf{3pc}}$ and receive $\langle z_i \rangle^{2^\ell}$.
  - P_0 invokes $\langle b_{x_i}^{(1)} \rangle^p \leftarrow \Pi^K_{\langle\cdot\rangle}(\mathsf{msb}(r_{x_i}))$, $\langle b_{y_i}^{(1)} \rangle^p \leftarrow \Pi^K_{\langle\cdot\rangle}(\mathsf{msb}(r_{y_i}))$, and $\langle b_{z_i}^{(1)} \rangle^p \leftarrow \Pi^K_{\langle\cdot\rangle}(\mathsf{smsb}(r_{z_i}))$;
  - P_0 invokes $\langle b_{x_i}^{(3)} \rangle^p \leftarrow \Pi^K_{\langle\cdot\rangle}(\mathsf{wrap}([r_{x_i}]_1, [r_{x_i}]_2))$, $\langle b_{y_i}^{(3)} \rangle^p \leftarrow \Pi^K_{\langle\cdot\rangle}(\mathsf{wrap}([r_{y_i}]_1, [r_{y_i}]_2))$, and $\langle b_{z_i}^{(3)} \rangle^p \leftarrow \Pi^K_{\langle\cdot\rangle}(\mathsf{wrap}^{2^{\ell-1}}([r_{z_i}]_1, [r_{z_i}]_2))$;
  - All parties set $\langle b_{x_i}^{(2)} \rangle^p := (\mathsf{msb}(m_{x_i}), 0)$, $\langle b_{y_i}^{(2)} \rangle^p := (\mathsf{msb}(m_{y_i}), 0)$, and $\langle b_{z_i}^{(2)} \rangle^p := (\mathsf{smsb}(m_{z_i}), 0)$;
  - All parties set $\langle \hat{z}_i \rangle^p = \langle z_i \rangle^{2^\ell} \pmod{2^{\ell-1}}$, $\langle \hat{x}_i \rangle^p = \langle x_i \rangle^{2^\ell}$, and $\langle \hat{y}_i \rangle^p = \langle y_i \rangle^{2^\ell}$;
  - All parties set $f(a,b,c,d) = a - 2^\ell \times (b + c - b \times c + d)$;
  - All parties send $(\mathsf{input}, f, \langle \hat{x}_i \rangle^p, \langle b_{x_i}^{(1)} \rangle, \langle b_{x_i}^{(2)} \rangle, \langle b_{x_i}^{(3)} \rangle)$, $(\mathsf{input}, f, \langle \hat{y}_i \rangle^p, \langle b_{y_i}^{(1)} \rangle, \langle b_{y_i}^{(2)} \rangle, \langle b_{y_i}^{(3)} \rangle)$, and $(\mathsf{input}, f, \langle \hat{z}_i \rangle^p, \langle b_{z_i}^{(1)} \rangle, \langle b_{z_i}^{(2)} \rangle, \langle b_{z_i}^{(3)} \rangle)$ to $\mathcal{F}_{\mathsf{3pc}}$ and receive $\langle \hat{x}_i \rangle^p$, $\langle \hat{y}_i \rangle^p$, and $\langle \hat{z}_i \rangle^p$, respectively.
- All parties agree on a random seed η and invoke PRG to generate $(r_1, \ldots, r_N) \leftarrow \mathbb{Z}_K^N$ using η.
- All parties set $\langle Z \rangle \leftarrow \langle 0 \rangle^p$, $\mathbf{X} \leftarrow ()$, $\mathbf{Y} \leftarrow ()$.
- For each $i \in \{1, \ldots, N\}$, all parties do:
  - If $\diamondsuit_i = +$, set $\langle Z \rangle \leftarrow \langle Z \rangle + r_i(\langle \hat{z}_i \rangle^p - \langle \hat{x}_i \rangle^p - \langle \hat{y}_i \rangle^p)$;
  - If $\diamondsuit_i = \times$, set $\mathbf{X} \leftarrow \mathbf{X} || (r_i \cdot \langle \hat{x}_i \rangle^p)$, $\mathbf{Y} \leftarrow \mathbf{Y} || (\langle \hat{y}_i \rangle^p)$, $\langle Z \rangle \leftarrow \langle Z \rangle + r_i \cdot \langle \hat{z}_i \rangle^p$;
- All parties send $\mathbf{X}$, $\mathbf{Y}$, and $\langle Z \rangle$ to $\mathcal{F}_{\mathsf{InnerVerify1}}$ to verify the inner-product relation and receive $b \in \{0,1\}$.

Figure 10: 3PC Batch Soft-boundary Overflow Detection Protocol.

F Actively Secure Inner Product Verification $\mathcal{F}_{\mathsf{InnerVerify2}}$

While $\mathcal{F}_{\mathsf{InnerVerify1}}$ provides a highly efficient verification mechanism in the semi-honest setting, its correctness relies on the parties faithfully constructing the verification residual and the subsequent zero test. This assumption no longer holds against a malicious adversary, who may inject additional errors during verification and correlate them with an error introduced in the original computation. To obtain active security, we therefore replace the direct residual verification with a randomized inner-product reduction. Since the preceding ring-to-field extension maps all arithmetic relations into $\mathbb{Z}_p$, the entire verification can be performed efficiently over the field. The resulting functionality $\mathcal{F}_{\mathsf{InnerVerify2}}$ consists of batch aggregation followed by recursive dimension reduction.

Batch Aggregation. After the online computation has been

Functionality $\mathcal{F}_{\mathsf{InnerVerify2}}$

$\mathcal{F}_{\mathsf{InnerVerify2}}$ interacts with parties $\mathcal{P} := \{P_0, P_1, P_2\}$ and an adversary $\mathcal{S}$. Let $\mathcal{V} := (X_0, \ldots, X_{n-1}, Y_0, \ldots, Y_{n-1}, Z)$. For every $v \in \mathcal{V}$, P_0 holds $([r_v]_1, [r_v]_2)$, P_1 holds $(m_v, [r_v]_1)$, and P_2 holds $(m_v, [r_v]_2)$ over $\mathbb{Z}_p$.

- Upon receiving $(\mathsf{input}, \mathsf{sid}, (\langle \mathbf{X} \rangle, \langle \mathbf{Y} \rangle, \langle Z \rangle)_i)$ from every $P_i \in \mathcal{P}$, the functionality checks that all submissions use the same session identifier and vector dimension.
- For every $v \in \mathcal{V}$, it checks that the copies of $[r_v]_1$ submitted by P_0 and P_1 agree, the copies of $[r_v]_2$ submitted by P_0 and P_2 agree, and the copies of m_v submitted by P_1 and P_2 agree. If any check fails, it sends $(\mathsf{abort}, \mathsf{sid})$ to all parties and halts.
- It reconstructs every $v \in \mathcal{V}$ as $v = m_v + [r_v]_1 + [r_v]_2 \pmod p$ and computes $\Delta = \sum_{j=0}^{n-1} X_j Y_j - Z \pmod p$. It sets $b = 0$ if $\Delta = 0$, and sets $b = 1$ otherwise.
- It notifies $\mathcal{S}$ that the result is ready, without revealing b. Upon receiving $(\mathsf{abort}, \mathsf{sid})$ from $\mathcal{S}$, it sends $(\mathsf{abort}, \mathsf{sid})$ to all parties and halts. Otherwise, it sends $(\mathsf{output}, \mathsf{sid}, b)$ to all parties and halts.

Figure 11: Actively Secure Inner-Product Verification Functionality with Abort.

completed, the parties jointly generate fresh random coefficients $r_i \xleftarrow{\$} \mathbb{Z}_p$. Sampling these coefficients after the corresponding arithmetic values have been fixed prevents a malicious party from choosing its computation errors as a function of the verification randomness. As in $\mathcal{F}_{\mathsf{InnerVerify1}}$, all multiplication relations are combined into one inner-product statement. For a multiplication tuple satisfying $x_i y_i = z_i$, the parties append $r_i x_i$ to $\mathbf{X}$ and y_i to $\mathbf{Y}$, while accumulating $r_i z_i$ into Z. Linear relations can be absorbed directly into Z. Consequently, the correctness of the entire batch is reduced to verifying $\mathbf{X} \cdot \mathbf{Y} = \sum_{i=0}^{M-1} X_i Y_i = Z \pmod p$. Here, M denotes the number of multiplicative terms after aggregation. When necessary, the vectors are padded with zero entries so that M is a power of two. If the original batch contains errors $e_i = x_i y_i - z_i$, then the aggregated statement instead satisfies $\mathbf{X} \cdot \mathbf{Y} - Z = \sum_i r_i e_i$. Therefore, unless all arithmetic relations are correct, the batch induces a nonzero random linear combination of the underlying errors except with negligible probability over the choice of $\{r_i\}$.

Recursive Dimension Reduction [35]. Directly opening $\mathbf{X} \cdot \mathbf{Y} - Z$ would again expose the verification procedure to malicious manipulation. Instead, $\mathcal{F}_{\mathsf{InnerVerify2}}$ recursively compresses the inner-product statement while preserving its validity. Consider one reduction round whose current dimension is L. The current verification target is $\sum_{i=0}^{L-1} x_i y_i = z$. The parties pair consecutive coordinates and define, for every $i \in \{0, \ldots, L/2-1\}$, two degree-one polynomials $f_i(t)$ and $g_i(t)$. Their evaluations at 0 and 1 encode the two coordinates in each pair $f_i(0) = x_{2i}, \quad f_i(1) = x_{2i+1}$, $g_i(0) = y_{2i}, \quad g_i(1) = y_{2i+1}$. Since f_i and g_i are linear, their values at 2 follow locally from interpolation $f_i(2) = 2f_i(1) - f_i(0) = 2x_{2i+1} - x_{2i}$ and $g_i(2) = 2g_i(1) - g_i(0) = 2y_{2i+1} - y_{2i}$. The parties then define the quadratic polynomial $h(t) = \sum_{i=0}^{L/2-1} f_i(t) g_i(t)$. The significance of this polynomial is that its evaluations at 0 and 1 separate the even and odd multiplication terms of the original inner product. In particular, $h(0) = \sum_{i=0}^{L/2-1} x_{2i} y_{2i}$, while $h(1) = \sum_{i=0}^{L/2-1} x_{2i+1} y_{2i+1}$. Hence, if the current inner-product claim is correct, these two evaluations must satisfy $h(0) + h(1) = z$. The parties compute a sharing of $h(0)$ using the underlying secure arithmetic over $\mathbb{Z}_p$ and bind the second evaluation directly to the current verification target by setting $\langle h(1) \rangle = \langle z \rangle - \langle h(0) \rangle$. This step is essential: $h(1)$ is not accepted as an independently supplied value. Instead, it is algebraically determined by the previous-round claim z. Therefore, any error already contained in the current relation is carried into the polynomial constraint rather than discarded during reduction.

A third evaluation is obtained from $h(2) = \sum_{i=0}^{L/2-1} f_i(2) g_i(2)$. The three shared values $\langle h(0) \rangle$, $\langle h(1) \rangle$, and $\langle h(2) \rangle$ uniquely determine the degree-two polynomial $h(t)$. Only after these values have been fixed do the parties jointly sample a fresh challenge $\zeta \xleftarrow{\$} \mathbb{Z}_p$. The parties then fold each pair of coordinates by evaluating the corresponding linear polynomials at ζ:

$$\langle x_i' \rangle = \langle f_i(\zeta) \rangle = \zeta \langle f_i(1) \rangle - (\zeta - 1) \langle f_i(0) \rangle,$$

$$\langle y_i' \rangle = \langle g_i(\zeta) \rangle = \zeta \langle g_i(1) \rangle - (\zeta - 1) \langle g_i(0) \rangle.$$

At the same challenge point, the new scalar target is obtained by interpolating h from its three fixed evaluations:

$$\langle z' \rangle = \langle h(\zeta) \rangle = \ell_0(\zeta) \langle h(0) \rangle + \ell_1(\zeta) \langle h(1) \rangle + \ell_2(\zeta) \langle h(2) \rangle,$$

where the Lagrange basis polynomials over the points $\{0, 1, 2\}$ are $\ell_0(\zeta) = \frac{(\zeta-1)(\zeta-2)}{2}$, $\ell_1(\zeta) = -\zeta(\zeta-2)$, $\ell_2(\zeta) = \frac{\zeta(\zeta-1)}{2}$. By construction,

$$\sum_{i=0}^{L/2-1} x_i' y_i' = \sum_{i=0}^{L/2-1} f_i(\zeta) g_i(\zeta) = h(\zeta) = z'.$$

Thus, a valid L-dimensional inner-product relation is transformed into a valid $L/2$-dimensional relation. The parties replace $(\mathbf{x}, \mathbf{y}, z)$ with $(\mathbf{x}', \mathbf{y}', z')$ and repeat the same procedure. After $\log_2 M$ rounds, the original batch relation is reduced to a single multiplication claim $x^\star y^\star = z^\star$. The final scalar relation is then checked using the actively secure multiplication consistency mechanism over $\mathbb{Z}_p$.

Soundness. The ordering of the reduction is crucial for active security. In every round, the values determining the quadratic polynomial $h(t)$ are fixed before the random challenge ζ is generated. Suppose a malicious execution makes the current relation inconsistent. The honest relation and the adversarially induced relation then differ by a nonzero polynomial $q(t)$

of degree at most two. For the incorrect relation to survive the reduction, the sampled challenge must satisfy $q(\zeta) = 0$. A nonzero degree-two polynomial over $\mathbb{Z}_p$ has at most two roots. Therefore, for a challenge sampled uniformly from the field, an inconsistent relation survives one folding round with probability at most $2/p$. Applying the reduction over $\log_2 M$ rounds gives an overall soundness error bounded by $O(\log M/p)$, in addition to the failure probability of the initial random batch aggregation.

The main advantage of $\mathcal{F}_{\mathsf{InnerVerify2}}$ is that active verification is performed on one aggregated relation rather than independently on every multiplication gate. Each folding round reduces the dimension by a factor of two, and the protocol reaches a scalar relation after only $\log_2 M$ rounds. Since all operations are carried out in the extended field $\mathbb{Z}_p$, the verification cost grows only logarithmically with the batch dimension and can be efficiently amortized when a large number of arithmetic operations are verified together.

Why the Semi-Honest Inner-Product Check Does Not Directly Extend to the Malicious Setting. Simply multiplying the verification residual by a random challenge α fails in the malicious setting because the verification computation itself is subject to errors. Consider a corrupted multiplication $XY = z + e$. To verify this, the protocol computes $\alpha X \cdot Y \stackrel{?}{=} \alpha z$. However, a malicious party can inject subsequent errors e', e'', e''' during these randomized operations:

- Randomized input: $X' = \alpha X - e'$
- Verification multiplication: $z'' = X'Y - e'' = \alpha XY - e'Y - e''$
- Randomized output: $z' = \alpha z - e'''$

The final equality check requires $z'' = z'$. Substituting $XY = z + e$ into this equation yields the acceptance condition: $\alpha e = e'Y + e'' - e'''$. If the adversary learns α before committing to these subsequent verification errors, they can adaptively choose them (e.g., $e''' = e'Y + e'' - \alpha e$) to satisfy this condition. This perfectly cancels the original error e, allowing an invalid multiplication to pass the check.This vulnerability motivates the structure of $\mathcal{F}_{\mathsf{InnerVerify2}}$. Instead of exposing a random challenge prematurely, the protocol ensures all polynomial relations are committed *before* a fresh challenge ζ is sampled. Consequently, any adversarial deviation becomes a fixed polynomial, and the adversary cannot adaptively construct challenge-dependent errors to bypass the verification.

