Dishonest Majority Multi-Party Arithmetic Garbling with Constant Rate

Tianyao Gu, Hanjun Li, and Elaine Shi

Carnegie Mellon University

May 29, 2026

Abstract

Minimizing round complexity is a central goal in secure Multi-Party Computation (MPC), particularly for deployment on high-latency networks. While constant-round protocols with concrete efficiency have been constructed, they are typically designed for Boolean circuits and each gate incurs a bandwidth cost linear in the security parameter. Moreover, for arithmetic-heavy applications such as privacy-preserving machine learning and statistical analysis, compiling arithmetic operations into Boolean gates incurs another substantial overhead in circuit size and communication. Conversely, existing arithmetic MPC protocols, such as SPDZ, require interaction rounds proportional to the circuit depth, imposing significant latency.

In this work, we bridge this gap by presenting the first concretely-efficient maliciously-secure MPC protocol that achieves both constant-round and constant-rate communication, where the rate is defined as the bandwidth cost per party divided by the number of gates and the size of the values each gate operates on. Our protocol computes over bounded integers and is secure against a static, malicious adversary corrupting up to $n - 1$ parties. The protocol is built upon the arithmetic garbling framework of Ball et al. (Eurocrypt 2023) and follows the BMR template, assuming the Decisional Composite Residuosity for the garbling phase and Learning Parity with Noise for preprocessing.

We evaluate our protocol on matrix-vector multiplication, a fundamental operation for data analysis. For standard computation parameters, we reduce communication bandwidth by 101× to 247× and improve end-to-end runtime by 4.4× to 10.7× compared to state-of-the-art constant-round Boolean MPC baselines, even when accounting for the overhead of a full bit-decomposition on the output vector.

Contents

1 Introduction **3**
- 1.1 Our Contributions . . . 4

2 Technical Roadmap **6**
- 2.1 Background: Garbling Scheme and BMR Blueprint . . . 6
- 2.2 Arithmetic Garbling from BLLL . . . 7
- 2.3 Our Multi-party Generalization . . . 10
- 2.4 Towards Malicious Security . . . 11

3 Preliminaries **14**
- 3.1 Secret Sharing and SPDZ . . . 14
- 3.2 Linearly Homomorphic Encryption (LHE) from DCR . . . 16

4 Definitions **17**
- 4.1 Security Definition . . . 17
- 4.2 Computational Model . . . 18

5 Protocol Specification **19**
- 5.1 Ideal Functionality of Offline Phase . . . 20
- 5.2 Main Protocol . . . 21

6 Evaluation of Concrete Costs **26**

7 Open Problems **29**

8 Acknowledgements **29**

A Full Offline Protocols **39**
- A.1 Preprocessing . . . 39
- A.2 Offline Protocol . . . 44

B Security Proofs **45**
- B.1 Security Proof for the Preprocessing Protocol . . . 45
- B.2 Security Proof for the Offline Protocol . . . 56
- B.3 Security Proof for the Main Protocol . . . 60
- B.4 DCR Assumption and Proof of Lemma 3.2 . . . 72

C Removing KDM Assumption for Truncation Gates **75**

D Detailed Cost Analysis **77**

1 Introduction

Secure Multi-Party Computation (MPC) enables a set of mutually distrusting parties to jointly compute a function over their private inputs while guaranteeing correctness and keeping their inputs private. Since the foundational works of Yao [Yao86] and Goldreich, Micali, and Wigderson [GMW87], MPC has transitioned from a theoretical curiosity to a viable technology for privacy-preserving applications, ranging from secure auctions to collaborative data analysis.

The communication cost of MPC is typically evaluated using two key metrics: *round complexity* and *bandwidth cost*. Specifically, bandwidth is often measured by *rate*, defined as the total bits communicated by each party divided by the product of the number of gates and the gate width — for a gate operating on ℓ-bit integers, the gate width is defined as ℓ.

Constant-round and constant-rate MPC. Early MPC protocols such as BGW [BOGW88], GMW [GMW87], and efficient variants like SPDZ [DPSZ12], require interaction rounds proportional to the circuit depth. On high-latency networks, this becomes a major bottleneck: beyond raw transmission delay, coordinating many sequential rounds among distributed parties with heterogeneous performance is also challenging in practice. To address this, a substantial body of work has aimed to reduce the round complexity to $O(1)$, beginning with the seminal construction of Beaver, Micali, and Rogaway (BMR) [BMR90], and followed by numerous improvements [LPSY15, BELO16, BLO17, HSS17, WRK17a, HOSS18a, HOSS18b, Ben18, YWZ20, BCO+21, BGH+23, GLM+24, HKN+25]. Constant-round MPC can also be achieved under stronger assumptions such as Fully Homomorphic Encryption (FHE) [Gen09, BGV14] or indistinguishability obfuscation (iO) [JLS21, GGH+13]. While FHE- or iO-based approaches can additionally obtain sub-constant rate $o(1)$, they face important limitations. FHE-based schemes incur substantial bootstrapping overhead for deeper circuits and are not known to be concretely efficient in the malicious setting, where constant-round zero-knowledge arguments of knowledge [FS89, BJKL21] are required to enforce honest behavior. Similarly, iO-based approaches remain primarily of theoretical interest in the near term. Without relying on assumptions that imply FHE or iO, the current state of the art [GLM+24] achieves a per-gate bandwidth of $O(n \cdot \lambda)$ bits against a malicious adversary corrupting up to $n-1$ parties, where n is the number of parties and λ is the security parameter. Even in the semi-honest model, all known constant-round MPC schemes suffer from at least $\Omega(\lambda)$ rate. This status quo raises two natural questions:

- *Can we construct an $O(1)$-round and $O(1)$-rate MPC scheme without relying on FHE or iO?* This question remains open even in the semi-honest model, and even without regard to concrete efficiency.
- *Can we construct an $O(1)$-round and $O(1)$-rate maliciously secure MPC scheme that is concretely efficient?*

In comparison, in the two-party (2PC) setting, several works have proposed $O(1)$-round and $O(1)$-rate garbling schemes for arithmetic circuits [BLLL23, MORS24, MORS25, ILL25, CHHK25, GGS26] and Boolean circuits [ILL25, LWYY25b, LWYY25a, ZLY+26, LLL26], based on Decisional Composite Residuosity (DCR) [Pai99], Ring Learning With Errors (RLWE) [LPR10], or NTRU [HPS98] assumptions. However, extending these constant-rate results to the multi-party setting remains an open problem.

MPC optimized for arithmetic circuits. Arithmetic circuits are often more efficient for representing scientific and AI computations. For instance, integer or fixed-point multiplication can be

expressed as a single arithmetic gate. In contrast, representing the same operation as a Boolean circuit is much more costly: while it requires $O(\ell \log \ell)$ size in theory [SS71, HvdH21], practical instantiations typically use Karatsuba's algorithm with $O(\ell^{1.585})$ cost or naïve long multiplication with $O(\ell^2)$ cost for lower precision (e.g., 64-bit).

To optimize MPC for arithmetic computations, Applebaum, Ishai, and Kushilevitz [AIK11] proposed arithmetic garbling — a class of *concretely efficient* techniques for *constant-round* computation. Assuming oblivious transfer [Rab05], these schemes immediately yield constant-round 2PC. Most prior arithmetic garbling works focused on the 2-party setting [AIK11, BLLL23, MORS24, Hea24, MORS25, ILL25, CHHK25, GGS26]. For bounded integer computation, these constructions achieve $O(1)$ rate from the DCR assumption [Pai99] (or a circular variant) and offer promising efficiency compared to constant-rate Boolean garbling [LWYY25b, ZLY+26, LLL26], which not only incurs the representation overhead of converting arithmetic computations to Boolean, but also requires evaluating cryptographic functions within FHE with known techniques.

However, existing arithmetic garbling schemes have two major limitations. First, even in the 2-party setting, they lack full malicious security. The best known result [HY24] achieves only one-sided malicious security and suffers from 1-bit leakage due to a selective-failure attack. Second, for the multi-party setting, no constant-rate arithmetic garbling scheme exists. Current constant-round multi-party protocols [BCO+21, BGH+23, GLM+24, HKN+25] support arithmetic computation only by compiling to Boolean circuits. This leads to the following open questions:

- *Can we construct a concretely efficient arithmetic garbling scheme with $O(1)$ rate and full malicious security?* This question remains open even for 2PC.
- *Can we construct a concretely efficient multi-party arithmetic garbling scheme with $O(1)$ rate?* This is open even for the semi-honest setting.

1.1 Our Contributions

We answer the questions above affirmatively. As stated in Theorem 1.1, we propose the first concretely-efficient and maliciously-secure MPC protocol for arithmetic circuits that achieves both constant round complexity and constant rate. Assuming DCR, our protocol is secure against a static malicious adversary corrupting up to $t = n - 1$ parties. We compare our protocol with prior constant-round MPC protocols in Table 1.

Theorem 1.1 (MPC for bounded-integer arithmetic circuits with gate set $\{\times, +\}$)**.** *Assuming DCR, LPN, and a trusted setup for generating the DCR modulus*[1]*, there is a computationally secure constant-round MPC protocol for arithmetic circuits over $\mathcal{B}$-bounded integers against a static malicious adversary corrupting up to $n-1$ parties. Moreover, the protocol's total bandwidth is at most $O(n \cdot (\lambda + \ell) \cdot (|\mathcal{C}| + n \cdot \log |\mathcal{C}|))$ bits, where λ is the computational security parameter, $\ell = \log \mathcal{B}$ is the bit length of the integer, n is the number of parties, and $|\mathcal{C}|$ denotes the circuit size, excluding the addition gates. Notably, when the circuit size $|\mathcal{C}| \geq n \log |\mathcal{C}|$ and the bit length $\ell \geq \lambda$, the protocol has a constant rate.*

In comparison with [HY24] at $n = 2$ (the only case they support), our protocol is secure when both parties are malicious and does not suffer from their 1-bit leakage problem[2]. Moreover, we support free addition gates and achieve around $4\times$ improvement in the concrete cost of multiplication gates.

[1] The trusted setup can also be replaced with a distributed key generation protocol [BDF+23].

[2] While [HY24] showed that 1-bit leakage is inevitable in their computational model, we consider a stricter bounded-integer model which assumes no wire will overflow upon any legal input.

Table 1: Comparison of our protocol with prior constant-round MPC protocols. The table lists the amortized communication costs for realizing ℓ-bit bounded integer arithmetic, where n denotes the number of parties, λ the security parameter of the corresponding security assumption, and c is a constant parameter between 0 and 1. All the protocols listed in the table assume a static, malicious adversary. For integer multiplication in prior works, we perform a Boolean circuit reduction via Karatsuba's algorithm [KO62]. In addition, for all the listed protocols, we assume a sufficiently large circuit and omit the costs sublinear in the circuit size. Both our protocol and [GLM+24] make black-box use of the LPN-based building blocks in [RS22] for preprocessing secret shares, and the LPN assumption is not required for the rest parts of our protocol.

Protocol	**Multiplication cost in bits**	**Addition cost in bits**	**Corruption Threshold**	**Assumption**
[BCO+21]	$O(n^2 \cdot \lambda \cdot \ell^{1.585})$	$O(n^2 \cdot \lambda \cdot \ell)$	$t = n - 1$	LPN
[BGH+23]	$O(\frac{1}{c} \cdot n \cdot \lambda \cdot \ell^{1.585})$	$O(\frac{1}{c} \cdot n \cdot \lambda \cdot \ell)$	$t = (\frac{1}{2} - c) \cdot n$	LPN
[GLM+24]	$O(n \cdot \lambda \cdot \ell^{1.585})$	$O(n \cdot \lambda \cdot \ell)$	$t = n - 1$	DDH, LPN
[LS25]	$O(n \cdot \lambda \cdot \ell^{1.585})$	$O(n \cdot \lambda \cdot \ell)$	$t = (n - 1)/2$	OWF
[HKN+25]	$O(\frac{1}{c} \cdot n \cdot \lambda \cdot \ell^{1.585})$	$O(\frac{1}{c} \cdot n \cdot \lambda \cdot \ell)$	$t = (1 - c) \cdot n$	OT, RO
Ours	$O(n \cdot (\lambda + \ell))$	0	$t = n - 1$	DCR, LPN

Beyond standard integer addition and multiplication, we also introduce a truncation gate construction (i.e., floor division by 2), as stated in Theorem 1.2, adapted from the work of Ball et al. [BLLL23]. The enriched gate set enables a wider range of computations, such as fixed-point arithmetic, modular arithmetic, as well as mixed Boolean and arithmetic circuits.

Since truncation gates are more expensive than other gate types, the concrete performance of our approach is particularly compelling in applications where truncation operations are used sparingly. To evaluate practical efficiency, we benchmarked our protocol on matrix-vector multiplication, a fundamental primitive that arises ubiquitously in scientific computing and AI workloads. For a 4096×4096 matrix and 16-bit integer/fixed-point precision, we achieve a 101× to 247× reduction in bandwidth compared to the state-of-the-art constant-round Boolean MPC schemes [GLM+24, HKN+25], even when accounting for the overhead of a full bit-decomposition on the output vector.

We note that maintaining the free addition property in our protocol with truncation gates requires a key-dependent message (KDM) security assumption on the underlying encryption scheme, which is not known to be implied by DCR alone. We provide a blueprint on how to remove this assumption by allowing each addition gate to incur a bandwidth cost of $O(n \cdot (\lambda + \ell))$.

Theorem 1.2 (MPC for bounded-integer arithmetic circuits with truncation)**.** *Assuming DCR, LPN, and a trusted setup for generating the DCR modulus, there is a computationally secure constant-round MPC protocol for arithmetic circuits over $\mathcal{B}$-bounded integers against a static malicious adversary corrupting up to $n - 1$ parties. Assuming a sufficiently large circuit, the protocol achieves a per-party bandwidth of $O(n \cdot (\lambda + \ell))$ bits per multiplication, addition gate, and input/output wire, and $O(n \cdot \lambda \cdot (\lambda + \ell))$ bits per truncation gate (i.e., floor division by 2), where λ is the computational security parameter, $\ell = \log \mathcal{B}$, and n is the number of parties. Moreover, the addition gates can be instantiated with no communication cost when the circuit $\mathcal{C}$ is a layered circuit, or under an additional KDM security assumption for the encryption scheme in Figure 3.*

2 Technical Roadmap

In this section, we provide an intuitive overview of our protocol. We first review the BMR template [BMR90, WRK17b] for converting a garbling scheme to a constant-round MPC protocol. Then, we describe our underlying arithmetic garbling scheme based on the work of Ball et al. [BLLL23]. Finally, we outline our tailored protocol for realizing the inner-MPC of the BMR template in the multi-party setting, starting with the semi-honest model and then extending to the malicious model.

2.1 Background: Garbling Scheme and BMR Blueprint

At a high level, our protocol follows the BMR template [BMR90, WRK17b] for constructing a constant-round MPC from a suitable garbling scheme. Abstractly, a garbling scheme consists of the following four algorithms:

- Garble: Given a circuit $\mathcal{C}$, the garbler computes a garbled circuit $\widehat{\mathcal{C}}$ and auxiliary information e and d for input encoding and output decoding.
- Encode: Given the encoding information e and an input string x of $\mathcal{C}$, the encoder computes an encoded input $\widehat{x}$.
- Eval: Given the garbled circuit $\widehat{\mathcal{C}}$ and the encoded input $\widehat{x}$, the evaluator computes an encoded output $\widehat{y}$.
- Decode: Given the decoding information d and the encoded output $\widehat{y}$, the decoder recovers the output $y = \mathcal{C}(x)$.

We assume the garbling scheme satisfies the following standard properties [BHR12b], which we informally describe here:

- *Obliviousness.* The evaluator learns nothing about the input x from $(\widehat{\mathcal{C}}, \widehat{x})$.
- *Privacy.* When the evaluator colludes with the decoder, they learn nothing about the input x from $(\widehat{\mathcal{C}}, \widehat{x}, d)$ beyond what is revealed by the output $\mathcal{C}(x)$.
- *Correctness.* In an honest execution, the output y equals $\mathcal{C}(x)$.
- *Authenticity.* When the garbler and encoder are honest, a malicious evaluator cannot cause an honest decoder to output an incorrect result $y \neq \mathcal{C}(x)$.

With these properties, we can convert a garbling scheme into an MPC protocol through the following template.

1. All the parties $P_1, \ldots, P_n$ jointly run Garble and Encode within an inner-MPC protocol. The protocol outputs $\widehat{C}$ and $\widehat{x}$ to a designated party P_1[3], and the decoding information d to the parties associated with the output.
2. P_1 runs $\mathsf{Eval}(\widehat{C}, \widehat{x})$ and sends the encoded output $\widehat{y}$ to the associated parties.
3. Each associated party locally runs $\mathsf{Decode}(\widehat{y}, d)$ to recover the final output.

[3]In the original BMR template [BMR90], every party needs to run Eval locally, incurring an n-fold blowup in the garbled circuit transmission. Here, we adopt the optimization from [WRK17b] and let a single party be the evaluator, leveraging the authenticity guarantee of the garbling scheme.

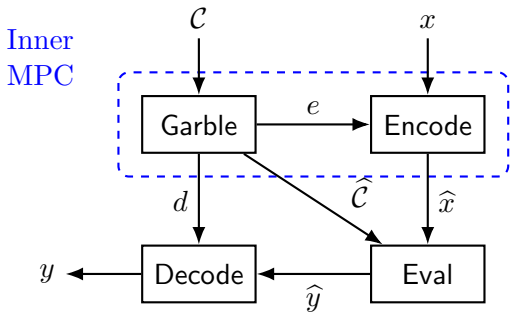

Figure 1: The BMR [BMR90, WRK17b] template for MPC from a garbling scheme.

As Beaver et al. [BMR90] point out, if the Garble and Encode algorithms have constant depth, then the inner-MPC can be realized in a constant number of rounds using standard techniques [GMW87, Bea91, DPSZ12], resulting in an overall constant-round MPC protocol. Moreover, it is not hard to see that if the inner-MPC protocol is secure against all-but-one corruption, then the overall MPC protocol is also secure against all-but-one corruption, thanks to the aforementioned properties of the garbling scheme.

2.2 Arithmetic Garbling from BLLL

Next, we give an overview of our underlying garbling scheme, which is adapted from the elegant work of Ball et al. [BLLL23] (hereafter referred to as BLLL).

Model of computation. While BLLL introduces multiple computational models, throughout this paper, we consider circuits comprising arithmetic gates that compute linear combinations, multiplication, and truncation (floor-division-by-2), defined over bounded integers in $[-\mathcal{B}, \mathcal{B}]$ for a global parameter $\mathcal{B}$. We provide more details of this model in Section 4.2 and show that it implies other computation models, such as fixed-point arithmetic, mixed arithmetic and Boolean circuits, and modular arithmetic.

Nonce-based LHE. A key building block in BLLL's garbling scheme is a Linearly Homomorphic Encryption (LHE) scheme. Informally, the scheme supports the following homomorphic properties over addition and scalar multiplication:

$$\mathsf{Enc}_{k_1}(m_1;\ \rho) \boxplus \mathsf{Enc}_{k_2}(m_2;\ \rho) = \mathsf{Enc}_{k_1+k_2}(m_1 + m_2;\ \rho)$$
$$s \boxdot \mathsf{Enc}_{k}(m;\ \rho) = \mathsf{Enc}_{s\cdot k}(s \cdot m;\ \rho)$$

where s is a scalar, k_1, k_2, k are secret keys, m_1, m_2, m are plaintexts, and ρ is the randomness used for encryption. Compared to the original BLLL construction, we additionally require the encryption scheme to be nonce-based [Rog04], meaning that the scheme remains secure even when the adversary learns the randomness ρ. As we will see, this requirement helps us design a more efficient protocol for the inner-MPC component of the BMR template. We provide a concrete instantiation of such an LHE scheme from the Decisional Composite Residuosity (DCR) assumption in Section 3.2.

Labels and masks. As a key invariant in BLLL's construction, for each wire x, the garbler chooses two random masks $r^{(x)}$ and $R^{(x)}$. Correspondingly, the evaluator should derive the *masked*

value $\widetilde{x} = x + r^{(x)} \in \mathbb{Z}$ and *label* $L^{(x)} = \Delta \cdot \widetilde{x} + R^{(x)} \in \mathbb{Z}$. Here, we overload x to denote the value of the wire, and Δ is a global secret key held by the garbler[4].

Free Linear Combination. For each linear gate computing $z = \mu \cdot x + \nu \cdot y$, where μ, ν are constant integers, the garbler sets $r^{(z)} = \mu \cdot r^{(x)} + \nu \cdot r^{(y)}$ and $R^{(z)} = \mu \cdot R^{(x)} + \nu \cdot R^{(y)}$. Then, the evaluator can compute the label and random mask of the output wire as $L^{(z)} = \mu \cdot L^{(x)} + \nu \cdot L^{(y)}$ and $\widetilde{z} = \mu \cdot \widetilde{x} + \nu \cdot \widetilde{y}$.

Core Building Block: Key-Extension. To support multiplication and truncation gates, BLLL relies on a *Key-Extension* operation. Given the input label $L^{(x)}$, the masked value $\widetilde{x}$, and the garbling materials of key extension, the evaluator can compute an affine function $\xi = A \cdot \widetilde{x} + B$ of the masked value, where A and B are secret coefficients chosen by the garbler. Informally, the key-extension protocol should ensure that the evaluator learns nothing about these coefficients beyond what is revealed by the output ξ, even when multiple key-extension operations are performed on the same wire x.

To achieve this, the garbler provides two LHE ciphertexts

$$\mathsf{ct}_1 = \mathsf{Enc}_{\Delta}(A;\ \rho), \quad \mathsf{ct}_2 = \mathsf{Enc}_{R^{(x)}}(B;\ \rho)$$

where the same public randomness ρ is used for both ct_1 and ct_2[5]. The linear homomorphism allows the evaluator to combine these ciphertexts and derive an encryption of ξ under the key $L^{(x)}$:

$$\mathsf{Enc}_{L^{(x)}}(\xi;\ \rho) = \mathsf{Enc}_{\Delta \cdot \widetilde{x} + R^{(x)}}(A \cdot \widetilde{x} + B;\ \rho) = (\widetilde{x} \boxdot \mathsf{ct}_1) \boxplus \mathsf{ct}_2$$

The evaluator then decrypts this ciphertext using $L^{(x)}$ to obtain ξ.

Multiplication gates. For a multiplication gate computing $z = x \cdot y$, the goal is to allow the evaluator to compute the label $L^{(z)}$ and the output masked value $\widetilde{z}$. To compute $L^{(z)}$, the evaluator starts by computing the product $\widetilde{x} \cdot L^{(y)}$. This expands to terms involving the desired $\Delta \cdot x \cdot y$, but also introduces unwanted cross-terms such as $\Delta \cdot x \cdot r^{(y)}$. To cancel these cross-terms and adjust the randomness to match the target label $L^{(z)}$, the garbler provides two correction values ξ_x and ξ_y using key-extension on wires x and y respectively. The evaluator then computes the target label as

$$L^{(z)} = \widetilde{x} \cdot L^{(y)} - \xi_x - \xi_y$$

where the corrections are defined as:

$$\begin{aligned}
\xi_x &= (\Delta \cdot r^{(y)} + R^{(y)}) \cdot \widetilde{x} + U \\
\xi_y &= (\Delta \cdot r^{(x)}) \cdot \widetilde{y} - (U + \Delta \cdot r^{(x)} \cdot r^{(y)} + \Delta \cdot r^{(z)} + R^{(z)})
\end{aligned}$$

where U is a random mask that ensures the evaluator learns nothing about ξ_x or ξ_y except for their sum. Note that the definition of ξ_x and ξ_y involves products of the global key Δ with other values. Consequently, we require the LHE scheme to satisfy a Key-Dependent Message (KDM) security notion where the message may be an affine function of the secret key. We show an efficient instantiation of such an LHE scheme from the standard DCR assumption in Section 3.2.

[4]The original BLLL protocol [BLLL23] uses a different key Δ for each wire. We adapt this to a global Δ to facilitate free linear combinations (additions) and simplify the construction.

[5]Note that ct_2 can be simulated given ct_1, $\widetilde{x}$, $L^{(x)}$, ξ, and ρ. Therefore, reusing ρ in ct_2 does not leak the secret coefficients (beyond what is revealed by ξ).

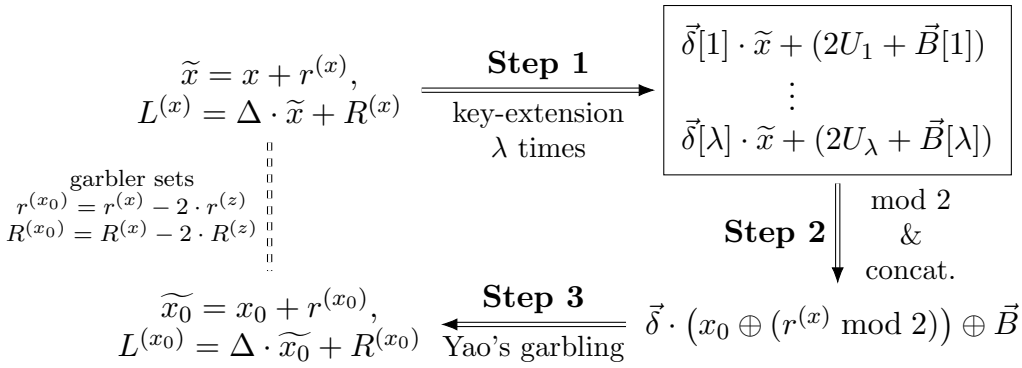

Figure 2: Truncation procedure of [BLLL23] and subsequent work [LL24] (with simplifications) from the evaluator's view. Steps 1 and 2 convert the arithmetic label to a Boolean label representing the input x's least significant bit x_0. The Boolean label is then converted back to the arithmetic mask and label through Yao's garbling [Yao86] with point-and-permute [BMR90]. Finally, the evaluator computes the mask and the label of the output wire z via the relation $z = (x - x_0)/2$.

A similar logic applies to computing the output masked value. The product $\widetilde{x} \cdot \widetilde{y}$ contains the target term $x \cdot y$ mixed with cross-terms like $x \cdot r^{(y)}$. To isolate the correct output $\widetilde{z} = x \cdot y + r^{(z)}$, the garbler provides corrections ξ'_x and ξ'_y such that $\widetilde{z} = \widetilde{x} \cdot \widetilde{y} - \xi'_x - \xi'_y$. These are defined as $\xi'_x = r^{(y)} \cdot \widetilde{x} + V$ and $\xi'_y = r^{(x)} \cdot \widetilde{y} - (V + r^{(x)} \cdot r^{(y)} + r^{(z)})$, where V is another random mask with a purpose similar to U.

Truncation gates. Figure 2 illustrates the truncation procedure from BLLL [BLLL23] and subsequent work [LL24], adapted to our setting with a global key Δ. For a truncation gate $z = \lfloor x/2 \rfloor$, the evaluator first computes an intermediate label $L^{(x_0)} = \Delta \cdot \widetilde{x_0} + R^{(x_0)}$ and masked value $\widetilde{x_0} = x_0 + r^{(x_0)}$ where $x_0 = (x \bmod 2)$. The intermediate masks $R^{(x_0)}, r^{(x_0)}$ are chosen so that $L^{(z)} = (L^{(x)} - L^{(x_0)})/2$ and $\widetilde{z} = (\widetilde{x} - \widetilde{x_0})/2$.

To compute the intermediate label $L^{(x_0)}$ and masked value $\widetilde{z}$, the evaluator further goes through three steps. First, it performs λ instances of key-extension on wire x and derives a λ-dimensional vector $\vec{\xi} = \vec{\delta} \cdot \widetilde{x} + 2 \cdot \vec{U} + \vec{B}$, where $\vec{\delta}$ and $\vec{B}$ are fresh secret bit-vectors, and the entries of $\vec{U}$ are fresh secret integer masks. Second, it reduces each entry of the vector modulo 2, and concatenates the resulting binary values. The concatenated string can be viewed as a garbled Boolean label for x_0. Finally, the evaluator can convert the Boolean label back into the arithmetic label $L^{(x_0)}$ and the masked value $\widetilde{x_0}$ using a two-row garbled truth table, where the rows are permuted based on the bit $r^{(x)} \bmod 2$. To facilitate the multi-party generalization later, we let the garbled truth table entries be generated using the same LHE scheme as in the key-extension.

Admittedly, the circular encryption of arithmetic and Boolean keys in our truncation gates necessitates a KDM security assumption for the LHE scheme, as detailed in Definition 3.1. Similar assumptions have been employed in prior works of arithmetic garbling [MORS24, CHHK25, MORS25] to reduce the cost of multiplication. In Section C, we show how to remove the KDM assumption by sampling a separate key per level of the circuit, at the cost of increasing the overhead of addition and multiplication gates.

Authenticated Decoding. While the original BLLL construction does not state any authenticity guarantee, we can extend our garbling scheme to achieve authenticity against a malicious evaluator by adding an authenticated decoding algorithm. Specifically, for each output wire y

of the circuit, the garbler includes in the decoding information an additional pair of ciphertexts $\mathsf{Enc}_{\Delta}(0;\ \rho^{(y)}), \mathsf{Enc}_{R^{(y)}}(0;\ \rho^{(y)})$, together with the random mask $r^{(y)}$ for recovering the output. Meanwhile, the decoder receives from the evaluator the encoded output, which consists of the masked value $\widetilde{y}$ and a ciphertext $\mathsf{Enc}_{L^{(y)}}(0;\ \rho^{(y)})$. The decoder can then check that $\widetilde{y}$ is in the correct range and satisfies the relation

$$\mathsf{Enc}_{L^{(y)}}(0;\ \rho^{(y)}) = (\widetilde{y} \boxdot \mathsf{Enc}_{\Delta}(0;\ \rho^{(y)})) \boxplus \mathsf{Enc}_{R^{(y)}}(0;\ \rho^{(y)}).$$

To see why this offers authenticity, suppose that the evaluator can inject an additive error ϵ to $\widetilde{y}$ while still passing the check. Then, it must learn $\Gamma = \epsilon \boxdot \mathsf{Enc}_{\Delta}(0;\ \rho^{(y)})$ and add it to the ciphertext in the encoded output. Moreover, our LHE instantiation ensures that Γ must be non-trivial given that $\widetilde{y}+\epsilon$ passes the decoder's range check. This means the evaluator can distinguish $\mathsf{Enc}_{\Delta}(0;\ \rho^{(y)})$ from a random ciphertext even though the randomness $\rho^{(y)}$ is fresh, contradicting the security of the LHE scheme.

Summary of Garbling Materials. Finally, we summarize the outputs of the garbling and encoding algorithms in the BLLL garbling scheme, as they will be computed via the inner MPC protocol in the BMR template.

- The garbled circuit $\widehat{C}$ consists solely of LHE ciphertexts of the form $\mathsf{ct} = \mathsf{Enc}_k(A;\ \rho)$, for some integer secret key k, message A, and public random value ρ of the gate.
- The decoding information d consists of $r^{(y)}$, $\mathsf{Enc}_{\Delta}(0;\ \rho^{(y)}), \mathsf{Enc}_{R^{(y)}}(0;\ \rho^{(y)})$, and the public randomness $\rho^{(y)}$ for each output wire y.
- The encoded input for each wire x consists of the masked value $\widetilde{x} = x + r^{(x)}$ and the label $L^{(x)} = \Delta \cdot \widetilde{x} + R^{(x)}$.

2.3 Our Multi-party Generalization

Next, we describe our tailored protocol for realizing the inner-MPC of the BMR template in the multi-party setting.

Client-server model. We assume a client-server model where the parties are divided into clients who hold (private) inputs and expect outputs, and servers who don't have inputs or outputs. The adversary can statically corrupt an arbitrary number of clients, and at most $n-1$ out of n servers. The model reduces to the classic MPC scenario when each server runs a client.

For simplicity, we first describe the protocol in the semi-honest model, where the parties follow the protocol honestly but try to learn additional information from the execution transcript.

Distributed generation of garbled circuit. Our main observation is that the homomorphic property of the LHE scheme suggests a very simple and efficient way of generating the ciphertexts in step 1 of the BMR template. For an LHE ciphertext of the form $\mathsf{Enc}_k(A;\ \rho)$, our protocol proceeds as follows:

(a) Invoke a sub-protocol that distributes additive shares $\langle k \rangle, \langle A \rangle$, and the public random value ρ to all n servers.

(b) Each server P_i locally computes $\mathsf{Enc}_{\langle k \rangle_i}(\langle A \rangle_i;\ \rho)$ and sends it to P_1.

(c) P_1 locally combines the received ciphertexts as

$$\mathsf{Enc}_k(A;\ \rho) = \mathsf{Enc}_{\langle k\rangle_1}(\langle A\rangle_1;\ \rho) \boxplus \cdots \boxplus \mathsf{Enc}_{\langle k\rangle_n}(\langle A\rangle_n;\ \rho)$$

Compared to using a generic MPC compiler, the above procedure avoids computing the heavyweight encryption algorithm in MPC. The sub-protocol in step (a) can be realized efficiently using standard MPC techniques, as we explain next.

For each multiplication gate $z = x \cdot y$, the LHE key k is either the global key Δ or random masks $R^{(x)}$ and $R^{(y)}$. It suffices to let each server locally sample its shares $\langle\Delta\rangle_i$, $\langle R^{(x)}\rangle_i$, and $\langle R^{(y)}\rangle_i$ at random from appropriate ranges. Intuitively, the randomness from a single honest server suffices for the security of the garbling scheme. Meanwhile, the encrypted message A contains constant-degree polynomials of Δ, $r^{(x)}$, and $r^{(y)}$, whose shares can be derived efficiently using a Beaver-triple-based interactive MPC protocol [Bea91] with modulus matching the message space of our LHE scheme.

The truncation gate follows a similar paradigm, with one additional requirement: we need to generate shares of random bits over the integers for garbling the Boolean truth table. To achieve this, we first generate shares of bits with respect to a sufficiently large prime modulus following the approach of [GLM+24], and then run a ModToInt protocol adapted from [DFK+06] to convert them into integer shares.

Finally, to sample a batch of m public random values ρ for our DCR-based LHE scheme, the parties only need to sample roughly $2m$ uniformly random elements in a finite ring of integers. The latter can be done efficiently using a standard coin-tossing protocol.

Input encoding. For every input wire x provided by a client, the servers reveal their shares $\langle r^{(x)}\rangle$ to that client. The client reconstructs the mask and broadcasts the masked input $\widetilde{x} = x + r^{(x)}$ to all servers. The servers then locally compute shares of the input label as $\langle L^{(x)}\rangle = \langle\Delta\rangle \cdot \widetilde{x} + \langle R^{(x)}\rangle$. Finally, they transmit these shares to P_1, who reconstructs $L^{(x)}$.

Output decoding. For each output wire y, the servers similarly reveal their shares $\langle r^{(y)}\rangle$ to the client, so that the client can run the Decode algorithm to recover the output.

Efficiency. We analyze the asymptotic communication cost of our protocol, assuming a sufficiently large circuit. In our DCR-based instantiation, each LHE ciphertext has size $O(\lambda + \log \mathcal{B})$, where λ is the bit length of the DCR modulus. For each multiplication gate, the n parties send a constant number of ciphertexts and invoke the SPDZ protocol [DPSZ12] to prepare key-extension coefficients, incurring $O(n \cdot (\lambda + \log \mathcal{B}))$ communication. Truncation gates are more expensive, requiring $O(n \cdot \lambda(\lambda + \log \mathcal{B}))$ bits due to λ instances of key-extension. Input and output wires each contribute $O(n \cdot (\lambda + \log \mathcal{B}))$ communication. Consequently, the total communication complexity is $O(n \cdot (\lambda + \log \mathcal{B}) \cdot (m_{\text{mul}} + m_{\text{in}} + m_{\text{out}} + \lambda \cdot m_{\text{div2}}))$, where $m_{\text{mul}}, m_{\text{div2}}, m_{\text{in}}, m_{\text{out}}$ denote the number of multiplication gates, truncation gates, input wires, and output wires respectively.

2.4 Towards Malicious Security

Finally, we outline the components required for malicious security. Since the servers do not own inputs or outputs, it suffices to consider the maximal corruption of servers where an adversary controls $n - 1$ servers, including the evaluator P_1, as prior work [GLM+24] observes.

Garbled circuit generation. Recall the garbled circuit consists solely of LHE ciphertexts and public randomness ρ. We begin by analyzing the LHE ciphertext generation in Section 2.3. Assuming the secure execution of step (a), the authenticity of the underlying garbling scheme ensures security against malicious behavior during steps (b) and (c). Namely, an adversary cannot force a client to accept an incorrect output. Note that since P_1 is corrupted, we need not worry about forged ciphertexts in step (b), as they belong to the internal communication among the corrupt servers.

Consequently, our focus lies in designing maliciously-secure sub-protocols for step (a)—specifically, for distributing additive shares $\langle k \rangle, \langle A \rangle$, and ρ to compute LHE ciphertexts. It turns out that we need to address the following attacks:

Attack 1 *Tampering secret shares.* The semi-honest MPC protocol [Bea91] is insufficient for computing the shares $\langle A \rangle$. For example, a malicious server can manipulate the computation to learn a label $\Delta \cdot \widetilde{w}' + R^{(w)}$ for a modified masked value $\widetilde{w}' \neq \widetilde{w}$, thereby effectively altering the wire's underlying value.

Attack 2 *Out-of-bound mask.* In the semi-honest protocol, we let each server choose their own share of the mask $r^{(w)}$. However, a malicious server can choose an arbitrarily large share of the mask $r^{(w)}$. This may cause the term $\Delta \cdot (w + r^{(w)})$ to exceed the noise range covered by $R^{(w)}$, leaking information about honest servers' shares of the global key Δ.

Attack 3 *Correlated randomness.* A malicious server can manipulate the semi-honest coin-tossing protocol to introduce correlated randomness ρ across different ciphertexts. Specifically, our DCR-based LHE instantiation is insecure if the adversary knows the discrete logarithm $\log_{\rho_1}(\rho_2)$ relating the randomness ρ_1, ρ_2 of two ciphertexts.

Note that the honest server's ciphertext share does not depend on the corrupt servers' shares of $R^{(w)}$. Therefore, we need not consider incorrect sampling of $R^{(x)}$ by malicious servers.

Addressing Attack 1. To ensure correct computation of the share $\langle A \rangle$, we use the SPDZ protocol [DPSZ12], where parties hold a MAC share $\langle \Delta' \cdot x \rangle$ for a secret-shared global MAC key Δ' together with each additive share $\langle x \rangle$. As an optimization, the MAC key Δ' can be set equal to the global key Δ of the LHE scheme[6], which simplifies our protocol and improves its concrete efficiency.

A technicality here is that when we instantiate the LHE, the plaintext space is not a finite field, but the ring $\mathbb{Z}_{N^\zeta}$ where $N = p \cdot q$ for two secret large primes p and q, and ζ is an arbitrary positive integer. Consequently, the SPDZ protocol must be adapted to this ring. While the adaptation is straightforward for multiplications and MAC-check protocols, we design a new protocol for sampling random bits, described in Section A.1.

Addressing Attack 2. In the semi-honest protocol, each party P_i locally samples a share $\langle r^{(x)} \rangle_i \leftarrow [0, r_{\max})$. Hence, a naive countermeasure to prevent out-of-bound masks would be to prove that the reconstructed $r^{(x)}$ lies in $[0, n \cdot (r_{\max} - 1)]$; however, such a range proof typically requires bit-decomposition, which is prohibitively expensive when every wire requires a separate proof.

To reduce overhead, we observe that it suffices to verify that $r^{(x)}$ lies within a relaxed interval $(-M \cdot n \cdot r_{\max}, M \cdot n \cdot r_{\max})$, provided that $R_{\max} \gg M \cdot n \cdot r_{\max} \cdot \Delta_h$. Furthermore, since $L^{(x)}$ is computed modulo N^ζ, adding multiples of N^ζ to the masks does not leak additional information. Thus, we can assume without loss of generality that $r^{(x)} \in [-N^\zeta/2, N^\zeta/2)$. These relaxations enable an efficient batch range check with cost independent of the number of wires.

[6] A subtle issue is that SPDZ MAC checks leak the global Δ in cases of abort. The servers therefore need to only deliver the garbled circuit, which contains ciphertexts under Δ, after all MAC checks pass.

Specifically, after obtaining SPDZ shares of $r^{(x)}$ for all wires, the servers perform the following randomized check:

> Repeat the following κ times, where κ is a statistical security parameter:
>
> – Sample a random subset of wires. Select each wire with probability 1/2.
>
> – Locally compute the SPDZ sharing of the sum of the selected masks.
>
> – Verify that the sum lies in $[0, M \cdot n \cdot r_{\max})$, where M is the total number of wires and n is the number of servers.

In an honest execution, the sum will always fall within the required range. However, if any mask $r^{(x)}$ lies outside $(-M \cdot n \cdot r_{\max}, M \cdot n \cdot r_{\max})$, the check fails with probability at least 1/2. This is because among the two cases where $r^{(x)}$ is either selected or not selected, at most one will result in a sum within the target range. As the trials are independent, the probability of an adversary cheating undetected is at most $2^{-\kappa}$.

In our formal protocol, we introduce another relaxation factor to ensure that the range check can be performed in constant rounds. We provide the full details in Section A.1.

Addressing Attack 3. To guarantee the independence of the randomness ρ, we can employ any maliciously-secure coin-tossing protocol. A standard approach involves generating authenticated SPDZ shares of random values, reconstructing them publicly, and verifying their MACs. In our concrete implementation, we adopt a more efficient strategy where parties coin-toss a random seed, which is then expanded via a random oracle to derive all necessary randomness.

Input encoding. Next, we consider attacks related to the input encoding phase. Recall that for each input wire x from a client, the servers reveal shares of the mask $r^{(x)}$ to the client, who then broadcasts the masked input $\widetilde{x} = x + r^{(x)}$ to all servers. We identify the following attacks:

Attack 4 Sending incorrect masks. A malicious server can send a falsified share of the mask $r^{(x)}$ to an honest client, which equivalently modifies the client's input.

Attack 5 Malicious client inputs. A malicious client may provide an input x that falls outside the allowed range. If such an out-of-bound x is multiplied by an honest client's input y, the product $x \cdot y$ might overflow the range protected by the output mask $r^{(z)}$, potentially leaking information about y. Moreover, if the masked value $\widetilde{x} = x + r^{(x)}$ is too large, it can also lead to leakage of the global key Δ as in *Attack 2*.

Note that since there is only one honest server, we don't need to worry about the malicious client sending inconsistent values to different servers.

Addressing Attack 4. To ensure clients receive correct masks, we adopt a similar method as prior work [GLM+24] to authenticate each mask $r^{(w)}$ to the client. At a high level, the protocol proceeds in two steps: first, the parties authentically open a random challenge α to the client by consuming a multiplication triple. Next, they compute shares of the product $\alpha \cdot r^{(w)}$ via (authenticated) SPDZ multiplication and open it to the client. The client then reconstructs the product and verifies the MAC on $r^{(w)}$ using the opened α.

Addressing Attack 5. Finally, to prevent overflow attacks from malicious inputs, we verify the range of each input wire from the clients. Without loss of generality, we assume all input wires are bits, as larger inputs can be built from single bits using linear gates for free.

To check if an input wire $x \in \{0,1\} \pmod{N^\zeta}$, the servers first obtain a SPDZ sharing of x using the public value $x + r^{(x)}$ and their shares of $r^{(x)}$. Then, they use the SPDZ protocol to verify that $x \cdot (x-1) \equiv 0 \pmod{N^\zeta}$. Although there are other solutions to this equation due to the composite modulus, we show that finding them is as hard as factoring N.

Finally, each server checks that the masked input $\widetilde{x}$ it receives from the client is in the valid range $[0, n \cdot r_{\max})$. This ensures that the server's share of the global key Δ will not be leaked when, for example, $\widetilde{x}$ wraps around modulo N^ζ.

Output decoding. For each output wire y, the decoding information provided to the client includes ciphertexts $\mathsf{Enc}_\Delta(0;\ \rho^{(y)})$, $\mathsf{Enc}_{R^{(y)}}(0;\ \rho^{(y)})$, and the random mask $r^{(y)}$. Additionally, the client receives the masked output $\widetilde{y}$ and the ciphertexts $\mathsf{Enc}_{L^{(y)}}(0;\ \rho^{(y)})$ from the evaluator P_1.

Since P_1 is corrupt and knows the true masked value $\widetilde{y}$, any tampering by P_i on the ciphertexts in the decoding information is equivalent to tampering with the ciphertext $\mathsf{Enc}_{L^{(y)}}(0;\ \rho^{(y)})$ in the encoded output. The authenticity of the garbling scheme ensures that such tampering will be detected during the decoding process.

Therefore, it suffices to ensure that the client receives the correct mask $r^{(y)}$. This can be achieved using the same authentication method as for *Attack 4*.

Efficiency. We note that each additional component for malicious security introduces either a constant factor multiplicative overhead or an additive overhead independent of the circuit size. Therefore, the overall asymptotic communication complexity remains unchanged from the semi-honest protocol. We provide a detailed analysis in Section D.

3 Preliminaries

3.1 Secret Sharing and SPDZ

Secret sharing. We first define the notation for secret sharing in our protocol. Our protocols primarily work over the ring $\mathbb{Z}_{N^\zeta}$, where N is a safe RSA modulus. We denote additive secret sharing of a value $x \in \mathbb{Z}_{N^\zeta}$ by $\langle x \rangle$, where each party P_i holds a share $\langle x \rangle_i$ such that $\sum_{i=1}^n \langle x \rangle_i = x \pmod{N^\zeta}$. Additionally, we use $\langle x \rangle^*$ to denote a secret sharing of x over the integers $\mathbb{Z}$; that is, each party P_i holds an integer share $\langle x \rangle_i^*$ such that $\sum_{i=1}^n \langle x \rangle_i^* = x$.

We denote the SPDZ-style authenticated sharing by $[\![x]\!] = (\langle x \rangle, \langle \Delta \cdot x \rangle)$, where Δ is the global MAC key. Similarly, we use $[\![x]\!]^* = (\langle x \rangle^*, \langle \Delta \cdot x \rangle)$ to denote an authenticated sharing where the value x is shared over the integers.

Parties can convert an integer sharing $\langle x \rangle^*$ to a modular sharing $\langle x \rangle$ by locally reducing each share modulo N^ζ. This naturally extends to converting $[\![x]\!]^*$ to $[\![x]\!]$. For brevity, we often omit explicit type conversion in our descriptions.

Finally, in the occasional instances where we require sharing modulo a prime p, we use $\langle x \rangle^p$ to denote an additive sharing of x in $\mathbb{Z}_p$, and $[\![x]\!]^p$ to denote the corresponding SPDZ-style authenticated sharing using a distinct global key Δ'.

Opening a secret share. To open a secret share, the parties can run a simple protocol where each party sends their share to a designated party (e.g., P_1), who reconstructs the value and sends it to all parties. The following protocol works similarly for integer sharing $\langle x \rangle^*$.

Protocol $\Pi_{\text{open}}(\langle x \rangle)$

1. All the parties send their shares of $\langle x \rangle$ to P_1.
2. P_1 reconstructs x and sends it to all the parties.

MAC check on opened values. When the secrets of some SPDZ sharings are opened to all parties, they can perform an efficient batch check on the opened values using the MACs via a standard SPDZ MAC check protocol $\Pi_{\text{SPDZ-MAC}}$ in the $\{\mathcal{F}_{\text{coin}}, \mathcal{F}_{\text{commit}}\}$-hybrid model, which we describe below.

Functionality $\mathcal{F}_{\text{coin}}$

- On input (`coin`) from all parties, the functionality samples a uniform random string $s \in \{0,1\}^\lambda$. The functionality sends s to the adversary $\mathcal{S}$.
- If the functionality receives `abort` from $\mathcal{S}$, it sends `abort` to all parties and halts. Otherwise, it sends s to all parties.

Functionality $\mathcal{F}_{\text{commit}}$

- On input $(\texttt{commit}, \text{id}, v)$ from P_i where id is an unused identifier, store (id, P_i, v) and output (id, P_i) to all.
- On input $(\texttt{open}, \text{id})$ from P_i, retrieve (id, P_i, v) and output v to all.

Protocol $\Pi^{\langle \Delta \rangle}_{\text{SPDZ-MAC}}$

Let all parties agree on a PRG $\mathsf{Prg} : \{0,1\}^\lambda \to (\mathbb{Z}_{N\zeta})^m$. Parties want to check the opened values $(A_1, \ldots, A_m)$ for sharings $([\![A_1]\!], \ldots, [\![A_m]\!])$ under the MAC key Δ.

1. The parties call $\mathcal{F}_{\text{coin}}$ to get a random seed s and expand it with Prg to get random values $\chi_1, \ldots, \chi_m \in \mathbb{Z}_{N\zeta}$.
2. The parties compute $A = \sum_{j=1}^m \chi_j \cdot A_j$ and $\langle \Delta \cdot A \rangle = \sum_{j=1}^m \chi_j \cdot \langle \Delta \cdot A_j \rangle$.
3. The parties compute $\langle \sigma \rangle = \langle \Delta \cdot A \rangle - \langle \Delta \rangle \cdot A$. Each P_i calls $\mathcal{F}_{\text{commit}}$ with input $(\texttt{commit}, P_i, \langle \sigma \rangle_i)$.
4. The parties open their commitments and check that $\sum_{i=1}^n \langle \sigma \rangle_i = 0$. If not, the parties output **abort** and abort the protocol.

Modular arithmetic. Next, we define the following modular arithmetic operations on SPDZ sharings.

- *Addition:* $[\![z]\!] = [\![x]\!] + [\![y]\!]$. Parties locally compute $\langle z \rangle = \langle x \rangle + \langle y \rangle$ and $\langle \Delta \cdot z \rangle = \langle \Delta \cdot x \rangle + \langle \Delta \cdot y \rangle$.
- *Multiplication:* $[\![z]\!] = [\![x]\!] \cdot [\![y]\!]$. This requires interaction and consumes a multiplication triple. The protocol Π_{mul} is described below.

Protocol $\Pi_{\text{mul}}([\![x]\!], [\![y]\!])$

1. Consume a preprocessed triple $([\![a]\!], [\![b]\!], [\![c]\!])$. The parties compute

$$[\![e]\!] = [\![x - a]\!], \qquad [\![d]\!] = [\![y - b]\!]$$

   and open the shares by invoking Π_{open}.

2. The parties compute the multiplication by

$$[\![z]\!] = [\![1]\!] \cdot e \cdot d + e \cdot [\![b]\!] + d \cdot [\![a]\!] + [\![c]\!].$$

Assuming the preprocessed triples are authenticated, the parties can ensure the correctness of the multiplication result by performing a MAC check on the opened values e and d using $\Pi^{\langle\Delta\rangle}_{\text{SPDZ}-\text{MAC}}$.

3.2 Linearly Homomorphic Encryption (LHE) from DCR

LHE for key-extension. In BLLL [BLLL23], the authors show that their key-extension operation can be instantiated from an encryption scheme LHE with the following property:

$$(s \boxdot \mathsf{LHE.Enc}_{k_1}(m_1;\ \rho)) \boxplus \mathsf{LHE.Enc}_{k_2}(m_2;\ \rho) = \mathsf{LHE.Enc}_{s \cdot k_1 + k_2}(s \cdot m_1 + m_2;\ \rho)$$

where s is a scalar, k_1, k_2 are secret keys, m_1, m_2 are plaintexts, and ρ is the randomness used for encryption. Here, $\boxdot$ and $\boxplus$ denote the scalar multiplication and addition operations on ciphertexts, respectively.

Compared to BLLL, we require two stronger security properties for the underlying LHE scheme.

1. *Nonce-based.* We require that security still holds when the adversary learns the random coins ρ used for encryption. This is unnecessary in the two-party setting of BLLL since the garbler can keep ρ secret. However, in our multi-party protocol, the parties jointly generate the garbled circuit. To aggregate their shares of each ciphertext, the parties must use a common randomness ρ that is known to all parties.

2. *KDM security:* We require that the encryption scheme is secure against a key-dependent message (KDM) attack where the message is an affine function of the secret key. This allows us to use the same global key across all the gates in the circuit, hence supporting free additions of wire values and simplifying the protocol.

Definition 3.1 (KDM security of nonce-based symmetric-key encryption)**.** An encryption scheme $(\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ is said to be nonce-based and KDM secure with respect to a class of functions $\mathbb{F}$ if for all PPT adversaries $\mathcal{A}$, the following two oracles are indistinguishable:

- $\mathcal{O}_0^{\mathsf{KDM},\Delta}(f \in \mathbb{F})$: returns $\{\rho, \mathsf{Enc}_k(f(\Delta);\ \rho)\}$, where ρ is the randomness sampled by Enc.
- $\mathcal{O}_1^{\mathsf{KDM},\Delta}(f \in \mathbb{F})$: returns $\{\rho, \mathsf{Enc}_k(z;\ \rho)\}$, where ρ is the randomness sampled by Enc, and z is a random plaintext sampled uniformly from the plaintext space.

Here, we omit the public parameters sampled from Setup, and Δ is a global secret key sampled from KeyGen.

Instantiation from DCR. We instantiate the above LHE scheme from a variant of the Damgård-Jurik cryptosystem [DJN10], which is based on the Decisional Composite Residuosity (DCR) assumption [Pai99].

Our instantiation differs from BLLL [BLLL23] in two main aspects:

1. In BLLL, they first sample a global generator g_{global} from $\mathbb{Z}^{\times}_{N^{\zeta+1}}$, a random exponent ρ for each key-extension operation, and reveal $g = g_{\text{global}}^{\rho}$ to all the parties. The security breaks when the randomness ρ is revealed. In our scheme, we take inspiration from prior works [CS02,LLPT23] and sample $g = \rho$ directly from a subgroup of $\mathbb{Z}^{\times}_{N^{\zeta+1}}$ with Jacobi symbol 1 modulo N, so the scheme satisfies the nonce-based requirement.

2. BLLL is based on the Paillier cryptosystem [Pai99], which can be viewed as a special case of the Damgård-Jurik cryptosystem with $\zeta = 1$. We generalize their scheme to work with an arbitrarily large message space $\mathbb{Z}_{N^{\zeta}}$.

$\mathsf{LHE} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$

- $\mathsf{Setup}(1^{\lambda})$: Set up the public parameter N where $N = p \cdot q$, and $p = 2p' + 1, q = 2q' + 1$ are distinct λ-bit safe primes sampled at random[a]. The plaintext space is $\mathbb{Z}_{N^{\zeta}}$ and the ciphertext space is $\mathbb{Z}^{\times}_{N^{\zeta+1}}$. We define the isomorphism $\mathsf{exp} : \mathbb{Z}_{N^{\zeta}} \to (1 + N\mathbb{Z}_{N^{\zeta+1}})^{\times}$ and its inverse $\mathsf{log} : (1 + N\mathbb{Z}_{N^{\zeta+1}})^{\times} \to \mathbb{Z}_{N^{\zeta}}$ as:

$$\mathsf{exp}(x) = \sum_{j=0}^{\zeta} \frac{(Nx)^j}{j!}, \quad \mathsf{log}(1 + Nx) = \sum_{j=1}^{\zeta} \frac{(-N)^{j-1} x^j}{j}$$

- $\mathsf{KeyGen}(1^{\lambda})$: Returns k where k is sampled uniformly from $[0, \lfloor N/2 \rfloor)$.

- $\mathsf{Enc}_k(m)$: Samples the randomness $g \leftarrow \{a \in \mathbb{Z}^{\times}_{N^{\zeta+1}} \mid (\frac{a}{N}) = 1\}$ and returns $(g, g^k \cdot \mathsf{exp}(m))$, where $(\frac{a}{N})$ is the Jacobi symbol of a modulo N.

- $\mathsf{Dec}_k(g, c)$: Computes and returns $m = \mathsf{log}(c/g^k)$.

[a] As [CS02] points out, standard conjectures and empirical evidence suggest that a random λ-bit integer is a safe prime with probability $\Omega(\lambda^{-2})$.

Figure 3: Linearly Homomorphic Encryption Scheme from DCR

Lemma 3.2 (KDM Security of the DCR-based LHE)**.** *Under the DCR assumption, the LHE scheme in Figure 3 is nonce-based and KDM secure with respect to the class of affine functions $f(x) = a \cdot x + b$ (Definition 3.1), where a, b are constants in $\mathbb{Z}_{N^{\zeta}}$.*

We defer the proof of Lemma 3.2 to Section B.4.

4 Definitions

4.1 Security Definition

We adopt the standard simulation-based security paradigm. A protocol Π securely realizes an ideal functionality $\mathcal{F}$ if for every real-world adversary $\mathcal{A}$, there exists an ideal-world simulator $\mathcal{S}$ such

that the output distributions of the real and ideal executions are computationally indistinguishable. We consider standard static, malicious adversaries in a synchronous network.

Real-World Execution. The protocol Π is executed by parties connected via secure synchronous channels. The adversary $\mathcal{A}$ corrupts a fixed set of parties at the onset, controlling their actions, while honest parties follow Π. The execution output consists of honest parties' inputs/outputs and the adversary's view.

Ideal-World Execution. Parties interact with a trusted functionality $\mathcal{F}$. The simulator $\mathcal{S}$, controlling corrupted parties in the ideal world, interacts with $\mathcal{F}$ (sending inputs and potentially aborting) and simulates the view of $\mathcal{A}$ in the real world. The execution output comprises honest parties' inputs/outputs and the simulated view.

4.2 Computational Model

We model our computation task as an arithmetic circuit $\mathcal{C}$ over the integers $\mathbb{Z}$ with input variables $x_1, \ldots, x_{m_{\text{in}}}$ and gates from the set $\{\texttt{lin}, \texttt{mul}, \texttt{div}_2\}$. The $\texttt{lin}$ gate computes a linear combination $z = \mu \cdot x + \nu \cdot y$ for constant scalars $\mu, \nu \in \mathbb{Z}$. The $\texttt{div}_2$ gate represents a floor-division by two, mapping an integer input v to $\lfloor v/2 \rfloor$. To simplify the maliciously secure protocol, we assume without loss of generality that each input wire to the circuit carries a binary value (since we can always compose the binary values into arithmetic values via $\texttt{lin}$ gates).

$\mathcal{B}$-boundedness. Furthermore, we enforce a boundedness constraint on the circuit. Let $\mathcal{B} \in \mathbb{Z}^+$ be a globally fixed bound parameter. We say that the circuit $\mathcal{C}$ is *$\mathcal{B}$-bounded* if on any valid input vector $\mathbf{x} \in \{0,1\}^{m_{\text{in}}}$, the value carried by every wire w in $\mathcal{C}$ satisfies $|w| \leq \mathcal{B}$. In practice, we can reduce a wide range of computations into this $\mathcal{B}$-bounded model, including:

- *Fixed-point arithmetic.* By representing a fixed-point number with f fractional bits as an integer scaled by 2^f, additions map directly to arithmetic gates. Multiplications are implemented by sequencing a multiplication gate with a truncation gate to discard the extra f fractional bits.
- *Mixed arithmetic and Boolean circuits.* Boolean operations can be implemented via arithmetic addition and multiplication with bound $\mathcal{B} = 1$ (e.g., XOR is computed as $x \oplus y = x - 2xy + y$). Furthermore, values can be converted between arithmetic and Boolean representations using repeated floor-division-by-2 gates (for bit decomposition) and linear combinations (for composition).
- *Modular arithmetic.* Using linear gates and floor-division-by-2 gates, we can implement Barrett reduction [Bar86] to emulate modular arithmetic over an arbitrary modulus by choosing a sufficiently large $\mathcal{B}$.

Client-Server Model. We adopt the *client-server model*, where clients provide inputs to the servers and receive outputs. Note that, if every party plays a single client and a single server, this corresponds to a protocol in the standard MPC model. As prior work observed [GLM+24], a benefit of the client-server model is that we only need to consider the maximal corruption of servers in the security proof.

Functionality $\mathcal{F}_{\text{main}}$
The functionality interacts with a set of clients, and an adversary $\mathcal{S}$ who controls an arbitrary number of corrupted clients. The functionality is parameterized by a $\mathcal{B}$-bounded arithmetic circuit $\mathcal{C}$ together with a fixed association between clients and the circuit's input/output wires.
Input Phase: Receive the value of each input wire x from the associated client. If $x \notin \{0, 1\}$, send `abort` to all parties and abort.
Evaluation Phase: Once all input values are defined, evaluate the circuit $\mathcal{C}$ topologically.

- For each `lin` gate with constants μ, ν and inputs x, y, compute the output $z = \mu \cdot x + \nu \cdot y$.
- For each `mul` gate with inputs x, y, compute the output $z = x \cdot y$.
- For each $\texttt{div}_2$ gate with input value x, compute the output $z = \lfloor x/2 \rfloor$.

Output Phase: Send the value of each output wire y to its associated client.
Abort: Upon receiving `abort` from $\mathcal{S}$, send `abort` to all parties and abort.

Figure 4: Ideal functionality for evaluating a $\mathcal{B}$-bounded arithmetic circuit.

Ideal Functionality. In Figure 4, we define the ideal functionality $\mathcal{F}_{\text{main}}$ for securely computing the circuit $\mathcal{C}$ in the client-server model. $\mathcal{F}_{\text{main}}$ receives inputs from clients, evaluates the circuit $\mathcal{C}$, and sends the output to clients.

5 Protocol Specification

We present our protocol in a modular framework to ensure clarity and summarize the dependencies among various functionalities and protocols in Figure 5. At a high level, the protocol consists of three phases: preprocessing, offline (excluding preprocessing), and online. The preprocessing phase is responsible for generating random authenticated shares that will be used later. The offline phase additionally prepares gate-dependent auxiliary data based on the circuit, while the online phase executes the main computation using the prepared data and the clients' inputs. We defer the details of the preprocessing phase and the remaining offline protocols to Section A.1 and Section A.2, respectively, and focus on the main protocol assuming an ideal offline functionality for now.

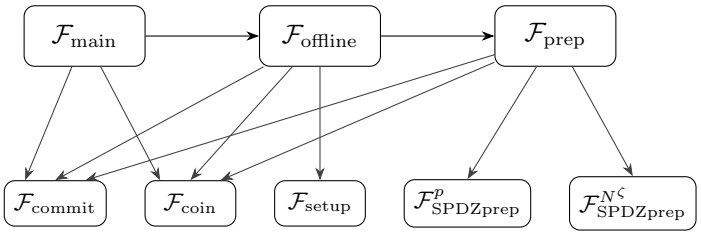

Figure 5: Dependencies between the modular components of our protocol. An arrow $\mathcal{F}_A \to \mathcal{F}_B$ indicates that the protocol for $\mathcal{F}_A$ calls $\mathcal{F}_B$. The functionalities on the second row can be instantiated from general MPC frameworks [DPSZ12, GLM+24, RS22].

5.1 Ideal Functionality of Offline Phase

We first define the functionality $\mathcal{F}_{\text{setup}}$ that generates the public parameters for the protocol.

Functionality $\mathcal{F}_{\text{setup}}$

Upon receiving $(\lambda, \kappa, \mathcal{B}, |\mathcal{C}|)$ from the servers where λ is the computational security parameter, κ is the statistical security parameter, $\mathcal{B}$ is the upper bound on the absolute value of wires, and $|\mathcal{C}|$ is the size of the circuit.

1. Sample random safe primes p, q of bit-length λ and let $N = p \cdot q$.
2. Let $r_{\max} = |\mathcal{C}| \cdot \mathcal{B} \cdot 2^{\kappa+1}$, $r_{\max}^{\text{relax}} = 2 \cdot |\mathcal{C}| \cdot r_{\max} \cdot (n + 2^{\kappa})$, $\Delta_{\max} = N$, $R_{\max} = |\mathcal{C}| \cdot \Delta_{\max} \cdot r_{\max}^{\text{relax}} \cdot 2^{\kappa+1}$, and $\zeta = \lceil \log_N(n \cdot R_{\max}) \rceil$.
3. Output $(N, \zeta, \Delta_{\max}, r_{\max}, r_{\max}^{\text{relax}}, R_{\max})$ to the servers.

Then, we define the ideal functionality $\mathcal{F}_{\text{offline}}$ that handles the offline phase of the main protocol. Most parts of the offline functionality can be instantiated from general MPC frameworks or techniques from prior works [DPSZ12, GLM+24, RS22]. The key challenges include performing efficient range checks on wire masks, which we explained in the roadmap (*Attack 2*), and generating SPDZ shares of random bits modulo N^{ζ}, which we detail in Section A.1.

Functionality $\mathcal{F}_{\text{offline}}$

Init: Let I denote the set of indices of the corrupt servers. Upon receiving $(\texttt{init}, \lambda, \kappa, \mathcal{B}, m_{\text{in}}, m_{\text{out}}, m_{\text{mul}}, m_{\text{div}})$ from all servers:

- Compute $(N, \zeta, \Delta_{\max}, r_{\max}, r_{\max}^{\text{relax}}, R_{\max})$ as described in $\mathcal{F}_{\text{setup}}$, with $|C| = m_{\text{in}} + m_{\text{out}} + m_{\text{mul}} + m_{\text{div}}$.
- Receive corrupted servers' key shares $\{\Delta_i\}_{i \in I}$ from $\mathcal{S}$, sample honest servers' key shares $\Delta_i \leftarrow [0, \Delta_{\max})$ for $i \notin I$, set the global key $\Delta = \sum_{i=1}^{n} \Delta_i$, and check that $\Delta < n \cdot \Delta_{\max}$. Abort otherwise.
- Send (N, ζ) and Δ_i to P_i for $i \in I$.

Sample Generators: Let $\lambda' = \lfloor \log N + \kappa + \log |C| \rfloor$. Sample $m_G = 4 \cdot m_{\text{mul}} + (2 + \lambda') \cdot m_{\text{div}} + m_{\text{out}}$ generators from the set $\{a \mid a \in \mathbb{Z}_{N^{\zeta+1}}^{\times}, (\frac{a}{N}) = 1\}$ where $(\frac{a}{N})$ is the Jacobi symbol of a modulo N. Output the generators to the servers.

Prepare Wire Masks: Prepare shares of masks on the output wire z of each gate as follows, and output the shares to the servers.