G Inner Product Overflow Detection Protocol

To accommodate multi-operator blocks such as multi-dimensional inner products, which are foundational to privacy-preserving inference, the single-operator boundary no longer suffices. Since an inner product involves cascading additions over multiple multiplication gates, the cumulative integer value scales with the vector dimension N. We formalize the exact ring extension security boundary for inner product circuits in Theorem 7, and specify the concrete execution and verification workflows of Protocol $\Pi_{\mathsf{inner\text{-}detect}}$ for 2PC and 3PC settings in Figure 12 and Figure 13, respectively.

Theorem 7. *Let* $\mathbf{x} = (x_1, \ldots, x_N) \in \mathbb{Z}_{2^\ell}^N$ *and* $\mathbf{y} = (y_1, \ldots, y_N) \in \mathbb{Z}_{2^\ell}^N$. *Let* $x \in \mathbb{Z}_{2^\ell}$ *and* $x' \in \mathbb{Z}_p$ *satisfy* $x = \sum_{i=1}^N x_i y_i \pmod{2^\ell}$ *and* $x' = \sum_{i=1}^N x_i y_i \pmod{p}$, *respectively, where* p *is a prime number satisfying* $p > \max\left(2^\ell, \left\lfloor \frac{N(2^\ell-1)^2}{2^\ell} \right\rfloor\right)$. *Then* $x = x'$ *if and only if* $\sum_{i=1}^N x_i y_i < 2^\ell$.

Proof. i. For $\sum_{i=1}^N x_i y_i < 2^\ell$, it follows directly that no modular reduction occurs in either $\mathbb{Z}_{2^\ell}$ or $\mathbb{Z}_p$. Since the boundary condition strictly enforces $p > \max\left(2^\ell, \left\lfloor \frac{N(2^\ell-1)^2}{2^\ell} \right\rfloor\right) \geq 2^\ell$, we have $p > 2^\ell > \sum_{i=1}^N x_i y_i$. Both local evaluations yield the exact true integer value, trivially implying $x = x'$.

ii. We prove by contradiction that an overflow ($\sum_{i=1}^N x_i y_i \geq 2^\ell$) cannot yield $x = x'$. By the division algorithm over $\mathbb{Z}$, the true integer sum is expressed as $\sum_{i=1}^N x_i y_i = a2^\ell + x = bp + x'$ for some wrap-around count $a \in \mathbb{Z}^+$ and $b \in \mathbb{N}$. Since $x_i, y_i \in \mathbb{Z}_{2^\ell}$, the wrap count a is upper-bounded by:

$$a = \left\lfloor \frac{\sum_{i=1}^N x_i y_i - x}{2^\ell} \right\rfloor \leq \left\lfloor \frac{N(2^\ell - 1)^2}{2^\ell} \right\rfloor.$$

Assuming $x = x'$, subtracting the two forms yields $a2^\ell = bp$. Since p is prime and $p > 2^\ell$, $\gcd(2^\ell, p) = 1$. By Euclid's Lemma, $a2^\ell = bp$ implies $p \mid a$. However, the upper bound enforces $1 \leq a \leq \left\lfloor \frac{N(2^\ell-1)^2}{2^\ell} \right\rfloor < p$. A positive integer $a \in [1, p-1]$ cannot be divisible by p, yielding a contradiction. Hence, $a = 0$, guaranteeing that $x = x'$ if and only if $\sum_{i=1}^N x_i y_i < 2^\ell$. □

Remark. The condition $p > \max\left(2^\ell, \left\lfloor \frac{N(2^\ell-1)^2}{2^\ell} \right\rfloor\right)$ guarantees soundness across all dimensions $N \geq 1$ and bit lengths $\ell \geq 1$, including the degenerate case $N = 1$. It enforces $\gcd(2^\ell, p) = 1$, establishing a collision-resistance argument via Euclid's Lemma that maps base-ring wrap-around errors to the impossibility of non-trivial integer solutions in $\mathbb{Z}_p$. Furthermore, the parameterized lifting function f_w must explicitly match the bit-width of each evaluated secret: input vector components $x_i, y_i \in \mathbb{Z}_{2^\ell}$ require f_ℓ with a 2^ℓ wrap-around step size, whereas the soft-boundary accumulated scalar z—truncated modulo $2^{\ell-1}$—strictly requires $f_{\ell-1}$ with a $2^{\ell-1}$ step size to guarantee correct integer alignment in $\mathbb{Z}_p$.

H Actively Secure Fixed-Point Arithmetic Soft-boundary Overflow Detection Protocol

This appendix presents the actively secure fixed-point overflow-detection protocol $\Pi_{\mathsf{fix\text{-}arithm}}$ (Figure 14) introduced in the main body. The protocol batches N addition or multiplication operations over $\mathbb{Z}_{2^\ell}$. It first computes the results over

Protocol $\Pi_{\mathsf{inner\text{-}detect}}(\otimes, [\mathbf{x}]^{2^\ell}, [\mathbf{y}]^{2^\ell})$

Input : Vector sharings $[\mathbf{x}]^{2^\ell} = ([x_1]^{2^\ell}, \ldots, [x_N]^{2^\ell})$ and $[\mathbf{y}]^{2^\ell} = ([y_1]^{2^\ell}, \ldots, [y_N]^{2^\ell})$; A prime modulus K satisfying $K > \max\left(2^\ell, \left\lfloor \frac{N(2^\ell-1)^2}{2^\ell} \right\rfloor\right)$.
Output : A soft-boundary interval detection result b over the inner product.
Execution:

- All parties send $(\mathsf{input}, \otimes, [\mathbf{x}]^{2^\ell}, [\mathbf{y}]^{2^\ell})$ to $\mathcal{F}_{\mathsf{2pc}}$ and receive $[z]^{2^\ell}$.
- For each component $i \in [N]$, P_j invokes $[b_{x_i}^{(j)}]^p \leftarrow \Pi_{[\cdot]}^p(\mathsf{msb}([x_i]_j))$ and $[b_{y_i}^{(j)}]^p \leftarrow \Pi_{[\cdot]}^p(\mathsf{msb}([y_i]_j))$ for $j \in \{1,2\}$;
- For the accumulated inner product scalar z:
  - P_j invokes $[b_z^{(j)}]^p \leftarrow \Pi_{[\cdot]}^p(\mathsf{smsb}([z]_j))$ for $j \in \{1,2\}$;
  - P_j set $[\bar{z}]_j^p = [z]_j^{2^\ell} \pmod{2^{\ell-1}}$;
- All parties define the parameterized 2PC lifting function $f_w(a,b,c) = a - 2^w \times (b + c - b \times c)$;
- For each $i \in [N]$, all parties send:
  - $(\mathsf{input}, f_\ell, [x_i]^p, [b_{x_i}^{(1)}], [b_{x_i}^{(2)}])$, $(\mathsf{input}, f_\ell, [y_i]^p, [b_{y_i}^{(1)}], [b_{y_i}^{(2)}])$ to $\mathcal{F}_{\mathsf{2pc}}$ and receive $[\hat{x}_i]^p$, and $[\hat{y}_i]^p$, respectively.
- All parties lift the soft-boundary scalar z, send $(\mathsf{input}, f_{\ell-1}, [\bar{z}]^p, [b_z^{(1)}], [b_z^{(2)}])$ to $\mathcal{F}_{\mathsf{2pc}}$ and receive $[\hat{z}]^p$.
- All parties re-evaluate the inner product, send $(\mathsf{input}, (\otimes, [\hat{\mathbf{x}}]^p, [\hat{\mathbf{y}}]^p)$ to $\mathcal{F}_{\mathsf{2pc}}$ and receive $[z']^p$.
- All parties send $[z']^p$ and $[\hat{z}]^p$ to $\mathcal{F}_{\mathsf{eq}}[K]$ to verify $z' = \hat{z}$.

Figure 12: 2PC Soft-boundary Inner Product Overflow Detection Protocol.

the ring, then verifies the arithmetic relations and the consistency of secret-shared sign and wraparound information in the field $\mathbb{Z}_p$ using pseudorandom linear combinations.

We show the security of $\Pi_{\mathsf{fix\text{-}arithm}}$ with the following theorem.

Theorem 8. *Let λ be a security parameter, p be a prime number and $p > 2^\lambda$. Let the PRG used in the sub-protocol $\Pi_{[\cdot]}$ (c.f. Sec. 2) in the protocol $\Pi_{\mathsf{fix\text{-}arithm}}$ be a secure peseudorandom geneartor with advantage $\mathsf{Adv}_{\mathsf{PRG}}$. The protocol $\Pi_{\mathsf{fix\text{-}arithm}}$ in Fig. 14 UC-realizes $\mathcal{F}_{\mathsf{fixed\text{-}arithm}}$ in Fig. 5 in $\{\mathcal{F}_{\mathsf{3pc}}, \mathcal{F}_{\mathsf{active3pc}}, \mathcal{F}_{\mathsf{InnerVerify2}}\}$-hybrid model against a malicious adversary who can statically corrupted up to 1 party.*

The proof of Theorem 8 is similar to that of Theorem 6 in Appendix N.

I Detailed Theoretical Analysis of Fixed-Point Overflow

Extended Boundary Derivation. Upon local cross-multiplication in $\mathbb{Z}_{2^\ell}$, the fractional scale quadratically expands to $2\ell_f$ bits, yielding an intermediate product

Protocol $\Pi_{\mathsf{inner\text{-}detect}}(\otimes, \langle\mathbf{x}\rangle^{2^\ell}, \langle\mathbf{y}\rangle^{2^\ell})$

Input : Vector sharings $\langle\mathbf{x}\rangle^{2^\ell} = (\langle x_1\rangle^{2^\ell}, \ldots, \langle x_N\rangle^{2^\ell})$ and $\langle\mathbf{y}\rangle^{2^\ell} = (\langle y_1\rangle^{2^\ell}, \ldots, \langle y_N\rangle^{2^\ell})$; A prime modulus K satisfying $K > \max\left(2^\ell, \left\lfloor \frac{N(2^\ell-1)^2}{2^\ell} \right\rfloor\right)$.
Output : A soft-boundary interval detection result b over the inner product.
Execution:

- All parties send $(\mathsf{input}, \otimes, \langle\mathbf{x}\rangle^{2^\ell}, \langle\mathbf{y}\rangle^{2^\ell})$ to $\mathcal{F}_{\mathsf{3pc_inner}}$ and receive $\langle z\rangle^{2^\ell}$.
- For each component $i \in [N]$, all parties do:
  - P_0 invokes $\langle b_{x_i}^{(1)}\rangle^p \leftarrow \Pi_{\langle\cdot\rangle}^K(\mathsf{msb}(r_{x_i}))$ and $\langle b_{y_i}^{(1)}\rangle^p \leftarrow \Pi_{\langle\cdot\rangle}^K(\mathsf{msb}(r_{y_i}))$;
  - P_0 invokes $\langle b_{x_i}^{(3)}\rangle^p \leftarrow \Pi_{\langle\cdot\rangle}^K(\mathsf{wrap}([r_{x_i}]_1, [r_{x_i}]_2))$ and $\langle b_{y_i}^{(3)}\rangle^p \leftarrow \Pi_{\langle\cdot\rangle}^K(\mathsf{wrap}([r_{y_i}]_1, [r_{y_i}]_2))$;
  - All parties set $\langle b_{x_i}^{(2)}\rangle^p := (\mathsf{msb}(m_{x_i}), 0)$ and $\langle b_{y_i}^{(2)}\rangle^p := (\mathsf{msb}(m_{y_i}), 0)$;
- For the accumulated inner product scalar z, all parties do:
  - P_0 invokes $\langle b_z^{(1)}\rangle^p \leftarrow \Pi_{\langle\cdot\rangle}^K(\mathsf{smsb}(r_z))$ and $\langle b_z^{(3)}\rangle^p \leftarrow \Pi_{\langle\cdot\rangle}^K(\mathsf{wrap}^{2^{\ell-1}}([r_z]_1, [r_z]_2))$;
  - All parties set $\langle b_z^{(2)}\rangle^p := (\mathsf{smsb}(m_z), 0)$;
  - All parties locally define the truncated scalar share $\langle\bar{z}\rangle^p = \langle z\rangle^{2^\ell} \pmod{2^{\ell-1}}$;
- All parties define the parameterized lifting function $f_w(a,b,c,d) = a - 2^w \times (b + c - b \times c + d)$;
- For each $i \in [N]$, all parties send:
  - $(\mathsf{input}, f_\ell, \langle x_i\rangle^p, \langle b_{x_i}^{(1)}\rangle, \langle b_{x_i}^{(2)}\rangle, \langle b_{x_i}^{(3)}\rangle)$ to $\mathcal{F}_{\mathsf{3pc}}$ and receive $\langle\hat{x}_i\rangle^p$.
  - $(\mathsf{input}, f_\ell, \langle y_i\rangle^p, \langle b_{y_i}^{(1)}\rangle, \langle b_{y_i}^{(2)}\rangle, \langle b_{y_i}^{(3)}\rangle)$ to $\mathcal{F}_{\mathsf{3pc}}$ and receive $\langle\hat{y}_i\rangle^p$.
- All parties lift the soft-boundary scalar z by sending $\mathsf{input}, (f_{\ell-1}, \langle\bar{z}\rangle^p, \langle b_z^{(1)}\rangle, \langle b_z^{(2)}\rangle, \langle b_z^{(3)}\rangle)$ to $\mathcal{F}_{\mathsf{3pc}}$ and receiving $\langle\hat{z}\rangle^p$.
- All parties send $(\mathsf{input}, \otimes, \langle\hat{\mathbf{x}}\rangle^p, \langle\hat{\mathbf{y}}\rangle^p)$ to $\mathcal{F}_{\mathsf{3pc}}$ and receive $\langle z'\rangle^p$.
- All parties send $\langle z'\rangle^p$ and $\langle\hat{z}\rangle^p$ to $\mathcal{F}_{\mathsf{eq}}[K]$ to verify $z' = \hat{z}$.

Figure 13: 3PC Inner Product Soft-boundary Overflow Detection Protocol.

bounded by $|\hat{x}_i \cdot \hat{y}_i| < M^2 \cdot 2^{2\ell_f}$. Under the worst-case scenario where all d dimensions saturate to M coherently, the sum expands to $d \cdot M^2 \cdot 2^{2\ell_f}$, leading directly to Equation 6.