- For a multiplication gate, sample shares of a bounded mask $[\![r^{(z)}]\!]$ (following the **Masks** interface of $\mathcal{F}_{\text{prep}}$).
- For an input gate or output gate: sample shares of a bounded mask $[\![r^{(z)}]\!]$. Additionally reveal $r^{(z)}$ to its associated client C.
- For a truncation gate: sample shares of two bounded masks $[\![r^{(z)}]\!]$, $[\![u^{(z)}]\!]$.
- For a linear gate (with input wires x, y and coefficients μ, ν): set $[\![r^{(z)}]\!] = \mu \cdot [\![r^{(x)}]\!] + \nu \cdot [\![r^{(y)}]\!]$.

Prepare Other Intermediate Values: Output shares of the following to the servers.

- For each truncation gate (with output wire z), sample shares of two λ'-dimensional random bit-vector $\vec{\delta}^{(z)}$, $[\![\vec{B}^{(z)}]\!]$, and another random bit $[\![b^{(z)}]\!]$ (following the **Bits** interface of $\mathcal{F}_{\text{prep}}$).
- For each multiplication gate (with input wires x, y), sample shares of $[\![r^{(x)}, r^{(y)}]\!]$ as follows.
  * Receive corrupted servers' shares of $[\![r^{(x)} \cdot r^{(y)}]\!]$ from $\mathcal{S}$.
  * Sample honest servers' shares to sum to $r^{(x)} \cdot r^{(y)}$ and $\Delta \cdot r^{(x)} \cdot r^{(y)}$.
- For each truncation gate (with output wire z), compute shares of $[\![\vec{\delta} \oplus \vec{B}^{(z)}]\!]$ as follows.
  * Receive corrupted servers' shares of $[\![\vec{\delta}^{(z)} \oplus \vec{B}^{(z)}]\!]$ from $\mathcal{S}$.
  * Sample honest servers' shares to sum to $\vec{\delta}^{(z)} \oplus \vec{B}^{(z)}$ and $\Delta \cdot \vec{\delta}^{(z)} \oplus \vec{B}^{(z)}$.
- For each input gate, sample a random triple $[\![\alpha]\!], [\![\beta]\!], [\![\gamma]\!]$ (following the **Triples** interface of $\mathcal{F}_{\text{prep}}$).

Abort: Upon receiving `abort` from $\mathcal{S}$, send Δ to $\mathcal{S}$ and abort.

5.2 Main Protocol

Now, we present the main protocol Π_{main}, which realizes the ideal MPC functionality $\mathcal{F}_{\text{main}}$ (Figure 4) in the $\{\mathcal{F}_{\text{offline}}, \mathcal{F}_{\text{coin}}, \mathcal{F}_{\text{commit}}\}$-hybrid model, assuming the hardness of DCR.

The main protocol first calls $\mathcal{F}_{\text{offline}}$ to perform the offline phase, and then proceeds to the online phase consisting of four sub-protocols: input, garbling, evaluation, and output. We describe each sub-protocol in detail below.

First, in the input protocol Π_{input}, each client sends the masked input to all the servers. The servers check that the input is a binary value and reconstruct the input label to the evaluator.

Protocol Π_{input}

1. **Mask Input:** For each input wire x, the associated client computes $\widetilde{x} = x + r^{(x)}$ and sends $\widetilde{x}$ to all servers, where $r^{(x)}$ has been received and authenticated during the offline phase.
2. **Prepare Range Check:** For each input wire x,
   - Servers locally compute $[\![x]\!] = \widetilde{x} - [\![r^{(x)}]\!]^*$, $[\![y]\!] = [\![x]\!] - 1$.
   - Servers consume a multiplication triple from $\mathcal{F}_{\text{prep}}$ and run $[\![z]\!] \leftarrow \Pi_{\text{mul}}([\![x]\!], [\![y]\!])$.
3. **Batch Verification:** After all input wires are processed
   - Servers check if each $\widetilde{x}$ value they receive from the client is in $[0, n \cdot r_{\max})$. If not, the servers abort the protocol.
   - Servers run $\Pi^{\langle\Delta\rangle}_{\text{SPDZ}-\text{MAC}}$ to authenticate all values opened during the Π_{mul} executions and to verify that all resulting $[\![z]\!]$ values open to 0.
4. **Construct Input Labels:** For every input wire x:

– Each server P_i locally samples its share of a mask $\langle R^{(x)} \rangle_i^* \leftarrow [0, R_{\max})$, and sends its share of the label: $\langle L^{(x)} \rangle_i^* = \Delta_i \cdot \widetilde{x} + \langle R^{(x)} \rangle_i^*$ to P_1.
– P_1 constructs the label $L^{(x)} = \sum_{i \in [n]} \langle L^{(x)} \rangle_i^*$.

Before going into the full garbling and evaluation protocol, we first describe the helper protocols for garbling and evaluating key extension operations, using the LHE scheme in Figure 3, where the random generator g serves as the nonce.

Helper Protocol $\Pi^{\langle \Delta \rangle^*}_{\text{garble}-\text{ext}}$
Global Input: Generators $\vec{g}_{\text{ext}}$ and shares of global key $\langle \Delta \rangle^*$.
Input: Shares of input wire mask $\langle R^{(w)} \rangle^*$, shares of target slope $\langle A \rangle$, shares of target shift $\langle B \rangle$.

1. Let g be the next generator from $\vec{g}_{\text{ext}}$.
2. Each server P_i sends the following ciphertext components to the evaluator P_1:

$$\mathsf{ct}_{i,1} = g^{\Delta_i} \mathsf{exp}(\langle A \rangle_i), \quad \mathsf{ct}_{i,2} = g^{\langle R^{(w)} \rangle_i^*} \mathsf{exp}(\langle B \rangle_i).$$

Helper Protocol $\Pi_{\text{eval}-\text{ext}}$
Global Input: Generators $\vec{g}_{\text{ext}}$.
Input: Label $L^{(w)}$, masked value $\widetilde{w}$.

1. Let g be the next generator from $\vec{g}_{\text{ext}}$, matching the one used in $\Pi_{\text{garble}-\text{ext}}$.
2. Evaluator P_1 aggregates the ciphertexts received from all servers:

$$\mathsf{ct}_1 = \prod_{i=1}^{n} \mathsf{ct}_{i,1}, \quad \mathsf{ct}_2 = \prod_{i=1}^{n} \mathsf{ct}_{i,2}$$

3. Output $\mathsf{DLog}(\mathsf{ct}_2 \cdot (\mathsf{ct}_1)^{\widetilde{w}} \cdot g^{-L^{(w)}})$.

Now, we define the main garbling and evaluation protocols, deferring the subroutines for truncation gates as they are more technically involved.

Protocol Π_{garble}

The servers retrieve the generators $\vec{g}$ from $\mathcal{F}_{\text{offline}}$ and parse it into three vectors $\vec{g}_{\text{ext}} \in (\mathbb{Z}^{\times}_{N\zeta+1})^{4 \cdot m_{\text{mul}} + \lambda' \cdot m_{\text{div}}}$, $\vec{g}_{\text{Bool}} \in (\mathbb{Z}^{\times}_{N\zeta+1})^{2 \cdot m_{\text{div}}}$ and $\vec{g}_{\text{out}} \in (\mathbb{Z}^{\times}_{N\zeta+1})^{m_{\text{out}}}$.

For each wire w that is either an input wire or the output wire of a multiplication or truncation gate, each server P_i locally samples a random mask $\langle R^{(w)} \rangle_i^* \leftarrow [0, R_{\max})$. For each output wire z of a linear gate with input wires x, y and constant scalars μ, ν, servers locally compute $\langle R^{(z)} \rangle^* = \mu \cdot \langle R^{(x)} \rangle^* + \nu \cdot \langle R^{(y)} \rangle^*$.

For each multiplication gate with input wires x, y and output wire z:

1. **Prepare Slopes:** Servers locally compute shares of the slopes for the label extraction:

$$\langle S \rangle = \langle \Delta \cdot r^{(y)} \rangle + \langle R^{(y)} \rangle^*, \quad \langle T \rangle = \langle \Delta \cdot r^{(x)} \rangle$$

2. **Garble Label Extension:**

– Servers sample a random target shift $\langle U \rangle$ locally.

– Servers compute the other target shift

$$\langle V \rangle = -\langle U \rangle - \langle \Delta \cdot r^{(x)} \cdot r^{(y)} \rangle - (\langle \Delta \cdot r^{(z)} \rangle + \langle R^{(z)} \rangle)$$

– Servers execute $\Pi_{\text{garble-ext}}^{\langle \Delta \rangle^*, \vec{g}_{\text{ext}}} \left(\langle R^{(x)} \rangle^*, \langle S \rangle, \langle U \rangle \right)$.

– Servers execute $\Pi_{\text{garble-ext}}^{\langle \Delta \rangle^*, \vec{g}_{\text{ext}}} \left(\langle R^{(y)} \rangle^*, \langle T \rangle, \langle V \rangle \right)$.

3. **Garble Masked Value Extension:**

– Servers sample a random target shift $\langle U' \rangle$ locally.

– Servers compute the other target shift

$$\langle V' \rangle = -\langle U' \rangle - \langle r^{(x)} \cdot r^{(y)} \rangle - \langle r^{(z)} \rangle^*$$

– Servers execute $\Pi_{\text{garble-ext}}^{\langle \Delta \rangle^*, \vec{g}_{\text{ext}}} \left(\langle R^{(x)} \rangle^*, \langle r^{(y)} \rangle^*, \langle U' \rangle \right)$.

– Servers execute $\Pi_{\text{garble-ext}}^{\langle \Delta \rangle^*, \vec{g}_{\text{ext}}} \left(\langle R^{(y)} \rangle^*, \langle r^{(x)} \rangle^*, \langle V' \rangle \right)$.

For each truncation gate, run $\Pi_{\text{garble-div}}$.

Protocol Π_{eval}

Evaluator P_1 processes gates in topological order.

– **Linear Gate:** For each linear gate $z = \mu \cdot x + \nu \cdot y$, P_1 computes the label $L^{(z)} = \mu \cdot L^{(x)} + \nu \cdot L^{(y)}$, and masked value $\widetilde{z} = \mu \cdot \widetilde{x} + \nu \cdot \widetilde{y}$.

– **Multiplication Gate:** For each multiplication gate $z = x \cdot y$, P_1 computes the correction terms

$$\xi_x \leftarrow \Pi_{\text{eval-ext}}^{\vec{g}_{\text{ext}}} \left(L^{(x)}, \widetilde{x} \right), \quad \xi_y \leftarrow \Pi_{\text{eval-ext}}^{\vec{g}_{\text{ext}}} \left(L^{(y)}, \widetilde{y} \right)$$
$$\xi'_x \leftarrow \Pi_{\text{eval-ext}}^{\vec{g}_{\text{ext}}} \left(L^{(x)}, \widetilde{x} \right), \quad \xi'_y \leftarrow \Pi_{\text{eval-ext}}^{\vec{g}_{\text{ext}}} \left(L^{(y)}, \widetilde{y} \right)$$

and sets the label $L^{(z)}$ and masked value $\widetilde{z}$ as

$$L^{(z)} = \widetilde{x} \cdot L^{(y)} - \xi_x - \xi_y, \quad \widetilde{z} = \widetilde{x} \cdot \widetilde{y} - \xi'_x - \xi'_y.$$

– **Truncation Gate:** For each truncation gate $z = \lfloor x/2 \rfloor$, run $\Pi_{\text{eval-div}}$ to compute $L^{(z)}$ and $\widetilde{z}$.

Now, we describe the garbling and evaluation protocol for truncation gates, following the high-level idea in the roadmap. A technicality here is that our current offline phase functionality does not provide the parties with a share of the least significant bit of the input mask $r^{(x)}$, which is needed for point-and-permuting the Boolean garbled truth table. Therefore, the parties need to perform a mask switching to use another mask $r'^{(x)}$ whose least significant bit has been authentically secret-shared.

Protocol $\Pi_{\text{garble}-\text{div}}$

1. **Intermediate Masks:** Each server P_i samples a share of a mask $\langle R^{(x_0)}\rangle_i^* \leftarrow [0, R_{\max})$, and computes the following intermediate masks:

$$[\![r^{(x_0)}]\!] = 2 \cdot [\![u^{(z)}]\!] + [\![b^{(z)}]\!], \quad [\![r'^{(x)}]\!] = [\![r^{(x_0)}]\!] + 2 \cdot [\![r^{(z)}]\!],$$
$$\langle R'^{(x)}\rangle^* = \langle R^{(x_0)}\rangle^* + 2 \cdot \langle R^{(z)}\rangle^*.$$

2. **Garble Arithmetic Label Extensions:**

   – Servers send the following target shifts to P_1.

$$\langle V\rangle = \langle R'^{(x)}\rangle - \langle R^{(x)}\rangle + \langle \Delta \cdot r'^{(x)}\rangle - \langle \Delta \cdot r^{(x)}\rangle, \quad \langle v\rangle = \langle r'^{(x)}\rangle - \langle r^{(x)}\rangle.$$

   – Each server P_i samples a share of a λ'-dimensional mask $\langle \vec{U}^{(z)}\rangle_i^* \leftarrow [0, R_{\max})^{\lambda'}$, and computes a target shift vector $\langle \vec{K}^{(z)}\rangle = 2 \cdot \langle \vec{U}^{(z)}\rangle + \langle \vec{B}^{(z)}\rangle$. For each $j \in [\lambda']$, servers execute $\Pi_{\text{garble}-\text{ext}}^{\langle\Delta\rangle^*, \vec{g}_{\text{ext}}}(\langle R'^{(x)}\rangle^*, \langle \vec{\delta}^{(z)}[j]\rangle, \langle \vec{K}^{(z)}[j]\rangle)$.

3. **Garble Boolean Label Extensions:**

   – Servers execute $\langle \vec{B}^{(z)}\rangle^* \leftarrow \Pi_{\text{ModToInt}}^{N^\zeta, t}(\langle \vec{B}^{(z)}\rangle)$ and $\langle \vec{\delta}^{(z)} \oplus \vec{B}^{(z)}\rangle^* \leftarrow \Pi_{\text{ModToInt}}^{N^\zeta, t}(\langle \vec{\delta}^{(z)} \oplus \vec{B}^{(z)}\rangle)$, with $t = \kappa + 3$, and then locally combine the obtained integer share vectors as

$$\langle \mathsf{sk}_0\rangle^* = \sum_k 2^k \cdot \langle \vec{B}^{(z)}[k]\rangle^*, \quad \langle \mathsf{sk}_1\rangle^* = \sum_k 2^k \cdot \langle (\vec{\delta}^{(z)} \oplus \vec{B}^{(z)})[k]\rangle^*.$$

   – Servers define intermediate label shares $\langle L_0\rangle, \langle L_1\rangle$ and masked value shares $\langle x_0\rangle$, $\langle x_1\rangle$ as follows:

$$\langle L_0^{(x_0)}\rangle = \langle R^{(x_0)}\rangle + \langle \Delta \cdot r^{(x_0)}\rangle + \langle \Delta \cdot b^{(z)}\rangle,$$
$$\langle L_1^{(x_0)}\rangle = \langle R^{(x_0)}\rangle + \langle \Delta \cdot r^{(x_0)}\rangle + \langle \Delta\rangle - \langle \Delta \cdot b^{(z)}\rangle,$$
$$\langle \widetilde{x_{0,0}}\rangle = \langle r^{(x_0)}\rangle + \langle b^{(z)}\rangle, \quad \langle \widetilde{x_{0,1}}\rangle = \langle r^{(x_0)}\rangle + \langle 1\rangle - \langle b^{(z)}\rangle.$$

   – Let $g, g' \in \mathbb{Z}_{N^{\zeta+1}}^\times$ be the next two group elements in $\vec{g}_{\text{Bool}}$. Each server P_i runs for $\mathsf{idx} \in \{0, 1\}$:

$$\vec{\mathsf{ct}}_{\mathsf{idx},i}^{(z)}[0] \leftarrow g^{\langle \mathsf{sk}_{\mathsf{idx}}\rangle_i^*} \cdot \mathsf{exp}(\langle L_{\mathsf{idx}}^{(x_0)}\rangle_i), \quad \vec{\mathsf{ct}}_{\mathsf{idx},i}^{(z)}[1] \leftarrow g'^{\langle \mathsf{sk}_{\mathsf{idx}}\rangle_i^*} \cdot \mathsf{exp}(\langle \widetilde{x_{0,\mathsf{idx}}}\rangle_i).$$

   and sends the ciphertexts $\{\vec{\mathsf{ct}}_{\mathsf{idx},i}^{(z)}\}_{\mathsf{idx}\in\{0,1\}}$ to the evaluator P_1.

Protocol $\Pi_{\text{eval}-\text{div}}$ For each $\mathtt{div}_2$ gate with input wire x and output wire z, the evaluator, P_1, holds a label $L^{(x)}$ and a masked value $\tilde{x}$.

1. **Evaluate Arithmetic Label Extensions:**

   – P_1 receives target shifts $\{\langle V\rangle_i\}$, $\{\langle v\rangle_i\}$ from all servers, reconstructs V, v, and com-

putes

$$L'^{(x)} \leftarrow L^{(x)} + V, \quad \tilde{x}' \leftarrow \tilde{x} + v,$$

– P_1 runs $\vec{L}[j] \leftarrow \Pi^{\vec{g}_{\text{ext}}}_{\text{eval}-\text{ext}}(L'^{(x)}, \tilde{x}')$ iteratively for each $j \in [\lambda']$.

Note that in an honest execution, we should have $\tilde{x}' = x + r'^{(x)}$, $L'^{(x)} = \Delta \cdot \tilde{x}' + R'^{(x)}$, and $\vec{L} = \vec{\delta}^{(z)} \cdot \tilde{x}' + \vec{K}^{(z)}$.

2. **Evaluate Boolean Label Extensions:** P_1 computes a decryption key sk and an index $\mathsf{idx} \in \{0, 1\}$ as follows

$$\mathsf{sk}^{(z)} = \mathsf{Int}(\vec{L} \bmod 2), \quad \mathsf{idx} = \tilde{x}' \bmod 2, \text{ where } \mathsf{Int}(\vec{x}) := \sum_i 2^i \cdot \vec{x}[i].$$

P_1 then decrypts the ciphertexts $\{\vec{\mathsf{ct}}^{(z)}_{\mathsf{idx},i}\}$ collected from every garbler P_i using sk:

$$\langle L^{(x_0)} \rangle \leftarrow \mathsf{DLog}((\prod_i \vec{\mathsf{ct}}^{(z)}_{\mathsf{idx},i}[0]) \cdot g_{\text{Bool}}^{-\mathsf{sk}}), \quad \langle \widetilde{x_0} \rangle \leftarrow \mathsf{DLog}((\prod_i \vec{\mathsf{ct}}^{(z)}_{\mathsf{idx},i}[1]) \cdot g'^{-\mathsf{sk}}_{\text{Bool}}).$$

where $g, g' \in \mathbb{Z}^{\times}_{N^{\zeta+1}}$ are the next two group elements in $\vec{g}_{\text{Bool}}$, matching those used in $\Pi_{\text{garble}-\text{div}}$. P_1 reconstructs the shares to obtain a label $L^{(x_0)}$ and a masked value $\widetilde{x_0}$.

Note that in an honest execution, we should have $\widetilde{x_0} = (x \bmod 2) + r^{(x_0)}$, and $L^{(x_0)} = \Delta \cdot \widetilde{x_0} + R^{(x_0)}$.

3. **Compute Output Arithmetic Label:** P_1 computes the output label $L^{(z)}$ and masked value $\tilde{z}$ as follows.

$$L^{(z)} = \lfloor (L'^{(x)} - L^{(x_0)})/2 \rfloor, \quad \tilde{z} = \lfloor (\tilde{x}' - \widetilde{x_0})/2 \rfloor.$$

Note that in an honest execution, we should have $\tilde{z} = \lfloor x/2 \rfloor + r^{(z)}$, and $L^{(z)} = \Delta \cdot \tilde{z} + R^{(z)}$.

Finally, we describe our output protocol, which follows from the authenticated decoding idea in the roadmap (Section 2.2).

Protocol Π_{output}

For every output wire y with an associated client C, let g be the next generator in $\vec{g}_{\text{out}}$ and proceed as follows.

1. **Decoding Information:** Every server P_i computes $\mathsf{ct}_{i,1} = g^{\Delta_i}$ and $\mathsf{ct}_{i,2} = g^{\langle R^{(y)} \rangle^*_i}$, and sends them to the client C.

2. **Evaluation Results:** P_1 sends $\mathsf{ct} = g^{L^{(y)}}$ and $\tilde{y}$ to C as the evaluation result.

3. **Client Output:** The client C aborts if $\tilde{y} \notin [0, n \cdot r_{\max})$ or if $\mathsf{ct} \neq (\prod_i \mathsf{ct}_{i,1})^{\tilde{y}} \cdot (\prod_i \mathsf{ct}_{i,2})$. Otherwise, output $\tilde{y} - r^{(y)}$, where $r^{(y)}$ is obtained from $\mathcal{F}_{\text{offline}}$.

Blueprint of security proof. We defer the security proofs to Section B and list the key ideas here.

First, we show that it suffices to consider the maximal corruption case where the adversary corrupts $n - 1$ servers, including the evaluator P_1, since the servers hold neither input nor output. To rule out a trivial simulator that always aborts, we also need to show that the protocol does not

abort (except with negligible probability) when the adversary behaves passively. The key challenge then becomes simulating the ciphertexts sent by the honest server P_h.

To this end, we leverage the fact that P_h's shares of $r^{(w)}$ and $R^{(w)}$ are significantly larger than the wire value w and the term $\Delta_h \cdot \widetilde{w}$, given that both w and $r^{(w)}$ are bounded (due to the $\mathcal{B}$-boundedness of the circuit and our range checks). Therefore, the simulator can instead sample the honest shares $\langle \widetilde{w} \rangle_h^* = w + \langle r^{(w)} \rangle_h^*$ and $\langle L^{(w)} \rangle_h^* = \Delta_h \cdot \widetilde{w} + \langle R^{(w)} \rangle_h^*$ from $[0, r_{\max})$ and $[0, R_{\max})$, and compute $\langle r^{(w)} \rangle_h^*$ and $\langle R^{(w)} \rangle_h^*$ by subtracting w and $\Delta_h \cdot \widetilde{w}$, respectively. These terms are used for computing encrypted messages in the honest party's first ciphertext $\mathsf{ct}_{h,1}$ in key extension instances. The output distribution of the execution remains statistically close to the real execution.

Next, we can show that the honest party's second ciphertext $\mathsf{ct}_{h,2}$ in all key-extension instances can be expressed using the first ciphertext $\mathsf{ct}_{h,1}$, the honest shares $\langle \widetilde{w} \rangle_h^*$ and $\langle L^{(w)} \rangle_h^*$, and the corrupt servers' shares from the offline phase.

After replacing the second ciphertexts, we observe that the random generator g is never reused across different ciphertexts. Therefore, we can replace all the real first ciphertexts with ideal ciphertexts of random messages using the KDM security of the encryption scheme (Definition 3.1). After the replacement, the simulator no longer needs to compute $\langle r^{(w)} \rangle_h^*$, $\langle R^{(w)} \rangle_h^*$ or know the value w of the wire. Note that if the circuit does not contain any truncation gates, then all encrypted messages are affine functions of the key Δ, and we can apply Lemma 3.2 to reduce the security assumption to DCR. In Section C, we describe how to remove the KDM assumption for circuits with truncation gates via level-wise key-switching.

Finally, since we use the same Δ as both the MAC key in the SPDZ protocol and the key for LHE, we must ensure consistency across both parts in the simulation, and the adversary should never see Δ if it also receives any ciphertexts. Fortunately, in the SPDZ protocol, the simulator only needs to query Δ when the MAC-check aborts, and all MAC checks in our protocol occur before the honest party sends any ciphertext. Therefore, if the simulation has not aborted by the time the honest party sends its first ciphertext, the simulator can simply sample the honest party's Δ_h following the real protocol and compute all honest ciphertexts accordingly.

6 Evaluation of Concrete Costs

In this section, we evaluate the concrete costs of our protocol for matrix-vector multiplication, which is fundamental to numerous applications, including LLM inference, Markov chain analysis, PageRank [PBMW99], Kalman filtering [Kal60], Principal Component Analysis, and various other spectral and linear-algebraic methods in machine learning and data analysis.

Baselines. We compare our protocol against two recent maliciously-secure multi-party garbling protocols: [GLM+24] and [HKN+25]. Both achieve constant-round MPC with bandwidth scaling linearly with the number of parties. Specifically, [GLM+24] provides $t = n - 1$ security under the Decisional Diffie-Hellman (DDH) assumption, while [HKN+25] secures against a constant fraction of corrupted parties assuming oblivious transfer (OT) in the random oracle model (we set their threshold to $t = 2n/3$ following their evaluation setup). As both schemes support only Boolean circuits, we convert our matrix-vector multiplication task using standard long multiplication and addition. For [HKN+25], which supports free XOR gates, we optimize the circuit to minimize non-XOR gates (e.g., using full-adders with a single AND gate).

Benchmark setting. We focus on the secure multiplication of a $D \times D$ matrix $\mathbf{A}$ and a vector $\vec{x}$, with integer entries ranging in precision from 1 to 128 bits. To ensure a fair comparison with

the Boolean baselines, we additionally perform a full bit-decomposition over output of the matrix multiplication. For both our protocol and the baselines, we exclude costs that are sublinear in the size of the full circuit, as they are effectively amortized in large-scale computations. Finally, we assume our computation overhead is dominated by the modular exponentiation operations[7], and we give advantage to the baselines by omitting their computation time.

Parameter choices and concrete optimizations. We target a 128-bit computational security level, setting the RSA modulus N to 3072 bits. The gate-level statistical security parameter is set to 80 (i.e., $\kappa + \log \mathcal{C} = 80$). For optimization, we adopt the small-exponent assumption [ADOS22, BLLL23], reducing the global key Δ_i to 256 bits per party, which maintains 128-bit security against the best-known baby-step-giant-step attack [Sha71]. Under this assumption, we set $\zeta = 1$ to minimize computation overhead. For truncation gates, we use a global key $\vec{\delta}$ as in the Free-XOR line of work [KS08, CKKZ12].

Furthermore, we can generate the public randomness $\vec{g}$ for the LHE scheme using an expander function modeled as a random oracle, such as AES in Counter (CTR) mode. Consequently, the servers only need to agree on a single random seed during the offline phase.

Finally, we use a packing optimization that exploits the gap between our bound $\mathcal{B}$ and the precision of the input entries. The idea is to pack several rows of $\mathbf{A}$ into a single row of a compressed matrix $\mathbf{A}'$, using disjoint bit ranges for the corresponding products. We then compute $\vec{y}' = \mathbf{A}' \cdot \vec{x}$ and recover the desired entries of $\vec{y} = \mathbf{A} \cdot \vec{x}$ by unpacking $\vec{y}'$ with $\mathtt{div}_2$ gates. Concretely, let ℓ_{out} denote the maximum bit-length of an output entry[8], and set the packing factor to $\eta = \lfloor \frac{\log B}{\ell_{\text{out}}} \rfloor$. We define each compressed row by $\mathbf{A}'_{i,j} = \sum_{k=0}^{\eta-1} \mathbf{A}_{i\cdot\eta+k,j} \cdot 2^{k\cdot\ell_{\text{out}}}$. The packing trick reduces the number of multiplication gates by a factor of η, but does not change the number of $\mathtt{div}_2$ gates needed.

When simulating the computation costs, we adopt a recent open-source implementation of modular exponentiation [LD25], which is built on the residual number system (RNS) [PP95] and utilizes vectorized instruction sets on modern CPUs. As a micro-benchmark, it takes 1.749 microseconds to perform a 6144-bit modular multiplication on a single CPU core of an AWS c7i.metal instance.

Evaluation results. Figure 6 summarizes the total communication bandwidth and end-to-end runtime of our protocol compared to prior works. While our protocol natively supports arithmetic computation over high-precision integers, it still outperforms prior Boolean garbling schemes when emulating low-precision integer arithmetic. For instance, for matrix dimension $D = 4096$, 16-bit integer precision, and 50 Gbps network bandwidth between each server and P_1, our protocol achieves a 101× and 247× reduction in communication and 4.41× and 10.73× speedup in end-to-end runtime compared to [GLM+24] and [HKN+25], respectively, assuming the end-to-end time of prior works is completely dominated by the communication overhead. Our speedups also grow with larger matrix sizes, as they help balance the overheads from our multiplication gates and $\mathtt{div}_2$ gates. Overall, our protocol demonstrates significant efficiency improvements over prior constant-round MPC protocols across various parameter settings.

Comparison with linear-round MPC. SPDZ-style protocols [Bea91, DPSZ12] remain extremely competitive for arithmetic computation, and we do not claim that our current implementation outperforms them in end-to-end runtime for the benchmark evaluated above under common

[7] As a sanity check, we benchmarked the open-source implementation of [GGS26], which garbles bounded-integer multiplication (for two parties) with similar DCR-based techniques. For them, modular exponentiation accounts for about 95% of the evaluation time.

[8] More concretely, $\ell_{\text{out}} = 2 \cdot \ell_{\text{in}} + \log D$ where ℓ_{in} is the input bit-length.

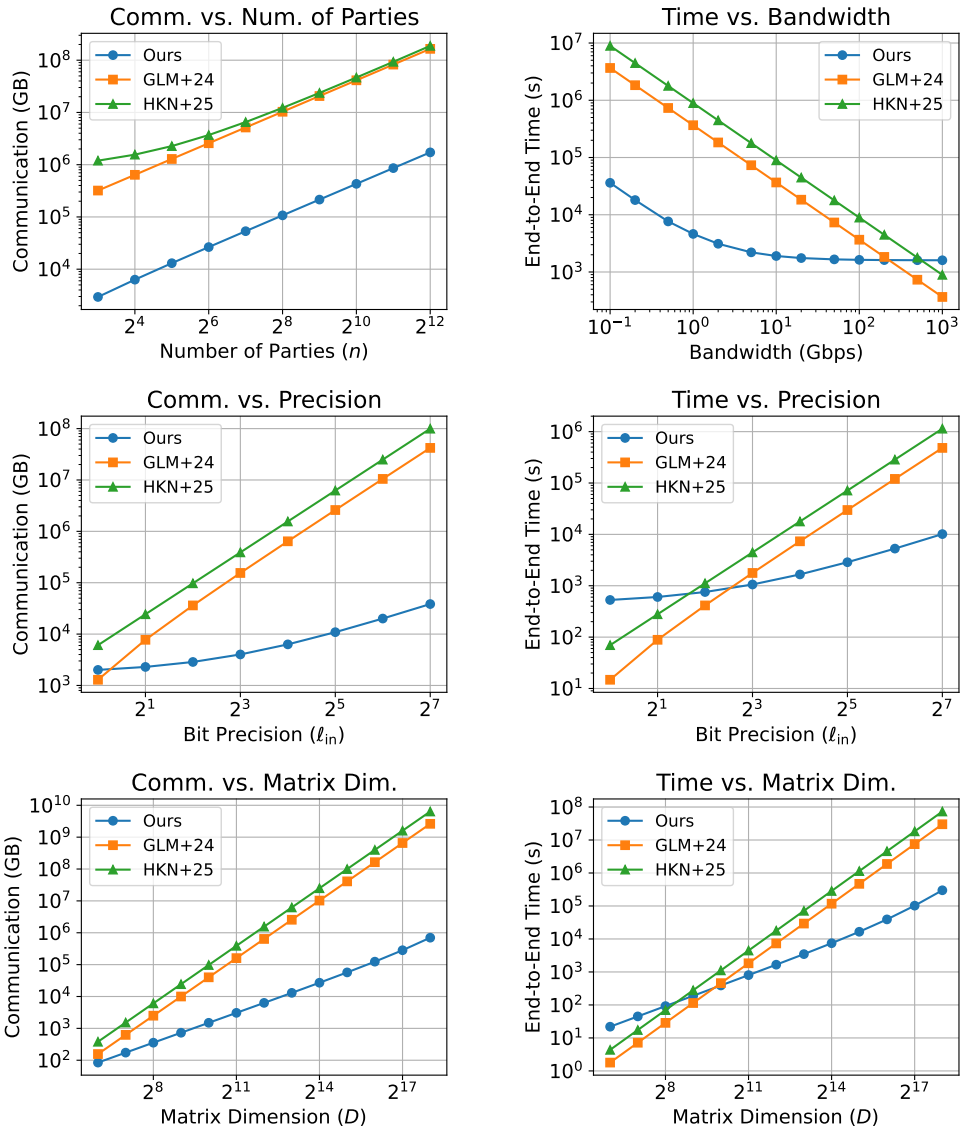

Figure 6: Amortized communication cost (left column) and end-to-end runtime (right column) of our protocol versus prior works for matrix-vector multiplication. By default, we set the number of parties $n = 16$, input precision $\ell_{\text{in}} = 16$, matrix dimension $D = 4096$, and assume a 50 Gbps network between each garbler and P_1. We assume each garbler runs in parallel on a separate AWS c7i.metal-48xl instance (192 vCPUs) and streams the garbled circuit to P_1 for immediate evaluation. Hence, the runtime is insensitive to the number of servers. Network latency is omitted due to the constant-round nature of all protocols. We model prior works as communication-bound only and omit their computation time; for our protocol, only modular exponentiation in the online phase is considered, as other computations are significantly faster in comparison.

network conditions.

Beyond theoretical interest, one main advantage of our approach lies in the online phase. As is standard for garbled circuits, we naturally support an offline-online execution: the parties may mask the garbled circuit during the offline phase with a (programmable) random oracle and unmask it online by revealing short seeds [BHR12a]. Consequently, our online communication can be made independent of the circuit size, depending only on the number of input and output wires.

7 Open Problems

Our work suggests several interesting directions for future research. We highlight a few below.

MPC from other succinct garbling schemes. Our construction builds on the constant-rate arithmetic garbling scheme of Ball et al. [BLLL23]. It remains open whether other succinct garbling schemes can also be transformed into maliciously secure, constant-round MPC protocols while preserving their concrete efficiency in the two-party setting. A key challenge is that these schemes typically rely on Fully Homomorphic Encryption (FHE) [Gen09, LWYY25b, LWYY25a, ZLY+26] or Homomorphic Secret Sharing (HSS) [BGI16, LLL26, MORS24, MORS25, ILL25], whose maliciously secure multi-party counterparts are not yet as well developed.

Faster truncation. In our protocol, truncating an arithmetic value by k bits requires k $\texttt{div}_2$ gates, each of which incurs $O(\lambda^2)$ communication per party. This makes truncation a bottleneck in applications such as fixed-point arithmetic. A natural question is whether truncation can be made more efficient in our setting. In the two-party case, several works reduce the communication cost of truncation by a factor of λ [LL24, GGS26, GN25]. However, these protocols still incur high computation costs, and their techniques rely on HSS, leaving open whether they can be extended efficiently to the multi-party setting.

Generalization to RAM. Another promising direction is to move beyond the circuit model by combining our techniques with recent work on practical Garbled RAM [HKO23, GTRS25, GGS26, LLLH26]. Since oblivious RAMs often have high evaluation depth, such an approach could yield RAM-model MPC with better concrete performance than ORAM-over-SPDZ [DPSZ12, Kel20], analogous to the recent two-party improvements of [GGS26].

8 Acknowledgements

This work is in part support by NSF under the award number 2212746, an Amazon Research Award, a Packard Fellowship, an ONR grant, a JP Morgan Faculty Research Award, a research grant from 0xPARC, and a seed grant from SBI. Tianyao's work is also supported by a Qualcomm Innovation Fellowship.

We thank Kaiwen He for providing benchmark results for modular exponentiation with vectorized instruction-set acceleration on AWS EC2 instances.

Use of AI. During the preparation of this work, the authors used AI-assisted technologies (specifically, Gemini 3.0 pro and ChatGPT 5.2) for language polishing, diagram plotting, and style formatting. The authors reviewed and edited the content as needed and take full responsibility for the content of the work.

References

[ACS02] Joy Algesheimer, Jan Camenisch, and Victor Shoup. Efficient computation modulo a shared secret with application to the generation of shared safe-prime products. pages 417–432, 2002. doi:10.1007/3-540-45708-9_27.

[ADOS22] Damiano Abram, Ivan Damgård, Claudio Orlandi, and Peter Scholl. An algebraic framework for silent preprocessing with trustless setup and active security. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022*, pages 421–452, Cham, 2022. Springer Nature Switzerland.

[AIK11] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. How to garble arithmetic circuits. In *Proceedings of the 2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, FOCS '11, page 120–129, USA, 2011. IEEE Computer Society. doi:10.1109/FOCS.2011.40.

[Bar86] Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In Andrew M. Odlyzko, editor, *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, Lecture Notes in Computer Science, pages 311–323. Springer, 1986. doi:10.1007/3-540-47721-7_24.

[BCG+20] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators from ring-lpn. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part II*, Lecture Notes in Computer Science, pages 387–416. Springer, 2020. doi:10.1007/978-3-030-56880-1_14.

[BCO+21] Aner Ben-Efraim, Kelong Cong, Eran Omri, Emmanuela Orsini, Nigel P. Smart, and Eduardo Soria-Vazquez. Large scale, actively secure computation from LPN and free-xor garbled circuits. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part III*, volume 12698 of *Lecture Notes in Computer Science*, pages 33–63. Springer, 2021. doi:10.1007/978-3-030-77883-5_2.

[BDF+23] Jakob Burkhardt, Ivan Damgård, Tore Kasper Frederiksen, Satrajit Ghosh, and Claudio Orlandi. Improved distributed RSA key generation using the miller-rabin test. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, pages 2501–2515. ACM, 2023. doi:10.1145/3576915.3623163.

[Bea91] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings*, Lecture Notes in Computer Science, pages 420–432. Springer, 1991. doi:10.1007/3-540-46766-1_34.

[BELO16] Aner Ben-Efraim, Yehuda Lindell, and Eran Omri. Optimizing semi-honest secure multiparty computation for the internet. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 578–590, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2976749.2978347.

[Ben18] Aner Ben-Efraim. On multiparty garbling of arithmetic circuits. In Thomas Peyrin and Steven D. Galbraith, editors, *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part III*, Lecture Notes in Computer Science, pages 3–33. Springer, 2018. doi:10.1007/978-3-030-03332-3_1.

[BFKL94] Avrim Blum, Merrick Furst, Michael Kearns, and Richard J. Lipton. Cryptographic primitives based on hard learning problems. In Douglas R. Stinson, editor, *Advances in Cryptology — CRYPTO' 93*, pages 278–291, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[BGH+23] Gabrielle Beck, Aarushi Goel, Aditya Hegde, Abhishek Jain, Zhengzhong Jin, and Gabriel Kaptchuk. Scalable multiparty garbling. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, pages 2158–2172. ACM, 2023. doi:10.1145/3576915.3623132.

[BGI16] Elette Boyle, Niv Gilboa, and Yuval Ishai. Breaking the circuit size barrier for secure computation under ddh. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology – CRYPTO 2016*, pages 509–539, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[BGV14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Trans. Comput. Theory*, 6(3):13:1–13:36, 2014. doi:10.1145/2633600.

[BHR12a] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Adaptively secure garbling with applications to one-time programs and secure outsourcing. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings*, Lecture Notes in Computer Science, pages 134–153. Springer, 2012. doi:10.1007/978-3-642-34961-4_10.

[BHR12b] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 784–796, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2382196.2382279.

[BJKL21] Fabrice Benhamouda, Aayush Jain, Ilan Komargodski, and Huijia Lin. Multiparty reusable non-interactive secure computation from LWE. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part II*,

Lecture Notes in Computer Science, pages 724–753. Springer, 2021. doi:10.1007/978-3-030-77886-6_25.

[BLLL23] Marshall Ball, Hanjun Li, Huijia Lin, and Tianren Liu. New ways to garble arithmetic circuits. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology - EUROCRYPT 2023 - 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part II*, Lecture Notes in Computer Science, pages 3–34. Springer, 2023. doi:10.1007/978-3-031-30617-4_1.

[BLO17] Aner Ben-Efraim, Yehuda Lindell, and Eran Omri. Efficient scalable constant-round MPC via garbled circuits. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part II*, Lecture Notes in Computer Science, pages 471–498. Springer, 2017. doi:10.1007/978-3-319-70697-9_17.

[BMO+26] Amik Raj Behera, Pierre Meyer, Claudio Orlandi, Lawrence Roy, and Peter Scholl. Privately constrained prfs from dcr: Puncturing and bounded waring rank. In Benny Applebaum and Huijia (Rachel) Lin, editors, *Theory of Cryptography*, pages 301–332, Cham, 2026. Springer Nature Switzerland.

[BMR90] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing*, STOC '90, page 503–513, New York, NY, USA, 1990. Association for Computing Machinery. doi:10.1145/100216.100287.

[BOGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, page 1–10, New York, NY, USA, 1988. Association for Computing Machinery. doi:10.1145/62212.62213.

[CHHK25] Geoffroy Couteau, Carmit Hazay, Aditya Hegde, and Naman Kumar. Breaking the $1/\lambda$-rate barrier for arithmetic garbling. In *Advances in Cryptology – EUROCRYPT 2025: 44th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Madrid, Spain, May 4–8, 2025, Proceedings, Part VI*, page 182–213, Berlin, Heidelberg, 2025. Springer-Verlag. doi:10.1007/978-3-031-91095-1_7.

[CKKZ12] Seung Geol Choi, Jonathan Katz, Ranjit Kumaresan, and Hong-Sheng Zhou. On the security of the "free-xor" technique. In Ronald Cramer, editor, *Theory of Cryptography*, pages 39–53, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[CS02] Ronald Cramer and Victor Shoup. Universal hash proofs and a paradigm for adaptive chosen ciphertext secure public-key encryption. In *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 - May 2, 2002, Proceedings*, volume 2332 of *Lecture Notes in Computer Science*, pages 45–64. Springer, 2002. doi:10.1007/3-540-46035-7_4.

[DFK+06] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. pages 285–304, 2006. doi:10.1007/11681878_15.

[DJN10] Ivan Damgård, Mads Jurik, and Jesper Buus Nielsen. A generalization of paillier's public-key system with applications to electronic voting. *Int. J. Inf. Secur.*, 9(6):371–385, December 2010. doi:10.1007/s10207-010-0119-9.

[DPSZ12] 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 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, Lecture Notes in Computer Science, pages 643–662. Springer, 2012. doi:10.1007/978-3-642-32009-5_38.

[EGP+23] Daniel Escudero, Vipul Goyal, Antigoni Polychroniadou, Yifan Song, and Chenkai Weng. Superpack: Dishonest majority MPC with constant online communication. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology - EUROCRYPT 2023 - 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part II*, Lecture Notes in Computer Science, pages 220–250. Springer, 2023. doi:10.1007/978-3-031-30617-4_8.

[FS89] Uriel Feige and Adi Shamir. Zero knowledge proofs of knowledge in two rounds. In Gilles Brassard, editor, *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, Lecture Notes in Computer Science, pages 526–544. Springer, 1989. doi:10.1007/0-387-34805-0_46.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, pages 169–178. ACM, 2009. doi:10.1145/1536414.1536440.

[GGH+13] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 40–49, 2013. doi:10.1109/FOCS.2013.13.

[GGS26] Tianyao Gu, Ashrujit Ghoshal, and Elaine Shi. Zebra: Arithmetic garbled RAM for large words from DCR. In Joan Daemen and Emmanuel Thomé, editors, *Advances in Cryptology - EUROCRYPT 2026 - 45th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Rome, Italy, May 10-14, 2026, Proceedings, Part III*, Lecture Notes in Computer Science, pages 276–306. Springer, 2026. doi:10.1007/978-3-032-25324-8_10.

[GLM+24] Vipul Goyal, Junru Li, Ankit Kumar Misra, Rafail Ostrovsky, Yifan Song, and Chenkai Weng. Dishonest majority constant-round MPC with linear communication from DDH. In Kai-Min Chung and Yu Sasaki, editors, *Advances in Cryptology - ASIACRYPT 2024 - 30th International Conference on the Theory and Application of*

Cryptology and Information Security, Kolkata, India, December 9-13, 2024, Proceedings, Part VI, Lecture Notes in Computer Science, pages 167–199. Springer, 2024. doi:10.1007/978-981-96-0938-3_6.

[GMW87] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, page 218–229, New York, NY, USA, 1987. Association for Computing Machinery. doi:10.1145/28395.28420.

[GN25] Jian Guo and Wenjie Nan. Efficient mixed garbling from homomorphic secret sharing and ggm-tree. In Serge Fehr and Pierre-Alain Fouque, editors, *Advances in Cryptology – EUROCRYPT 2025*, pages 152–181, Cham, 2025. Springer Nature Switzerland.

[GTRS25] Tianyao Gu, Afonso Tinoco, Sri Harish G. Rajan, and Elaine Shi. Picogram: Practical garbled ram from decisional diffie-hellman. In Yael Tauman Kalai and Seny F. Kamara, editors, *Advances in Cryptology – CRYPTO 2025*, pages 247–281, Cham, 2025. Springer Nature Switzerland.

[Hea24] David Heath. Efficient arithmetic in garbled circuits. In *Advances in Cryptology - EUROCRYPT 2024: 43rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zurich, Switzerland, May 26-30, 2024, Proceedings, Part V*, pages 3–31, Berlin, Heidelberg, 2024. Springer-Verlag. doi:10.1007/978-3-031-58740-5_1.

[HKN+25] David Heath, Vladimir Kolesnikov, Varun Narayanan, Rafail Ostrovsky, and Akash Shah. Multiparty garbling from ot with linear scaling and ram support. In *Advances in Cryptology – CRYPTO 2025: 45th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17–21, 2025, Proceedings, Part IV*, page 556–588, Berlin, Heidelberg, 2025. Springer-Verlag. doi:10.1007/978-3-032-01884-7_18.

[HKO23] David Heath, Vladimir Kolesnikov, and Rafail Ostrovsky. Tri-state circuits: A circuit model that captures ram. In *Advances in Cryptology - CRYPTO 2023: 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part IV*, pages 128–160, Berlin, Heidelberg, 2023. Springer-Verlag. doi:10.1007/978-3-031-38551-3_5.

[HOSS18a] Carmit Hazay, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez. Concretely efficient large-scale MPC with active security (or, tinykeys for tinyot). In Thomas Peyrin and Steven D. Galbraith, editors, *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part III*, volume 11274 of *Lecture Notes in Computer Science*, pages 86–117. Springer, 2018. doi:10.1007/978-3-030-03332-3_4.

[HOSS18b] Carmit Hazay, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez. Tinykeys: A new approach to efficient multi-party computation. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part III*, volume 10993 of *Lecture Notes in Computer Science*, pages 3–33. Springer, 2018. doi:10.1007/978-3-319-96878-0_1.

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: A ring-based public key cryptosystem. In Joe Buhler, editor, *Algorithmic Number Theory, Third International Symposium, ANTS-III, Portland, Oregon, USA, June 21-25, 1998, Proceedings*, volume 1423 of *Lecture Notes in Computer Science*, pages 267–288. Springer, 1998. doi:10.1007/BFB0054868.

[HSS17] Carmit Hazay, Peter Scholl, and Eduardo Soria-Vazquez. Low cost constant round MPC combining BMR and oblivious transfer. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*, Lecture Notes in Computer Science, pages 598–628. Springer, 2017. doi:10.1007/978-3-319-70694-8_21.

[HvdH21] David Harvey and Joris van der Hoeven. Integer multiplication in time o(n log n). *Annals of Mathematics*, 2021. URL: https://api.semanticscholar.org/CorpusID:109934776.

[HY24] Carmit Hazay and Yibin Yang. Toward malicious constant-rate 2pc via arithmetic garbling. In *Advances in Cryptology – EUROCRYPT 2024: 43rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zurich, Switzerland, May 26–30, 2024, Proceedings, Part V*, page 401–431, Berlin, Heidelberg, 2024. Springer-Verlag. doi:10.1007/978-3-031-58740-5_14.

[ILL25] Yuval Ishai, Hanjun Li, and Huijia Lin. A unified framework for succinct garbling from homomorphic secret sharing. In Yael Tauman Kalai and Seny F. Kamara, editors, *Advances in Cryptology - CRYPTO 2025 - 45th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2025, Proceedings, Part IV*, Lecture Notes in Computer Science, pages 390–425. Springer, 2025. doi:10.1007/978-3-032-01884-7_13.

[JLS21] Aayush Jain, Huijia Lin, and Amit Sahai. Indistinguishability obfuscation from well-founded assumptions. In Samir Khuller and Virginia Vassilevska Williams, editors, *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*, pages 60–73. ACM, 2021. doi:10.1145/3406325.3451093.

[Kal60] R. E. Kalman. A new approach to linear filtering and prediction problems. *Journal of Basic Engineering*, 82(1):35–45, 03 1960. arXiv:https://asmedigitalcollection.asme.org/fluidsengineering/article-pdf/82/1/35/5518977/35_1.pdf, doi:10.1115/1.3662552.

[Kel20] Marcel Keller. Mp-spdz: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1575–1590, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3372297.3417872.

[KO62] Anatolii Karatsuba and Yu Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 7:595, 12 1962.

[KS08] Vladimir Kolesnikov and T. Schneider. Improved garbled circuit: Free xor gates and applications. In *International Colloquium on Automata, Languages and Programming*, 2008. URL: https://api.semanticscholar.org/CorpusID:7746167.

[LD25] Simon Langowski and Srini Devadas. Efficient modular multiplication using vector instructions on commodity hardware. Cryptology ePrint Archive, Paper 2025/1068, 2025. URL: https://eprint.iacr.org/2025/1068.

[LL24] Hanjun Li and Tianren Liu. How to garble mixed circuits that combine boolean and arithmetic computations. In *Advances in Cryptology – EUROCRYPT 2024: 43rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zurich, Switzerland, May 26–30, 2024, Proceedings, Part VI*, page 331–360, Berlin, Heidelberg, 2024. Springer-Verlag. doi:10.1007/978-3-031-58751-1_12.

[LLL26] Hanjun Li, Huijia Lin, and George Lu. Succinct garbled circuits with low-depth garbling algorithms. In Joan Daemen and Emmanuel Thomé, editors, *Advances in Cryptology - EUROCRYPT 2026 - 45th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Rome, Italy, May 10-14, 2026, Proceedings, Part III*, Lecture Notes in Computer Science, pages 246–275. Springer, 2026. doi:10.1007/978-3-032-25324-8_9.

[LLLH26] Can Liu, Lenny Liu, Ning Luo, and David Heath. Improved garbled RAM via garbled merge. Cryptology ePrint Archive, Paper 2026/761, 2026. URL: https://eprint.iacr.org/2026/761.

[LLPT23] Hanjun Li, Huijia Lin, Antigoni Polychroniadou, and Stefano Tessaro. Lerna: Secure single-server aggregation via key-homomorphic masking. In *Advances in Cryptology – ASIACRYPT 2023: 29th International Conference on the Theory and Application of Cryptology and Information Security, Guangzhou, China, December 4–8, 2023, Proceedings, Part I*, page 302–334, Berlin, Heidelberg, 2023. Springer-Verlag. doi:10.1007/978-981-99-8721-4_10.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Monaco / French Riviera, May 30 - June 3, 2010. Proceedings*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2010. doi:10.1007/978-3-642-13190-5_1.

[LPSY15] Yehuda Lindell, Benny Pinkas, Nigel P. Smart, and Avishay Yanai. Efficient constant round multi-party computation combining BMR and SPDZ. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, Lecture Notes in Computer Science, pages 319–338. Springer, 2015. doi:10.1007/978-3-662-48000-7_16.

[LS25] Junru Li and Yifan Song. Honest majority constant-round MPC with linear communication from one-way functions. In Benny Applebaum and Huijia (Rachel) Lin, editors, *Theory of Cryptography - 23rd International Conference, TCC 2025, Aarhus, Denmark, December 1-5, 2025, Proceedings, Part I*, Lecture Notes in Computer Science, pages 3–32. Springer, 2025. doi:10.1007/978-3-032-12287-2_1.

[LWYY24] Hanlin Liu, Xiao Wang, Kang Yang, and Yu Yu. The hardness of LPN over any integer ring and field for PCG applications. In Marc Joye and Gregor Leander, editors,

EUROCRYPT 2024, Part VI, volume 14656 of *LNCS*, pages 149–179. Springer, Heidelberg, may 2024. Artifact available at https://artifacts.iacr.org/tches/2022/a1. doi:10.1007/978-3-031-58751-1_6.

[LWYY25a] Hanlin Liu, Xiao Wang, Kang Yang, and Yu Yu. Authenticated bitgc for actively secure rate-one 2pc. In Yael Tauman Kalai and Seny F. Kamara, editors, *Advances in Cryptology - CRYPTO 2025 - 45th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2025, Proceedings, Part IV*, Lecture Notes in Computer Science, pages 652–687. Springer, 2025. doi:10.1007/978-3-032-01884-7_21.

[LWYY25b] Hanlin Liu, Xiao Wang, Kang Yang, and Yu Yu. Bitgc: Garbled circuits with 1 bit per gate. In *Advances in Cryptology – EUROCRYPT 2025: 44th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Madrid, Spain, May 4–8, 2025, Proceedings, Part VII*, page 437–466, Berlin, Heidelberg, 2025. Springer-Verlag. doi:10.1007/978-3-031-91098-2_16.

[MORS24] Pierre Meyer, Claudio Orlandi, Lawrence Roy, and Peter Scholl. Rate-1 arithmetic garbling from homomorphic secret sharing. In *Theory of Cryptography: 22nd International Conference, TCC 2024, Milan, Italy, December 2–6, 2024, Proceedings, Part IV*, page 71–97, Berlin, Heidelberg, 2024. Springer-Verlag. doi:10.1007/978-3-031-78023-3_3.

[MORS25] Pierre Meyer, Claudio Orlandi, Lawrence Roy, and Peter Scholl. Silent circuit relinearisation: Sublinear-size (boolean and arithmetic) garbled circuits from DCR. In Yael Tauman Kalai and Seny F. Kamara, editors, *Advances in Cryptology - CRYPTO 2025 - 45th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2025, Proceedings, Part IV*, Lecture Notes in Computer Science, pages 426–458. Springer, 2025. doi:10.1007/978-3-032-01884-7_14.

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, Lecture Notes in Computer Science, pages 223–238. Springer, 1999. doi:10.1007/3-540-48910-X_16.

[PBMW99] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120. URL: http://ilpubs.stanford.edu:8090/422/.

[PP95] Karl C. Posch and Reinhard Posch. Modulo reduction in residue number systems. *IEEE Trans. Parallel Distrib. Syst.*, 6(5):449–454, May 1995. doi:10.1109/71.382314.

[Rab05] Michael O. Rabin. How to exchange secrets with oblivious transfer. *IACR Cryptol. ePrint Arch.*, 2005:187, 2005. URL: http://eprint.iacr.org/2005/187.

[Rog04] Phillip Rogaway. Nonce-based symmetric encryption. In Bimal K. Roy and Willi Meier, editors, *Fast Software Encryption, 11th International Workshop, FSE 2004, Delhi, India, February 5-7, 2004, Revised Papers*, Lecture Notes in Computer Science, pages 348–359. Springer, 2004. doi:10.1007/978-3-540-25937-4_22.

[RS22] Rahul Rachuri and Peter Scholl. Le mans: Dynamic and fluid MPC for dishonest majority. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology - CRYPTO 2022 - 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15-18, 2022, Proceedings, Part I*, Lecture Notes in Computer Science, pages 719–749. Springer, 2022. doi:10.1007/978-3-031-15802-5_25.

[Sha71] Daniel Shanks. Class number, a theory of factorization and genera. In *Proc. Symp. Pure Math.*, volume 20, pages 415–440, Providence, R.I., 1971. American Mathematical Society.

[SS71] A. Schönhage and V. Strassen. Schnelle multiplikation großer zahlen. *Computing*, 7(3):281–292, Sep 1971. doi:10.1007/BF02242355.

[WRK17a] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 39–56, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3133956.3133979.

[WRK17b] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 39–56, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3133956.3133979.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 162–167, 1986. doi:10.1109/SFCS.1986.25.

[YWZ20] Kang Yang, Xiao Wang, and Jiang Zhang. More efficient MPC from improved triple generation and authenticated garbling. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 1627–1646. ACM, 2020. doi:10.1145/3372297.3417285.

[ZLY+26] Wenhao Zhang, Hanlin Liu, Kang Yang, Wen-jie Lu, Yu Yu, Xiao Wang, and Chenkai Weng. Bitgc made (more) efficient. In Joan Daemen and Emmanuel Thomé, editors, *Advances in Cryptology - EUROCRYPT 2026 - 45th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Rome, Italy, May 10-14, 2026, Proceedings, Part III*, Lecture Notes in Computer Science, pages 214–245. Springer, 2026. doi:10.1007/978-3-032-25324-8_8.

A Full Offline Protocols

A.1 Preprocessing

To further modularize the offline phase protocol, we introduce the preprocessing protocol Π_{prep}, which realizes the ideal functionality $\mathcal{F}_{\text{prep}}$ in the $\{\mathcal{F}^{p}_{\text{SPDZprep}}, \mathcal{F}^{N^\zeta}_{\text{SPDZprep}}, \mathcal{F}_{\text{coin}}, \mathcal{F}_{\text{commit}}\}$-hybrid model. At a high level, the protocol generates the necessary random shares for subsequent phases. The preprocessing phase is independent of both the circuit (except for its size) and the inputs. We also note that it requires no communication with the clients.

We require the following preprocessing functionality $\mathcal{F}_{\text{prep}}$ for preparing SPDZ shares of random values, random multiplication triples, random bits, and random masks from a bounded range. While using a preprocessing phase to prepare SPDZ shares of random values has been considered in prior works, e.g. [RS22, GLM+24], we note two distinctions in our use case: (1) our SPDZ shares are w.r.t. a composite (RSA) modulus N^ζ rather than a prime p; (2) in addition to shares of uniformly random values, we also need shares of bounded integers. Existing preprocessing techniques [RS22, GLM+24] for preparing (mod$p$)-SPDZ shares of random values and multiplication triples can be straightforwardly adapted to modN^ζ. Specifically, these techniques rely on pseudorandom correlation generators based on the Learning Parity with Noise (LPN) assumption [BFKL94]. Recent work [LWYY24, HY24] establishes that the security of LPN over $\mathbb{Z}_{N^\zeta}$ simply reduces to its security over the prime factors of N.

However, the protocol for generating shares of random bits from [GLM+24] no longer works modN^ζ, because it requires computing square roots, which is efficient mod$p$, but not modN^ζ. At a high level, our solution is to use an independent helper SPDZ instance mod$p$ for generating SPDZ shares of random bits mod$p$, and then design a protocol for converting them into SPDZ shares in our main instance modN^ζ. In the following, we write a superscript p to highlight the helper instance.

In more detail, assuming each server P_i holds an additive share Δ_i of the MAC key Δ in the main instance, our goal is to derive additive shares $\langle \Delta \cdot b \rangle$, $\langle b \rangle$ for random bits b. Towards this, the servers first obtain authenticated shares of random bits $[\![b]\!]^p$ in the helper instance, and then run a modulus-switching protocol adapted from prior work [DFK+06] to obtain $\langle b \rangle$ from $\langle b \rangle^p$. Next, in order to obtain the shares $\langle \Delta \cdot b \rangle$, the servers input their local key shares Δ_i to the helper instance to obtain $[\![\Delta_i]\!]^p$, and linearly combine them into $[\![\Delta]\!]^p = \sum_{i\in[n]} [\![\Delta_i]\!]^p$. Assuming each Δ_i is sufficiently small, no wrap-around happens modN^ζ or mod$p$. Finally, the servers perform authenticated multiplications between $[\![\Delta]\!]^p$ and $[\![b]\!]^p$ in the helper instance to obtain $[\![\Delta \cdot b]\!]^p$, and modulus-switch to obtain $\langle \Delta \cdot b \rangle$.

The outlined steps above suffice for the semi-honest setting. However, there are multiple vulnerabilities in the malicious setting. First, the corrupted servers could use $\Delta^*_i \neq \Delta_i$ as their inputs to the helper SPDZ instance, leading to a share $[\![\Delta^*]\!]^p$ of an incorrect MAC key. As a result, the obtained shares $\langle \Delta^* \cdot b \rangle$ and $\langle b \rangle$ may not form valid SPDZ shares in the main instance. We address this via a MAC-check step in the main instance. Second, the modulus-switching protocol from [DFK+06] only has privacy assuming the underlying shared values are sufficiently small. The corrupted servers could use large Δ^*_i, which breaks privacy during modulus-switching from $\langle \Delta^* \cdot b \rangle^p$ to $\langle \Delta^* \cdot b \rangle$. We address this via a standard range-check step in the helper instance over $[\![\Delta^*]\!]^p$ before running modulus-switching. [9]

For generating SPDZ shares of bounded integers $[\![r]\!]$, we first let the servers locally sample ran-

[9] Existing range-check techniques require first performing a bit-decomposition of the value Δ^*, which takes $\log p$ rounds. Alternatively we can apply our optimized range check (in Π_{masks}) in the helper instance to achieve constant round. The trade-off is a relaxed range bound on Δ^*, which suffices for our purpose.

dom shares $\langle r \rangle_i$ from a bounded range, and authenticate them to obtain $[\![r]\!]$. To prevent malicious servers from choosing out-of-bound local shares, we implement the optimized range-checking idea outlined in the technical roadmap (Section 2.4) over $[\![r]\!]$.

Functionality $\mathcal{F}_{\text{prep}}$

Init. Upon receiving $(\texttt{init}, N, \zeta, \Delta_{\max}, r_{\max}, r_{\max}^{\text{relax}}, m_R, m_T, m_B, m_M)$ from all servers:

- Receive corrupted servers' key shares $\{\Delta_i\}_{i \in I}$ from $\mathcal{S}$.
- Sample honest servers' key shares $\Delta_i \leftarrow [0, \Delta_{\max})$ for $i \notin I$.
- Compute global key $\Delta = \sum_{i=1}^{n} \Delta_i$ and check that $\Delta < n \cdot \Delta_{\max}$. Abort otherwise.
- Send Δ_i to P_i.

Random Values. Repeat m_R times:

- Sample $u \leftarrow \mathbb{Z}_{N^\zeta}$.
- Receive corrupted servers' shares of $[\![u]\!]$ from $\mathcal{S}$.
- Sample honest servers' shares such that they sum to u and $\Delta \cdot u$.
- Output $[\![u]\!]$ to servers.

Triples. Repeat m_T times:

- Sample $a, b \leftarrow \mathbb{Z}_{N^\zeta}$, and compute $c = a \cdot b \pmod{N^\zeta}$.
- Receive corrupted servers' shares of $[\![a]\!], [\![b]\!], [\![c]\!]$ from $\mathcal{S}$.
- Sample honest servers' shares such that they sum to a, b, c and $\Delta \cdot a, \Delta \cdot b, \Delta \cdot c$.
- Output $([\![a]\!], [\![b]\!], [\![c]\!])$ to servers.

Bits. Repeat m_B times:

- Sample $b \leftarrow \{0, 1\}$.
- Compute $\gamma_b = \Delta \cdot b \pmod{N^\zeta}$.
- Receive corrupted servers' shares of $[\![b]\!]$ from $\mathcal{S}$.
- Sample honest servers' shares such that they sum to b and $\gamma_b \pmod{N^\zeta}$.
- Output $[\![b]\!]$ to servers.

(Bounded) Masks. Repeat m_M times:

- Receive corrupted servers' key shares $\{[\![r]\!]_i\}_{i \in I}$ from $\mathcal{S}$.
- Sample honest servers' shares $\langle r \rangle_i \leftarrow [0, r_{\max})$ for $i \notin I$, and compute $r = \sum_{i \in [n]} \langle r \rangle_i$. Abort if $r \bmod N^\zeta \in [r_{\max}^{\text{relax}},\ N^\zeta - r_{\max}^{\text{relax}}]$.

– Sample honest servers' shares $\langle \Delta \cdot r \rangle_i$ for $i \notin I$ such that they sum to $\Delta \cdot r \pmod{N^\zeta}$.

– Output $[\![r]\!]$ to servers.

Abort. Upon receiving `abort` from $\mathcal{S}$, send Δ to $\mathcal{S}$, output `abort` to all servers, and abort.

As our new protocol builds on existing techniques for preparing ($\mathrm{mod}\, p$)-SPDZ shares, we summarize those into a functionality $\mathcal{F}^p_{\mathrm{SPDZprep}}$ and omit its implementation. We also summarize the techniques that can be straightforwardly adapted to $\mathrm{mod}\, N^\zeta$ into $\mathcal{F}^{N^\zeta}_{\mathrm{SPDZprep}}$. Here, we overload the notation p to denote an arbitrary prime that is not related to the RSA modulus N.

Functionality $\mathcal{F}^p_{\mathrm{SPDZprep}}$
Init. Upon receiving $(\mathtt{init}, p, m_R, m_T, m_B)$ from all servers, proceed analogously to $\mathcal{F}_{\mathrm{prep}}$, except sampling $\Delta_i \leftarrow [p]$ for $i \notin I$.
Range Check: Upon receiving $(\mathsf{range}, B, [\![v]\!]_i)$ from every server P_i, check if $\{[\![v]\!]_i\}$ form a valid SPDZ share of some value $v < B$. Abort otherwise.
Random Values, Triples, Random Bits, Abort: Analogous to $\mathcal{F}_{\mathrm{prep}}$, except over mod$p$.

Functionality $\mathcal{F}^{N^\zeta}_{\mathrm{SPDZprep}}$
Init. Upon receiving $(\mathtt{init}, N, \zeta, \Delta_{\max}, m_R, m_T)$ from all servers, proceed analogously to $\mathcal{F}_{\mathrm{prep}}$.
Random Values, Triples, Abort: Analogous to $\mathcal{F}_{\mathrm{prep}}$.

We now present our preprocessing protocol Π_{prep} implementing $\mathcal{F}_{\mathrm{prep}}$.

Protocol Π_{prep}
The protocol uses $\mathcal{F}^p_{\mathrm{SPDZprep}}, \mathcal{F}^{N^\zeta}_{\mathrm{SPDZprep}}, \mathcal{F}_{\mathrm{coin}}, \mathcal{F}_{\mathrm{commit}}$ as ingredients.
Servers hold $N, \zeta, \Delta_{\max}, r_{\max}, r^{\mathrm{relax}}_{\max}, m_R, m_T, m_B, m_M$ as common inputs.
Init. Let $\ell = \lceil \log r^{\mathrm{relax}}_{\max} \rceil$, $m'_B = m_B + \kappa \cdot \ell$, and $m'_R = m_R + m_M + m'_B$.