Statistical Average-Case Bounds. In practical machine learning workloads, positive and negative terms partially cancel out. Assuming i.i.d. Gaussian inputs with variance $\sigma^2 \propto M^2$, the Central Limit Theorem dictates that the aggregate standard deviation scales with $\sqrt{d}$. Under a 3σ confidence interval, the probabilistic three-standard-deviation operating threshold $d_{3\sigma}$ relaxes to:

Protocol $\langle z\rangle \leftarrow \Pi_{\mathsf{fix\text{-}arithm}}(\diamondsuit, \langle x\rangle^{2^\ell}, \langle y\rangle^{2^\ell})$

Input : N tuples $\{\diamondsuit_i, \langle x_i\rangle^{2^\ell}, \langle y_i\rangle^{2^\ell}\}_{i\in[N]}$

Output : a soft-boundary overflow detection result b, which holds $b \leftarrow \bigvee_{i=0}^{N-1} \mathcal{G}(x_i \diamondsuit y_i, \mathbb{B} = (2^{\ell-2}, 2^{\ell-1}))$.

Execution: For each $i \in [N]$, all parties do

- All parties invoke $\langle z_i\rangle^{2^\ell} \leftarrow \Pi_{\mathsf{3pc}}^{2^\ell}(\diamondsuit_i, \langle x_i\rangle^{2^\ell}, \langle y_i\rangle^{2^\ell})$;

Verification:

- All parties set $\mathbf{A} = ()$, $\mathbf{B} = ()$ and $C = \langle 0\rangle$;
- For each $i \in [N]$, all parties do
  - P_0 invokes $\langle b_{x_i}^{(1)}\rangle^p \leftarrow \Pi_{\langle\cdot\rangle}^p(\mathsf{msb}(r_{x_i}))$, $\langle b_{y_i}^{(1)}\rangle^p \leftarrow \Pi_{\langle\cdot\rangle}^p(\mathsf{msb}(r_{y_i}))$ and $\langle b_{z_i}^{(1)}\rangle^p \leftarrow \Pi_{\langle\cdot\rangle}^p(\mathsf{smsb}(r_{z_i}))$;
  - P_0 invokes $\langle b_{x_i}^{(3)}\rangle^p \leftarrow \Pi_{\langle\cdot\rangle}^p(\mathsf{wrap}([r_{x_i}]_1, [r_{x_i}]_2))$, $\langle b_{y_i}^{(3)}\rangle^p \leftarrow \Pi_{\langle\cdot\rangle}^p(\mathsf{wrap}([r_{y_i}]_1, [r_{y_i}]_2))$ and $\langle b_{z_i}^{(3)}\rangle^p \leftarrow \Pi_{\langle\cdot\rangle}^p(\mathsf{wrap}^{2^{\ell-1}}([r_{z_i}]_1, [r_{z_i}]_2))$;
  - P_1 invokes $\langle c_{x_i}^{(1)}\rangle^p \leftarrow \Pi_{\langle\cdot\rangle}^p(\mathsf{msb}([r_{x_i}]_1 + m_{x_i}))$, $\langle c_{y_i}^{(1)}\rangle^p \leftarrow \Pi_{\langle\cdot\rangle}^p(\mathsf{msb}([r_{y_i}]_1 + m_{y_i}))$ and $\langle c_{z_i}^{(1)}\rangle^p \leftarrow \Pi_{\langle\cdot\rangle}^p(\mathsf{smsb}([r_{z_i}]_1 + m_{z_i}))$;
  - P_1 invokes $\langle c_{x_i}^{(3)}\rangle^p \leftarrow \Pi_{\langle\cdot\rangle}^p(\mathsf{wrap}([r_{x_i}]_1, m_{x_i}))$, $\langle c_{y_i}^{(3)}\rangle^p \leftarrow \Pi_{\langle\cdot\rangle}^p(\mathsf{wrap}([r_{y_i}]_1, m_{y_i}))$ and $\langle c_{z_i}^{(3)}\rangle^p \leftarrow \Pi_{\langle\cdot\rangle}^p(\mathsf{wrap}^{2^{\ell-1}}([r_{z_i}]_1, m_{z_i}))$;
  - All parties set $\langle b_{x_i}^{(2)}\rangle^p := (\mathsf{msb}(m_{x_i}), 0, 0)$, $\langle b_{y_i}^{(2)}\rangle^p := (\mathsf{msb}(m_{y_i}), 0, 0)$ and $\langle b_{z_i}^{(2)}\rangle^p := (\mathsf{smsb}(m_{z_i}), 0, 0)$;
  - All parties set $\langle c_{x_i}^{(2)}\rangle^p := (0, \mathsf{msb}([r_{x_i}]_1), 0)$, $\langle c_{y_i}^{(2)}\rangle^p := (0, \mathsf{msb}([r_{y_i}]_1), 0)$ and $\langle c_{z_i}^{(2)}\rangle^p := (0, \mathsf{smsb}([r_{z_i}]_1), 0)$;
  - All parties set $\langle \hat{z}_i\rangle_j^p = \langle z_i\rangle_j^{2^\ell} \pmod{2^{\ell-1}}$, $\langle \hat{x}_i\rangle_j^p = \langle x_i\rangle_j^{2^\ell}$ and $\langle \hat{y}_i\rangle_j^p = \langle y_i\rangle_j^{2^\ell}$ for $j \in [3]$;
  - All parties set $f(a,b,c,d) = a - 2^\ell \times (b + c - b \times c + d)$;
  - All parties invoke $\langle \hat{x}_i\rangle^p \leftarrow \Pi_{\mathsf{3pc}}^p(f, \langle \hat{x}_i\rangle^p, \langle b_{x_i}^{(1)}\rangle, \langle b_{x_i}^{(2)}\rangle, \langle b_{x_i}^{(3)}\rangle)$, $\langle \hat{y}_i\rangle^p \leftarrow \Pi_{\mathsf{3pc}}^p(f, \langle \hat{y}_i\rangle^p, \langle b_{y_i}^{(1)}\rangle, \langle b_{y_i}^{(2)}\rangle, \langle b_{y_i}^{(3)}\rangle)$, and $\langle \hat{z}_i\rangle^p \leftarrow \Pi_{\mathsf{3pc}}^p(f, \langle \hat{z}_i\rangle^p, \langle b_{z_i}^{(1)}\rangle, \langle b_{z_i}^{(2)}\rangle, \langle b_{z_i}^{(3)}\rangle)$;
- All parties compute $\langle r\rangle \leftarrow \Pi_{\langle\cdot\rangle}$.
- For each $i \in [N]$, all parties do
  - If $\diamondsuit = +$, set $\langle C\rangle \leftarrow \langle C\rangle + r_i(\langle z_i\rangle - \langle x_i\rangle - \langle y_i\rangle)$;
  - If $\diamondsuit = \times$, set $\mathbf{A} \leftarrow \mathbf{A}||(r_i \cdot \langle x_i\rangle)$, $\mathbf{B} \leftarrow \mathbf{B}||(\langle y_i\rangle)$, $\langle C\rangle \leftarrow \langle C\rangle + r_i \cdot \langle \hat{z}_i\rangle^p$;
  - $\mathbf{A} \leftarrow \mathbf{A}||(r^{N+i} \cdot \langle b_{x_i}^{(1)}\rangle, r^{2N+i} \cdot \langle b_{y_i}^{(1)}\rangle, r^{3N+i} \cdot \langle b_{z_i}^{(1)}\rangle)$;
  - $\mathbf{B} \leftarrow \mathbf{B}||(\langle b_{x_i}^{(2)}\rangle, \langle b_{y_i}^{(2)}\rangle, \langle b_{z_i}^{(2)}\rangle)$;
  - $\langle C\rangle \leftarrow \langle C\rangle + r^{N+i} \cdot \langle b_{x_i}^{(1)}\rangle + r^{2N+i} \cdot \langle b_{y_i}^{(1)}\rangle + r^{3N+i} \cdot \langle b_{z_i}^{(1)}\rangle$;
- For each $i \in [N]$, all parties do
  - If $\diamondsuit = +$, set $\langle C\rangle \leftarrow \langle C\rangle + r_i(\langle z_i\rangle - \langle x_i\rangle - \langle y_i\rangle)$;
  - If $\diamondsuit = \times$, set $\mathbf{A} \leftarrow \mathbf{A}||(r_i \cdot \langle x_i\rangle)$, $\mathbf{B} \leftarrow \mathbf{B}||(\langle y_i\rangle)$, $\langle C\rangle \leftarrow \langle C\rangle + r_i \cdot \langle \hat{z}_i\rangle^p$;
  - $\mathbf{A} \leftarrow \mathbf{A}||(r^{N+i} \cdot \langle b_{x_i}^{(1)}\rangle, r^{N+i} \cdot \langle c_{x_i}^{(1)}\rangle, r^{2N+i} \cdot \langle b_{y_i}^{(1)}\rangle, r^{2N+i} \cdot \langle c_{y_i}^{(1)}\rangle, r^{3N+i} \cdot \langle b_{z_i}^{(1)}\rangle, r^{3N+i} \cdot \langle c_{z_i}^{(1)}\rangle)$;
  - $\mathbf{B} \leftarrow \mathbf{B}||(\langle b_{x_i}^{(2)}\rangle, -\langle c_{x_i}^{(2)}\rangle, \langle b_{y_i}^{(2)}\rangle, -\langle c_{y_i}^{(2)}\rangle, \langle b_{z_i}^{(2)}\rangle, -\langle c_{z_i}^{(2)}\rangle)$;
  - $\langle C\rangle \leftarrow \langle C\rangle + r^{N+i} \cdot \big(\langle b_{x_i}^{(1)}\rangle + \langle b_{x_i}^{(2)}\rangle + \langle b_{x_i}^{(3)}\rangle - \langle c_{x_i}^{(1)}\rangle - \langle c_{x_i}^{(2)}\rangle - \langle c_{x_i}^{(3)}\rangle\big) + r^{2N+i} \cdot \big(\langle b_{y_i}^{(1)}\rangle + \langle b_{y_i}^{(2)}\rangle + \langle b_{y_i}^{(3)}\rangle - \langle c_{y_i}^{(1)}\rangle - \langle c_{y_i}^{(2)}\rangle - \langle c_{y_i}^{(3)}\rangle\big) + r^{3N+i} \cdot \big(\langle b_{z_i}^{(1)}\rangle + \langle b_{z_i}^{(2)}\rangle + \langle b_{z_i}^{(3)}\rangle - \langle c_{z_i}^{(1)}\rangle - \langle c_{z_i}^{(2)}\rangle - \langle c_{z_i}^{(3)}\rangle\big)$;
- All parties send $\mathbf{A}, \mathbf{B}, C$ to $\mathcal{F}_{\mathsf{InnerVerify2}}$ and receive $b \in \{0,1\}$.
- If $b = 1$, all parties output abort and halt.
- If $b = 0$, all parties receive the secret shares of z_i from $\mathcal{F}_{\mathsf{InnerVerify2}}$ and set them as the protocol output.

Figure 14: Actively Secure Fixed-Point Arithmetic Soft-boundary Overflow Detection Protocol.

$$d_{3\sigma} \approx \left(\frac{2^{\ell-1-2\ell_f}}{3\sigma^2}\right)^2 = 2^{2(\ell-1-2\ell_f)-2\log_2 3-4\log_2 \sigma}. \quad (7)$$

This is a probabilistic operating estimate rather than a deterministic safety guarantee.

J Actively Secure PPML

This appendix presents the actively secure transformer-layer protocol $\Pi_{\mathsf{Active\text{-}Transformer}}$ 15. The protocol evaluates the layer over secret shares, records the required arithmetic and verifies the aggregated relations in the field $\mathbb{Z}_p$ before reconstructing the output.

K Performance of $\Pi_{\mathsf{active\text{-}batch\text{-}detect}}$

To evaluate the performance of the 3PC-$\Pi_{\mathsf{active\text{-}batch\text{-}detect}}$ protocol, we benchmark its batched addition and multiplication operations under LAN, MAN, and WAN settings across varying batch sizes N. The results shown in Table 8.

L Performance Breakdown of the Individual Operations

Tables 9 and 10 present the LAN performance breakdown of Li *et al.* [31] and MP-SPDZ [27] for BERT-base and GPT-2, respectively. Tables 11 and 12 report the corresponding results under MAN and WAN settings. Table 13 provides the operation-level performance breakdown of our protocol.

Protocol $\Pi_{\text{Active-Transformer}}$

Parties: P_0, P_1, P_2, holding secret-shared inputs $\mathbf{X}$ and $\{\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V, \mathbf{W}_O\}$. Each pair of parties holds a common seed $\eta_{j,k} \in \{0,1\}^\lambda$.
Params: Block topology $\mathcal{C}$, prime field $\mathbb{F}_p$, and extension field $\mathbb{F}_{p^r}$. Unless explicitly opened, all intermediate values below remain secret-shared; sharing notation is omitted for clarity.

Phase 1: Arithmetic Pipeline Evaluation

1. **Linear Projections (QKV):** For $i \in \{Q, K, V\}$, compute $\mathbf{M}_i = \mathbf{X}\mathbf{W}_i$. For every underlying multiplication gate, invoke $\mathcal{F}_{\mathsf{mult}}$ to obtain the untruncated intermediate value $z \in \mathbb{Z}_{2^{\ell+\ell_f}}$, and append (u, v, z) to $\mathcal{G}_{\mathsf{mult}}$.
2. **Attention Score Computation:** Compute $\mathbf{A}_{\mathsf{raw}} \leftarrow \Pi_{\mathsf{3pc}}(\mathbf{Q}\mathbf{K}^T)$. Record the untruncated intermediate values of all underlying multiplication gates in $\mathcal{G}_{\mathsf{mult}}$.
3. **Activation and Context Computation:** Compute $\mathbf{A}_{\mathsf{activated}} \leftarrow \mathsf{Softmax}(\mathbf{A}_{\mathsf{raw}})$, using element-wise piecewise protocols, and record the relevant addition gates in $\mathcal{G}_{\mathsf{add}}$. Then compute $\mathbf{H} \leftarrow \Pi_{\mathsf{3pc}}(\mathbf{A}_{\mathsf{activated}}\mathbf{V})$, and record its multiplication gates in $\mathcal{G}_{\mathsf{mult}}$.
4. **Output Projection and Residual Addition:** Compute $\mathbf{O} \leftarrow \Pi_{\mathsf{3pc}}(\mathbf{H}\mathbf{W}_O)$, $\mathbf{X}_{\mathsf{res}} = \mathbf{O} + \mathbf{X}$. Record the multiplication gates of the output projection in $\mathcal{G}_{\mathsf{mult}}$ and the residual additions in $\mathcal{G}_{\mathsf{add}}$.