1. Servers invoke $\mathcal{F}^{N^\zeta}_{\mathrm{SPDZprep}}$ with $(\mathtt{init}, N, \zeta, \Delta_{\max}, m'_R, m_T)$ to obtain Δ_i, and ($\mathrm{mod} N^\zeta$)-SPDZ shares of

   – m_R random values and m_T random triples required by the **Random Values**, and **Triples** interfaces;

   – m_M random values $[\![\vec{v}]\!]$ and m'_B random values $[\![\vec{u}]\!]$.

   Servers locally form $[\![1]\!]$ using Δ_i.

2. Servers invoke $\mathcal{F}_{\mathrm{coin}}$ to sample a random prime $p > n^2 \cdot \Delta_{\max} \cdot 2^{\kappa+3}$, and invoke $\mathcal{F}^p_{\mathrm{SPDZprep}}$ with $(\mathtt{init}, p, 1, m'_B, m'_B)$ to obtain Δ'_i and ($\mathrm{mod}p$)-SPDZ shares of

   – 1 random value $[\![u]\!]^p$;

   – m'_B random triples $[\![\vec{\alpha}]\!]^p, [\![\vec{\beta}]\!]^p, [\![\vec{\gamma}]\!]^p$;

   – m'_B random bits $[\![\vec{b}]\!]^p$.

   Servers locally form $[\![1]\!]^p$ using Δ'_i.

Obtain Shares of $[\![\Delta]\!]^p$**:** for every $i \in [n]$

1. Servers locally define $\langle \Delta \rangle_i^p = \Delta_i$, where Δ_i is viewed as an integer.
2. Servers run Π_{auth}^p to obtain $[\![\Delta]\!]^p$, consuming (mod p)-SPDZ shares of a random value.
3. Servers invoke $\mathcal{F}_{\text{SPDZprep}}^p$ with $(\mathsf{range}, \Delta_{\max} \cdot n, [\![\Delta]\!]^p)$ to verify the underlying shared value satisfies $\Delta < \Delta_{\max} \cdot n$.

Obtain Shares of Random Bits $[\![\vec{b}]\!]$: Servers run Π_{Bits}, to obtain $m'_B = m_B + \kappa \cdot \ell$ shares of random bits, consuming (mod p)-SPDZ shares of m'_B triples and m'_B random bits and (mod N^ζ) shares of m'_B random values. The results are organized as (mod N^ζ)-SPDZ shares of

- m_B random bits as required by the **Bits** interface;
- κ random bit vectors $\{[\![\vec{b}_j]\!]\}_{j \in [\kappa]}$ of dimension ℓ.

Obtain Shares of Masks $[\![\vec{r}]\!]$: Servers run Π_{Masks} to obtain m_M shares of bounded masks as required by **Masks** interface, consuming (mod N^ζ)-SPDZ shares of m_M random values and κ random bit vectors of dimension ℓ.

Helper Protocol Π_{auth}^R
Global Input: SPDZ share of $[\![1]\!]$ (mod R).
Input: m additive shares $\langle \vec{v} \rangle$ (mod R), and m SPDZ shares of random values $[\![\vec{u}]\!]$ (mod R).

1. Servers compute $\langle \vec{s} \rangle \leftarrow \langle \vec{v} \rangle + \langle \vec{u} \rangle$ and run Π_{open}^R to reconstruct s.
2. Servers compute $[\![\vec{v}]\!] \leftarrow [\![1]\!] \cdot \vec{s} - [\![\vec{u}]\!]$ as their output.

Helper Protocol Π_{bits}
Global Input: (mod N^ζ) share of $[\![1]\!]$; (mod p)-SPDZ shares of $[\![1]\!]^p$ and the MAC key $[\![\Delta]\!]^p$; The parameters $n, \Delta_{\max}$.
Input: (mod p)-SPDZ shares of m triples $[\![\vec{\alpha}]\!]^p$, $[\![\vec{\beta}]\!]^p$, $[\![\vec{\gamma}]\!]^p$, and m random bits $[\![\vec{b}]\!]^p$; (mod N^ζ)-SPDZ shares of m random values $[\![\vec{u}]\!]$.

1. Servers run $[\![\Delta \cdot \vec{b}]\!]^p \leftarrow \Pi_{\text{mul}}^p([\![\Delta]\!]^p, [\![\vec{b}]\!]^p)$, consuming the m triples, and then run $\Pi_{\text{SPDZ-MAC}}^p$ to batch authenticate all opened values during Π_{mul}^p.
2. Let $t = \kappa + \lceil \log n \rceil + \lceil \log \Delta_{\max} \rceil + 1$. Servers run $\langle \vec{b} \rangle^* \leftarrow \Pi_{\text{ModToInt}}^{p,t}(\langle \vec{b} \rangle^p)$ and $\langle \Delta \cdot \vec{b} \rangle^* \leftarrow \Pi_{\text{ModToInt}}^{p,t}(\langle \Delta \cdot \vec{b} \rangle^p)$, and locally derive (mod N^ζ)-SPDZ shares $[\![\vec{b}]\!]$ (by taking modN^ζ).
3. Servers compute $[\![\vec{s}]\!] \leftarrow [\![\vec{b}]\!] + [\![\vec{u}]\!]$, run Π_{open} to reconstruct $\vec{s}$, and run $\Pi_{\text{SPDZ-MAC}}^{N^\zeta}$ to authenticate the opened values $\vec{s}$.

Helper Protocol Π_{masks}
Global Input: The (mod N^ζ) share $[\![1]\!]$; the parameters $n, r_{\max}, r_{\max}^{\text{relax}}, \ell$.
Input: (mod N^ζ)-SPDZ shares of m random values $[\![\vec{v}]\!]$ and κ random bit vectors $\{[\![\vec{b}_j]\!]\}_{j \in [\kappa]}$ of dimension ℓ. (All SPDZ shares below have modulus N^ζ, hence we omit the superscript.)
Sample Masks:

1. Servers locally sample shares of $\langle \vec{r} \rangle$, where P_i samples $\langle \vec{r} \rangle_i \leftarrow [0, r_{\max})^m$.
2. Servers run $\Pi_{\text{auth}}^{N^\zeta}$ to obtain $[\![\vec{r}]\!]$, consuming (mod N^ζ)-SPDZ shares of m random values.

Range-check Masks:

1. Servers combine each bit vector $[\![\vec{b}_j]\!]$ into a mask $[\![M_j]\!] \leftarrow \sum_{k \in [\ell]} 2^k \cdot [\![\vec{b}_j[k]]\!]$ for $j \in [\kappa]$.
2. Servers invoke $\mathcal{F}_{\text{coin}}$ to sample a random seed, and expand it into κ bit vectors $\{\vec{\chi}_j\}_{j \in [\kappa]}$ of dimension m.
3. For each "trial" $j \in [\kappa]$, servers compute $[\![V_j]\!] \leftarrow \chi_j^\intercal \cdot [\![\vec{r}]\!] + [\![M_j]\!]$, run Π_{open} to reconstruct V_j. If $V_j \notin [0, r_{\max}^{\text{relax}})$, servers abort.
4. Servers run $\Pi_{\text{SPDZ-MAC}}^{N^\zeta}$ to batch authenticate all opened values $\{V_j\}$.

The following helper protocol converts an additive share over any modulus R into one over $\mathbb{Z}$. It's based on ideas from [DFK+06, ACS02], and requires the input share to be of a bounded value much smaller than R.

Helper Protocol $\Pi_{\text{ModToInt}}^{R,t}$
Global Input: A modulus R, and a parameter t.
Input: Shares of a bounded value $\langle v \rangle$ (mod R).

1. Servers invoke $\mathcal{F}_{\text{coin}}$ to sample a public random share $\langle 0 \rangle$, [a] and re-randomize the input share as $\langle v \rangle \leftarrow \langle v \rangle + \langle 0 \rangle$.
2. Each server P_i computes $\theta_i = \lfloor \langle v \rangle_i / 2^t \rfloor$ and sends θ_i to P_1.
3. P_1 computes an "overflow counter" $l = \lfloor 2^t \sum_i \theta_i / R \rceil$, and returns its integer share as $\langle v \rangle_1^* = \langle v \rangle_1 - l \cdot R$. Other servers P_i return their integer shares as $\langle v \rangle_i^* = \langle v \rangle_i$.

[a] As an optimization, we can instead invoke $\mathcal{F}_{\text{coin}}$ to sample a seed, and expand it using a PRG to derive all public shares $\langle 0 \rangle$ in our overall protocol.

Remark on the efficiency of triple generation. Similarly to [HY24], the random multiplication-triple generation in our preprocessing phase can be based purely on LPN modulo an RSA modulus. We make several observations about the concrete efficiency of this approach. First, although the security of LPN may degrade over integer moduli of the form 2^λ, this degradation does not occur for RSA moduli $N = pq$ [LWYY24, Theorem 10]. More concretely, known attacks against LPN modulo pq with noise rate t proceed by attacking LPN modulo p or modulo q, with respective noise rates $t \cdot (p-1)/p$ or $t \cdot (q-1)/q$. Since LPN becomes harder over larger finite fields, the large prime factors of N are not expected to require a larger noise rate or otherwise less favorable LPN parameters.

Second, despite the larger modulus, our preprocessing can be concretely more efficient than prior Boolean MPC schemes such as [GLM+24] for arithmetic workloads, because our protocol requires far fewer multiplication triples. For example, multiplying two 64-bit integers uses only a single multiplication triple over the RSA modulus in our protocol, whereas [GLM+24] requires at least 10,000 triples over a modulus matching the DDH group order.

Finally, PCG-based preprocessing has been adopted as a standard building block in many recent works [RS22, EGP+23, GLM+24, HY24]. We leave the concrete optimization and implementation of our preprocessing phase as an important direction for future work.

For better computational efficiency, one can alternatively instantiate the underlying PCGs using Ring-LPN rather than standard LPN, following the approaches of [BCG+20, RS22]. As observed in [BCG+20, Remark 3.3], using Ring-LPN over a composite modulus $N = pq$ does not introduce significant weaknesses against known attacks. To ensure correctness, including for the packing technique of [BCG+20], it suffices to work over a polynomial ring $R_N = \mathbb{Z}_N[X]/(F(X))$, where $F(X) = \prod_i (X - r_i)$ for independently sampled $r_i \in \mathbb{Z}_N$. With overwhelming probability, the ideals generated by the factors $(X - r_i)$ are pairwise coprime, satisfying the required conditions.

A.2 Offline Protocol

Now, we construct the offline protocol Π_{offline}, which realizes the ideal functionality $\mathcal{F}_{\text{offline}}$ in the $\{\mathcal{F}_{\text{setup}}, \mathcal{F}_{\text{prep}}, \mathcal{F}_{\text{coin}}, \mathcal{F}_{\text{commit}}\}$-hybrid model. Given the circuit $\mathcal{C}$, the protocol prepares gate-dependent auxiliary data for the main protocol while remaining independent of the circuit's inputs. It also distributes the necessary auxiliary data to the clients. The techniques we use here are largely based on prior works on SPDZ-based MPC with malicious security [DPSZ12, GLM+24].

Protocol Π_{offline}
The protocol uses $\mathcal{F}_{\text{setup}}$, $\mathcal{F}_{\text{prep}}$, $\mathcal{F}_{\text{coin}}$, and $\mathcal{F}_{\text{commit}}$ as ingredients.
Servers hold $\lambda, \kappa, \mathcal{B}, m_{\text{in}}, m_{\text{out}}, m_{\text{mul}}, m_{\text{div}}$ as common inputs.
Initialization:

1. Servers invoke $\mathcal{F}_{\text{setup}}$ with $(\texttt{init}, 1^\lambda, \kappa, \mathcal{B}, m_{\text{in}} + m_{\text{out}} + m_{\text{mul}} + m_{\text{div}})$ to obtain $(N, \zeta, \Delta_{\max}, r_{\max}, r_{\max}^{\text{relax}}, R_{\max})$.
2. Let $\lambda' = \lfloor \log N \rfloor$. Define the following parameters:
   - $m_G = 4 \cdot m_{\text{mul}} + (2+\lambda') \cdot m_{\text{div}} + m_{\text{out}}$: Number of random generators for key-extension and output verification.
   - $m'_G = 2 \cdot (m_G + \kappa + \sqrt{m_G \cdot \kappa})$: Number of random values in $[0, N^{\zeta+1})$ needed.
   - $m_R = \lceil \frac{\zeta+1}{\zeta} \cdot m'_G \rceil$: Number of random values in $\mathbb{Z}_{N^\zeta}$ needed for sampling generators.
   - $m_T = m_{\text{mul}} + \lambda' \cdot m_{\text{div}} + 3 \cdot m_{\text{in}} + 2 \cdot m_{\text{out}}$: Number of authenticated triples.
   - $m_B = (2 \cdot \lambda' + 1) \cdot m_{\text{div}}$: Number of random authenticated bits needed.
   - $m_M = m_{\text{in}} + m_{\text{mul}} + 2 \cdot m_{\text{div}}$: Number of bounded random masks needed.
3. Servers invoke $\mathcal{F}_{\text{prep}}$ with $(\texttt{init}, n, |\mathcal{C}|, N, \zeta, \Delta_{\max}, m_R, m_T, m_B, m_M)$ to obtain the following that directly satisfy the interface of $\mathcal{F}_{\text{offline}}$:
   - Global key shares $\langle \Delta \rangle^*$.
   - SPDZ shares of random bits.
   - SPDZ shares of masks for input, multiplication, and truncation gates. (Masks for linear gates can also be simply computed as required.)
   - SPDZ triples, one per input gate.

   Additionally, servers obtain the following for internal computations:
   - SPDZ shares $\{[\![u]\!]_j\}_{j \in [m_R]}$ of m_R random values.

– SPDZ triples $\{[\![\alpha]\!], [\![\beta]\!], [\![\gamma]\!]\}$, one per multiplication gate, λ' per truncation gate, and two per input or output gate.

Sample Generators:

1. For each $j \in [m_R]$, the servers run Π_{open} on $[\![u_j]\!]$ to reconstruct u_j.
2. The servers locally cast the m_R opened values as m'_G values in $\mathbb{Z}^{\times}_{N^{\zeta+1}}$, and output the first m_G among those with Jacobi symbol 1 mod N. Abort if there are not enough such elements.

Computing Intermediate Values $[\![r^{(x)} \cdot r^{(y)}]\!]$, $[\![\vec{\delta}^{(z)} \oplus \vec{B}^{(z)}]\!]$:

– For each multiplication gate with input wires x, y and output wire z, servers run $[\![r^{(x)} \cdot r^{(y)}]\!] \leftarrow \Pi_{\text{mul}}([\![r^{(x)}]\!], [\![r^{(y)}]\!])$ consuming one multiplication triple per gate.

– For each truncation gate with output wire z, and for $k \in [\lambda']$, servers run $[\![\vec{\delta}^{(z)}[k] \cdot \vec{B}^{(z)}[k]]\!] \leftarrow \Pi_{\text{mul}}([\![\vec{\delta}^{(z)}]\!][k], [\![\vec{B}^{(z)}]\!][k])$, consuming λ' multiplication triples per gate. Servers then set $[\![\vec{\delta}^{(z)} \oplus \vec{B}^{(z)}]\!][k] = [\![\vec{\delta}^{(z)}]\!][k] + [\![\vec{B}^{(z)}]\!][k] - 2 \cdot [\![\vec{\delta}^{(z)}[k] \cdot \vec{B}^{(z)}[k]]\!]$.

Reveal Input and Output Masks: For each input or output wire w:

1. Servers consume a triple $\left([\![\alpha^{(w)}]\!], [\![\beta^{(w)}]\!], [\![\gamma^{(w)}]\!]\right)$ and run $[\![\tau^{(w)}]\!] \leftarrow \Pi_{\text{mul}}([\![r^{(w)}]\!]^*, [\![\alpha^{(w)}]\!])$ consuming another triple.
2. Servers send their shares $\langle r^{(w)} \rangle$, $\langle \alpha^{(w)} \rangle$, $\langle \beta^{(w)} \rangle$, $\langle \gamma^{(w)} \rangle$, and $\langle \tau^{(w)} \rangle$ to the associated client C.
3. C reconstructs the values and verifies that $\gamma^{(w)} = \alpha^{(w)} \cdot \beta^{(w)}$ and $\tau^{(w)} = r^{(w)} \cdot \alpha^{(w)}$. If either check fails, C aborts the protocol.

Batch Verify Openings: Servers run $\Pi^{\langle\Delta\rangle}_{\text{SPDZ-MAC}}$ to batch authenticate all opened values during sampling generators and all instances of Π_{mul}.

B Security Proofs

B.1 Security Proof for the Preprocessing Protocol

Lemma B.1. *Suppose that in the presence of a static malicious adversary corrupting exactly $n-1$ servers, the protocol Π_{prep}*

(1) securely realizes the functionality $\mathcal{F}_{\text{prep}}$ in the $\{\mathcal{F}^{p}_{\text{SPDZprep}}, \mathcal{F}^{N^\zeta}_{\text{SPDZprep}}, \mathcal{F}_{\text{coin}}, \mathcal{F}_{\text{commit}}\}$-hybrid model, and

(2) does not abort except with negligible probability when all parties follow the protocol honestly.

Then, Π_{prep} also securely realizes $\mathcal{F}_{\text{prep}}$ in the same hybrid model in the presence of a static malicious adversary corrupting any $t < n$ servers.

Proof. Let $\mathcal{A}$ be a static malicious adversary corrupting a set of servers I with $|I| = t < n$. We distinguish two cases:

– **Case 1:** $t > 0$**.** We can construct an adversary $\mathcal{A}'$ that corrupts a set I' of size $n-1$ such that $I \subset I'$. The adversary $\mathcal{A}'$ simulates $\mathcal{A}$ for the servers in I, and runs the honest protocol

strategy for the servers in $I' \setminus I$. By Condition (1), the protocol is secure against any adversary corrupting $n-1$ servers. Thus, there exists a simulator $\mathcal{S}'$ for $\mathcal{A}'$ such that the real execution with $\mathcal{A}'$ is indistinguishable from the ideal execution with $\mathcal{S}'$. Since the additional servers in $I' \setminus I$ corrupted by $\mathcal{A}'$ follow the protocol honestly, the view of the environment and honest parties in the real execution with $\mathcal{A}'$ is identical to that with $\mathcal{A}$. Therefore, security against $\mathcal{A}'$ implies security against $\mathcal{A}$.

– **Case 2:** $t = 0$**.** In this case, all servers are honest. Consider a hypothetical functionality $\mathcal{F}'_{\text{prep}}$ that is identical to $\mathcal{F}_{\text{prep}}$ but allows the simulator to send an `abort` command even when no servers are corrupted. Using the same reduction argument as in Case 1 (constructing an adversary $\mathcal{A}'$ that corrupts $n-1$ servers), we can show that there exists a simulator $\mathcal{S}'$ such that the real-world execution is computationally indistinguishable from the ideal-world execution with $\mathcal{F}'_{\text{prep}}$.

  Since the real protocol does not abort (except with negligible probability) when all parties are honest, the ideal execution with $\mathcal{S}'$ and $\mathcal{F}'_{\text{prep}}$ must also not abort (except with negligible probability) due to the indistinguishability. Conditioned on $\mathcal{S}'$ not aborting, the execution with the hypothetical $\mathcal{F}'_{\text{prep}}$ is identical to the execution with the standard $\mathcal{F}_{\text{prep}}$ (which forbids aborts for $t = 0$). Thus, we can remove all the aborts in $\mathcal{S}'$ to get a simulator $\mathcal{S}$ for $\mathcal{F}_{\text{prep}}$ and show that Π_{prep} securely realizes $\mathcal{F}_{\text{prep}}$.

□

Lemma B.2 (Soundness of Randomized Range Check)**.** *Let κ be a statistical parameter, B be a target range, $R > 2B$ be a modulus, and $\vec{r}$ be a vector (of dimension m) of integers to be range-checked. Consider the following experiment:*

– *Choose an arbitrary pad $M \in \mathbb{Z}$, and sample a random bit vector $\vec{\chi} \leftarrow \{0,1\}^m$.*

– *Compute $V \leftarrow M + \vec{\chi}^{\intercal} \cdot \vec{r} \bmod R$. Abort if $V \notin [0, B)$.*

If no abort happens after κ independent runs of the experiment, then all entries in $\vec{r}$ are within $[-B, B] \pmod R$, except with $2^{-\kappa}$ probability.

Proof. Suppose there exists an index w^* such that $\vec{r}[w^*] \notin [-B, B] \bmod R$. Then, consider the two cases in an experiment: $\vec{\chi}[w^*] = 0$ or $\vec{\chi}[w^*] = 1$. Only one of these two cases can result in $V \bmod R \in [0, B)$, since the difference between the two cases is at least B away from 0 modulo N^{ζ}.

Thus, each experiment aborts with probability $\geq 1/2$. And we conclude with probability $\geq 1 - 2^{-\kappa}$, there exists at least one experiment that aborts. □

Lemma B.3 (Correctness of ModToInt)**.** *Let B be a bound on the shared value, R be a modulus, and t be an integer such that*

$$R > 2 \cdot (n \cdot 2^t + B).$$

For all additive shares $\langle v \rangle \pmod R$ of a value $v \in [B]$, the output of $\langle v \rangle^* \leftarrow \Pi^{R,t}_{\text{ModToInt}}(\langle v \rangle)$ satisfies

$$\sum_{i \in [n]} \langle v \rangle^*_i = v \text{ (over } \mathbb{Z}).$$

Proof. Recall that in the protocol, the i-th output share $\langle v \rangle^*$ is defined as $\langle v \rangle^*_i = \langle v \rangle_i$ for $i \neq 1$, and $\langle v \rangle^*_1 = \langle v \rangle_1 - l \cdot R$. On the other hand, by assumption we have

$$\sum_i \langle v \rangle_i = v \bmod R \implies \sum_i \langle v \rangle_i = v + c \cdot R,$$

for some "overflow" value c. Therefore, it suffices to show $l = c$. Recall l is computed as follows.

$$l = \lfloor 2^t \sum_i \theta_i / R \rceil, \quad \theta_i = \lfloor \langle v \rangle_i / 2^t \rfloor.$$

By construction, we have $\langle v \rangle_i = 2^t \cdot \theta_i + (\langle v \rangle_i \bmod 2^t)$. So we can re-write

$$l = \lfloor \sum_i \langle v \rangle_i / R - \sum_i (\langle v \rangle_i \bmod 2^t)/R \rceil = c + \lfloor v/R - \sum_i (\langle v \rangle_i \bmod 2^t)/R \rceil.$$

By the assumption we have $|v - \sum_i (\langle v \rangle_i \bmod 2^t)| < B + n \cdot 2^t < R/2$. Hence, we have $l = c$ as desired. □

Lemma B.4 (Simulation of ModToInt)**.** *Let κ be a statistical security parameter, B be a bound on the shared value, R be a modulus, and t be an integer such that*

$$2^t > 2^\kappa \cdot B.$$

There exists an efficient simulator $\mathsf{Sim}_{\text{ModToInt}}$ such that for all $v_h \in [B]$, $v_{\text{adv}} \in [R]$, and any additive shares $\langle v \rangle$ of $v = v_h + v_{\text{adv}}$, the view of any subset $I \subset [n]$ of $n-1$ servers during $\Pi^{R,t}_{\text{ModToInt}}(\langle v \rangle)$ can be simulated (with statistical distance $< 2^{-\kappa}$) as $\mathsf{Sim}^t_{\text{ModToInt}}(v_{\text{adv}}, \langle v \rangle_{\text{adv}})$, where $\langle v \rangle_{\text{adv}} = \sum_{i \in I} \langle v \rangle_i$.

Proof. If the subset I of $n-1$ servers doesn't include P_1, then its view is $\emptyset$ and is trivially simulated. Otherwise, let $P_h \neq P_1$ be the honest server. Its message during Π^R_{ModToInt} is

$$\theta_h \leftarrow \lfloor (\langle v \rangle_h + \langle 0 \rangle_h \bmod R)/2^t \rfloor = \lfloor (v_h + \underbrace{v_{\text{adv}} - (\langle v \rangle_{\text{adv}} - \langle 0 \rangle_h \bmod R)}_{u})/2^t \rfloor,$$

where $\langle 0 \rangle$ is a public (pseudo-)random share of 0. We note that θ_h is independent of v_h if the term u satisfies

$$u < 2^t - B \bmod 2^t. \tag{1}$$

By the randomness of $\langle 0 \rangle_h$, the term u has R equally likely possible values, where at most $(R/2^t) \cdot B$ of them violate Eq 1. Therefore, we can simulate θ_h (without using v_h) as $\theta_h \leftarrow \lfloor u/2^t \rfloor$, which is correct except with probability $(R/2^t) \cdot B/R = B/2^t < 2^{-\kappa}$. □

Lemma B.5. *Assuming a PRG, the protocol Π_{prep} securely realizes the functionality $\mathcal{F}_{\text{prep}}$ in the $\{\mathcal{F}^p_{\text{SPDZprep}}, \mathcal{F}^{N^\zeta}_{\text{SPDZprep}}, \mathcal{F}_{\text{coin}}, \mathcal{F}_{\text{commit}}\}$-hybrid model in the presence of a static malicious adversary corrupting at most $t = n-1$ servers and any number of clients.*

Proof. For any adversary $\mathcal{A}$, let $I \subset [n]$ be the set of corrupted servers by $\mathcal{A}$. In the following we consider the most complicated case where all-but-one servers are corrupted, and the remaining honest server P_h is not P_1. We first describe a simulator that interacts with $\mathcal{F}_{\text{prep}}$, and runs $\mathcal{A}$ internally.

Simulator $\mathcal{S}$:
Init.:

1. Emulate $\mathcal{F}^{N^\zeta}_{\text{SPDZprep}}$ by accepting from $\mathcal{A}$ $\{\Delta_i\}_{i \neq h}$, and the following corrupted servers' shares of:
   - m_R random values and m_T random triples required by the **Random Values**, and **Triples** interfaces. Directly forward those to $\mathcal{F}_{\text{prep}}$.
   - m_M random values $\{[\![\vec{v}]\!]_i\}_{i \neq h}$ and m'_B random values $\{[\![\vec{u}]\!]_i\}_{i \neq h}$.

Forward corrupted server's shares $\{\Delta_i\}_{i\neq h}$ to $\mathcal{F}_{\text{prep}}$. Define Δ_{adv}, where $\Delta_{\text{adv}} = \sum_{i\neq h} \Delta_i (\bmod N^\zeta)$, $[\![\vec{v}]\!]_{\text{adv}}$, where

$$[\![\vec{v}]\!]_{\text{adv}} = (\langle\vec{v}\rangle_{\text{adv}}, \langle\Delta\cdot\vec{v}\rangle_{\text{adv}}), \quad \langle\vec{v}\rangle_{\text{adv}} = \sum_{i\neq h}\langle\vec{v}\rangle_i, \quad \langle\Delta\cdot\vec{v}\rangle_{\text{adv}} = \sum_{i\neq h}\langle\Delta\cdot\vec{v}\rangle_i,$$

and $[\![\vec{u}]\!]_{\text{adv}}$ analogously.

2. Emulate $\mathcal{F}^p_{\text{SPDZprep}}$ by accepting from $\mathcal{A}$ $\{\Delta'_i\}_{i\neq h}$ and the following corrupted servers' shares:
   - 1 random value $\{[\![u]\!]^p_i\}_{i\neq h}$
   - m'_B random triples $\{[\![\vec{\alpha}]\!]^p_i, [\![\vec{\beta}]\!]^p_i, [\![\vec{\gamma}]\!]^p_i\}_{i\neq h}$;
   - m'_B random bits $\{[\![\vec{b}]\!]^p_i\}_{i\neq h}$.

   Define Δ'_{adv}, $[\![u]\!]^p_{\text{adv}}$, $[\![\vec{\alpha}]\!]^p_{\text{adv}}$, $[\![\vec{\beta}]\!]^p_{\text{adv}}$, $[\![\vec{\gamma}]\!]^p_{\text{adv}}$, $[\![\vec{b}]\!]^p_{\text{adv}}$ analogously to the above.

Obtain Shares of $[\![\Delta]\!]^p$:

1. Run $[\![\Delta^*]\!]^p_{\text{adv}}, \langle\vec{u}\rangle^p_h \leftarrow \mathsf{Sim}^p_{\text{auth}}(\Delta'_{\text{adv}}, [\![u]\!]^p_{\text{adv}})$.

Obtain Shares of Random Bits:

1. Run $[\![\vec{b'}]\!]_{\text{adv}} \leftarrow \mathsf{Sim}_{\text{bits}}(\Delta_{\text{adv}}, \Delta'_{\text{adv}}, [\![\Delta^*]\!]^p_{\text{adv}}, [\![\vec{\alpha}]\!]^p_{\text{adv}}, [\![\vec{\beta}]\!]^p_{\text{adv}}, [\![\vec{\gamma}]\!]^p_{\text{adv}}, [\![\vec{b}]\!]^p_{\text{adv}}, [\![\vec{u}]\!]_{\text{adv}})$, and split $[\![\vec{b'}]\!]_{\text{adv}}$ into a vector $[\![\vec{b}]\!]_{\text{adv}}$ of dimension m_B, and κ vectors $\{[\![\vec{b}_j]\!]_{\text{adv}}\}_{j\in[\kappa]}$ of dimension ℓ.
2. Choose arbitrary corrupted servers' shares $[\![\vec{b}]\!]_i$ for $i \neq h$ such that they sum to $[\![\vec{b}]\!]_{\text{adv}}$, and send those to $\mathcal{F}_{\text{prep}}$.

Obtain Shares of Masks:

1. Run $[\![\vec{r}]\!]_{\text{adv}} \leftarrow \mathsf{Sim}_{\text{masks}}(\Delta_{\text{adv}}, [\![\vec{v}]\!]_{\text{adv}}, \{[\![\vec{b}_j]\!]_{\text{adv}}\})$.
2. Choose arbitrary corrupted servers' shares $[\![\vec{r}]\!]_i$ for $i \neq h$ such that they sum to $[\![\vec{r}]\!]_{\text{adv}}$, and send those to $\mathcal{F}_{\text{prep}}$.

Simulator $\mathsf{Sim}^R_{\text{auth}}(\Delta_{\text{adv}}, [\![\vec{u}]\!]^R_{\text{adv}})$:

1. Sample $\langle\vec{u}\rangle^R_h$ at random (over $\mathbb{Z}_R$), and send $\langle\vec{s}\rangle^R_h = \langle\vec{u}\rangle^R_h$ to $\mathcal{A}$. Receive $\vec{s}$ from $\mathcal{A}$.
2. Let $[\![1]\!]^R_{\text{adv}} = (1, \Delta_{\text{adv}})$. Output $[\![\vec{v}]\!]^R_{\text{adv}} = [\![1]\!]^R_{\text{adv}} \cdot \vec{s} - [\![\vec{u}]\!]^R_{\text{adv}}$ and $\langle\vec{u}\rangle^R_h$.

Simulator $\mathsf{Sim}^R_{\text{bits}}(\Delta_{\text{adv}}, \Delta'_{\text{adv}}, [\![\Delta^*]\!]^p_{\text{adv}}, [\![\vec{\alpha}]\!]^p_{\text{adv}}, [\![\vec{\beta}]\!]^p_{\text{adv}}, [\![\vec{\gamma}]\!]^p_{\text{adv}}, [\![\vec{b}]\!]^p_{\text{adv}}, [\![\vec{u}]\!]_{\text{adv}})$:

1. Run $([\![\Delta^*\cdot\vec{b}]\!]^p_{\text{adv}}, \vec{e}, \vec{e'}, \vec{d}, \vec{d'}) \leftarrow \mathsf{Sim}_{\text{mul}}([\![\Delta^*]\!]^p_{\text{adv}}, [\![\vec{b}]\!]^p_{\text{adv}}, [\![\vec{\alpha}]\!]^p_{\text{adv}}, [\![\vec{\beta}]\!]^p_{\text{adv}}, [\![\vec{\gamma}]\!]^p_{\text{adv}})$.
2. Let $\vec{f} = \vec{e}\|\vec{d}$, $\vec{f'} = \vec{e'}\|\vec{d'}$, and $[\![\vec{f}]\!]^p_{\text{adv}} = [\![\vec{e}]\!]^p_{\text{adv}}\|[\![\vec{d}]\!]^p_{\text{adv}}$, where

$$[\![\vec{e}]\!]^p_{\text{adv}} \leftarrow [\![\Delta^*]\!]^p_{\text{adv}} - [\![\vec{\alpha}]\!]^p_{\text{adv}}, \quad [\![\vec{d}]\!]^p_{\text{adv}} \leftarrow [\![\vec{b}]\!]^p_{\text{adv}} - [\![\vec{\beta}]\!]^p_{\text{adv}}.$$

Run $\mathsf{Sim}^{p}_{\mathrm{MAC}}(\Delta'_{\mathrm{adv}}, \vec{f}, \vec{f}', [\![\vec{f}]\!]^{p}_{\mathrm{adv}})$.

3. Emulate $\mathcal{F}_{\mathrm{coin}}$ to sample a seed and send it to $\mathcal{A}$. Expand it to two public random shares of zeros denoted $\langle\vec{o}\rangle^{p}, \langle\vec{o}'\rangle^{p}$. Define $\langle\vec{o}\rangle^{p}_{\mathrm{adv}} = \sum_{i\neq h}\langle\vec{o}\rangle^{p}_{i}$, and $\langle\vec{o}'\rangle^{p}_{\mathrm{adv}}$ analogously.