Phase 2: Active Batch Verification

1. **Offline Preparation:** Using the pairwise seeds $\eta_{j,k}$, the parties emulate the offline phases of the addition- and multiplication-detection protocols. They invoke $\mathcal{F}_{\mathsf{share}}$ to generate the random masks and wraparound information required for all gates in $\mathcal{G}_{\mathsf{add}}$ and $\mathcal{G}_{\mathsf{mult}}$.
2. **Online Algebraic Reduction:**
   - For $\mathcal{G}_{\mathsf{add}}$, extract the masked values, derive a and $\hat{a}$ over $\mathbb{F}_p$, and obtain audited field triples $(\mathcal{X}_i, \mathcal{Y}_i, \mathcal{Z}_i)$ through $\mathcal{F}_{\mathsf{share}}$.
   - For $\mathcal{G}_{\mathsf{mult}}$, perform the inner-product reduction $c = b^1 a$ over $\mathbb{F}_p$ and convert the recorded wraparound information into field relations $(\mathcal{X}_k, \mathcal{Y}_k, \mathcal{Z}_k)$.
3. **Batch Enforcement:** The parties jointly sample a challenge $\eta \in \mathbb{F}_p$, aggregate all reduced field relations using random linear combinations, and invoke $\mathcal{F}_{\mathsf{verify}}[K]$:
   - **Abort:** If $\mathcal{F}_{\mathsf{verify}}[K] \to b = 0$, indicating share tampering or boundary overflow, all parties abort and output $\perp$.
   - **Success:** If $\mathcal{F}_{\mathsf{verify}}[K] \to b = 1$, the parties execute the output-reconstruction step of Π_{3pc} and open $\mathbf{X}_{\mathsf{res}}$.

Figure 15: Actively Secure Transformer Layer Pipeline with Batch Verification.

M Proof of Theorem 4

Theorem 4 (restated). The protocol Π_{detect} in Figure 2 UC-realizes $\mathcal{F}_{\mathsf{detect}}$ in Figure 1 in the $\{\mathcal{F}_{\mathsf{3pc}}, \mathcal{F}_{\mathsf{eq}}\}$-hybrid model against a semi-honest adversary that can statically corrupt at most one party.

Proof. To prove Thm. 4, we construct a simulator $\mathcal{S}$ that internally runs a semi-honest adversary $\mathcal{A}$, interacts with $\mathcal{F}_{\mathsf{detect}}$ and forwarding messages to/from the environment $\mathcal{Z}$ in the ideal world. Then we shows that any PPT environment $\mathcal{Z}$ cannot distinguish the simulation in the ideal world execution from the real-world execution more than a negligible probability. We consider the following cases.

Case 1: Corrupted P_0. In this case, we build the simulator $\mathcal{S}$ as follows, which emulates the interface of $\mathcal{F}_{\mathsf{3pc}}$, $\mathcal{F}_{\mathsf{eq}}$ as well as the honest parties P_1, P_2:

- In the initialization phase, $\mathcal{S}$ plays the role of P_1 and P_2 to agree on the seeds $\eta_{0,1}, \eta_{0,2}$ with P_0 as real world;
- $\mathcal{S}$ records P_0's input $(([r_x]_1^{2^\ell}, [r_x]_2^{2^\ell}), ([r_y]_1^{2^\ell}, [r_y]_2^{2^\ell}))$;
- Upon receiving $(\mathsf{input}, \mathsf{sid}, \diamondsuit, (([r_x]_1^{2^\ell}, [r_x]_2^{2^\ell}), ([r_y]_1^{2^\ell}, [r_y]_2^{2^\ell})))$ from P_0 to $\mathcal{F}_{\mathsf{3pc}}$, $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{3pc}}$ to randomly sample $[r_z]_1^{2^\ell}, [r_z]_2^{2^\ell} \xleftarrow{\$} \mathbb{Z}_{2^\ell}$, and send them to P_0;
- Upon receiving $m_{b_x^{(1)}}, m_{b_x^{(2)}}, m_{b_x^{(3)}}$ from P_0 to P_1, P_2, and receiving $(\mathsf{input}, \mathsf{sid}, f, (([r_x]_1^p, [r_x]_2^p), ([r_{b_x^{(1)}}]_1, [r_{b_x^{(1)}}]_2), ([r_{b_x^{(2)}}]_1, [r_{b_x^{(2)}}]_2), ([r_{b_x^{(3)}}]_1, [r_{b_x^{(3)}}]_2)))$ from P_0 to $\mathcal{F}_{\mathsf{3pc}}$,

Table 8: Performance of 3PC-$\Pi_{\mathsf{active\text{-}batch\text{-}detect}}$ under different network settings ($\ell = 32$). The reported results are averaged over five independent runs.

Protocol $\Pi^{\mathsf{active\text{-}batch}}_{\mathsf{detect}}$	N	Offline (ms) LAN	Offline (ms) MAN	Offline (ms) WAN	Online (ms) LAN	Online (ms) MAN	Online (ms) WAN	Offline Comm. (KB)	Online Comm. (KB)
+	10^2	44.36	49.57	57.75	128.13	147.36	346.52	10.55	16.25
+	10^3	45.69	59.65	59.98	214.58	234.93	482.03	105.47	135.50
+	10^4	47.18	66.53	143.84	317.97	321.64	720.28	1054.69	1159.16
×	10^2	53.54	55.51	56.03	189.52	217.70	599.15	10.94	20.25
×	10^3	54.72	56.30	60.64	345.68	404.53	962.38	109.38	167.63
×	10^4	84.33	78.82	144.51	611.69	685.19	1309.22	1093.75	1621.72

Table 9: LAN performance breakdown of the measured BERT-base operations using Li *et al.*'s [31] and MP-SPDZ's [27] backends.

Operation	Runtime (s) over LAN Online	Runtime (s) over LAN Offline	Communication Cost (GB) Online	Communication Cost (GB) Offline
BERT-base - Li *et al.* [31]				
MatrixMul $(\mathbb{Q}^{128\times768\times768}) \times 3$	9.796	8.726	0.014	0.340
MatrixMul $(\mathbb{Q}^{128\times64\times128}) \times 12$	0.332	1.249	0.009	0.227
SoftMax $(\mathbb{Q}^{128\times128}) \times 12$	12.975	21.23	0.03	0.81
MatrixMul $(\mathbb{Q}^{128\times128\times64}) \times 12$	0.352	0.639	0.005	0.113
MatrixMul $\mathbb{Q}^{128\times768\times768}$	1.647	0.939	0.005	0.113
LayerNorm $\mathbb{Q}^{128\times768}$	1.059	3.638	0.016	0.342
MatrixMul $\mathbb{Q}^{128\times768\times3072}$	6.048	2.470	0.019	0.453
GeLU $\mathbb{Q}^{128\times3072}$	7.533	37.623	0.167	4.814
MatrixMul $\mathbb{Q}^{128\times3072\times768}$	5.798	1.121	0.005	0.113
LayerNorm $\mathbb{Q}^{128\times768}$	0.922	3.492	0.016	0.342
Total (12 layers)	**401.837**	**718.766**	**3.056**	**82.295**
BERT-base - SP-MPDZ [27]				
MatrixMul $(\mathbb{Q}^{128\times768\times768}) \times 3$	67.353	362.182	10.879	28.679
MatrixMul $(\mathbb{Q}^{128\times64\times128}) \times 12$	2.022	35.533	0.609	2.919
SoftMax $(\mathbb{Q}^{128\times128}) \times 12$	34.342	1340.050	1.965	67.066
MatrixMul $(\mathbb{Q}^{128\times128\times64}) \times 12$	1.959	22.703	0.606	2.196
MatrixMul $\mathbb{Q}^{128\times768\times768}$	12.944	54.619	3.626	9.560
LayerNorm $\mathbb{Q}^{128\times768}$	1.536	49.190	0.025	2.222
MatrixMul $\mathbb{Q}^{128\times768\times3072}$	64.608	247.452	14.505	38.237
GeLU $\mathbb{Q}^{128\times3072}$	51.522	3433.360	9.042	181.528
MatrixMul $\mathbb{Q}^{128\times3072\times768}$	66.246	186.729	14.498	36.070
LayerNorm $\mathbb{Q}^{128\times768}$	1.215	44.959	0.025	2.222
Total (12 layers)	**3644.965**	**69321.322**	**669.341**	**4448.382**

Table 10: LAN performance breakdown of the measured GPT-2 operations using Li *et al.*'s [31] and MP-SPDZ's [27] backends.

Operation	Runtime (s) over LAN		Communication Cost (GB)	
	Online	Offline	Online	Offline
GPT-2 - Li *et al.* [31]				
MatrixMul $(\mathbb{Q}^{64\times768\times768})\times3$	3.041	2.166	0.007	0.170
MatrixMul $(\mathbb{Q}^{64\times64\times64})\times12$	0.130	0.495	0.002	0.057
Masked SoftMax $(\mathbb{Q}^{64\times64})\times12$	0.943	0.634	0.004	0.112
MatrixMul $(\mathbb{Q}^{64\times64\times64})\times12$	0.155	0.723	0.002	0.057
MatrixMul $\mathbb{Q}^{64\times768\times768}$	0.720	0.473	0.002	0.057
LayerNorm $\mathbb{Q}^{64\times768}$	0.462	1.568	0.008	0.171
MatrixMul $\mathbb{Q}^{64\times768\times3072}$	3.302	1.235	0.009	0.227
GeLU $\mathbb{Q}^{64\times3072}$	4.008	19.323	0.084	2.407
MatrixMul $\mathbb{Q}^{64\times3072\times768}$	3.171	0.460	0.002	0.057
LayerNorm $\mathbb{Q}^{64\times768}$	1.081	4.153	0.008	0.171
Total (12 layers)	**192.834**	**367.150**	**1.506**	**40.470**
GPT-2 – MP-SPDZ [27]				
MatrixMul $(\mathbb{Q}^{64\times768\times768})\times3$	22.281	90.972	5.439	14.339
MatrixMul $(\mathbb{Q}^{64\times64\times64})\times12$	0.498	8.833	0.152	0.730
Masked SoftMax $(\mathbb{Q}^{64\times64})\times12$	12.423	335.078	0.498	16.794
MatrixMul $(\mathbb{Q}^{64\times64\times64})\times12$	0.828	12.743	0.152	0.730
MatrixMul $\mathbb{Q}^{64\times768\times768}$	6.687	27.638	1.813	4.780
LayerNorm $\mathbb{Q}^{64\times768}$	1.225	27.771	0.012	1.112
MatrixMul $\mathbb{Q}^{64\times768\times3072}$	28.979	119.542	7.252	19.119
GeLU $\mathbb{Q}^{64\times3072}$	33.537	1840.435	4.521	90.765
MatrixMul $\mathbb{Q}^{64\times3072\times768}$	34.652	105.248	7.249	18.036
LayerNorm $\mathbb{Q}^{64\times768}$	0.777	23.792	0.012	1.112
Total (12 layers)	**1702.660**	**31104.611**	**325.225**	**2010.216**

Table 11: MAN and WAN performance breakdown of BERT-base operations using Li *et al.* [31].

Operation	Runtime (s) over MAN		Runtime (s) over WAN		Communication Cost (GB)	
	Online	Offline	Online	Offline	Online	Offline
$\mathsf{op}_1\times3$	7.529	16.252	7.313	58.125	0.014	0.340
$\mathsf{op}_2\times12$	0.766	8.785	1.498	35.628	0.009	0.227
$\mathsf{op}_3\times12$	56.110	52.484	377.798	195.090	0.029	0.807
$\mathsf{op}_4\times12$	0.514	4.429	1.282	24.377	0.005	0.113
op_5	1.702	4.826	2.155	25.891	0.005	0.113
op_6	11.002	16.925	84.001	58.997	0.016	0.342
op_7	7.596	17.842	13.046	72.295	0.019	0.453
op_8	16.338	204.653	27.641	722.442	0.167	4.814
op_9	7.320	4.836	14.824	19.677	0.005	0.113
op_{10}	12.396	19.446	83.741	58.955	0.016	0.342
Total (× 12)	**1455.274**	**4205.732**	**7359.577**	**15257.721**	**3.406**	**91.975**

Note: $\mathsf{op}_1,\ldots,\mathsf{op}_{10}$ correspond one-to-one, in the same order, to the operations listed in Table 9.

Table 12: MAN and WAN performance breakdown of GPT-2 operations using Li *et al.* [31].

Operation	Runtime (s) over MAN		Runtime (s) over WAN		Communication Cost (GB)	
	Online	Offline	Online	Offline	Online	Offline
$\mathsf{op}_1\times3$	2.752	7.420	4.060	28.834	0.007	0.170
$\mathsf{op}_2\times12$	0.266	2.206	0.756	11.076	0.002	0.057
$\mathsf{op}_3\times12$	19.869	11.043	165.388	39.463	0.007	0.204
$\mathsf{op}_4\times12$	0.281	2.316	0.607	8.844	0.002	0.057
op_5	0.960	2.504	1.168	11.677	0.002	0.057
op_6	5.472	8.294	41.658	28.931	0.008	0.171
op_7	4.128	8.965	5.721	43.024	0.009	0.227
op_8	8.158	98.992	14.511	365.633	0.084	2.407
op_9	4.142	2.501	4.393	9.935	0.002	0.057
op_{10}	5.561	8.245	42.079	28.395	0.008	0.171
Total (× 12)	**619.067**	**1829.835**	**3364.080**	**6909.739**	**1.590**	**42.918**

Note: $\mathsf{op}_1,\ldots,\mathsf{op}_{10}$ correspond one-to-one, in the same order, to the operations listed in Table 9.

$\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{3pc}}$ to randomly sample $[r'_{\hat{x}}]^p_1,[r'_{\hat{x}}]^p_2 \xleftarrow{\$} \mathbb{Z}_p$, and send them to P_0;

- Upon receiving $m_{b_y^{(1)}}, m_{b_y^{(2)}}, m_{b_y^{(3)}}$ from P_0 to P_1,P_2, and receiving $(\mathsf{input},\mathsf{sid},f,(([r_y]^p_1,[r_y]^p_2),([r_{b_y^{(1)}}]_1,[r_{b_y^{(1)}}]_2),([r_{b_y^{(2)}}]_1,[r_{b_y^{(2)}}]_2),([r_{b_y^{(3)}}]_1,[r_{b_y^{(3)}}]_2)))$ from P_0 to $\mathcal{F}_{\mathsf{3pc}}$, $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{3pc}}$ to randomly sample $[r'_{\hat{y}}]^p_1,[r'_{\hat{y}}]^p_2 \xleftarrow{\$} \mathbb{Z}_p$, and send them to P_0;
- Upon receiving $m_{b_z^{(1)}}, m_{b_z^{(2)}}, m_{b_z^{(3)}}$ from P_0 to P_1,P_2, and receiving $(\mathsf{input},\mathsf{sid},f,(([r_z]^p_1,[r_z]^p_2),([r_{b_z^{(1)}}]_1,[r_{b_z^{(1)}}]_2),([r_{b_z^{(2)}}]_1,[r_{b_z^{(2)}}]_2),([r_{b_z^{(3)}}]_1,[r_{b_z^{(3)}}]_2)))$ from P_0 to $\mathcal{F}_{\mathsf{3pc}}$, $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{3pc}}$ to randomly sample $[r'_{\hat{z}}]^p_1,[r'_{\hat{z}}]^p_2 \xleftarrow{\$} \mathbb{Z}_p$, and send them to P_0;
- Upon receiving $(\mathsf{input},\mathsf{sid},\Diamond,(([r_{\hat{x}}]^p_1,[r_{\hat{x}}]^p_2),([r_{\hat{y}}]^p_1,[r_{\hat{y}}]^p_2))$ from P_0 to $\mathcal{F}_{\mathsf{3pc}}$, $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{3pc}}$ to randomly sample $[r'_{z'}]^p_1,[r'_{z'}]^p_2 \xleftarrow{\$} \mathbb{Z}_p$, and send them to P_0;
- Upon receiving $(\mathsf{equationCheck},\mathsf{sid},([r_{z'}]^p_1,[r_{z'}]^p_2),([r_{\hat{z}}]^p_1,[r_{\hat{z}}]^p_2))$ from P_0 to $\mathcal{F}_{\mathsf{eq}}$, $\mathcal{S}$ forwards P_0's input $(([r_x]^{2^\ell}_1,[r_x]^{2^\ell}_2),([r_y]^{2^\ell}_1,[r_y]^{2^\ell}_2))$ to the functionality $\mathcal{F}_{\mathsf{detect}}$ and receives the result b. Finally, $\mathcal{S}$ sends b to P_0 as $\mathcal{F}_{\mathsf{eq}}$.