4. Compute

$$\mathrm{tmp}_{\mathrm{adv}} = (\langle\vec{b}\rangle^{p}_{\mathrm{adv}} + \langle\vec{o}\rangle^{p}_{\mathrm{adv}} \bmod p), \quad \mathrm{tmp}'_{\mathrm{adv}} = (\langle\Delta^{*}\cdot\vec{b}\rangle^{p}_{\mathrm{adv}} + \langle\vec{o}'\rangle^{p}_{\mathrm{adv}} \bmod p),$$

and run $\mathsf{Sim}^{t}_{\mathrm{ModToInt}}(\mathrm{tmp}_{\mathrm{adv}})$ and $\mathsf{Sim}^{t}_{\mathrm{ModToInt}}(\mathrm{tmp}'_{\mathrm{adv}})$.

5. Compute $[\![\vec{b}]\!]_{\mathrm{adv}} = (\langle\vec{b}\rangle_{\mathrm{adv}}, \langle\Delta\cdot\vec{b}\rangle_{\mathrm{adv}})$ as follows

$$\langle\vec{b}\rangle_{\mathrm{adv}} = \mathrm{tmp}_{\mathrm{adv}} - p, \quad \langle\Delta\cdot\vec{b}\rangle_{\mathrm{adv}} = \mathrm{tmp}'_{\mathrm{adv}} - c\cdot p,$$

where the "overflow" bit c is determined by

$$c = \begin{cases} 1 & \text{if } \mathrm{tmp}'_{\mathrm{adv}} \geq \Delta_{\mathrm{adv}} \\ 0 & \text{o/w} \end{cases} \tag{2}$$

6. Sample $\langle\vec{u}\rangle_{h}$ at random (over $\mathbb{Z}_{N^{\zeta}}$), and send $\langle\vec{s}\rangle_{h} = \langle\vec{u}\rangle_{h}$ to $\mathcal{A}$. Receive $\vec{s}'$ from $\mathcal{A}$.

7. Compute

$$\vec{s} \leftarrow \langle\vec{s}\rangle_{h} + \langle\vec{u}\rangle_{\mathrm{adv}} + \langle\vec{b}\rangle_{\mathrm{adv}}, \quad [\![\vec{s}]\!]_{\mathrm{adv}} \leftarrow [\![\vec{b}]\!]_{\mathrm{adv}} + [\![\vec{u}]\!]_{\mathrm{adv}},$$

and run $\mathsf{Sim}'^{N^{\zeta}}_{\mathrm{MAC}}(\Delta_{\mathrm{adv}}, \vec{s}, \vec{s}', [\![\vec{s}]\!]_{\mathrm{adv}})$.

Simulator $\mathsf{Sim}_{\mathrm{mul}}([\![x]\!]_{\mathrm{adv}}, [\![y]\!]_{\mathrm{adv}}, [\![a]\!]_{\mathrm{adv}}, [\![b]\!]_{\mathrm{adv}}, [\![c]\!]_{\mathrm{adv}})$**:**

1. $\mathcal{S}$ computes corrupted servers' shares $[\![e]\!]_{\mathrm{adv}} = [\![x]\!]_{\mathrm{adv}} - [\![a]\!]_{\mathrm{adv}}$ and $[\![d]\!]_{\mathrm{adv}} = [\![y]\!]_{\mathrm{adv}} - [\![b]\!]_{\mathrm{adv}}$.

2. $\mathcal{S}$ samples random honest shares $\langle e\rangle_{h}, \langle d\rangle_{h}$ from $\mathbb{Z}_{N^{\zeta}}$, and computes e and d based on the corrupted servers' shares of $\langle e\rangle$ and $\langle d\rangle$.

3. $\mathcal{S}$ simulates $\Pi_{\mathrm{open}}([\![e]\!])$ and $\Pi_{\mathrm{open}}([\![d]\!])$:

   (a) $\mathcal{S}$ sends $\langle e\rangle_{h}, \langle d\rangle_{h}$ to P_1 (controlled by $\mathcal{A}$).

   (b) $\mathcal{S}$ receives the reconstructed values e' and d' from $\mathcal{A}$.

4. $\mathcal{S}$ computes the corrupt parties' SPDZ shares of the product $z = x\cdot y$ as

$$[\![z]\!]_{\mathrm{adv}} = [\![c]\!]_{\mathrm{adv}} + d\cdot[\![a]\!]_{\mathrm{adv}} + e\cdot[\![b]\!]_{\mathrm{adv}} + e\cdot d$$

5. Output $[\![z]\!]_{\mathrm{adv}}$ and e, e', d, d'.

Simulator $\mathsf{Sim}^{R}_{\mathrm{MAC}}(\Delta'_{\mathrm{adv}}, \vec{s}, \vec{s}', [\![\vec{s}]\!]^{R}_{\mathrm{adv}})$**:**

1. Emulate $\mathcal{F}_{\mathrm{coin}}$ to sample a seed and send it to $\mathcal{A}$. Expand it into coefficients $\vec{\chi}$ over $\mathbb{Z}_R$.

2. Compute $s \leftarrow \vec{\chi}^{\intercal}\cdot\vec{s}$, $s' \leftarrow \vec{\chi}^{\intercal}\cdot\vec{s}'$ (mod R), and $[\![s]\!]^{R}_{\mathrm{adv}} \leftarrow \vec{\chi}^{\intercal}\cdot[\![\vec{s}]\!]^{R}_{\mathrm{adv}}$.

3. Emulate $\mathcal{F}_{\text{commit}}$ by notifying $\mathcal{A}$ that P_h has committed, and receiving committed values $\{\langle\sigma\rangle_i^R\}_{i\neq h}$ from $\mathcal{A}$.

4. Compute $\langle\sigma\rangle_{\text{adv}}^R \leftarrow \sum_{i\neq h}\langle\sigma\rangle_i^R$, and check: $\langle\sigma\rangle_{\text{adv}}^R \stackrel{?}{=} \langle\Delta\cdot s\rangle_{\text{adv}}^R - s\cdot\Delta_{\text{adv}}$.

   – If $\vec{s}=\vec{s}'$ and check passes, set $\langle\sigma\rangle_h^R \leftarrow -\langle\sigma\rangle_{\text{adv}}^R$.
   – Otherwise, send abort to $\mathcal{F}_{\text{prep}}$, (and receive Δ from $\mathcal{F}_{\text{prep}}$). Sample $\Delta' \leftarrow \mathbb{Z}_R$, and set $\Delta'_h = \Delta' - \Delta'_{\text{adv}}$. Finally, compute $\langle\sigma\rangle_h^R \leftarrow \Delta'\cdot s - \langle\Delta'\cdot s\rangle_{\text{adv}}^R - \Delta'_h\cdot s'$.

5. Emulate $\mathcal{F}_{\text{commit}}$ by opening $\langle\sigma\rangle_i^R$ to $\mathcal{A}$ for every $i\in[n]$. If the previous check failed, abort the simulation with $\mathcal{A}$.

Simulator $\mathsf{Sim}'^{N\zeta}_{\text{MAC}}(\Delta_{\text{adv}}, \vec{s}, \vec{s}', [\![\vec{s}]\!]_{\text{adv}})$**:** The same as $\mathsf{Sim}^{N\zeta}_{\text{MAC}}$ except in Step 4.

4. – Otherwise, send abort to $\mathcal{F}_{\text{prep}}$, (and receive Δ from $\mathcal{F}_{\text{prep}}$). Compute $\Delta_h = \Delta - \Delta_{\text{adv}}$, and $\langle\sigma\rangle_h \leftarrow \Delta\cdot s - \langle\Delta\cdot s\rangle_{\text{adv}} - \Delta_h\cdot s'$.

Simulator $\mathsf{Sim}^R_{\text{masks}}(\Delta_{\text{adv}}, [\![\vec{v}]\!]_{\text{adv}}, \{[\![\vec{b}_j]\!]_{\text{adv}}\})$**:**

1. Run $[\![\vec{r}]\!]_{\text{adv}}, \langle\vec{v}\rangle_h \leftarrow \mathsf{Sim}^{N\zeta}_{\text{auth}}(\Delta_{\text{adv}}, [\![\vec{v}]\!]_{\text{adv}})$.
2. Compute $[\![M_j]\!]_{\text{adv}} \leftarrow \sum_{k\in[\ell]} 2^k\cdot[\![\vec{b}_j[k]]\!]_{\text{adv}}$ for $j\in[\kappa]$.
3. Emulate $\mathcal{F}_{\text{coin}}$ to sample a seed and send it to $\mathcal{A}$. Expand it into coefficient vectors $\{\vec{\chi}_j\}_{j\in[\kappa]}$ over $\{0,1\}$.
4. For $j\in[\kappa]$: Sample $M_j \leftarrow [0,2^\ell)$, set $\langle M_j\rangle_h \leftarrow M_j - \langle M_j\rangle_{\text{adv}}$, and send $\langle V_j\rangle_h = \langle M_j\rangle_h$ to $\mathcal{A}$. Receive $\{V'_j\}$ from $\mathcal{A}$. Abort the simulation if there exists $V'_j \notin [0, r^{\text{relax}}_{\max})$.
5. Compute

$$V_j \leftarrow \chi_j^\intercal\cdot(\langle\vec{r}\rangle_{\text{adv}} - \vec{v}_h) + M_j, \quad [\![V_j]\!]_{\text{adv}} \leftarrow \chi_j^\intercal\cdot[\![\vec{r}]\!]_{\text{adv}} + [\![M_j]\!]_{\text{adv}}.$$

   Organize $\{V'_j\}$ as a vector $\vec{V}'$, $\{V_j\}$ as $\vec{V}$, and $\{[\![V_j]\!]_{\text{adv}}\}$ as $[\![\vec{V}]\!]_{\text{adv}}$.
6. Run $\mathsf{Sim}'^{N\zeta}_{\text{MAC}}(\Delta_{\text{adv}}, \vec{V}, \vec{V}', [\![\vec{V}]\!]_{\text{adv}})$.

We define the following hybrid experiments. Hyb_1 corresponds to the real-world execution of Π_{prep} between an honest server P_h, where $h\neq 1$, and an adversary $\mathcal{A}$ corrupting all other servers. Hyb_4 corresponds to the ideal-world execution where $\mathcal{F}_{\text{prep}}$ interacts with a dummy honest server P_h and the simulator $\mathcal{S}$ (which runs $\mathcal{A}$ internally).

Hyb_1: The experiment executes the protocol as P_h with $\mathcal{A}$.

Hyb_2: The experiment proceeds in the same way as the simulator $\mathcal{S}$ during **Initialization** and **Obtaining Shares of** $[\![\Delta]\!]^p$. In particular, it obtains corrupted servers' shares Δ_{adv}, Δ'_{adv}, $[\![\vec{v}]\!]_{\text{adv}}$, $[\![\vec{u}]\!]_{\text{adv}}$, $[\![\vec{\alpha}]\!]^p_{\text{adv}}$, $[\![\vec{\beta}]\!]^p_{\text{adv}}$, $[\![\vec{\gamma}]\!]^p_{\text{adv}}$, $[\![\vec{b}]\!]^p_{\text{adv}}$, and $[\![\Delta^*]\!]^p_{\text{adv}}$. It has also sampled $\langle u\rangle_h^p$ during $\mathsf{Sim}^p_{\text{auth}}$.

It then proceeds as in the real execution during **Obtaining Shares of Random Bits** and **Obtaining Shares of Masks**:

1. Sample $\vec{v} \leftarrow \mathbb{Z}_{N^\zeta}^{m_M}$, $\vec{u} \leftarrow \mathbb{Z}_{N^\zeta}^{m'_B}$, $\vec{\alpha}, \vec{\beta} \leftarrow \mathbb{Z}_p^{m'_B}$, $\vec{b} \leftarrow \{0,1\}^{m'_B}$, $\Delta_h \leftarrow [0, \Delta_{\max})$, and $\Delta'_h \leftarrow \mathbb{Z}_p$ (which defines $[\![1]\!]_h$ and $[\![1]\!]_h^p$). Set $\Delta = \Delta_h + \Delta_{\text{adv}}$, $\Delta' = \Delta'_h + \Delta'_{\text{adv}}$ and $\vec{\gamma} = \vec{\alpha} \odot \vec{\beta}$, where $\odot$ denotes component-wise multiplication.
2. Derive $[\![\vec{v}]\!]_h$ (from $\vec{v}$, Δ, and $[\![\vec{v}]\!]_{\text{adv}}$, i.e., $[\![\vec{v}]\!]_h + [\![\vec{v}]\!]_{\text{adv}} = (\vec{v}, \Delta \cdot \vec{v})$). Similarly, derive $[\![\vec{u}]\!]_h$, $[\![\vec{\alpha}]\!]_h^p$, $[\![\vec{\beta}]\!]_h^p$, $[\![\vec{\gamma}]\!]_h^p$, and $[\![\vec{b}]\!]_h^p$.
3. Set $\Delta^* = \langle \Delta^* \rangle_{\text{adv}}^p - \langle u \rangle_h^p + \Delta_h$, and derive $[\![\Delta^*]\!]_h^p$ from Δ^*, Δ' and $[\![\Delta^*]\!]_{\text{adv}}^p$.
4. Run Π_{bits}, Π_{masks} as P_h with $\mathcal{A}$ following the protocol.

Hyb_3: The experiment proceeds in the same way as the simulator $\mathcal{S}$ during **Initialization**, **Obtaining Shares of** $[\![\Delta]\!]^p$, and **Obtaining Shares of Random Bits**. In particular, it obtains corrupted servers' shares Δ_{adv}, $[\![\vec{v}]\!]_{\text{adv}}$, and $\{[\![\vec{b}_j]\!]_{\text{adv}}\}_{j \in [\kappa]}$.

It then proceeds as in the real execution during **Obtaining Shares of Masks**:

1. Sample $\vec{v} \leftarrow \mathbb{Z}_{N^\zeta}^{m_M}$, $\vec{b}_j \leftarrow \{0,1\}^\ell$ for $j \in [\kappa]$, and $\Delta_h \leftarrow [0, \Delta_{\max})$ (which defines $[\![1]\!]_h$). Set $\Delta = \Delta_h + \Delta_{\text{adv}}$.
2. Derive $[\![\vec{v}]\!]_h$ from $\vec{v}$, Δ, $[\![\vec{v}]\!]_{\text{adv}}$, and $[\![\vec{b}_j]\!]_h$ from $\vec{b}$, Δ, $[\![\vec{b}]\!]_{\text{adv}}$ for $j \in [\kappa]$.
3. Run Π_{masks} as P_h with $\mathcal{A}$ following the protocol.

Hyb_4: The experiment proceeds in the same way as the simulator $\mathcal{S}$.

We conclude the proof by showing (statistical) indistinguishability between the adjacent hybrid experiments.

Claim B.6. $\mathsf{Hyb}_1 \equiv \mathsf{Hyb}_2$.

Proof. We need to argue the simulated honest server's shares Δ_h, Δ'_h, $[\![\vec{v}]\!]_h$, $[\![\vec{u}]\!]_h$, $[\![\vec{\alpha}]\!]_h^p$, $[\![\vec{\beta}]\!]_h^p$, $[\![\vec{\gamma}]\!]_h^p$, $[\![\vec{b}]\!]_h^p$, and $[\![\Delta^*]\!]_h^p$ during **Initialization** and **Obtaining Shares of** $[\![\Delta]\!]^p$ are identical to those in the real execution. All shares except $[\![\Delta^*]\!]_h^p$ are indeed computed exactly as in the real execution. We now examine $[\![\Delta^*]\!]_h^p$:

- In the real execution (Hyb_1), P_h sends $\langle s \rangle_h^p \leftarrow \Delta_h + \langle u \rangle_h^p$ to $\mathcal{A}$, where $\langle u \rangle_h^p$ is uniformly random. It receives s from $\mathcal{A}$ and computes $u = \langle u \rangle_h^p + \langle u \rangle_{\text{adv}}^p$ and

$$[\![\Delta^*]\!]_h^p \leftarrow [\![1]\!]_h^p \cdot s - [\![u]\!]_h^p = (1, \Delta') \cdot (s - u) - ([\![1]\!]_{\text{adv}}^p \cdot s - [\![u]\!]_{\text{adv}}^p).$$

  We define $\Delta^* = (s - u)$ and $[\![\Delta^*]\!]_{\text{adv}}^p = [\![1]\!]_{\text{adv}}^p \cdot s - [\![u]\!]_{\text{adv}}^p$.

- Consider another hybrid Hyb'_1 where we define an intermediate value $u_h = \langle u \rangle_h^p + \Delta_h$. We now have $\langle s \rangle_h^p = u_h$, and $u = u_h + \langle u \rangle_{\text{adv}}^p - \Delta_h$ and

$$\Delta^* = s - u = s - u_h - \langle u \rangle_{\text{adv}}^p + \Delta_h = \langle \Delta^* \rangle_{\text{adv}} + \underbrace{\langle 1 \rangle_h^p}_{0} - u_h + \Delta_h.$$

- We note that Hyb'_1 is exactly the same as Hyb_2 when renaming u_h to $\langle u \rangle_h^p$.

□

Claim B.7. $\mathsf{Hyb}_2 \approx_c \mathsf{Hyb}_3$.

Proof. We need to argue the honest server's shares Δ_h, $[\![\vec{v}]\!]_h$, and $[\![\vec{b}]\!]_h$ in Hyb_3 are statistically indistinguishable from those in Hyb_2. Since Δ_h, $[\![\vec{v}]\!]_h$ are indeed computed exactly in the same way, we focus on $[\![\vec{b}]\!]_h$. We show the following hybrid experiments.

– Hyb_2: Recall that $[\![\vec{b}]\!]_h$ are computed in three steps. First, P_h participates in Π^p_{Mul} with inputs $[\![\Delta^*]\!]^p_h, [\![\vec{b}]\!]^p_h, [\![\vec{\alpha}]\!]^p, [\![\vec{\beta}]\!]^p, [\![\vec{\gamma}]\!]^p$. In particular, P_h computes $\langle\vec{e}\rangle^p_h \leftarrow \langle\Delta^*\rangle^p_h \cdot \vec{1} - \langle\vec{\alpha}\rangle^p_h$, $\langle\vec{d}\rangle^p_h \leftarrow \langle\vec{b}\rangle^p_h - \langle\vec{\beta}\rangle^p_h$, and sends them to $\mathcal{A}$. It receives $\vec{e}', \vec{d}'$ from $\mathcal{A}$, and computes $\vec{\gamma}' = \vec{e}' \odot \vec{d}'$,

$$[\![\Delta^* \cdot \vec{b}]\!]^p_h \leftarrow [\![1]\!]^p_h \cdot \vec{\gamma}' + [\![\vec{b}]\!]^p_h \odot \vec{e}' + [\![\Delta^*]\!]^p_h \odot \vec{d}' + [\![\vec{\gamma}]\!]^p_h.$$

It then computes $[\![\vec{e}]\!]^p_h \leftarrow [\![\Delta^*]\!]^p_h \cdot \vec{1} - [\![\vec{\alpha}]\!]^p_h$, $[\![\vec{d}]\!]^p_h \leftarrow [\![b]\!]^p_h - [\![\vec{\beta}]\!]^p_h$, and participates in $\Pi^p_{\text{SPDZ}-\text{MAC}}$ with inputs $\vec{e'}$, $\vec{d'}$, $[\![\vec{e}]\!]_h$, $[\![\vec{d}]\!]_h$.

Second, P_h participates in $\Pi^{p,t}_{\text{ModToInt}}$ with inputs $\langle\vec{b}\rangle^p_h$ and $\langle\Delta^* \cdot \vec{b}\rangle^p_h$. In particular, P_h obtains a seed from $\mathcal{F}_{\text{coin}}$, and expands it into two public random shares of zeros $\langle\vec{z}\rangle^p$, $\langle\vec{z}'\rangle^p$. It computes

$$\begin{aligned} \text{tmp}_h = \langle\vec{b}\rangle^p_h + \langle\vec{o}\rangle^p_h \bmod p, \quad \text{tmp}'_h = \langle\Delta^* \cdot \vec{b}\rangle^p_h + \langle\vec{o}\rangle'^p_h \bmod p, \\ \vec{\theta} \leftarrow \lfloor \text{tmp}_h / 2^t \rfloor, \qquad\qquad \vec{\theta}' \leftarrow \lfloor \text{tmp}'_h / 2^t \rfloor, \end{aligned} \tag{3}$$

and sends them to $\mathcal{A}$. It defines $[\![\vec{b}]\!]_h = (\text{tmp}_h, \text{tmp}'_h)$.

Third, P_h computes $\langle\vec{s}\rangle_h \leftarrow \langle\vec{b}\rangle_h + \langle\vec{u}\rangle_h$ and sends them to $\mathcal{A}$. It receives $\vec{s}'$ from $\mathcal{A}$, computes $[\![\vec{s}]\!]_h \leftarrow [\![\vec{b}]\!]_h + [\![\vec{u}]\!]_h$, and participates in $\Pi^{N\zeta}_{\text{SPDZ}-\text{MAC}}$ with inputs $\vec{s}'$, $[\![\vec{s}]\!]_h$.

– $\mathsf{Hyb}_{2,1}$: We define

$$\begin{aligned} [\![\vec{e}]\!]^p_{\text{adv}} \leftarrow [\![\Delta^*]\!]^p_{\text{adv}} \cdot \vec{1} - [\![\vec{\alpha}]\!]^p_{\text{adv}}, \qquad [\![\vec{d}]\!]^p_{\text{adv}} \leftarrow [\![\vec{b}]\!]^p_{\text{adv}} - [\![\vec{\beta}]\!]_{\text{adv}} \\ \vec{e} \leftarrow \langle\vec{e}\rangle^p_h + \langle\vec{e}\rangle^p_{\text{adv}} = \Delta^* \cdot \vec{1} - \vec{\alpha}, \qquad \vec{d} \leftarrow \langle\vec{d}\rangle^p_h + \langle\vec{d}\rangle^p_{\text{adv}} = \vec{b} - \vec{\beta}, \end{aligned}$$

and replace invocation of $\Pi^p_{\text{SPDZ}-\text{MAC}}$ with $\mathsf{Sim}^p_{\text{MAC}}(\Delta'_{\text{adv}}, \vec{e} \| \vec{d}, \vec{e}' \| \vec{d}', [\![\vec{e}]\!]^p_{\text{adv}} \| [\![\vec{d}]\!]^p_{\text{adv}})$. By standard MAC-check arguments for SPDZ shares (see Hyb_6–Hyb_9 in Section B.2 for example, which relies on Δ'_h being hidden from $\mathcal{A}$ prior to invoking $\Pi^p_{\text{SPDZ}-\text{MAC}}$), we have $\mathsf{Hyb}_2 \approx_c \mathsf{Hyb}_{2,1}$, assuming the security of the PRG used for seed expansion in $\Pi^p_{\text{SPDZ}-\text{MAC}}$.

– $\mathsf{Hyb}_{2,2}$: We define intermediate values

$$\vec{z} \leftarrow \vec{\gamma}' + \vec{b} \odot \vec{e}' + \Delta^* \cdot \vec{d}' + \vec{\gamma}, \quad [\![\Delta^* \cdot \vec{b}]\!]^p_{\text{adv}} \leftarrow [\![1]\!]^p_{\text{adv}} \cdot \vec{\gamma}' + [\![\vec{b}]\!]^p_{\text{adv}} \odot \vec{e}' + [\![\Delta^*]\!]^p_{\text{adv}} \odot \vec{d}' + [\![\vec{\gamma}]\!]^p_{\text{adv}}.$$

We have $[\![\Delta^* \cdot \vec{b}]\!]^p_h = (1, \Delta') \otimes \vec{z} - [\![\Delta^* \cdot \vec{b}]\!]^p_{\text{adv}}$. Since $\mathsf{Sim}^p_{\text{MAC}}$ aborts unless $\vec{e}' = \vec{e}$ and $\vec{d}' = \vec{d}$, we can directly replace $\vec{e}', \vec{d}'$ with $\vec{e}, \vec{d}$ and set

$$\vec{z} = \vec{\gamma} + \vec{b} \odot \vec{e} + \Delta^* \cdot \vec{d} + \vec{\gamma} = \Delta^* \cdot \vec{b}.$$

We have $\mathsf{Hyb}_{2,1} \equiv \mathsf{Hyb}_{2,2}$. Note that $\vec{\gamma} = \vec{\alpha} \odot \vec{\beta}$ are no longer used anywhere in $\mathsf{Hyb}_{2,2}$.

– $\mathsf{Hyb}_{2,3}$: Instead of computing $\langle\vec{e}\rangle^p_h \leftarrow \langle\Delta^*\rangle^p_h \cdot \vec{1} - \langle\alpha\rangle^p_h$ and $\langle\vec{d}\rangle^p_h \leftarrow \langle\vec{b}\rangle^p_h - \langle\vec{\beta}\rangle^p_h$, compute them as $\langle\vec{e}\rangle^p_h \leftarrow -\langle\alpha\rangle^p_h$ and $\langle\vec{d}\rangle^p_h \leftarrow -\langle\vec{\beta}\rangle^p_h$. By the randomness of $\vec{\alpha}, \vec{\beta}$ (and hence that of $\langle\vec{\alpha}\rangle^p_h, \langle\vec{\beta}\rangle^p_h$), we have $\mathsf{Hyb}_{2,3} \equiv \mathsf{Hyb}_{2,2}$.

Note that in $\mathsf{Hyb}_{2,3}$, the steps for running Π^p_{Mul} and $\Pi^p_{\text{SPDZ}-\text{MAC}}$ have now been replaced by running $\mathsf{Sim}_{\text{mul}}$ and $\mathsf{Sim}_{\text{MAC}}$. In particular, it first runs $\mathsf{Sim}_{\text{mul}}$ and $\mathsf{Sim}^p_{\text{MAC}}$ as in the

simulator $\mathcal{S}$ to derive $[\![\Delta^* \cdot \vec{b}]\!]^p_{\text{adv}}$ using only Δ'_{adv}, $[\![\Delta^*]\!]^p_{\text{adv}}$, $[\![\vec{b}]\!]^p_{\text{adv}}$, $[\![\vec{\alpha}]\!]^p_{\text{adv}}$, $[\![\vec{\beta}]\!]^p_{\text{adv}}$, and $[\![\vec{\gamma}]\!]^p_{\text{adv}}$. It then computes

$$[\![\Delta^* \cdot \vec{b}]\!]^p_h \leftarrow (1, \Delta') \otimes \Delta^* \cdot \vec{b} - [\![\Delta^* \cdot \vec{b}]\!]^p_{\text{adv}},$$

and continues as in Hyb_2.

– $\mathsf{Hyb}_{2,4}$: Instead of computing $\vec{\theta}, \vec{\theta}'$ from $\langle \vec{b} \rangle^p_h, \langle \Delta^* \cdot \vec{b} \rangle^p_h$ as in Eq 3, run $\mathsf{Sim}^t_{\text{ModToInt}}(\text{tmp}_{\text{adv}})$, $\mathsf{Sim}^t_{\text{ModToInt}}(\text{tmp}'_{\text{adv}})$ instead, where

$$\text{tmp}_{\text{adv}} = (\langle \vec{b} \rangle^p_{\text{adv}} + \langle \vec{o} \rangle^p_{\text{adv}} \bmod p), \quad \text{tmp}'_{\text{adv}} = (\langle \Delta^* \cdot \vec{b} \rangle^p_{\text{adv}} + \langle \vec{o}' \rangle^p_{\text{adv}} \bmod p).$$

By Lemma B.4, we have $\mathsf{Hyb}_{2,4} \approx_s \mathsf{Hyb}_{2,3}$.

– $\mathsf{Hyb}_{2,5}$: Instead of setting $[\![\vec{b}]\!]_h = (\text{tmp}_h, \text{tmp}'_h)$ as in Eq 3, set it as $[\![\vec{b}]\!]_h = (\vec{b} - \langle \vec{b} \rangle_{\text{adv}}, \Delta^* \cdot \vec{b} - \langle \Delta^* \cdot \vec{b} \rangle_{\text{adv}})$, where

$$\langle \vec{b} \rangle_{\text{adv}} \leftarrow \text{tmp}_{\text{adv}} - p, \quad \langle \Delta^* \cdot \vec{b} \rangle_{\text{adv}} \leftarrow \text{tmp}'_{\text{adv}} - c \cdot p,$$

and

$$c = \begin{cases} 1 & \text{if } \text{tmp}'_{\text{adv}} \geq (\Delta^* - \Delta_h) = \langle \Delta^* \rangle^p_{\text{adv}} - \langle u \rangle^p_h, \\ 0 & \text{o/w.} \end{cases} \tag{4}$$

By our setting, we have $\|\vec{b}\|_\infty \ll p$, so over the (pseudo-)randomness of $\langle \vec{o} \rangle^p_h$, we have $\text{tmp}_h > \vec{b}$ (except with negligible probability), hence

$$\text{tmp}_h = \vec{b} - \text{tmp}_{\text{adv}} + p \text{ (over } \mathbb{Z}).$$

This shows that the new way of computing $\langle \vec{b} \rangle_h$ is equivalent to the old way. Similarly, by our setting we have $\|\Delta_h \cdot \vec{b}\|_\infty \ll p$, so $\text{tmp}_h > \Delta_h \cdot \vec{b}$. We have

$$\text{tmp}'_h = \Delta_h \cdot \vec{b} - \left(\text{tmp}'_{\text{adv}} - (\Delta^* - \Delta_h) \cdot \vec{b} \bmod p\right) + p \text{ (over } \mathbb{Z}).$$

We can verify that in both cases of $c = 1$ and $c = 0$, the new way of computing $\langle \Delta^* \cdot \vec{b} \rangle_h$ is equivalent to the old way. We conclude that $\mathsf{Hyb}_{2,5} \approx_s \mathsf{Hyb}_{2,4}$.

Note that in $\mathsf{Hyb}_{2,5}$, the steps for running Π^t_{ModToInt} have now been replaced by running $\mathsf{Sim}^t_{\text{ModToInt}}$ and deriving $[\![\vec{b}]\!]_{\text{adv}}$ as in the simulator $\mathcal{S}$, except during which the "overflow" bit c is computed by checking against $\langle \Delta^* \rangle^p_{\text{adv}} - \langle u \rangle^p_h$ (Eq 4) rather than against Δ_{adv} (Eq 2). The experiment then computes

$$[\![\vec{b}]\!]_h \leftarrow (1, \Delta^*) \otimes \vec{b} - [\![\vec{b}]\!]_{\text{adv}},$$

and continues as in Hyb_2 with the MAC-check steps which we recap below.

* P_h computes $[\![\vec{s}]\!]_h \leftarrow [\![\vec{b}]\!]_h + [\![\vec{u}]\!]_h$, and sends $\langle \vec{s} \rangle_h$ to $\mathcal{A}$. It receives $\vec{s}'$ from $\mathcal{A}$.
* P_h receives a seed from $\mathcal{F}_{\text{coin}}$ and expands it into (pseudo-)random coefficients $\vec{\chi}$ over $\mathbb{Z}_{N^\zeta}$. It computes

$$w \leftarrow \vec{\chi}^\intercal \cdot \vec{s}', \quad \langle \Delta \cdot w \rangle_h \leftarrow \vec{\chi}^\intercal \cdot \langle \Delta^* \cdot \vec{s} \rangle_h.$$

* P_h commits $\sigma_h \leftarrow \langle \Delta \cdot w \rangle_h - \Delta_h \cdot w$ to $\mathcal{F}_{\text{commit}}$, and then receives the corrupted server's commitments from $\mathcal{F}_{\text{commit}}$, whose sum is σ_{adv}.
* P_h checks if $\sigma_h = -\sigma_{\text{adv}} \pmod{N^\zeta}$, and aborts if not.

We next show that it must hold that $\langle \Delta^* \rangle^p_{\text{adv}} - \langle u \rangle^p_h = \Delta_{\text{adv}}$, and hence $\Delta^* = \Delta$ due to the MAC-check steps.

– $\mathsf{Hyb}_{2,6}$: Expand $\langle \Delta^* \cdot \vec{s} \rangle_h$ as

$$\langle \Delta^* \cdot \vec{s} \rangle_h = \underbrace{(\Delta_h + \langle \Delta^* \rangle^p_{\text{adv}} - \langle u \rangle^p_h)}_{\Delta^*} \cdot \vec{b} - \langle \Delta^* \cdot \vec{b} \rangle_{\text{adv}} + \underbrace{(\Delta_h + \Delta_{\text{adv}})}_{\Delta} \cdot \vec{u} - \langle \Delta \cdot \vec{u} \rangle_{\text{adv}},$$

we have

$$\begin{aligned}
\sigma_h &= \langle \Delta \cdot w \rangle_h - \Delta_h \cdot w \\
&= \Delta_h \cdot \vec{\chi}^\intercal \cdot (\vec{b} + \vec{u} - \vec{s}') && \text{// term 1} \\
&\quad + \vec{\chi}^\intercal \cdot \left((\langle \Delta^* \rangle^p_{\text{adv}} + \langle u \rangle^p_h) \cdot \vec{b} + \Delta_{\text{adv}} \cdot \vec{u} \right) && \text{// term 2} \\
&\quad - \vec{\chi}^\intercal \cdot (\langle \Delta^* \cdot \vec{b} \rangle_{\text{adv}} + \langle \Delta \cdot \vec{u} \rangle_{\text{adv}}). && \text{// term 3}
\end{aligned}$$

Due to the randomness of Δ_h, which is independent of $\mathcal{A}$'s view, the check can only pass (except with negligible probability) if term 1 is zero. Due to the (pseudo-)randomness of $\vec{\chi}$, this only happens when $\vec{s}' = \vec{b} + \vec{u}$ (except with negligible probability). So in $\mathsf{Hyb}_{2,6}$, the experiment instead checks $\vec{s}' = \vec{b} + \vec{u}$, and then sets

$$\begin{aligned}
\sigma_h &= \text{term 2} - \text{term 3} \\
&= (\langle \Delta^* \rangle^p_{\text{adv}} + \langle u \rangle^p_h - \Delta_{\text{adv}}) \cdot \vec{\chi}^\intercal \cdot \vec{b} && \text{// term } 2a \\
&\quad + \Delta_{\text{adv}} \cdot \vec{\chi}^\intercal \cdot \vec{s}' - \text{term 3}
\end{aligned}$$

We have $\mathsf{Hyb}_{2,6} \approx_c \mathsf{Hyb}_{2,5}$.