We argue that the probability of ideal execution described above being distinguished from the real execution is negligible, by the following hybrids:

Hybrid $\mathcal{H}_0$: It is the real execution $\mathrm{EXEC}_{\Pi_{\mathsf{detect}},\mathcal{A},\mathcal{Z}}$.

Hybrid $\mathcal{H}_1$: Same as $\mathcal{H}_0$ except that in $\mathcal{H}_1$, except that $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{eq}}$:

- Upon receiving $(\mathsf{equationCheck},\mathsf{sid},([r_{z'}]^p_1,[r_{z'}]^p_2),([r_{\hat{z}}]^p_1,[r_{\hat{z}}]^p_2))$ from P_0 to $\mathcal{F}_{\mathsf{eq}}$, $\mathcal{S}$ forwards P_0's input $(([r_x]^{2^\ell}_1,[r_x]^{2^\ell}_2),([r_y]^{2^\ell}_1,[r_y]^{2^\ell}_2))$ to the functionality $\mathcal{F}_{\mathsf{detect}}$ and receives the result b. Finally, $\mathcal{S}$ sends b to P_0 as $\mathcal{F}_{\mathsf{eq}}$.

instead of P_0 getting $(z=z')$ from $\mathcal{F}_{\mathsf{eq}}$.

According to Theorem 3, given any $\langle x\rangle^{2^\ell}$, $\langle y\rangle^{2^\ell}$, the distribution of the variable $(z=z')$ is equal to the distribution of the response b from $\mathcal{F}_{\mathsf{detect}}$. Thus, $\mathcal{H}_0$ and $\mathcal{H}_1$ are perfect indistinguishable.

Hybrid $\mathcal{H}_2$: Same as $\mathcal{H}_1$ except that in $\mathcal{H}_2$, except that $\mathcal{S}$ plays the role of P_1 and P_2 to agree on the seeds $\eta_{0,1},\eta_{0,2}$ with P_0. Perfect indistinguishability holds since the simulator modifies nothing.

Hybrid $\mathcal{H}_3$: Same as $\mathcal{H}_2$ except that in $\mathcal{H}_3$, except that $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{3pc}}$ and sends randomly sampled values to P_0.

Easy to note that, in $\mathcal{H}_2$, the shares of real results from $\mathcal{F}_{\mathsf{3pc}}$ obtained by P_0 are uniformly random. Thus, the distributions of $\mathcal{H}_2$ and $\mathcal{H}_3$ are perfect indistinguishable.

The adversary's view of $\mathcal{H}_3$ is identical to the simulated view. Therefore, two worlds are perfectly indistinguishable.

Case 2: Corrupted P_1 (or P_2). In this case, we build the simulator $\mathcal{S}$ as follows, which emulates the interface of $\mathcal{F}_{\mathsf{3pc}}$, $\mathcal{F}_{\mathsf{eq}}$ as well as the honest parties P_0,P_2:

- In the initialization phase, $\mathcal{S}$ plays the role of P_0 and P_1 to agree on the seeds $\eta_{0,2},\eta_{1,2}$ with P_1 as real world;
- $\mathcal{S}$ records P_1's input $((m_x,[r_x]^{2^\ell}_1),(m_y,[r_y]^{2^\ell}_1))$;
- Upon receiving $(\mathsf{input},\mathsf{sid},\Diamond,((m_x,[r_x]^{2^\ell}_1),(m_y,[r_y]^{2^\ell}_1)))$ from P_1 to $\mathcal{F}_{\mathsf{3pc}}$, $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{3pc}}$ to randomly

Table 13: Performance breakdown of our method under different network settings.

Operation	Runtime (s) over LAN		Runtime (s) over MAN		Runtime (s) over WAN		Communication Cost (GB)	
	Online	Offline	Online	Offline	Online	Offline	Online	Offline
BERT-base								
MatrixMul $(\mathbb{Q}^{128\times768\times768})\times3$	0.540	0.431	1.144	1.720	2.404	4.096	0.022	0.046
MatrixMul $(\mathbb{Q}^{128\times64\times128})\times12$	1.763	0.030	5.123	0.302	11.549	0.768	0.107	0.009
SoftMax $(\mathbb{Q}^{128\times128})\times12$	0.321	-	8.991	-	33.641	-	0.243	-
MatrixMul $(\mathbb{Q}^{128\times128\times64})\times12$	0.030	0.033	0.157	0.274	0.461	0.700	0.005	0.009
MatrixMul $\mathbb{Q}^{128\times768\times768}$	0.864	0.148	2.488	0.582	5.548	1.384	0.058	0.017
LayerNorm $\mathbb{Q}^{128\times768}$	0.027	-	0.873	-	2.526	-	0.026	-
MatrixMul $\mathbb{Q}^{128\times768\times3072}$	0.776	0.506	1.423	2.330	2.712	5.226	0.029	0.060
GeLU $\mathbb{Q}^{128\times3072}$	0.866	-	13.577	-	34.301	-	0.420	-
MatrixMul $\mathbb{Q}^{128\times3072\times768}$	3.861	0.513	10.144	2.186	20.957	4.986	0.226	0.058
LayerNorm $\mathbb{Q}^{128\times768}$	0.027	-	0.873	-	2.526	-	0.026	-
Total (12 layers)	**108.930**	**19.923**	**537.523**	**88.722**	**1399.488**	**205.930**	**13.938**	**2.378**
GPT-2								
MatrixMul $(\mathbb{Q}^{64\times768\times768})\times3$	0.329	0.299	0.879	1.439	1.930	3.527	0.018	0.041
MatrixMul $(\mathbb{Q}^{64\times64\times64})\times12$	1.659	0.016	4.841	0.107	10.908	0.275	0.103	0.003
Masked SoftMax $(\mathbb{Q}^{64\times64})\times12$	0.106	-	3.287	-	20.251	-	0.061	-
MatrixMul $(\mathbb{Q}^{64\times64\times64})\times12$	0.027	0.015	0.071	0.109	0.343	0.273	0.002	0.003
MatrixMul $\mathbb{Q}^{64\times768\times768}$	0.503	0.104	1.369	0.484	3.092	1.180	0.031	0.014
LayerNorm $\mathbb{Q}^{64\times768}$	0.010	-	0.459	-	1.486	-	0.013	-
MatrixMul $\mathbb{Q}^{64\times768\times3072}$	0.461	0.354	1.019	1.881	2.116	5.142	0.024	0.054
GeLU $\mathbb{Q}^{64\times3072}$	0.418	-	6.807	-	17.136	-	0.210	-
MatrixMul $\mathbb{Q}^{64\times3072\times768}$	3.521	0.356	9.162	1.865	19.862	4.486	0.223	0.052
LayerNorm $\mathbb{Q}^{64\times768}$	0.010	-	0.459	-	1.486	-	0.013	-
Total (12 layers)	**84.519**	**13.733**	**340.236**	**70.619**	**943.322**	**178.593**	**8.380**	**2.010**

sample $m_x, [r_z]_1^{2^\ell} \xleftarrow{\$} \mathbb{Z}_{2^\ell}$, and send them to P_1;

- Upon receiving $(\mathsf{input}, \mathsf{sid}, f, ((m_x,[r_x]_1^p),(m_{b_x^{(1)}},[r_{b_x^{(1)}}]_1),(m_{b_x^{(2)}},[r_{b_x^{(2)}}]_1),(m_{b_x^{(3)}}[r_{b_x^{(3)}}]_1)))$ from P_1 to $\mathcal{F}_{\mathsf{3pc}}$, $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{3pc}}$ to randomly sample $[m'_{\hat{x}}, r'_{\hat{x}}]_1^p \xleftarrow{\$} \mathbb{Z}_p$, and send them to P_1;
- Upon receiving $(\mathsf{input}, \mathsf{sid}, f, ((m_y,[r_y]_1^p),(m_{b_y^{(1)}},[r_{b_y^{(1)}}]_1),(m_{b_y^{(2)}},[r_{b_y^{(2)}}]_1),(m_{b_y^{(3)}}[r_{b_y^{(3)}}]_1)))$ from P_1 to $\mathcal{F}_{\mathsf{3pc}}$, $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{3pc}}$ to randomly sample $[m'_{\hat{y}}, r'_{\hat{y}}]_1^p \xleftarrow{\$} \mathbb{Z}_p$, and send them to P_1;
- Upon receiving $(\mathsf{input}, \mathsf{sid}, f, ((m_z,[r_z]_1^p),(m_{b_z^{(1)}},[r_{b_z^{(1)}}]_1),(m_{b_z^{(2)}},[r_{b_z^{(2)}}]_1),(m_{b_z^{(3)}}[r_{b_z^{(3)}}]_1)))$ from P_1 to $\mathcal{F}_{\mathsf{3pc}}$, $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{3pc}}$ to randomly sample $[m'_{\hat{z}}, r'_{\hat{z}}]_1^p \xleftarrow{\$} \mathbb{Z}_p$, and send them to P_1;
- Upon receiving $(\mathsf{input}, \mathsf{sid}, \Diamond, ((m_{\hat{x}},[r_{\hat{x}}]_1^p),(m_{\hat{y}},[r_{\hat{y}}]_1^p))$ from P_1 to $\mathcal{F}_{\mathsf{3pc}}$, $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{3pc}}$ to randomly sample $m'_{z'}, [r'_{z'}]_1^p \xleftarrow{\$} \mathbb{Z}_p$, and send them to P_1;
- Upon receiving $(\mathsf{equationCheck}, \mathsf{sid}, (m_{z'},[r_{z'}]_1^p), (m_{\hat{z}},[r_{\hat{z}}]_1^p))$ from P_1 to $\mathcal{F}_{\mathsf{eq}}$, $\mathcal{S}$ forwards P_1's input $((m_x,[r_x]_1^{2^\ell}),(m_y,[r_y]_1^{2^\ell}))$ to the functionality $\mathcal{F}_{\mathsf{detect}}$ and receives the result b. Finally, $\mathcal{S}$ sends b to P_1 as $\mathcal{F}_{\mathsf{eq}}$.

We argue that the probability of ideal execution described above being distinguished from the real execution is negligible, by the following hybrids:

Hybrid $\mathcal{H}_0$: It is the real execution $\mathrm{EXEC}_{\Pi_{\mathsf{detect}},\mathcal{A},\mathcal{Z}}$.

Hybrid $\mathcal{H}_1$: Same as $\mathcal{H}_0$ except that in $\mathcal{H}_1$, except that $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{eq}}$:

- Upon receiving $(\mathsf{equationCheck}, \mathsf{sid}, (m_{z'},[r_{z'}]_1^p), (m_{\hat{z}},[r_{\hat{z}}]_1^p))$ from P_1 to $\mathcal{F}_{\mathsf{eq}}$, $\mathcal{S}$ forwards P_1's input $((m_x,[r_x]_1^{2^\ell}),(m_y,[r_y]_1^{2^\ell}))$ to the functionality $\mathcal{F}_{\mathsf{detect}}$ and receives the result b.
- $\mathcal{S}$ sends b to P_1 as $\mathcal{F}_{\mathsf{eq}}$.

instead of P_1 getting $(z = z')$ from $\mathcal{F}_{\mathsf{eq}}$.

According to Theorem 3, given any $\langle x\rangle^{2^\ell}$, $\langle y\rangle^{2^\ell}$, the distribution of the variable $(z = z')$ is equal to the distribution of the response b from $\mathcal{F}_{\mathsf{detect}}$. Thus, $\mathcal{H}_0$ and $\mathcal{H}_1$ are perfect indistinguishable.

Hybrid $\mathcal{H}_2$: Same as $\mathcal{H}_1$ except that in $\mathcal{H}_2$, except that $\mathcal{S}$ plays the role of P_0 and P_2 to agree on the seeds $\eta_{0,1}, \eta_{1,2}$ with P_1. Perfect indistinguishability holds since the simulator modifies nothing.

Hybrid $\mathcal{H}_3$: Same as $\mathcal{H}_2$ except that in $\mathcal{H}_3$, except that $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{3pc}}$ and sends randomly sampled values to P_0.

Easy to note that, in $\mathcal{H}_2$, the shares of real results from $\mathcal{F}_{\mathsf{3pc}}$ obtained by P_1 are uniformly random. Thus, the distributions of $\mathcal{H}_2$ and $\mathcal{H}_3$ are perfect indistinguishable.

The adversary's view of $\mathcal{H}_3$ is identical to the simulated view. Therefore, two worlds are perfectly indistinguishable.

In conclusion, the probability by which any PPT environment $\mathcal{Z}$ can distinguish $\mathrm{EXEC}_{\Pi_{\mathsf{detect}},\mathcal{A},\mathcal{Z}}$. and $\mathrm{EXEC}_{\mathcal{F}_{\mathsf{detect}},\mathcal{S},\mathcal{Z}}$ is negligible. This concludes the proof. □

N Proof of Theorem 6

Theorem 6 (restated). Let λ be a security parameter, p be a prime number and $p > 2^\lambda$. Let the PRG used in the sub-

protocol $\Pi_{[\cdot]}$ (c.f. Sec. 2) in the protocol $\Pi_{\mathsf{active\text{-}batch\text{-}detect}}$ be a secure peseudo-random geneartor with advantage $\mathsf{Adv}_{\mathsf{PRG}}$. The protocol $\Pi_{\mathsf{active\text{-}batch\text{-}detect}}$ in Fig. 4 UC-realizes $\mathcal{F}_{\mathsf{active\text{-}batch\text{-}detect}}$ in Fig. 1 in $\{\mathcal{F}_{\mathsf{3pc}}, \mathcal{F}_{\mathsf{active3pc}}, \mathcal{F}_{\mathsf{InnerVerify2}}\}$-hybrid model against a malicious adversary who can statically corrupted up to 1 party.

Proof. To prove Thm. 6, we construct a simulator $\mathcal{S}$ that internally runs a malicious adversary $\mathcal{A}$, interacts with $\mathcal{F}_{\mathsf{active\text{-}batch\text{-}detect}}$ and forwarding messages to/from the environment $\mathcal{Z}$ in the ideal world. Then we shows that any PPT environment $\mathcal{Z}$ cannot distinguish the simulation in the ideal world execution from the real-world execution more than a negligible probability. We consider the following cases.

Case 1: Corrupted P_0. In this case, we build the simulator $\mathcal{S}$ as follows, which emulates the interface of $\mathcal{F}_{\mathsf{3pc}}, \mathcal{F}_{\mathsf{active3pc}}, \mathcal{F}_{\mathsf{InnerVerify2}}$ as well as the honest parties P_1, P_2:

- In the initialization phase, $\mathcal{S}$ plays the role of P_1 and P_2 to agree on the seeds $\eta_{0,1}, \eta_{0,2}$ with P_0 as real world;
- $\mathcal{S}$ expand the PRG seeds $\eta_{0,1}, \eta_{0,2}$ to the secret shares $\{[r'_{b^{(1)}_{x_i}}]^p, [r'_{b^{(1)}_{y_i}}]^p, [r'_{b^{(1)}_{z_i}}]^p, [r'_{b^{(3)}_{x_i}}]^p, [r'_{b^{(3)}_{y_i}}]^p, [r'_{b^{(3)}_{z_i}}]^p\}_{i\in[N]}$ and $\{[r'_{c^{(1)}_{y_i}}]^p, [r'_{c^{(3)}_{y_i}}]^p\}_{i\in[N]}$, following the share generation protocol;
- For each $i \in [N]$:
  - $\mathcal{S}$ sets $[r'_{b^{(2)}_{x_i}}]^p_1 := 0, [r'_{b^{(2)}_{x_i}}]^p_2 := 0, [r'_{b^{(2)}_{y_i}}]^p_1 := 0, [r'_{b^{(2)}_{y_i}}]^p_2 := 0, [r'_{b^{(2)}_{z_i}}]^p_1 := 0$ and $[r'_{b^{(2)}_{z_i}}]^p_2 := 0$;
  - Upon receiving $(\mathsf{input}, \mathsf{sid}, \Diamond, (([r_{x_i}]^{2^\ell}_1, [r_{x_i}]^{2^\ell}_2), ([r_{y_i}]^{2^\ell}_1, [r_{y_i}]^{2^\ell}_2)))$ from P_0 to $\mathcal{F}_{\mathsf{3pc}}$, the simulator $\mathcal{S}$ records P_0's input, then randomly samples and sends $[r'_{z_i}]^{2^\ell}_1, [r'_{z_i}]^{2^\ell}_2 \xleftarrow{\$} \mathbb{Z}_{2^\ell}$ to P_0;
  - During the share generation, upon receiving $m_{b^{(1)}_{y_i}}, m_{b^{(2)}_{y_i}}, m_{b^{(3)}_{y_i}}$ from P_0 to P_1, P_2, the simulator $\mathcal{S}$ extracts $b'^{(1)}_{y_i} := m_{b^{(1)}_{y_i}} + [r_{b^{(1)}_{y_i}}]^p_1 + [r_{b^{(1)}_{y_i}}]^p_2$; similarly for $b'^{(2)}_{y_i}$ and $b'^{(3)}_{y_i}$;
  - Upon receiving $(\mathsf{input}, \mathsf{sid}, f, (([r_{\hat{x}_i}]^p_1, [r_{\hat{x}_i}]^p_2), ([r_{b^{(1)}_{x_i}}]^p_1, [r_{b^{(1)}_{x_i}}]^p_2), ([r_{b^{(2)}_{x_i}}]^p_1, [r_{b^{(2)}_{x_i}}]^p_2), ([r_{b^{(3)}_{x_i}}]^p_1, [r_{b^{(3)}_{x_i}}]^p_2)))$ from P_0 to $\mathcal{F}_{\mathsf{active3pc}}$, if $\mathcal{S}$ receives abort from $\mathcal{A}$, or the input share is not equal to the corresponding PRG-generated/extracted value (i.e. $[r_{\hat{x}_i}]^p_1 \neq [r_{x_i}]^p_1$, or $[r_{\hat{x}_i}]^p_2 \neq [r_{x_i}]^p_2$, or ...), then it sends $(\mathsf{leak}, \mathsf{sid}, f)$ to $\mathcal{F}_{\mathsf{active\text{-}batch\text{-}detect}}$; otherwise, $\mathcal{S}$ randomly sample and sends $[r'_{\hat{x}}]^p_1, [r'_{\hat{x}}]^p_2 \xleftarrow{\$} \mathbb{Z}_p$ to P_0.
  - Upon receiving $(\mathsf{input}, \mathsf{sid}, f, (([r_{y_i}]^p_1, [r_{y_i}]^p_2), ([r_{b^{(1)}_{y_i}}]^p_1, [r_{b^{(1)}_{y_i}}]^p_2), ([r_{b^{(2)}_{y_i}}]^p_1, [r_{b^{(2)}_{y_i}}]^p_2), ([r_{b^{(3)}_{y_i}}]^p_1, [r_{b^{(3)}_{y_i}}]^p_2)))$ from P_0 to $\mathcal{F}_{\mathsf{active3pc}}$, if $\mathcal{S}$ receives abort from $\mathcal{A}$, or the input share is not equal to the corresponding PRG-generated/extracted value (i.e. $[r_{\hat{y}_i}]^p_1 \neq [r_{y_i}]^p_1$, or $[r_{\hat{y}_i}]^p_2 \neq [r_{y_i}]^p_2$, or ...), then it sends $(\mathsf{leak}, \mathsf{sid}, f)$ to $\mathcal{F}_{\mathsf{active\text{-}batch\text{-}detect}}$; otherwise, $\mathcal{S}$ randomly samples and sends $[r'_{\hat{y}}]^p_1, [r'_{\hat{y}}]^p_2 \xleftarrow{\$} \mathbb{Z}_p$ to P_0.
  - Upon receiving $(\mathsf{input}, \mathsf{sid}, f, (([r_{\hat{z}_i}]^p_1, [r_{\hat{z}_i}]^p_2), ([r_{b^{(1)}_{z_i}}]^p_1, [r_{b^{(1)}_{z_i}}]^p_2), ([r_{b^{(2)}_{z_i}}]^p_1, [r_{b^{(2)}_{z_i}}]^p_2), ([r_{b^{(3)}_{z_i}}]^p_1, [r_{b^{(3)}_{z_i}}]^p_2)))$ from P_0 to $\mathcal{F}_{\mathsf{active3pc}}$, if $\mathcal{S}$ receives abort from $\mathcal{A}$, or the input share is not equal to the corresponding PRG-generated/extracted value (i.e. $[r_{\hat{z}_i}]^p_1 \mod 2^\ell \neq [r'_{z_i}]^p_1$, or $[r_{\hat{z}_i}]^p_2 \mod 2^\ell \neq [r'_{z_i}]^p_2$, or ...), then it sends $(\mathsf{leak}, \mathsf{sid}, f)$ to $\mathcal{F}_{\mathsf{active\text{-}batch\text{-}detect}}$; otherwise, $\mathcal{S}$ randomly samples and sends $[r'_{\hat{y}}]^p_1, [r'_{\hat{y}}]^p_2 \xleftarrow{\$} \mathbb{Z}_p$ to P_0.
- $\mathcal{S}$ computes $[r_r]_1, [r_r]_2$ using PRG with seeds $\eta_{0,1}, \eta_{0,2}$, and samples $m_r \xleftarrow{\$} \mathbb{Z}_p$, then open $r := m_r + [r_r]_1 + [r_r]_2 \pmod K$ as P_1 and P_2.
- Upon receiving $\langle \mathbf{A} \rangle_0 := (r^{N+i} \cdot [r_{b^{(1)}_{x_i}}]^p, r^{N+i} \cdot [r_{c^{(1)}_{x_i}}]^p, r^{2N+i} \cdot [r_{b^{(1)}_{y_i}}]^p, r^{2N+i} \cdot [r_{c^{(1)}_{y_i}}]^p, r^{3N+i} \cdot [r_{b^{(1)}_{z_i}}]^p, r^{3N+i} \cdot [r_{c^{(1)}_{z_i}}]^p)_{i\in[N]}$, $\langle \mathbf{B} \rangle_0 := ([r_{b^{(2)}_{x_i}}]^p, -[r_{c^{(2)}_{x_i}}]^p, [r_{b^{(2)}_{y_i}}]^p, -[r_{c^{(2)}_{y_i}}]^p, [r_{b^{(2)}_{z_i}}]^p, -[r_{c^{(2)}_{z_i}}]^p)\}_{i\in[N]}$, $\langle C \rangle_0 := [r_C]^p$ from P_0 to $\mathcal{F}_{\mathsf{InnerVerify2}}$, the simulator $\mathcal{S}$ compares them with generated shares before. More specifically, if $\langle \mathbf{A} \rangle_0$, $\langle \mathbf{B} \rangle_0$, and $\langle C \rangle_0$ does not consists with $\{[r'_{b^{(j)}_{x_i}}]^p, [r'_{b^{(j)}_{y_i}}]^p, [r'_{b^{(j)}_{z_i}}]^p, b'^{(j)}_{y_i}\}_{i\in[N], j\in\{1,2,3\}}$ according to the protocol, then $\mathcal{S}$ sends $(\mathsf{leak}, \mathsf{sid}, f)$ to $\mathcal{F}_{\mathsf{active\text{-}batch\text{-}detect}}$ and responds P_0 with $b := 1$;
- Finally, $\mathcal{S}$ forwards $\{([r_{x_i}]^{2^\ell}_1, [r_{x_i}]^{2^\ell}_2), ([r_{y_i}]^{2^\ell}_1, [r_{y_i}]^{2^\ell}_2)\}_{i\in[N]}$ to $\mathcal{F}_{\mathsf{active\text{-}batch\text{-}detect}}$ to obtain b, and sends b to P_0 as the functionality $\mathcal{F}_{\mathsf{InnerVerify2}}$.

We argue that the probability of ideal execution described above being distinguished from the real execution is negligible, by the following hybrids:

Hybrid $\mathcal{H}_0$: It is the real execution $\mathrm{EXEC}_{\Pi_{\mathsf{detect}}, \mathcal{A}, \mathcal{Z}}$.

Hybrid $\mathcal{H}_1$: Same as $\mathcal{H}_0$ except that in $\mathcal{H}_1$, except that in $\mathcal{H}_1$ the mask m_r is uniformly sampled while in $\mathcal{H}_0$ it is computed by PRG.

Claim 1. *If PRG be a secure pseudo-random generator with advantage* $\mathsf{Adv}_{\mathsf{PRG}}$*, then* $\mathcal{H}_1$ *and* $\mathcal{H}_0$ *are indistinguishable with advantage* $\varepsilon_1 := \mathsf{Adv}_{\mathsf{PRG}}$.

Proof. Easy to note that, we replace one PRG output by a uniformly random value. Thus, the overall advantage is $\varepsilon_1 := \mathsf{Adv}_{\mathsf{PRG}}$ by hybrid argument via reduction. □

Hybrid $\mathcal{H}_2$: Same as $\mathcal{H}_1$ except that in $\mathcal{H}_2$ we add a check

- If $\langle \mathbf{A} \rangle_0$, $\langle \mathbf{B} \rangle_0$, and $\langle C \rangle_0$ does not consist with $\{[r'_{b^{(j)}_{x_i}}]^p, [r'_{b^{(j)}_{y_i}}]^p, [r'_{b^{(j)}_{z_i}}]^p, b'^{(j)}_{y_i}\}_{i\in[N], j\in\{1,2,3\}}$, the parties output abort and halt.

Claim 2. *If* p *is a prime and* $p > 2^\lambda$*, then* $\mathcal{H}_1$ *and* $\mathcal{H}_0$ *are indistinguishable with advantage* $\varepsilon_2 \leq \frac{4N}{p-1}$.

Proof. The bad event occurs only when the adversary changes the input share but the check of $\mathcal{F}_{\mathsf{InnnerVerify2}}$ passes. For instance, if $\Diamond = \times$, it holds that $\Sigma_{i\in[N]} r^i x_i \cdot y_i + r^{N+i} b_{x_i}^{(1)} \cdot b_{x_i}^{(1)} - r^{N+i} c_{x_i}^{(1)} \cdot c_{x_i}^{(2)} + r^{2N+i} b_{y_i}^{(1)} \cdot b_{y_i}^{(2)} - r^{2N+i} c_{y_i}^{(1)} \cdot c_{y_i}^{(2)} + r^{3N+i} b_{z_i}^{(1)} \cdot b_{z_i}^{(2)} + r^{3N+i} c_{z_i}^{(1)} \cdot c_{z_i}^{(2)} = \Sigma_{i\in[N]} r^i \hat{z}_i + r^{N+i}(b_{x_i}^{(1)} + b_{x_i}^{(2)} + b_{x_i}^{(3)} - c_{x_i}^{(1)} - c_{x_i}^{(2)} - c_{x_i}^{(3)}) + r^{2N+i}(b_{y_i}^{(1)} + b_{y_i}^{(2)} + b_{y_i}^{(3)} - c_{y_i}^{(1)} - c_{y_i}^{(2)} - c_{y_i}^{(3)}) + r^{3N+i}(b_{z_i}^{(1)} + b_{z_i}^{(2)} + b_{z_i}^{(3)} - c_{z_i}^{(1)} - c_{z_i}^{(2)} - c_{z_i}^{(3)})$ after modifying the shares.

Since $r := m_r + [r_r]_1 + [r_r]_2$ is uniformly random, thus the probability of the bad event is bounded by $4N/p - 1$. Due to $p > 2^\ell$ and N is a positive integer number, $\mathcal{H}_2$ and $\mathcal{H}_1$ are indistinguishable with a negligible advantage $\varepsilon_2 \leq \frac{4N}{p-1}$. □

Hybrid $\mathcal{H}_3$: Same as $\mathcal{H}_2$ except that in $\mathcal{H}_3$ the simulator $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{InnerVerify2}}$:

- Upon receiving $\langle \mathbf{A} \rangle_0, \langle \mathbf{B} \rangle_0, \langle C \rangle$ from P_0 to $\mathcal{F}_{\mathsf{InnerVerify2}}$, if they consist with shares generated before (c.f. $\mathcal{H}_2$), $\mathcal{S}$ forwards P_0's input to $\mathcal{F}_{\mathsf{active\text{-}batch\text{-}detect}}$ to obtain b, and then sends b to P_0 as $\mathcal{F}_{\mathsf{InnerVerify2}}$.

instead of in $\mathcal{H}_2$ P_0 getting b from $\mathcal{F}_{\mathsf{InnerVerify2}}$.