– $\mathsf{Hyb}_{2,7}$: Due to the randomness of $\vec{b}$, which is independent of $\mathcal{A}$'s view and the (pseudo-)randomness of $\vec{\chi}$, the check only passes (except with negligible probability) if term $2a$ is zero, which means $\langle \Delta^* \rangle^p_{\text{adv}} - \langle u \rangle^p_h = \Delta_{\text{adv}}$ and $\Delta^* = \Delta$ as desired. [10] So in $\mathsf{Hyb}_{2,7}$, the experiment instead checks $\langle \Delta^* \rangle^p_{\text{adv}} - \langle u \rangle^p_h = \Delta_{\text{adv}}$ and sets

$$\sigma_h = \Delta_{\text{adv}} \cdot \vec{\chi}^\intercal \cdot \vec{s}' - \vec{\chi}^\intercal \cdot (\langle \Delta^* \cdot \vec{b} \rangle_{\text{adv}} + \langle \Delta \cdot \vec{u} \rangle_{\text{adv}}).$$

We have $\mathsf{Hyb}_{2,7} \approx_c \mathsf{Hyb}_{2,6}$.

Note that $\mathsf{Hyb}_{2,7}$ is now equivalently distributed as Hyb_3, which concludes the proof of the claim.

□

Claim B.8. $\mathsf{Hyb}_3 \approx_c \mathsf{Hyb}_4$.

Proof. We need to argue the simulated honest server's shares of Δ_h, and of the bounded range masks $[\![\vec{r}]\!]_h$ in Hyb_3 are statistically indistinguishable from those in the ideal-world (Hyb_4) where $\mathcal{F}_{\text{prep}}$ interacts with the simulator $\mathcal{S}$. As Δ_h is indeed computed in the same way, we focus on $[\![\vec{r}]\!]_h$. We show the following hybrid experiments.

[10] This argument needs $\vec{b}$ to have dimension at least κ, which is true in our preprocessing protocol.

– Hyb_3: Recall that $[\![\vec{r}]\!]_h$ are computed in three steps. First, P_h locally samples $\langle \vec{r} \rangle_h \leftarrow [0, r_{\max})^m$ and participates in $\Pi_{\text{auth}}^{N^\zeta}$ with $\langle \vec{r} \rangle_h$ and $[\![\vec{v}]\!]_h$. It obtains $[\![\vec{r}]\!]_h$.

Second, P_h receives a seed from $\mathcal{F}_{\text{coin}}$, and expands it into κ (pseudo-)random bit vectors $\vec{\chi}_j$ for $j \in [\kappa]$. It then computes

$$[\![M_j]\!]_h \leftarrow \sum_{k \in [\ell]} 2^k \cdot [\![\vec{b}_j[k]]\!]_h, \quad [\![V_j]\!]_h \leftarrow \vec{\chi}^\intercal \cdot [\![\vec{r}]\!]_h + [\![M_j]\!]_h.$$

For $j \in [\kappa]$, it sends $\langle V_j \rangle_h$ to $\mathcal{A}$, receives V_j', and aborts if $V_j' \notin [0, r_{\max}^{\text{relax}})$.

Third, P_h organizes $\{V_j'\}$ and $\{[\![V_j]\!]_h\}$ into vectors $\vec{V}$, $[\![\vec{V}]\!]_h$, and participates in $\Pi_{\text{SPDZ-MAC}}^{N^\zeta}$ with inputs $\vec{V}$, $[\![\vec{V}]\!]_h$.

– $\mathsf{Hyb}_{3,1}$: Instead of $\Pi_{\text{auth}}^{N^\zeta}$, run $[\![\vec{r}]\!]_{\text{adv}}, \langle \vec{v} \rangle_h \leftarrow \mathsf{Sim}_{\text{auth}}^{N^\zeta}(\Delta_{\text{adv}}, [\![\vec{v}]\!]_{\text{adv}})$. Further, compute

$$\vec{r} \leftarrow \langle \vec{r} \rangle_{\text{adv}} - \langle v \rangle_h + \langle \vec{r} \rangle_h, \quad [\![\vec{r}]\!]_h \leftarrow (1, \Delta) \otimes \vec{r} - [\![\vec{r}]\!]_{\text{adv}}.$$

By analogous arguments to Claim B.6, we have $\mathsf{Hyb}_{3,1} \equiv \mathsf{Hyb}_3$.

– $\mathsf{Hyb}_{3,2}$: Define the following:

$$M_j \leftarrow \sum_{k \in [\ell]} 2^k \cdot \vec{b}_j[k], \qquad V_j \leftarrow \vec{\chi}^\intercal \cdot \vec{r} + M_j,$$
$$[\![M_j]\!]_{\text{adv}} \leftarrow \sum_{k \in [\ell]} 2^k \cdot [\![\vec{b}_j[k]]\!]_{\text{adv}}, \quad [\![V_j]\!]_{\text{adv}} \leftarrow \vec{\chi}^\intercal \cdot [\![\vec{r}]\!]_{\text{adv}} + [\![M_j]\!]_{\text{adv}},$$

and compute $\langle V_j \rangle$ as follows to send to $\mathcal{A}$:

$$\langle M_j \rangle_h \leftarrow M_j - \langle M_j \rangle_{\text{adv}}, \quad \langle V_j \rangle_h \leftarrow V_j - \langle V_j \rangle_{\text{adv}}.$$

Replace the invocation of $\Pi_{\text{SPDZ-MAC}}^{N^\zeta}$ with $\mathsf{Sim}_{\text{MAC}}^{N^\zeta}(\Delta_{\text{adv}}, \vec{V}, \vec{V}', [\![\vec{V}]\!]_{\text{adv}})$, where $\vec{V}$, $[\![\vec{V}]\!]_{\text{adv}}$ are re-organized from $\{V_j\}$ and $[\![\vec{V}]\!]_{\text{adv}}$. By standard MAC-check arguments for SPDZ shares (which rely on Δ_h being hidden from $\mathcal{A}$ prior to invoking $\Pi_{\text{SPDZ-MAC}}^{N^\zeta}$), we have $\mathsf{Hyb}_{3,2} \approx_c \mathsf{Hyb}_{3,1}$.

– $\mathsf{Hyb}_{3,3}$: Since $\mathsf{Sim}_{\text{MAC}}^{N^\zeta}$ aborts unless $V_j = V_j'$ for all $j \in [\kappa]$, the experiment can now directly check $V_j \notin [0, r_{\max}^{\text{relax}})$ instead of $V_j' \notin [0, r_{\max}^{\text{relax}})$. Additionally, the experiment also aborts unless all entries in $\vec{r} \pmod{N^\zeta}$ are within range $[r_{\max}^{\text{relax}}, N^\zeta - r_{\max}^{\text{relax}}]$. By Lemma B.2, the additional abort condition on $\vec{r}$ has no effect except with negligible probability, so we have $\mathsf{Hyb}_{3,3} \approx_s \mathsf{Hyb}_{3,2}$.

– $\mathsf{Hyb}_{3,4}$: Sample M_j instead as $M_j \leftarrow [0, 2^\ell)$. By the randomness of the bit vectors $\{\vec{b}_j\}$, we have $\mathsf{Hyb}3, 4 \equiv \mathsf{Hyb}3, 3$.

– $\mathsf{Hyb}_{3,5}$: Define $\tilde{M}_j = M_j + \vec{\chi}^\intercal \cdot \langle \vec{r} \rangle_h$. We have

$$V_j = \vec{\chi}^\intercal \cdot (\vec{r} - \langle \vec{r} \rangle_h) + \tilde{M}_j = \vec{\chi}^\intercal \cdot (\langle \vec{r} \rangle_{\text{adv}} - \langle v \rangle_h) + \tilde{M}_j,$$

and $\mathsf{Hyb}_{3,5} \equiv \mathsf{Hyb}_{3,4}$.

– $\mathsf{Hyb}_{3,6}$: Sample $\tilde{M}_j \leftarrow [0, 2^\ell)$ directly. By our setting, we have $\vec{\chi}^\intercal \cdot \langle \vec{r} \rangle_h \ll 2^\ell$, so the distribution of $\tilde{M}_j$ is statistically close to that in $\mathsf{Hyb}_{3,5}$, i.e., $\mathsf{Hyb}_{3,6} \approx_s \mathsf{Hyb}_{3,5}$.

  Note that $\mathsf{Hyb}_{3,6}$ is now equivalently distributed as Hyb_4, which concludes the proof of the claim.

Finally, it's straightforward to verify an honest execution of the protocol does not abort (except with negligible failure probability in Π_{ModToInt}, Lemma B.3) due to the correctness of the sub-protocols. □

B.2 Security Proof for the Offline Protocol

Lemma B.9. *Suppose that in the presence of a static malicious adversary corrupting exactly $n-1$ servers and any number of clients, the protocol Π_{offline}*

(1) securely realizes $\mathcal{F}_{\text{offline}}$ in the $\{\mathcal{F}_{\text{prep}}, \mathcal{F}_{\text{setup}}, \mathcal{F}_{\text{coin}}, \mathcal{F}_{\text{commit}}\}$-hybrid model, and

(2) does not abort except with negligible probability when all parties follow the protocol honestly.

Then, Π_{offline} also securely realizes $\mathcal{F}_{\text{offline}}$ in the same hybrid model in the presence of a static malicious adversary corrupting any $t < n$ servers and any number of clients.

Proof. We omit the proof since it is similar to Lemma B.1. □

Lemma B.10. *Let m, κ be parameters and $m' = 2(m + \kappa + \sqrt{m\kappa})$. Let $X \sim B(m', 1/2)$ be a random variable following the binomial distribution with m' trials and success probability $p = 1/2$. Then*

$$\Pr[X \le m] \le \mathsf{exp}(-\kappa).$$

Proof. The expected value of X is $\mathbb{E}[X] = m'/2$. By Hoeffding's inequality, the probability that $X \le m$ is:

$$\Pr[X \le m] = \Pr[X \le \mathbb{E}[X] - (\mathbb{E}[X] - m)] \le \mathsf{exp}\left(-\frac{2(\mathbb{E}[X] - m)^2}{m'}\right).$$

Substituting $\mathbb{E}[X] = m'/2 = m + \kappa + \sqrt{m\kappa}$, we have $\mathbb{E}[X] - m = \kappa + \sqrt{m\kappa}$. The exponent becomes:

$$-\frac{2(\kappa + \sqrt{m\kappa})^2}{2(m + \kappa + \sqrt{m\kappa})} = -\kappa \cdot \frac{\kappa + m + 2\sqrt{m\kappa}}{\kappa + m + \sqrt{m\kappa}} \le -\kappa.$$

Thus, the probability is bounded by $\mathsf{exp}(-\kappa)$. □

Lemma B.11. *Assuming a PRG, the protocol Π_{offline} securely realizes the functionality $\mathcal{F}_{\text{offline}}$ in the $\{\mathcal{F}_{\text{setup}}, \mathcal{F}_{\text{prep}}, \mathcal{F}_{\text{coin}}, \mathcal{F}_{\text{commit}}\}$-hybrid model in the presence of a static malicious adversary corrupting at most $t = n - 1$ servers and any number of clients.*

Proof. By Lemma B.9, we only need to consider the case when the adversary corrupts exactly $n-1$ servers. First, we construct an ideal-world simulator $\mathcal{S}$.

Simulator $\mathcal{S}$:
Initialization:

1. $\mathcal{S}$ emulates $\mathcal{F}_{\text{setup}}$ with $(\lambda, \kappa, \mathcal{B}, |\mathcal{C}| = m_{\text{in}} + m_{\text{out}} + m_{\text{mul}} + m_{\text{div}})$ and receives $(N, \zeta, \Delta_{\max}, r_{\max}, r^{\text{relax}}_{\max}, R_{\max})$.
2. $\mathcal{S}$ computes m_R, m_T, m_B, m_M, m_G, and m'_G following the protocol and emulates $\mathcal{F}_{\text{prep}}$ with input $(\texttt{init}, |\mathcal{C}|, N, \zeta, \Delta_{\max}, m_R, m_T, m_B, m_M)$.

3. $\mathcal{S}$ emulates $\mathcal{F}_{\text{prep}}$ and receives the corrupted parties' shares of the global key Δ, the random values, random bits, multiplication triples, and bounded random masks from $\mathcal{A}$. $\mathcal{S}$ forwards the corrupted shares of Δ, random bits, masks, and the triples for input gates to $\mathcal{F}_{\text{offline}}$.

4. If $\mathcal{S}$ receives `abort` from $\mathcal{A}$, $\mathcal{S}$ forwards `abort` to $\mathcal{F}_{\text{offline}}$ to receive Δ_h, computes $\Delta = \sum_{i\in[n]} \Delta_i$, sends Δ to $\mathcal{A}$, and aborts.

5. $\mathcal{S}$ initializes a list $\mathsf{ToCheck} = \emptyset$ for later MAC checks.

Sample Generators:

1. $\mathcal{S}$ identifies the next $m = \lceil \frac{\zeta+1}{\zeta} \cdot m'_G \rceil$ random values $[\![u_1]\!], \ldots, [\![u_m]\!]$ from **Initialization**.

2. For each $j \in [m]$, $\mathcal{S}$ simulates $\Pi_{\text{open}}([\![u_j]\!])$:

   (a) $\mathcal{S}$ samples the honest share $\langle u_j \rangle_h \leftarrow \mathbb{Z}_{N^\zeta}$ and reconstructs u_j based on the corrupted servers' shares.

   (b) $\mathcal{S}$ sends $\langle u_j \rangle_h$ to $\mathcal{A}$.

   (c) $\mathcal{S}$ receives the reconstructed value u'_j from $\mathcal{A}$.

   (d) $\mathcal{S}$ adds $(u_j, u'_j, [\![u_j]\!]_{\text{adv}})$ to $\mathsf{ToCheck}$, where $[\![u_j]\!]_{\text{adv}} = \sum_{i\neq h} [\![u_j]\!]_i$.

3. $\mathcal{S}$ follows the protocol and locally casts $\{u_1, \ldots, u_m\}$ to a random vector $\vec{g'} = (g'_1, \ldots, g'_{m'_G})$ where each $g'_j \in \mathbb{Z}^\times_{N^{\zeta+1}}$.

4. $\mathcal{S}$ follows the protocol and locally computes the Jacobi symbol of each g'_j modulo N. Set $\vec{g}$ to be the first m_G elements in $\{g'_j \mid (\frac{g'_j}{N}) = 1\}$. If there are not enough such elements, send `abort` to $\mathcal{F}_{\text{offline}}$ and abort the simulation on behalf of the servers.

Computing Intermediate Values $[\![r^{(x)} \cdot r^{(y)}]\!]$, $[\![\vec{\delta} \oplus \vec{B}^{(z)}]\!]$:

1. For each multiplication gate with input wires x, y and output wire z, $\mathcal{S}$ runs

$$[\![r^{(x)} \cdot r^{(y)}]\!]_{\text{adv}}, e, e', d, d' \leftarrow \mathsf{Sim}_{\text{mul}}([\![r^{(x)}]\!]_{\text{adv}}, [\![r^{(y)}]\!]_{\text{adv}}, [\![\alpha^{(z)}]\!]_{\text{adv}}, [\![\beta^{(z)}]\!]_{\text{adv}}, [\![\gamma^{(z)}]\!]_{\text{adv}}).$$

   It then samples arbitrary $[\![r^{(x)} \cdot r^{(y)}]\!]_i$ for $i \neq h$ such that they sum to $[\![r^{(x)} \cdot r^{(y)}]\!]_{\text{adv}}$, and sends them to $\mathcal{F}_{\text{offline}}$.

2. $\mathcal{S}$ computes $[\![e]\!]_{\text{adv}} = [\![r^{(x)}]\!]_{\text{adv}} - [\![\alpha^{(z)}]\!]_{\text{adv}}$, $[\![d]\!]_{\text{adv}} = [\![r^{(y)}]\!]_{\text{adv}} - [\![\beta^{(z)}]\!]_{\text{adv}}$, and adds $(e, e', [\![e]\!]_{\text{adv}})$, $(d, d', [\![d]\!]_{\text{adv}})$, to $\mathsf{ToCheck}$.

3. For each truncation gate with output wire z, $\mathcal{S}$ runs

$$[\![\vec{\delta} \odot \vec{B}^{(z)}]\!]_{\text{adv}}, \vec{e}, \vec{e'}, \vec{d}, \vec{d'} \leftarrow \mathsf{Sim}_{\text{mul}}([\![\vec{\delta}]\!]_{\text{adv}}, [\![\vec{B}^{(z)}]\!]_{\text{adv}}, [\![\vec{\alpha}^{(z)}]\!]_{\text{adv}}, [\![\vec{\beta}^{(z)}]\!]_{\text{adv}}, [\![\vec{\gamma}^{(z)}]\!]_{\text{adv}}),$$

where $\odot$ denotes component-wise products. It then computes $[\![\vec{\delta} \oplus \vec{B}^{(z)}]\!]_{\text{adv}}$ as in the protocol, samples arbitrary corrupted shares satisfying this sum, and sends those to $\mathcal{F}_{\text{offline}}$.

4. $\mathcal{S}$ computes $[\![\vec{e}]\!]_{\text{adv}} \leftarrow [\![\vec{\delta}]\!]_{\text{adv}} - [\![\vec{\alpha}^{(z)}]\!]_{\text{adv}}$, $[\![\vec{d}]\!]_{\text{adv}} \leftarrow [\![\vec{B}^{(z)}]\!]_{\text{adv}} - [\![\vec{\beta}^{(z)}]\!]_{\text{adv}}$, and adds $(\vec{e}, \vec{e}', [\![\vec{e}]\!]_{\text{adv}})$, $(\vec{d}, \vec{d}', [\![\vec{d}]\!]_{\text{adv}})$ to $\mathsf{ToCheck}$.

Reveal Input and Output Masks: For each input or output wire w associated with a client C:

1. $\mathcal{S}$ runs
$$[\![r^{(w)} \cdot \alpha^{(w)}]\!]_{\text{adv}}, e, e', d, d' \leftarrow \mathsf{Sim}_{\text{mul}}([\![r^{(w)}]\!], [\![\alpha^{(w)}]\!], [\![\alpha'^{(w)}]\!]_{\text{adv}}, [\![\beta'^{(w)}]\!]_{\text{adv}}, [\![\gamma'^{(w)}]\!]_{\text{adv}}).$$

2. $\mathcal{S}$ computes $[\![e]\!]_{\text{adv}} = [\![r^{(w)}]\!]_{\text{adv}} - [\![\alpha'^{(w)}]\!]_{\text{adv}}$, $[\![d]\!]_{\text{adv}} = [\![\alpha^{(w)}]\!]_{\text{adv}} - [\![\beta'^{(w)}]\!]_{\text{adv}}$, and adds $(e, e', [\![e]\!]_{\text{adv}})$, $(d, d', [\![d]\!]_{\text{adv}})$, to $\mathsf{ToCheck}$.

3. *If C is honest:* $\mathcal{S}$ receives corrupted shares $\langle r^{(w)} \rangle_{\text{adv}}$, $\langle \alpha^{(w)} \rangle_{\text{adv}}$, $\langle \beta^{(w)} \rangle_{\text{adv}}$, $\langle \gamma^{(w)} \rangle_{\text{adv}}$, $\langle r^{(w)} \cdot \alpha^{(w)} \rangle_{\text{adv}}$ from $\mathcal{A}$. It checks if they match the sums of the simulated ones from previous steps, and aborts if not.

4. *If C is corrupt:*
   - $\mathcal{S}$ samples $\langle r^{(w)} \rangle_h \leftarrow [0, r_{\max})$ and $\langle \alpha^{(w)} \rangle_h, \langle \beta^{(w)} \rangle_h \leftarrow \mathbb{Z}_{N^\zeta}$.
   - $\mathcal{S}$ computes the honest server's shares of $\gamma^{(w)}$ and $r^{(w)} \cdot \alpha^{(w)}$ such that:
$$\langle \gamma^{(w)} \rangle_h = \alpha^{(w)} \cdot \beta^{(w)} - \langle \gamma^{(w)} \rangle_{\text{adv}}$$
$$\langle r^{(w)} \cdot \alpha^{(w)} \rangle_h = r^{(w)} \cdot \alpha^{(w)} - \langle r^{(w)} \cdot \alpha^{(w)} \rangle_{\text{adv}}$$
   - $\mathcal{S}$ sends $\langle r^{(w)} \rangle_h, \langle \alpha^{(w)} \rangle_h, \langle \beta^{(w)} \rangle_h, \langle \gamma^{(w)} \rangle_h, \langle r^{(w)} \cdot \alpha^{(w)} \rangle_h$ to $\mathcal{A}$. If $\mathcal{A}$ sends `abort`, then $\mathcal{S}$ aborts.

Batch Verify Openings: $\mathcal{S}$ runs $\mathsf{Sim}'_{\text{MAC}}(\Delta_{\text{adv}}, \mathsf{ToCheck})$, where $\Delta_{\text{adv}} = \sum_{i \neq h} \Delta_i$.

Next, we show that the real-world execution with any adversary $\mathcal{A}$ is computationally indistinguishable from the ideal-world execution with the simulator $\mathcal{S}$.

Hyb_0: $\mathcal{S}$ runs the protocol honestly, corresponding to the real-world execution.

Hyb_1: When emulating $\mathcal{F}_{\text{prep}}$, $\mathcal{S}$ delays the sampling of the honest server's shares of the random values, random bits, multiplication triples, and bounded random masks until they are used. This does not change the output distribution.

Hyb_2: $\mathcal{S}$ initializes a $\mathsf{ToCheck}$ list. When simulating the random generators, let u'_j be the reconstructed value $\mathcal{S}$ receives from $\mathcal{A}$ for each consumed random SPDZ share $[\![u_j]\!]$. $\mathcal{S}$ adds $(u_j, u'_j, [\![u_j]\!]_{\text{adv}})$ to $\mathsf{ToCheck}$. This does not change the output distribution.

Hyb_3: $\mathcal{S}$ appends $(V_j, V'_j, [\![V_j]\!]_{\text{adv}})$ to $\mathsf{ToCheck}$ for each $j \in [\kappa]$. This does not affect the output distribution.

Hyb_4: For each SPDZ multiplication on shares $[\![x]\!]$ and $[\![y]\!]$, $\mathcal{S}$ directly samples the honest shares of $\langle e \rangle = \langle x - a \rangle$ and $\langle d \rangle = \langle y - b \rangle$ from $\mathbb{Z}_{N^\zeta}$, where $[\![a]\!], [\![b]\!], [\![c]\!]$ is the triple consumed. Then,

$\mathcal{S}$ reconstructs $a = e + x$, $b = d + y$, $c = a \cdot b$, as well as $\Delta \cdot a$, $\Delta \cdot b$, $\Delta \cdot c$, and computes the honest server's shares of $[\![a]\!]$, $[\![b]\!]$, and $[\![c]\!]$ by subtracting corrupt servers' shares. Since the honest server's shares of a and b are uniformly random in both cases, this change does not affect the output distribution.

Hyb_5: For each instance of Π_{mul}, let e', d' be the value opened to the honest party P_h. $\mathcal{S}$ appends $(e, e', [\![e]\!]_{\text{adv}})$ and $(d, d', [\![d]\!]_{\text{adv}})$ to the $\mathsf{ToCheck}$ list. This does not change the output distribution.

Hyb_6: For each entry $(A_j, A'_j, [\![A_j]\!]_{\text{adv}})$ in the $\mathsf{ToCheck}$ list, $\mathcal{S}$ checks if A_j equals A'_j. If so, $\mathcal{S}$ sets flag $\mathsf{MACCheck} = 0$. Otherwise, $\mathcal{S}$ sets flag $\mathsf{MACCheck} = 1$. This does not change the output distribution.

Hyb_7: If $\mathsf{MACCheck} = 0$, $\mathcal{S}$ does not compute the honest server's share of $\langle\sigma\rangle$ following the MAC-Check protocol, but instead computes $\langle\sigma\rangle_h = -\sum_{i\neq h}\langle\sigma\rangle_i$. Since each value is opened correctly, it's guaranteed that $\sum_{i\in[n]}\langle\sigma\rangle_i = 0$. Therefore, the output distribution is unchanged.

Hyb_8: If $\mathsf{MACCheck} = 1$, $\mathcal{S}$ does not compute the honest server's share of $\langle\Delta \cdot A_j\rangle$ for each opened value A_j following the protocol. Instead, $\mathcal{S}$ computes $\Delta \cdot A_j$ and subtracts the corrupt servers' shares. This only changes the way of computing the honest server's shares, and does not affect the output distribution. Note that up to this point, the simulation no longer needs to compute the honest shares of $[\![a]\!]$, $[\![b]\!]$, $[\![c]\!]$ in Hyb_4.

Hyb_9: After checking if $\sum_{i\in[n]}\langle\sigma\rangle_i = 0$ following the protocol, $\mathcal{S}$ also aborts the simulation if $\mathsf{MACCheck} = 1$. We show that for any adversary $\mathcal{A}$, the probability that $\mathsf{MACCheck} = 1$ but $\sum_{i\in[n]}\langle\sigma\rangle_i = 0$ in Hyb_4 is negligible.

Suppose that the values opened to the honest server P_h are $\{A'_j\}_{j\in[m]}$ and let $\delta_j = A'_j - A_j$. Since $\mathsf{MACCheck} = 1$, there must exist a $j \in [m]$ such that $\delta_j \neq 0$. To ensure $\langle\sigma\rangle$ still sum-checks to zero, the adversary must adjust the sum of the corrupted servers' committed shares to contain an additive error δ_σ such that

$$\delta_\sigma = -\Delta_h \cdot \sum_{j\in[m]} \chi_j \cdot \delta_j \pmod{N^\zeta}$$

First, since $N = p \cdot q$ where p, q are large primes, and the seed for generating $\chi_1, \ldots, \chi_m$ is sampled after the errors $\{\delta_j\}_{j\in[m]}$ are fixed, the probability that $\sum_{j\in[m]} \chi_j \cdot \delta_j = 0$ is upper-bounded by $1/\min(p, q) + \mathsf{negl}(\lambda)$ assuming the pseudorandomness of the PRG.

Conditioned on $\sum_{j\in[m]} \chi_j \cdot \delta_j \neq 0$, the adversary needs to at least obtain the value of $\Delta_h \bmod p$ or $\Delta_h \bmod q$ to compute the required δ_σ. However, since Δ_h is never used in the simulation to compute any transcript sent to any corrupted server before the corrupted servers commit their shares of $\langle\sigma\rangle$, the probability that the adversary guesses the correct Δ_h is at most $\max(\frac{1}{\Delta_{\max}}, \frac{2}{p+1}, \frac{2}{q+1})$, which is again negligible in the security parameter.

Therefore, Hyb_9 is computationally indistinguishable from Hyb_4.

Hyb_{10}: For each input or output wire w attached to an honest client C, $\mathcal{S}$ does not follow the protocol to check $r^{(w)}$. Instead, $\mathcal{S}$ checks whether the sums of the corrupted servers' shares of $\langle r^{(w)}\rangle$, $\langle\alpha^{(w)}\rangle$, $\langle\beta^{(w)}\rangle$, $\langle\gamma^{(w)}\rangle$, $\langle r^{(w)} \cdot \alpha^{(w)}\rangle$ from the previous simulation match the sum of the shares sent by the corrupt servers. Since $\mathsf{MACCheck} = 0$ in the verification step (or the simulation would have aborted), the shares $\langle r^{(w)} \cdot \alpha^{(w)}\rangle$ $\mathcal{S}$ holds is a correct share of $r^{(w)} \cdot \alpha^{(w)}$. Therefore, the change only affects distribution if the corrupt servers inject additive errors $\delta_r, \delta_\alpha, \delta_\beta, \delta_\gamma, \delta_{r\cdot\alpha}$ (not all zero) into these values such that

$$(r^{(w)} + \delta_r) \cdot (\alpha^{(w)} + \delta_\alpha) = r^{(w)} \cdot \alpha^{(w)} + \delta_{r\cdot\alpha} \pmod{N^\zeta}$$
$$(\alpha^{(w)} + \delta_\alpha) \cdot (\beta^{(w)} + \delta_\beta) = \gamma^{(w)} + \delta_\gamma \pmod{N^\zeta}$$

or equivalently,

$$\delta_r \cdot \alpha^{(w)} + \delta_\alpha \cdot r^{(w)} = \delta_{r\cdot\alpha} - \delta_r \cdot \delta_\alpha \pmod{N^\zeta}$$
$$\delta_\alpha \cdot \beta^{(w)} + \delta_\beta \cdot \alpha^{(w)} = \delta_\gamma - \delta_\alpha \cdot \delta_\beta \pmod{N^\zeta}$$

Note that if $\delta_{r\cdot\alpha}$ is non-zero, then one of δ_r, δ_α must be non-zero. Similarly, if δ_γ is non-zero, then one of $\delta_\alpha, \delta_\beta$ must be non-zero. Therefore, it suffices to show that the probability that the adversary can inject non-zero errors into any of $r^{(w)}, \alpha^{(w)}, \beta^{(w)}$ is negligible.

Due to our replacement in Hyb_4 and Hyb_7, the honest party's share of $[\![\alpha^{(w)}]\!]$ is no longer used to generate any transcript sent to the corrupted servers before the errors are injected. Therefore, we can view $\alpha^{(w)}$ as sampled after the errors are fixed.

If $\delta_r \neq 0$, then there are at most $\frac{N^\zeta}{\min(p,q)}$ choices of $\alpha^{(w)}$ that satisfy the first equation above. Since $\alpha^{(w)}$ is uniformly random in $\mathbb{Z}_{N^\zeta}$, this happens with at most $1/\min(p,q)$ probability, which is negligible in the security parameter. Similarly, if $\delta_\beta \neq 0$, there is at most $1/\min(p,q)$ probability the second equation above holds. Finally, if $\delta_\alpha \neq 0$, since $\beta^{(w)}$ is also uniformly random and fresh, there is also at most $1/\min(p,q)$ probability that the second equation above holds.

Hyb_{11}: For each input or output wire w attached to a corrupt client C, the honest server does not follow the protocol to obtain the honest server's shares of $\langle r^{(w)} \cdot \alpha^{(w)} \rangle$ or $\langle \gamma^{(w)} \rangle$. Instead, $\mathcal{S}$ computes these shares based on the reconstructed $\alpha^{(w)}$, $\beta^{(w)}$, $r^{(w)}$ and the corrupt servers' shares of $\langle r^{(w)} \cdot \alpha^{(w)} \rangle$ and $\langle \gamma^{(w)} \rangle$. This only changes the order of generating $\{r^{(w)} \cdot \alpha^{(w)}, \gamma^{(w)}\}$ and the honest server's shares of them. Therefore, the output distribution is unchanged. Note that the honest server's shares of $\alpha^{(w)}$, $\beta^{(w)}$, and $\gamma^{(w)}$ are no longer used, so $\mathcal{S}$ no longer computes them.

We note that Hyb_{11} is the same as the ideal-world execution. Thus, the real-world execution is computationally indistinguishable from the ideal world.

Finally, we show that the protocol does not abort when all parties follow the protocol honestly. We identify the possible abort points in the protocol:

- The protocol may abort when invoking $\mathcal{F}_{\text{prep}}$. However, by Lemma B.5, this happens with only negligible probability when all parties are honest.
- If there are not enough numbers in $\vec{g}$ whose Jacobi is 1, the protocol aborts. By Lemma B.16, there are $2p'q' \cdot N^\zeta$ numbers in $\mathbb{Z}_{N^{\zeta+1}}$ with Jacobi 1, which accounts for $\frac{2p'q'}{N} = \frac{1}{2} - \frac{p+q-1}{N} \approx \frac{1}{2}$ fraction of all the numbers. Given $2(m_G + \kappa + \sqrt{m_G \cdot \kappa})$ uniformly and independently sampled numbers, by Lemma B.10, the probability of finding fewer than m_G generators is bounded by $\mathsf{exp}(-\kappa)$, which is negligible in κ.
- The protocol may abort during the SPDZ MAC checks. However, when all parties are honest, all opened values are consistent, so the MAC checks always pass.
- The protocol may abort by the clients if their reconstructed values don't form valid triples. However, when all parties are honest, the reconstructed values are always valid triples.

To summarize, the protocol only aborts with negligible probability when all parties are honest. □

B.3 Security Proof for the Main Protocol

We first show that it suffices to consider the maximal corruption case with a corrupted evaluator P_1.