According to Theorem 3, if the adversary does not modify the shares, the distribution of the variable b from $\mathcal{F}_{\mathsf{InnerVerify2}}$ is equal to the distribution of the response b from $\mathcal{F}_{\mathsf{active\text{-}batch\text{-}detect}}$. Thus, $\mathcal{H}_2$ and $\mathcal{H}_3$ are perfect indistinguishable.

Hybrid $\mathcal{H}_4$: Same as $\mathcal{H}_3$ except that in $\mathcal{H}_4$, except that $\mathcal{S}$ plays the role of P_1 and P_2 to agree on the PRG seeds $\eta_{0,1}, \eta_{0,2}$ with P_0. Then $\mathcal{S}$ locally generates the corresponding shares using the PRG seeds. Perfect indistinguishability holds since the simulator modifies nothing.

Hybrid $\mathcal{H}_5$: Same as $\mathcal{H}_4$ except that in $\mathcal{H}_5$ $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{3pc}}$ and sends randomly sampled values to P_0.

Easy to note that, in $\mathcal{H}_4$, the shares of real results from $\mathcal{F}_{\mathsf{3pc}}$ obtained by P_0 are uniformly random. Thus, the distributions of $\mathcal{H}_4$ and $\mathcal{H}_5$ are perfect indistinguishable.

Hybrid $\mathcal{H}_6$: Same as $\mathcal{H}_4$, except that in $\mathcal{H}_5$, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{active3pc}}$. More specifically, upon receiving an input message from P_0, $\mathcal{S}$ first checks whether the input values are equal to the corresponding PRG-generated shares. After that, if the check passes and the adversary does not send leak message, $\mathcal{S}$ sends randomly sampled values to P_0.

It is straightforward that if bad event does not happen, the distribution of the response uniformly random in both $\mathcal{H}_5$ and $\mathcal{H}_6$. The check and the leak interface are same as $\mathcal{F}_{\mathsf{active3pc}}$. Thus, the distributions of $\mathcal{H}_5$ and $\mathcal{H}_6$ are perfect indistinguishable.

The adversary's view of $\mathcal{H}_6$ is identical to the simulated view. Therefore, the advantage is $\varepsilon \leq \mathsf{Adv}_{\mathsf{PRG}} + \frac{4N}{p-1}$.

Case 2: Corrupted P_1. In this case, we build the simulator $\mathcal{S}$ as follows, which emulates the interface of $\mathcal{F}_{\mathsf{3pc}}, \mathcal{F}_{\mathsf{active3pc}}, \mathcal{F}_{\mathsf{InnerVerify2}}$ as well as the honest parties P_0, P_2:

- In the initialization phase, $\mathcal{S}$ plays the role of P_0 and P_2 to agree on the seeds $\eta_{0,1}, \eta_{1,2}$ with P_1 as real world;
- $\mathcal{S}$ expand the PRG seeds $\eta_{0,1}, \eta_{1,2}$ to the secret shares $\{[r'_{b_{x_i}^{(j)}}]_1^p, [r'_{b_{y_i}^{(j)}}]_1^p, [r'_{b_{z_i}^{(j)}}]_1^p, m'_{c_{x_i}^{(j)}}, [r'_{c_{x_i}^{(j)}}]_1^p, m_{c_{y_i}^{(j)}}, [r'_{c_{y_i}^{(j)}}]_1^p, m_{c_{z_i}^{(j)}}, [r'_{c_{z_i}^{(j)}}]_1^p\}_{i\in[N], j\in\{1,3\}}$;
- For $i \in [N]$:
  - Upon receiving $(\mathsf{input}, \mathsf{sid}, \Diamond, ((m_{x_i}, [r_{x_i}]_1^{2^\ell}), (m_{y_i}, [r_{y_i}]_1^{2^\ell})))$ from P_2 to $\mathcal{F}_{\mathsf{3pc}}$, $\mathcal{S}$ records the input, and then plays the role of $\mathcal{F}_{\mathsf{3pc}}$ to randomly sample $m'_{z_i}, [r'_{z_i}]_1^\ell \xleftarrow{\$} \mathbb{Z}_{2^\ell}$, and send them to P_1;
  - During the share generation, the simulator $\mathcal{S}$ plays the role of P_0 to randomly sample $m'_{b_{x_i}^{(1)}}, m'_{b_{y_i}^{(1)}}, m'_{b_{z_i}^{(1)}}, m'_{b_{x_i}^{(3)}}, m'_{b_{y_i}^{(3)}}, m'_{b_{z_i}^{(3)}} \xleftarrow{\$} \mathbb{Z}_p$ and send them to P_1.
  - During the share generation, upon receiving $[r_{c_{x_i}^{(j)}}]_2^p$ for $j \in \{1,3\}$, $\mathcal{S}$ records them and denotes the values by $[r'_{c_{x_i}^{(j)}}]_2^p$.
  - Upon receiving $(\mathsf{input}, \mathsf{sid}, f, ((m_{\hat{x}_i}, [r_{\hat{x}_i}]_1^p), (m_{b_{x_i}^{(1)}}, [r_{b_{x_i}^{(1)}}]_1^p), (m_{b_{x_i}^{(2)}}, [r_{b_{x_i}^{(2)}}]_1^p), (m_{b_{x_i}^{(3)}} [r_{b_{x_i}^{(3)}}]_1^p)))$ from P_1 to $\mathcal{F}_{\mathsf{active3pc}}$, if $\mathcal{S}$ receives abort from $\mathcal{A}$, or the input share is not equal to the corresponding PRG-generated/extracted value (i.e. $r_{\hat{x}_i} \neq r_{x_i}$, or $[r_{b_{x_i}^{(1)}}]_1 \neq [r'_{b_{x_i}^{(1)}}]_1$, or ...), then it sends $(\mathsf{leak}, \mathsf{sid}, f)$ to $\mathcal{F}_{\mathsf{active\text{-}batch\text{-}detect}}$; otherwise, $\mathcal{S}$ randomly samples and sends $[m'_{\hat{x}}]^p, [r'_{\hat{x}}]_1^p \xleftarrow{\$} \mathbb{Z}_p$ to P_1.
  - Upon receiving $(\mathsf{input}, \mathsf{sid}, f, ((m_{\hat{y}_i}, [r_{\hat{y}_i}]_1^p), (m_{b_{y_i}^{(1)}}, [r_{b_{y_i}^{(1)}}]_1^p), (m_{b_{y_i}^{(2)}}, [r_{b_{y_i}^{(2)}}]_1^p), (m_{b_{y_i}^{(3)}} [r_{b_{y_i}^{(3)}}]_1^p)))$ from P_1 to $\mathcal{F}_{\mathsf{active3pc}}$, if $\mathcal{S}$ receives abort from $\mathcal{A}$, or the input share is not equal to the corresponding PRG-generated/extracted value (i.e. $m_{\hat{y}_i} \neq m_{y_i}$, or $[r_{\hat{y}_i}]_1^p \neq [r_{y_i}]_1^p$, or ...), then it sends $(\mathsf{leak}, \mathsf{sid}, f)$ to $\mathcal{F}_{\mathsf{active\text{-}batch\text{-}detect}}$; otherwise, $\mathcal{S}$ randomly samples and sends $[m'_{\hat{y}}]^p, [r'_{\hat{y}}]_1^p \xleftarrow{\$} \mathbb{Z}_p$ to P_1.
  - Upon receiving $(\mathsf{input}, \mathsf{sid}, f, ((m_{\hat{z}_i}, [r_{\hat{z}_i}]_1^p), (m_{b_{z_i}^{(1)}}, [r_{b_{z_i}^{(1)}}]_1^p), (m_{b_{z_i}^{(2)}}, [r_{b_{z_i}^{(2)}}]_1^p), (m_{b_{z_i}^{(3)}} [r_{b_{z_i}^{(3)}}]_1^p)))$ from P_1 to $\mathcal{F}_{\mathsf{3pc}}$, if $\mathcal{S}$ receives abort from $\mathcal{A}$, or the input share is not equal to the corresponding PRG-generated/extracted value (i.e. $m_{\hat{z}_i} \neq m'_{z_i}$, or $[r_{\hat{z}_i}]_1^p \neq [r'_{z_i}]_1^p$, or ...), then it sends $(\mathsf{leak}, \mathsf{sid}, f)$ to $\mathcal{F}_{\mathsf{active\text{-}batch\text{-}detect}}$; otherwise, $\mathcal{S}$ randomly samples and sends $[m'_{\hat{z}}, r'_{\hat{z}}]_1^p \xleftarrow{\$} \mathbb{Z}_p$, and send them to P_1;
- $\mathcal{S}$ computes $[r_r]_1, m_r$ using PRG with seeds $\eta_{0,1}, \eta_{1,2}$, and samples $[r_r]_2 \xleftarrow{\$} \mathbb{Z}_p$, then open $r := m_r + [r_r]_1 + [r_r]_2 \pmod K$ as P_0 and P_2.

- Upon receiving $\langle \mathbf{A} \rangle_1 := (r^{N+i} \cdot (m_{b_{x_i}^{(1)}}, [r_{b_{x_i}^{(1)}}]_1^p), r^{N+i} \cdot (m_{c_{x_i}^{(1)}}, [r_{c_{x_i}^{(1)}}]_1^p), r^{2N+i} \cdot (m_{b_{y_i}^{(1)}}, [r_{b_{y_i}^{(1)}}]_1^p), r^{2N+i} \cdot (m_{c_{y_i}^{(1)}}, [r_{c_{y_i}^{(1)}}]_1)^K, r^{3N+i} \cdot (m_{b_{z_i}^{(1)}}, [r_{b_{z_i}^{(1)}}]_1)^K, r^{3N+i} \cdot (m_{c_{z_i}^{(1)}}, [r_{c_{z_i}^{(1)}}]_1^p))_{i\in[N]}$, $\langle \mathbf{B} \rangle_1 := (m_{b_{x_i}^{(2)}}, [r_{b_{x_i}^{(2)}}]_1^p, -m_{c_{x_i}^{(2)}}, -[r_{c_{x_i}^{(2)}}]_1^p, m_{b_{y_i}^{(2)}}, [r_{b_{y_i}^{(2)}}]_1^p, -m_{c_{y_i}^{(2)}}, -[r_{c_{y_i}^{(2)}}]_1^p, m_{b_{z_i}^{(2)}}, [r_{b_{z_i}^{(2)}}]_1^p, -m_{c_{z_i}^{(2)}}, -[r_{c_{z_i}^{(2)}}]_1^p))_{i\in[N]}$, $\langle C \rangle_1 := (m_C, [r_C]_1)$ from P_1 to $\mathcal{F}_{\mathsf{InnerVerify2}}$, the simulator $\mathcal{S}$ compares them with generated shares before. More specifically, if $\langle \mathbf{A} \rangle_0$, $\langle \mathbf{B} \rangle_0$, and $\langle C \rangle_0$ does not consists with $\{m'_{c_{x_i}^{(j)}}, [r'_{c_{x_i}^{(j)}}]^p, m'_{c_{y_i}^{(j)}}, [r'_{c_{y_i}^{(j)}}]^p, m'_{c_{z_i}^{(j)}}, [r'_{c_{z_i}^{(j)}}]^p\}_{i\in[N],j\in\{1,3\}}$ and $\{m'_{b_{x_i}^{(1)}}, m'_{b_{y_i}^{(1)}}, m'_{b_{z_i}^{(1)}}, m'_{b_{x_i}^{(3)}}, m'_{b_{y_i}^{(3)}}, m'_{b_{z_i}^{(3)}}\}$ according to the protocol, then $\mathcal{S}$ sends $(\mathsf{leak}, \mathsf{sid}, f)$ to $\mathcal{F}_{\mathsf{active\text{-}batch\text{-}detect}}$ and responds P_0 with $b := 1$;
- Finally, $\mathcal{S}$ forwards $\{(m_{x_i}]^{2^\ell}, [r_{x_i}]_1^{2^\ell}), (m_{y_i}^{2^\ell}, [r_{y_i}]_1^{2^\ell})\}_{i\in[N]}$ to $\mathcal{F}_{\mathsf{active\text{-}batch\text{-}detect}}$ to obtain b, and sends b to P_0 as the functionality $\mathcal{F}_{\mathsf{InnerVerify2}}$.

We argue that the probability of ideal execution described above being distinguished from the real execution is negligible, by the following hybrids:

Hybrid $\mathcal{H}_0$: It is the real execution $\mathrm{EXEC}_{\Pi_{\mathsf{detect}},\mathcal{A},\mathcal{Z}}$.

Hybrid $\mathcal{H}_1$: Same as $\mathcal{H}_0$ except that in $\mathcal{H}_1$, except that in $\mathcal{H}_1$ the mask $[r_r]_2$ is uniformly sampled while in $\mathcal{H}_0$ it is computed by PRG.

Claim 3. *If PRG be a secure pseudo-random generator with advantage* $\mathsf{Adv}_{\mathsf{PRG}}$*, then* $\mathcal{H}_1$ *and* $\mathcal{H}_0$ *are indistinguishable with advantage* $\varepsilon_1 := \mathsf{Adv}_{\mathsf{PRG}}$.

Proof. Easy to note that, we replace one PRG output by a uniformly random value. Thus, the overall advantage is $\varepsilon_1 := \mathsf{Adv}_{\mathsf{PRG}}$ by hybrid argument via reduction. □

Hybrid $\mathcal{H}_2$: Same as $\mathcal{H}_1$ except that in $\mathcal{H}_2$ we add a check

- If $\langle \mathbf{A} \rangle_0$, $\langle \mathbf{B} \rangle_0$, and $\langle C \rangle_0$ does not consist with $\{[r'_{c_{x_i}^{(j)}}]^p, [r'_{c_{y_i}^{(j)}}]^p, [r'_{c_{z_i}^{(j)}}]^p\}_{i\in[N],j\in\{1,3\}}$ and $\{m'_{b_{x_i}^{(1)}}, m'_{b_{y_i}^{(1)}}, m'_{b_{z_i}^{(1)}}, m'_{b_{x_i}^{(3)}}, m'_{b_{y_i}^{(3)}}, m'_{b_{z_i}^{(3)}}\}$ according to the protocol, the parties output abort and halt.

Claim 4. *If* p *is a prime and* $p > 2^\lambda$*, then* $\mathcal{H}_1$ *and* $\mathcal{H}_0$ *are indistinguishable with advantage* $\varepsilon_2 \leq \frac{4N}{p-1}$.

Proof. The bad event occurs only when the adversary changes the input share but the check of $\mathcal{F}_{\mathsf{InnnerVerify2}}$ passes. For instance, if $\Diamond = \times$, it holds that $\Sigma_{i\in[N]} r^i x_i \cdot y_i + r^{N+i} b_{x_i}^{(1)} \cdot b_{x_i}^{(1)} - r^{N+i} c_{x_i}^{(1)} \cdot c_{x_i}^{(2)} + r^{2N+i} b_{y_i}^{(1)} \cdot b_{y_i}^{(2)} - r^{2N+i} c_{y_i}^{(1)} \cdot c_{y_i}^{(2)} + r^{3N+i} b_{z_i}^{(1)} \cdot b_{z_i}^{(2)} + r^{3N+i} c_{z_i}^{(1)} \cdot c_{z_i}^{(2)} = \Sigma_{i\in[N]} r^i \hat{z}_i + r^{N+i}(b_{x_i}^{(1)} + b_{x_i}^{(2)} + b_{x_i}^{(3)} - c_{x_i}^{(1)} - c_{x_i}^{(2)} - c_{x_i}^{(3)}) + r^{2N+i}(b_{y_i}^{(1)} + b_{y_i}^{(2)} + b_{y_i}^{(3)} - c_{y_i}^{(1)} - c_{y_i}^{(2)} - c_{y_i}^{(3)}) + r^{3N+i}(b_{z_i}^{(1)} + b_{z_i}^{(2)} + b_{z_i}^{(3)} - c_{z_i}^{(1)} - c_{z_i}^{(2)} - c_{z_i}^{(3)})$ after modifying the shares own by P_1.

Since P_1 learn no information of uniformly random $[r_r]_2$, the value $r := m_r + [r_r]_1 + [r_r]_2$ is also uniformly random. Therefore, the probability of the bad event is bounded by $4N/p - 1$. Due to $p > 2^\ell$ and N is a positive integer number, $\mathcal{H}_2$ and $\mathcal{H}_1$ are indistinguishable with a negligible advantage $\varepsilon_2 \leq \frac{4N}{p-1}$. □

Hybrid $\mathcal{H}_3$: Same as $\mathcal{H}_2$ except that in $\mathcal{H}_3$ the simulator $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{InnerVerify2}}$:

- Upon receiving $\langle \mathbf{A} \rangle_0$, $\langle \mathbf{B} \rangle_0$, $\langle C \rangle$ from P_0 to $\mathcal{F}_{\mathsf{InnerVerify2}}$, if they consist with shares generated before (c.f. $\mathcal{H}_2$), $\mathcal{S}$ forwards P_0's input to $\mathcal{F}_{\mathsf{active\text{-}batch\text{-}detect}}$ to obtain b, and then sends b to P_0 as $\mathcal{F}_{\mathsf{InnerVerify2}}$.

instead of in $\mathcal{H}_2$ P_0 getting b from $\mathcal{F}_{\mathsf{InnerVerify2}}$.

According to Theorem 3, if the adversary does not modify the shares, the distribution of the variable b from $\mathcal{F}_{\mathsf{InnerVerify2}}$ is equal to the distribution of the response b from $\mathcal{F}_{\mathsf{active\text{-}batch\text{-}detect}}$. Thus, $\mathcal{H}_2$ and $\mathcal{H}_3$ are perfect indistinguishable.

Hybrid $\mathcal{H}_4$: Same as $\mathcal{H}_3$ except that in $\mathcal{H}_4$, except that $\mathcal{S}$ plays the role of P_0 and P_2 to agree on the PRG seeds $\eta_{0,1}, \eta_{1,2}$ with P_1. Then $\mathcal{S}$ locally generates the corresponding shares using the PRG seeds. Perfect indistinguishability holds since the simulator modifies nothing.

Hybrid $\mathcal{H}_5$: Same as $\mathcal{H}_4$ except that in $\mathcal{H}_5$ $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{3pc}}$ and sends randomly sampled values to P_0.

Easy to note that, in $\mathcal{H}_4$, the shares of real results from $\mathcal{F}_{\mathsf{3pc}}$ obtained by P_1 are uniformly random. Thus, the distributions of $\mathcal{H}_4$ and $\mathcal{H}_5$ are perfect indistinguishable.

Hybrid $\mathcal{H}_6$: Same as $\mathcal{H}_4$, except that in $\mathcal{H}_5$, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{active3pc}}$. More specifically, upon receiving an input message from P_0, $\mathcal{S}$ first checks whether the input values are equal to the corresponding PRG-generated shares. After that, if the check passes and the adversary does not send leak message, $\mathcal{S}$ sends randomly sampled values to P_0.

It is straightforward that if bad event does not happen, the distribution of the response uniformly random in both $\mathcal{H}_5$ and $\mathcal{H}_6$. The check and the leak interface are same as $\mathcal{F}_{\mathsf{active3pc}}$. Thus, the distributions of $\mathcal{H}_5$ and $\mathcal{H}_6$ are perfect indistinguishable.

The adversary's view of $\mathcal{H}_6$ is identical to the simulated view. Therefore, the advantage is no more than $\mathsf{Adv}_{\mathsf{PRG}} + \frac{4N}{p-1}$.

Case 3: Corrupted P_2. In this case, we build the simulator $\mathcal{S}$ as follows, which emulates the interface of $\mathcal{F}_{\mathsf{3pc}}, \mathcal{F}_{\mathsf{active3pc}}, \mathcal{F}_{\mathsf{InnerVerify2}}$ as well as the honest parties P_0, P_1:

- In the initialization phase, $\mathcal{S}$ plays the role of P_0 and P_1 to agree on the seeds $\eta_{0,2}, \eta_{1,2}$ with P_2 as real world;
- $\mathcal{S}$ expand the PRG seeds $\eta_{0,2}, \eta_{1,2}$ to the secret shares $\{[r'_{b_{x_i}^{(j)}}]_2^p, [r'_{b_{y_i}^{(j)}}]_2^p, [r'_{b_{z_i}^{(j)}}]_2^p, m'_{c_{x_i}^{(j)}}, m_{c_{y_i}^{(j)}}, m_{c_{z_i}^{(j)}}\}_{i\in[N],j\in\{1,3\}}$;
- For $i \in [N]$:

– Upon receiving $(\mathsf{input}, \mathsf{sid}, \diamondsuit, ((m_{x_i}, [r_{x_i}]_2^{2^\ell}), (m_{y_i}, [r_{y_i}]_2^{2^\ell})))$ from P_2 to $\mathcal{F}_{\mathsf{3pc}}$, $\mathcal{S}$ records the input, and then plays the role of $\mathcal{F}_{\mathsf{3pc}}$ to randomly sample $m'_{z_i}, [r'_{z_i}]_2^{2^\ell} \xleftarrow{\$} \mathbb{Z}_{2^\ell}$, and send them to P_1;

– During the share generation, the simulator $\mathcal{S}$ plays the role of P_1 and P_2 to randomly sample $m'_{b_{x_i}^{(1)}}, m'_{b_{y_i}^{(1)}}, m'_{b_{z_i}^{(1)}}, m'_{b_{x_i}^{(3)}}, m'_{b_{y_i}^{(3)}}, m'_{b_{z_i}^{(3)}}, [r'_{c_{x_i}^{(1)}}]_2^p, [r'_{c_{y_i}^{(1)}}]_2^p, [r'_{c_{z_i}^{(1)}}]_2^p, [r'_{c_{x_i}^{(3)}}]_2^p, [r'_{c_{y_i}^{(3)}}]_2^p, [r'_{c_{z_i}^{(3)}}]_2^p \xleftarrow{\$} \mathbb{Z}_p$ and send them to P_2;

– Upon receiving $(\mathsf{input}, \mathsf{sid}, f, ((m_{\hat{x}_i}, [r_{\hat{x}_i}]_2^p), (m_{b_{x_i}^{(1)}}, [r_{b_{x_i}^{(1)}}]_2^p), (m_{b_{x_i}^{(2)}}, [r_{b_{x_i}^{(2)}}]_2^p), (m_{b_{x_i}^{(3)}} [r_{b_{x_i}^{(3)}}]_2^p)))$ from P_2 to $\mathcal{F}_{\mathsf{active3pc}}$, if $\mathcal{S}$ receives abort from $\mathcal{A}$, or the input share is not equal to the corresponding PRG-generated/extracted value (i.e. $r_{\hat{x}_i} \neq r_{x_i}$, or $[r_{b_{x_i}^{(1)}}]_2 \neq [r'_{b_{x_i}^{(1)}}]_2$, or ...), then it sends $(\mathsf{leak}, \mathsf{sid}, f)$ to $\mathcal{F}_{\mathsf{active\text{-}batch\text{-}detect}}$; otherwise, $\mathcal{S}$ randomly samples and sends $[m'_{\hat{x}}]^p, [r'_{\hat{x}}]_2^p \xleftarrow{\$} \mathbb{Z}_p$ to P_1.

– Upon receiving $(\mathsf{input}, \mathsf{sid}, f, ((m_{\hat{y}_i}, [r_{\hat{y}_i}]_2^p), (m_{b_{y_i}^{(1)}}, [r_{b_{y_i}^{(1)}}]_2^p), (m_{b_{y_i}^{(2)}}, [r_{b_{y_i}^{(2)}}]_2^p), (m_{b_{y_i}^{(3)}} [r_{b_{y_i}^{(3)}}]_2^p)))$ from P_2 to $\mathcal{F}_{\mathsf{active3pc}}$, if $\mathcal{S}$ receives abort from $\mathcal{A}$, or the input share is not equal to the corresponding PRG-generated/extracted value (i.e. $m_{\hat{y}_i} \neq m_{y_i}$, or $[r_{\hat{y}_i}]_2^p \neq [r_{y_i}]_2^p$, or ...), then it sends $(\mathsf{leak}, \mathsf{sid}, f)$ to $\mathcal{F}_{\mathsf{active\text{-}batch\text{-}detect}}$; otherwise, $\mathcal{S}$ randomly samples and sends $[m'_{\hat{y}}]^p, [r'_{\hat{y}}]_2^p \xleftarrow{\$} \mathbb{Z}_p$ to P_2.

– Upon receiving $(\mathsf{input}, \mathsf{sid}, f, ((m_{\hat{z}_i}, [r_{\hat{z}_i}]_2^p), (m_{b_{z_i}^{(1)}}, [r_{b_{z_i}^{(1)}}]_2^p), (m_{b_{z_i}^{(2)}}, [r_{b_{z_i}^{(2)}}]_2^p), (m_{b_{z_i}^{(3)}} [r_{b_{z_i}^{(3)}}]_2^p)))$ from P_2 to $\mathcal{F}_{\mathsf{3pc}}$, if $\mathcal{S}$ receives abort from $\mathcal{A}$, or the input share is not equal to the corresponding PRG-generated/extracted value (i.e. $m_{\hat{z}_i} \neq m'_{z_i}$, or $[r_{\hat{z}_i}]_2^p \neq [r'_{z_i}]_2^p$, or ...), then it sends $(\mathsf{leak}, \mathsf{sid}, f)$ to $\mathcal{F}_{\mathsf{active\text{-}batch\text{-}detect}}$; otherwise, $\mathcal{S}$ randomly samples and sends $[m'_{\hat{z}}, r'_{\hat{z}}]_2^p \xleftarrow{\$} \mathbb{Z}_p$, and send them to P_1;

- $\mathcal{S}$ computes $[r_r]_2, m_r$ using PRG with seeds $\eta_{0,2}, \eta_{1,2}$, and samples $[r_r]_1 \xleftarrow{\$} \mathbb{Z}_p$, then open $r := m_r + [r_r]_1 + [r_r]_2 \pmod K$ as P_0 and P_2.
- Upon receiving
  $\langle \mathbf{A} \rangle_2 := (r^{N+i} \cdot (m_{b_{x_i}^{(1)}}, [r_{b_{x_i}^{(1)}}]_2^p), r^{N+i} \cdot (m_{c_{x_i}^{(1)}}, [r_{c_{x_i}^{(1)}}]_2^p), r^{2N+i} \cdot (m_{b_{y_i}^{(1)}}, [r_{b_{y_i}^{(1)}}]_2^p), r^{2N+i} \cdot (m_{c_{y_i}^{(1)}}, [r_{c_{y_i}^{(1)}}]_2)^K, r^{3N+i} \cdot (m_{b_{z_i}^{(1)}}, [r_{b_{z_i}^{(1)}}]_2)^K, r^{3N+i} \cdot (m_{c_{z_i}^{(1)}}, [r_{c_{z_i}^{(1)}}]_2^p))_{i \in [N]}$,
  $\langle \mathbf{B} \rangle_2 := (m_{b_{x_i}^{(2)}}, [r_{b_{x_i}^{(2)}}]_2^p, -m_{c_{x_i}^{(2)}}, -[r_{c_{x_i}^{(2)}}]_2^p, m_{b_{y_i}^{(2)}}, [r_{b_{y_i}^{(2)}}]_2^p, -m_{c_{y_i}^{(2)}}, -[r_{c_{y_i}^{(2)}}]_2^p, m_{b_{z_i}^{(2)}}, [r_{b_{z_i}^{(2)}}]_2^p, -m_{c_{z_i}^{(2)}}, -[r_{c_{z_i}^{(2)}}]_2^p))_{i \in [N]}$,
  $\langle C \rangle_1 := (m_C, [r_C]_2)$ from P_1 to $\mathcal{F}_{\mathsf{InnerVerify2}}$, the simulator $\mathcal{S}$ compares them with generated shares before. More specifically, if $\langle \mathbf{A} \rangle_0$, $\langle \mathbf{B} \rangle_0$, and $\langle C \rangle_0$ does not consists with the sent shares and P_0's input according to the protocol, then $\mathcal{S}$ sends $(\mathsf{leak}, \mathsf{sid}, f)$ to $\mathcal{F}_{\mathsf{active\text{-}batch\text{-}detect}}$ and responds P_0 with $b := 1$;
- Finally, $\mathcal{S}$ forwards $\{(m_{x_i}]^{2^\ell}, [r_{x_i}]_2^{2^\ell}), (m_{y_i}^{2^\ell}, [r_{y_i}]_2^{2^\ell})\}_{i \in [N]}$ to $\mathcal{F}_{\mathsf{active\text{-}batch\text{-}detect}}$ to obtain b, and sends b to P_0 as the functionality $\mathcal{F}_{\mathsf{InnerVerify2}}$.

By the similar method to that in the case 2, we argue that the advantage of case 3 is no more than $\mathsf{Adv}_{\mathsf{PRG}} + \frac{4N}{p-1}$.

In conclusion, the probability by which any PPT environment $\mathcal{Z}$ can distinguish $\text{EXEC}_{\Pi_{\mathsf{detect}}, \mathcal{A}, \mathcal{Z}}$. and $\text{EXEC}_{\mathcal{F}_{\mathsf{active\text{-}batch\text{-}detect}}, \mathcal{S}, \mathcal{Z}}$ is $\mathsf{Adv}_{\mathsf{PRG}} + \frac{4N}{p-1}$. Since PRG is a secure pseudo-random generator, $\mathsf{Adv}_{\mathsf{PRG}}$ is negligible. For $p > 2^\ell$ and constant number N, the value $\frac{4N}{p-1}$ is also negligible. Therefore, any PPT environment $\mathcal{Z}$ cannot distinguish the simulation in the ideal world execution from the real-world execution more than a negligible probability. This concludes the proof. □