Lemma B.12. *Suppose that in the presence of a static malicious adversary corrupting exactly $n-1$ servers including P_1 (the evaluator), and any number of clients, the protocol Π_{main}*

(1) securely realizes the functionality $\mathcal{F}_{\text{main}}$ in $\{\mathcal{F}_{\text{offline}}, \mathcal{F}_{\text{coin}}, \mathcal{F}_{\text{commit}}\}$-hybrid model, and

(2) does not abort except with negligible probability when all parties follow the protocol honestly.

Then, Π_{main} also securely realizes $\mathcal{F}_{\text{main}}$ in the same hybrid model in the presence of a static malicious adversary corrupting any $t < n$ servers and any number of clients.

Proof. We first extend the security guarantee from Condition (1) (which covers $n-1$ corruptions including P_1) to the case of $n-1$ corruptions *excluding* P_1. Specifically, we view the honest server P_1 as two virtual parties P_1^G and P_1^E, where P_1^G acts exactly like the other servers (e.g., receiving inputs from clients and computing shares of the garbled circuit), and P_1^E performs the rest of the tasks that are exclusively to P_1 (e.g., aggregating shares and evaluating the garbled circuit). Suppose that S is the simulator for the case when the honest server $P_h \neq P_1$. We can construct a new simulator S' for the case when P_1 is honest by simulating P_1^G the same way as S simulates P_h, and simulating P_1^E by following the honest protocol. Since P_1^G and P_1^E do not share any internal state, the indistinguishability between the real and ideal executions is preserved.

Given the security for any set of $n-1$ corrupt servers, the reduction for $t < n-1$ and $t = 0$ then proceeds exactly as in Lemma B.1. □

Then, we introduce a lemma that shows the circular security assumption we need for truncation gates can be reduced to the standard KDM security assumption for symmetric-key encryption.

Lemma B.13. *Assuming that the encryption scheme* LHE *in Figure 3 is nonce-based and KDM secure with respect to any efficiently computable function, then for any m polynomial in the security parameter, no efficient P.P.T adversary can distinguish the following two oracles:*

– $\mathcal{O}_{\text{real}}$:

  * *Sample a global key Δ from $[0, \Delta_{\max})$ as in* $\mathsf{LHE.KeyGen}$*, and $k_1, \ldots, k_m$ from $[0, \Delta_{\max} \cdot m \cdot 2^\kappa)$, where κ is the security parameter.*
  * *On each oracle call with an index $i \in [m]$ and an efficiently computable function f on Δ, return* $\mathsf{LHE.Enc}_{k_i}(f(\Delta))$.
  * *On each oracle call with a function f on $k_1, \ldots, k_m$, return*

$$\mathsf{LHE.Enc}_\Delta(f(k_1, \ldots, k_m)).$$

  * *On each oracle call with a function f on Δ, return* $\mathsf{LHE.Enc}_\Delta(f(\Delta))$.

– $\mathcal{O}_{\text{random}}$:

  * *Sample a global key Δ from $[0, \Delta_{\max})$ as in* $\mathsf{LHE.KeyGen}$*, and $k_1, \ldots, k_m$ from $[0, \Delta_{\max} \cdot m \cdot 2^\kappa)$, where κ is the security parameter.*
  * *On each oracle call with an index $i \in [m]$ and a function f on Δ, sample z randomly from $\mathbb{Z}_{N^\zeta}$ and return* $\mathsf{LHE.Enc}_{k_i}(z)$.
  * *On each oracle call with a function f on $k_1, \ldots, k_m$, sample z randomly from $\mathbb{Z}_{N^\zeta}$ and return* $\mathsf{LHE.Enc}_\Delta(z)$.

* *On each oracle call with a function f on Δ, sample z randomly from $\mathbb{Z}_{N^\zeta}$ and return* $\mathsf{LHE.Enc}_\Delta(z)$.

Proof. Suppose that there is a distinguisher $\mathcal{A}$ that can distinguish $\mathcal{O}_{\text{real}}$ and $\mathcal{O}_{\text{random}}$ with non-negligible advantage. We construct a distinguisher $\mathcal{B}$ for the oracles in Definition 3.1 as follows:

- Sample n random values $\delta_1, \ldots, \delta_m$ from $[0, \Delta_{\max} \cdot m \cdot 2^\kappa)$.
- Run $\mathcal{A}$ internally.
- Every time $\mathcal{A}$ makes an oracle call with input i and f, query the oracle in Definition 3.1 with f. Upon receiving $(g, g^\Delta \cdot \mathsf{exp}(f(\Delta)))$, $\mathcal{B}$ forwards $(g, g^{\delta_i} \cdot g^\Delta \cdot \mathsf{exp}(\Delta))$ to $\mathcal{A}$.
- Every time $\mathcal{A}$ makes an oracle call with only a function f on $k_1, \ldots, k_m$, $\mathcal{B}$ sets $f'(\Delta) = f(\Delta + \delta_1, \ldots, \Delta + \delta_m)$, queries the oracle in Definition 3.1 with f', and forwards the output $g, g^\Delta \cdot \mathsf{exp}(f'(\Delta))$ to $\mathcal{A}$.
- Every time $\mathcal{A}$ makes an oracle call with only a function f on Δ, $\mathcal{B}$ queries the oracle in Definition 3.1 with f and forwards the output to $\mathcal{A}$.
- Output whatever $\mathcal{A}$ outputs.

Since $\Delta_{\max} \ll m \cdot 2^\kappa \cdot \Delta_{\max}$, the joint distribution of $\Delta, k_1, \ldots, k_m$ is statistically close to the joint distribution of $\Delta, \Delta + \delta_1, \ldots, \Delta + \delta_m$. Therefore, $\mathcal{B}$ simulates the view of $\mathcal{A}$ correctly. Therefore, $\mathcal{B}$ can distinguish the two oracles in Definition 3.1 with non-negligible advantage, which contradicts the KDM security assumption of the encryption scheme. □

Finally, we prove our main theorem.

Theorem B.14. *Assuming KDM security of the encryption scheme in Figure 3 (with respect to the class of any efficiently computable function), the protocol* Π_{main} *securely realizes the functionality* $\mathcal{F}_{\text{main}}$ *in the* $\{\mathcal{F}_{\text{offline}}, \mathcal{F}_{\text{coin}}, \mathcal{F}_{\text{commit}}\}$*-hybrid model in the presence of a static malicious adversary corrupting at most* $t = n - 1$ *servers and any number of clients. Moreover, when the input circuit* $\mathcal{C}$ *does not contain* $\texttt{div}_2$ *gates, the KDM assumption can be weakened to standard DCR.*

Proof. By Lemma B.12, we only need to consider the case when the adversary corrupts exactly $n - 1$ servers, including P_1. First, we construct an ideal-world simulator $\mathcal{S}$.

Simulator $\mathcal{S}$:
Offline Phase with $\mathcal{F}_{\text{offline}}$: $\mathcal{S}$ emulates $\mathcal{F}_{\text{offline}}$ with inputs $(\texttt{init}, m_{\text{in}}, m_{\text{out}}, m_{\text{mul}}, m_{\text{div}})$:

1. During **Init**, $\mathcal{S}$ receives the corrupted key shares $\{\Delta_i\}_{i \neq h}$ to $\mathcal{F}_{\text{offline}}$, and samples $\Delta_h \leftarrow [0, \Delta_{\max})$. It then sends (N, ζ) to the corrupt parties.
2. During **Sample Generators**, $\mathcal{S}$ emulates $\mathcal{F}_{\text{offline}}$ to sample random generators $\vec{g}$ (parsed as $\vec{g}_{\text{ext}}$, $\vec{g}_{\text{Bool}}$ and $\vec{g}_{\text{out}}$ as in the protocol) and sends them to $\mathcal{S}$.
3. During **Prepare Wire Masks**:
   - For each wire w that's not an output of a linear gate, receive corrupted server's shares $[\![r^{(w)}]\!]_i$ for $i \neq h$. $\mathcal{S}$ samples the honest server's share $[\![r^{(w)}]\!]_h$ and computes $r^{(w)} = \sum_{i \in [n]} \langle r^{(w)} \rangle_i$. $\mathcal{S}$ aborts if $r^{(w)}$ is out-of-bound.
   - For each wire z that's an output of a truncation gate, additionally receive corrupted shares $[\![u^{(z)}]\!]_i$ for $i \neq h$. $\mathcal{S}$ similarly samples $[\![u^{(z)}]\!]_h$ and computes $u^{(z)}$. $\mathcal{S}$ aborts if

$u^{(z)}$ is out-of-bound.

- For each output wire z of a linear gate, $\mathcal{S}$ emulates $\mathcal{F}_{\text{offline}}$ and computes corrupted shares $[\![r^{(z)}]\!]_i$, honest shares $[\![r^{(z)}]\!]_h$ and the mask value $r^{(z)}$.
- For each input or output wire w associated with a corrupt Client C, $\mathcal{S}$ sends $r^{(w)}$ to C.

4. During **Prepare Other Intermediate Values**, $\mathcal{S}$ receives the following corrupted shares for $i \neq h$:
   - $[\![\vec{\delta}^{(z)}]\!], [\![\vec{B}^{(z)}]\!]_i, [\![b^{(z)}]\!]_i, [\![\vec{\delta}^{(z)} \oplus \vec{B}^{(z)}]\!]_i$ for each output wire z of a truncation gate,
   - $[\![r^{(x)} \cdot r^{(y)}]\!]_i$ for each multiplication gate with input wires x, y,
   - $[\![\alpha^{(w)}]\!]_i, [\![\beta^{(w)}]\!]_i, [\![\gamma^{(w)}]\!]_i$ for each input wire w.

   $\mathcal{S}$ then samples $\vec{\delta}^{(z)}, \vec{B}^{(z)}, \vec{b}^{(z)}, \alpha^{(w)}, \beta^{(w)}$ (which determines $\gamma^{(w)} = \alpha^{(w)} \cdot \beta^{(w)}$) as specified in $\mathcal{F}_{\text{offline}}$, and derive the honest servers' shares accordingly.

5. Upon receiving `abort` from $\mathcal{A}$, $\mathcal{S}$ emulates $\mathcal{F}_{\text{offline}}$ to send Δ to $\mathcal{A}$ and aborts the protocol on behalf of the honest clients and servers.

Input Phase: $\mathcal{S}$ initializes a list $\mathsf{ToCheck} = \emptyset$.

1. When simulating **Mask Input**, for each input wire x associated with Client C:
   - *If C is honest:* $\mathcal{S}$ sets $\widetilde{x} = r^{(x)}$, and sends it to $\mathcal{A}$.
   - *If C is corrupt:* $\mathcal{S}$ receives $\widetilde{x}$ from $\mathcal{A}$, and extracts $x = \widetilde{x} - r^{(x)} (\mathrm{mod} N^{\zeta})$. $\mathcal{S}$ sends x to $\mathcal{F}_{\text{main}}$ as the input for C.

   After extracting all corrupted clients' input and sending them to $\mathcal{F}_{\text{main}}$, $\mathcal{S}$ receives an output value for each output wire w from $\mathcal{F}_{\text{main}}$.

2. When simulating **Prepare Range Check**:
   - $\mathcal{S}$ computes the corrupt servers' shares of $[\![x]\!] = \widetilde{x} - [\![r^{(x)}]\!]$ and $[\![y]\!] = [\![x]\!] - 1$, and defines their sums as $[\![x]\!]_{\text{adv}}$, $[\![y]\!]_{\text{adv}}$. Similarly, define the sum of corrupted servers' shares of the prepared triple for this wire $[\![\alpha^{(x)}]\!]_{\text{adv}}$, $[\![\beta^{(x)}]\!]_{\text{adv}}$, $[\![\gamma^{(x)}]\!]_{\text{adv}}$. It then runs
   $$(\![z]\!]_{\text{adv}}, e, e', d, d')_{\text{mul}} \leftarrow \mathsf{Sim}_{\text{mul}}([\![x]\!]_{\text{adv}}, [\![y]\!]_{\text{adv}}, [\![\alpha^{(x)}]\!]_{\text{adv}}, [\![\beta^{(x)}]\!]_{\text{adv}}, [\![\gamma^{(x)}]\!]_{\text{adv}}).$$
   - $\mathcal{S}$ computes $[\![e]\!]_{\text{adv}} = [\![x]\!]_{\text{adv}} - [\![\alpha^{(x)}]\!]_{\text{adv}}$, $[\![d]\!]_{\text{adv}} = [\![y]\!]_{\text{adv}} - [\![\beta^{(x)}]\!]_{\text{adv}}$, and appends $(e, e', [\![e]\!]_{\text{adv}})$, $(d, d', [\![d]\!]_{\text{adv}})$, $(0, 0, [\![z]\!]_{\text{adv}})$ to $\mathsf{ToCheck}$.

3. $\mathcal{S}$ checks that for every input wire x of the circuit, the masked value $\widetilde{x}$ is in range $[0, n \cdot r_{\max})$. If not, $\mathcal{S}$ aborts.

4. $\mathcal{S}$ runs $\mathsf{Sim}_{\text{MAC}}(\Delta_{\text{adv}}, \mathsf{ToCheck})$ for the input range check, where $\Delta_{\text{adv}} = \sum_{i \neq h} \Delta_i$.

5. For each input wire x, $\mathcal{S}$ samples $\langle L^{(x)} \rangle_h^* \leftarrow [0, R_{\max})$ and sends it to P_1 (controlled by $\mathcal{A}$).

Garbling Phase:

1. **Simulate Masks and Labels:**
   - For each output wire w of the circuit associated with a corrupt client C, $\mathcal{S}$ sets $\widetilde{w} = r^{(w)} + w$, where $r^{(w)}$ is recorded in the offline phase.
   - For each output wire z of a multiplication gate that is not an output wire of the circuit:
     * $\mathcal{S}$ sets $\langle \widetilde{z} \rangle_h \leftarrow \langle r^{(z)} \rangle_h$ and samples $\langle L^{(z)} \rangle_h^* \leftarrow [0, R_{\max})$.
     * $\mathcal{S}$ computes $\widetilde{z} = \langle \widetilde{z} \rangle_h + \langle \widetilde{z} \rangle_{\text{adv}}$.
   - For each output wire z of a linear gate with input wires x, y and constant scalars μ, ν:
     * $\mathcal{S}$ computes $\widetilde{z} = \mu \cdot \widetilde{x} + \nu \cdot \widetilde{y}$ and $\langle L^{(z)} \rangle_h^* = \mu \cdot \langle L^{(x)} \rangle_h^* + \nu \cdot \langle L^{(y)} \rangle_h^*$.
2. **Garble Multiplication Gates:** For each multiplication gate in the circuit with input wires x, y and output wire z:
   (a) **Determine Targets:** $\mathcal{S}$ samples random $\langle \xi_x \rangle_h, \langle \xi'_x \rangle_h$ from $\mathbb{Z}_{N^\zeta}$.
   (b) Calculate targets

$$\begin{aligned}
\langle \xi_y \rangle_h &= \widetilde{x} \cdot \langle L^{(y)} \rangle_h^* - \widetilde{x} \cdot \langle \Delta \cdot r^{(y)} \rangle_{\text{adv}} - \widetilde{y} \cdot \langle \Delta \cdot r^{(x)} \rangle_{\text{adv}} \\
&\quad + (\widetilde{x} \cdot \widetilde{y} - \widetilde{z}) \cdot \langle \Delta \rangle_{\text{adv}}^* + \langle \Delta \cdot r^{(x)} \cdot r^{(y)} \rangle_{\text{adv}} \\
&\quad + \langle \Delta \cdot r^{(z)} \rangle_{\text{adv}} - \langle L^{(z)} \rangle_h^* - \langle \xi_x \rangle_h \pmod{N^\zeta} \\
\langle \xi'_y \rangle_h &= \widetilde{x} \cdot \widetilde{y} - \widetilde{x} \cdot \langle r^{(y)} \rangle_{\text{adv}} - \widetilde{y} \cdot \langle r^{(x)} \rangle_{\text{adv}} - \widetilde{z} \\
&\quad + \langle r^{(x)} \cdot r^{(y)} \rangle_{\text{adv}} + \langle r^{(z)} \rangle_{\text{adv}} - \langle \xi'_x \rangle_h \pmod{N^\zeta}
\end{aligned}$$

   where $\langle X \rangle_{\text{adv}} = \sum_{i \neq h} \langle X \rangle_i$ for any shared value $\langle X \rangle$.
   (c) **Simulate Key-Extension:**
   - $\mathcal{S}$ calls $\mathsf{Sim}_{\text{ext}}^{\Delta_h, \vec{g}_{\text{ext}}}(\widetilde{x}, \langle L^{(x)} \rangle_h^*, \langle \xi_x \rangle_h)$.
   - $\mathcal{S}$ calls $\mathsf{Sim}_{\text{ext}}^{\Delta_h, \vec{g}_{\text{ext}}}(\widetilde{y}, \langle L^{(y)} \rangle_h^*, \langle \xi_y \rangle_h)$.
   - $\mathcal{S}$ calls $\mathsf{Sim}_{\text{ext}}^{\Delta_h, \vec{g}_{\text{ext}}}(\widetilde{x}, \langle L^{(x)} \rangle_h^*, \langle \xi'_x \rangle_h)$.
   - $\mathcal{S}$ calls $\mathsf{Sim}_{\text{ext}}^{\Delta_h, \vec{g}_{\text{ext}}}(\widetilde{y}, \langle L^{(y)} \rangle_h^*, \langle \xi'_y \rangle_h)$.

   where the subroutine $\mathsf{Sim}_{\text{ext}}^{\Delta_h, \vec{g}_{\text{ext}}}$ is defined as follows:

$\mathsf{Sim}_{\mathrm{ext}}^{\Delta_h,\vec{g}_{\mathrm{ext}}}(\widetilde{w}, \langle L^{(w)}\rangle_h^*, \langle \xi\rangle_h)$:

i. Let g be the next generator from $\vec{g}_{\mathrm{ext}}$.

ii. $\mathcal{S}$ samples a random $\langle A\rangle'_h \in \mathbb{Z}_{N^\zeta}$ and computes the honest server's first component:

$$\mathsf{ct}_{h,1} = g^{\Delta_h}\mathsf{exp}(\langle A\rangle'_h)$$

iii. Let $\widetilde{w}' \in [-N^\zeta/2, N^\zeta/2)$ be such that $\widetilde{w}' \equiv \widetilde{w} \mod N^\zeta$. $\mathcal{S}$ computes the honest server's second component $\mathsf{ct}_{h,2}$ to satisfy the constraint:

$$(\mathsf{ct}_{h,1})^{\widetilde{w}'} \cdot \mathsf{ct}_{h,2} = g^{\langle L^{(w)}\rangle_h^*} \cdot \mathsf{exp}(\langle \xi\rangle_h)$$

iv. $\mathcal{S}$ sends $\mathsf{ct}_{h,1}, \mathsf{ct}_{h,2}$ to P_1 (controlled by $\mathcal{A}$).

3. **Garble Truncation Gates:** For each truncation gate in the circuit with input wire x and output wire z, the simulator has prepared masked wire values $\widetilde{x}, \widetilde{z}$ and honest shares $\langle L^{(x)}\rangle_h^*, \langle L^{(z)}\rangle_h^*$. Additionally, $\mathcal{S}$ samples honest shares $\langle R^{(x_0)}\rangle_h^*, \langle \vec{U}^{(z)}\rangle_h^*$ from $[0, R_{\max})$.

   (a) **Intermediate Values:**
   - Set $\langle L^{(x_0)}\rangle_h^* = \langle R^{(x_0)}\rangle_h^*$, $\langle L'\rangle_h^* = \langle L^{(x_0)}\rangle_h^* + 2\cdot\langle L^{(z)}\rangle_h^*$, $\langle \vec{L}\rangle_h^* = 2\cdot\langle \vec{U}^{(z)}\rangle_h + \langle \vec{B}^{(z)}\rangle_h$, $[\![r^{(x_0)}]\!]_{\mathrm{adv}} = 2\cdot[\![u^{(z)}]\!]_{\mathrm{adv}} + [\![b^{(z)}]\!]_{\mathrm{adv}}$, and $[\![r'^{(x)}]\!]_{\mathrm{adv}} = [\![r^{(x_0)}]\!]_{\mathrm{adv}} + 2\cdot[\![r^{(z)}]\!]_{\mathrm{adv}}$.
   - Sample $\langle r^{(x_0)}\rangle_h \leftarrow [0, 2\cdot r_{\max})$, and $\widetilde{x_0} = r^{(x_0)} = \langle r^{(x_0)}\rangle_h^* + \langle r^{(x_0)}\rangle_{\mathrm{adv}}$.
   - Set $\tilde{x}' = \widetilde{x_0} + 2\cdot\tilde{z}$, and $\mathsf{idx} = \widetilde{x_0} \bmod 2$.

   (b) **Arithmetic Label Extensions:** To avoid cluttering, we omit the superscript (z) in the following.
   - Compute the honest server's target shifts

$$\begin{aligned}\langle V\rangle_h = {} & \langle L'^{(x)}\rangle_h - \langle L^{(x)}\rangle_h + \Delta_{\mathrm{adv}}\cdot(\tilde{x}' - \tilde{x}) \\ & - 2\cdot(\langle \Delta\cdot u^{(z)}\rangle_{\mathrm{adv}} + \langle \Delta\cdot r^{(z)}\rangle_{\mathrm{adv}}) - \langle \Delta\cdot b^{(z)}\rangle_{\mathrm{adv}} \\ & + \langle \Delta\cdot r^{(x)}\rangle_{\mathrm{adv}}, \\ \langle v\rangle_h = {} & \tilde{x}' - \tilde{x} - 2\cdot(\langle u^{(z)}\rangle_{\mathrm{adv}} + \langle r^{(z)}\rangle_{\mathrm{adv}}) - \langle b^{(z)}\rangle_{\mathrm{adv}} + \langle r^{(x)}\rangle_{\mathrm{adv}}.\end{aligned}$$

   - Run helper procedure

$$\mathsf{Sim}_{\mathrm{ext}}^{\Delta_h,\vec{g}_{\mathrm{ext}}}(\tilde{x}', \langle L'^{(x)}\rangle_h^*, \langle \vec{L}[j]\rangle_h) \text{ for each } j \in [\lambda'].$$

   (c) **Boolean Label Extensions:**
   - Compute $\langle \mathsf{sk}_{\mathsf{idx}}\rangle_h^*, \langle \mathsf{sk}_{1-\mathsf{idx}}\rangle_h^*$ as

$$\begin{aligned}\mathsf{sk}_{\mathsf{idx}} = \mathrm{Int}((\vec{\delta}\cdot\mathsf{idx})\oplus\vec{B}), \quad \mathsf{sk}_{1-\mathsf{idx}} &= \mathrm{Int}(\vec{\delta}\oplus(\vec{\delta}\cdot\mathsf{idx})\oplus\vec{B}), \\ \langle \mathsf{sk}_{\mathsf{idx}}\rangle_h^* = {} & \mathsf{sk}_{\mathsf{idx}} - (1-\mathsf{idx})\cdot\mathrm{Int}(\langle\vec{B}\rangle_{\mathrm{adv}} + \langle\vec{0}\rangle_{\mathrm{adv}}) \\ & - \mathsf{idx}\cdot\mathrm{Int}(\langle\vec{\delta}\oplus\vec{B}\rangle_{\mathrm{adv}} + \langle\vec{0}\rangle_{\mathrm{adv}}) \\ \langle \mathsf{sk}_{1-\mathsf{idx}}\rangle_h^* = {} & \mathsf{sk}_{1-\mathsf{idx}} - \mathsf{idx}\cdot\mathrm{Int}(\langle\vec{B}\rangle_{\mathrm{adv}} + \langle\vec{0}\rangle_{\mathrm{adv}}) \\ & - (1-\mathsf{idx})\cdot\mathrm{Int}(\langle\vec{\delta}\oplus\vec{B}\rangle_{\mathrm{adv}} + \langle\vec{0}\rangle_{\mathrm{adv}}),\end{aligned}$$

where $\langle\vec{0}\rangle_{\text{adv}}$ is a fixed public random share.

– Run helper procedure $\mathsf{Sim}_{\text{ModToInt}}(\langle\vec{B}\rangle_{\text{adv}})$ and $\mathsf{Sim}_{\text{ModToInt}}(\langle\vec{\delta}\oplus\vec{B}\rangle_{\text{adv}})$, to simulate communications during Π_{ModToInt}

– Let $g, g' \in \mathbb{Z}^{\times}_{N^{\zeta+1}}$ be two additional group elements from $\vec{g}_{\text{Bool}}$ prepared in $\mathcal{F}_{\text{offline}}$ for the current gate. Simulate ciphertexts as

$$\vec{\mathsf{ct}}_{\mathsf{idx},h}[0] = g^{\langle \mathsf{sk}_{\mathsf{idx}}\rangle^*_h} \cdot \mathsf{exp}(\langle\xi\rangle_h), \quad \vec{\mathsf{ct}}_{1-\mathsf{idx},h}[0] = g^{\langle \mathsf{sk}_{1-\mathsf{idx}}\rangle^*_h} \cdot \mathsf{exp}(m),$$
$$\vec{\mathsf{ct}}_{\mathsf{idx},h}[1] = g'^{\langle \mathsf{sk}_{\mathsf{idx}}\rangle^*_h} \cdot \mathsf{exp}(\langle\xi'\rangle_h), \quad \vec{\mathsf{ct}}_{1-\mathsf{idx},h}[1] = g'^{\langle \mathsf{sk}_{1-\mathsf{idx}}\rangle^*_h} \cdot \mathsf{exp}(m'),$$

where m, m' are random plaintexts and

$$\begin{aligned}\langle\xi\rangle_h =& \langle L^{(x_0)}\rangle_h + \Delta_{\text{adv}} \cdot \widetilde{x_0} \\ & - \langle\Delta \cdot r^{(x_0)}\rangle_{\text{adv}} - \mathsf{idx} \cdot \Delta_{\text{adv}} - (1 - 2 \cdot \mathsf{idx}) \cdot \langle\Delta \cdot b\rangle_{\text{adv}}, \\ \langle\xi'\rangle_h =& \widetilde{x_0} - \mathsf{idx} \cdot \langle 1\rangle_{\text{adv}} - (1 - 2 \cdot \mathsf{idx}) \cdot \langle b\rangle_{\text{adv}}.\end{aligned}$$

Evaluation Phase:

1. $\mathcal{S}$ does nothing in this phase since $\mathcal{A}$ controls the evaluator P_1.

Output Phase: For every output wire y associated with an honest client C:

1. **Decoding Information:** $\mathcal{S}$ receives $\mathsf{ct}_{i,1}$, $\mathsf{ct}_{i,2}$ from $\mathcal{A}$ for $i \neq h$, and sets $\mathsf{ct}_{\text{adv},1} = \prod_{i\neq h} \mathsf{ct}_{i,1}$, and similarly $\mathsf{ct}_{\text{adv},2}$.
2. **Evaluation Results:** $\mathcal{S}$ receives ct, $\widetilde{y}^*$ from $\mathcal{A}$.
3. **Client Output:** $\mathcal{S}$ checks if the received $\widetilde{y}^*$ equals $\widetilde{y}$ computed in the simulation, and if $\mathsf{ct} = (\mathsf{ct}_{\text{adv},1})^{\tilde{y}} \cdot \mathsf{ct}_{\text{adv},2} \cdot g^{\langle L^{(y)}\rangle^*_h}$. If not, abort.

For every output wire y associated with a corrupted client C, $\mathcal{S}$ computes $\mathsf{ct}_{h,1} = g^{\Delta_h}$ and $\mathsf{ct}_{h,2} = g^{\langle L^{(y)}\rangle^*_h}/(\mathsf{ct}^{(\tilde{y})}_{h,1})$, where $\tilde{y}$ is simulated during Garbling Phase. It sends $\mathsf{ct}_{h,1}, \mathsf{ct}_{h,2}$ to $\mathcal{A}$ as the decoding information.

Next, we show that the real-world execution with any adversary $\mathcal{A}$ is computationally indistinguishable from the ideal-world execution with the simulator $\mathcal{S}$.

Hyb_0: $\mathcal{S}$ runs the protocol honestly, corresponding to the real-world execution.

Hyb_1: When emulating $\mathcal{F}_{\text{offline}}$, $\mathcal{S}$ delays the sampling of the honest server's shares of the masks and random triples until used. This does not affect the output distribution.

Hyb_2: For each input wire x attached to a corrupted client, after receiving the masked input value $\widetilde{x}$ from the corrupted client, $\mathcal{S}$ uses the mask value $r^{(x)}$ delivered to the corrupted client in the offline phase to extract the input value $x = \widetilde{x} - r^{(x)} \bmod N^{\zeta}$ and store it. This extraction is internal to the simulation and does not affect the distribution of the adversary's view.

Hyb_3: Similar to the offline phase, for each SPDZ multiplication on shares $[\![x]\!]$ and $[\![y]\!]$ during the input range check, $\mathcal{S}$ directly samples the honest shares of $\langle e\rangle = \langle x - a\rangle$ and $\langle d\rangle = \langle y - b\rangle$ from $\mathbb{Z}_{N^{\zeta}}$, where $[\![a]\!], [\![b]\!], [\![c]\!]$ is the triple consumed. Then, $\mathcal{S}$ reconstructs a, b, c, $\Delta \cdot a$, $\Delta \cdot b$, $\Delta \cdot c$, and computes the honest server's shares of $[\![a]\!]$, $[\![b]\!]$, and $[\![c]\!]$ by subtracting corrupt servers' shares. Since the honest server's shares of a and b are uniformly random in both cases, this change does not affect the output distribution.

Hyb_4: $\mathcal{S}$ initializes an empty list $\mathsf{ToCheck}$ for MAC checks. For each instance of $[\![z]\!] \leftarrow \Pi_{\text{mul}}([\![x]\!], [\![y]\!])$ during the input range check, let e', d' be the value opened to the honest server P_h. $\mathcal{S}$ appends $(e, e', [\![e]\!]_{\text{adv}})$ and $(d, d', [\![d]\!]_{\text{adv}})$ to $\mathsf{ToCheck}$. Additionally, $\mathcal{S}$ appends $(0, 0, [\![z]\!]_{\text{adv}})$ to $\mathsf{ToCheck}$. This is internal bookkeeping and does not affect the distribution of the adversary's view.

Hyb_5: We simulate the subsequent batch MAC check on the opened values recorded in $\mathsf{ToCheck}$ using the same hybrid argument as in the offline phase (Hyb_6–Hyb_9 in Section B.2). Therefore, the simulated transcript is computationally indistinguishable, assuming the pseudorandomness of the PRG.

Hyb_6: $\mathcal{S}$ initializes a flag $\mathsf{InputCheck} = 0$. $\mathcal{S}$ checks if any input value x from the corrupt client does not satisfy $x \in \{0, 1\}$. If so, $\mathcal{S}$ sets $\mathsf{InputCheck} = 1$. This does not affect the distribution of the adversary's view.

Hyb_7: If $\mathsf{InputCheck} = 1$, $\mathcal{S}$ aborts the simulation (after producing the transcript of the MAC check) on behalf of the honest servers and outputs the adversary's view. We show that the probability that the MAC check passes while $\mathsf{InputCheck} = 1$ is negligible under the DCR assumption. Due to the MAC check, this event only happens when there is an input $x \in [0, N^\zeta)$ such that $x \cdot (x-1) \equiv 0 \pmod{N^\zeta}$ and $x \notin \{0, 1\}$. We show that if there is an adversary that can find such an x, then it can factorize N and hence break the DCR assumption. Suppose that $N^\zeta \mid x \cdot (x-1)$ (where $a \mid b$ means a divides b). Since x is co-prime with $x - 1$, and N^ζ can be factorized to $p^\zeta \cdot q^\zeta$, we can assume, without loss of generality, that $p^\zeta \mid x$ and $q \nmid x$ (as p and q are symmetric). Then, the adversary can efficiently compute p as the greatest common divisor of x and N, thus finding the factorization of N. Hence, Hyb_7 is computationally indistinguishable from Hyb_6.

Hyb_8: After receiving all the input wire values and performed the input range check, $\mathcal{S}$ further computes the value of every wire in the circuit $\mathcal{C}$. Since $\mathcal{C}$ is $\mathcal{B}$-bounded, every wire value is an integer in $[-\mathcal{B}, \mathcal{B}]$. The computation is internal and does not affect the output distribution.

Hyb_9: After computing all wire values, $\mathcal{S}$ now assigns to every wire a masked value and an honest server's share of the label:

(a) For every wire w that's not an output of a linear gate, assign $\tilde{w} = w + r^{(w)} \pmod{N}^\zeta$, and $\langle L^{(w)} \rangle_h^* = \Delta_h \cdot \tilde{w} + \langle R^{(w)} \rangle_h^*$.

(b) For every output wire z of a linear gate with input wires x, y and coefficients μ, ν, set $\tilde{w} = \mu \cdot \tilde{x} + \nu \cdot \tilde{y}$, $\langle L^{(w)} \rangle_h^* = \mu \cdot \langle L^{(x)} \rangle_h^* + \nu \cdot \langle L^{(y)} \rangle_h^*$.

(c) For every truncation gate with input wire x and output wire z, let $x_0 = x \bmod 2$ and assign internal masked values and labels: $\widetilde{x_0} = x_0 + 2 \cdot u^{(z)} + b^{(z)} \pmod{N^\zeta}$, $\langle L^{(x_0)} \rangle_h^* = \Delta_h \cdot \widetilde{x_0} + \langle R^{(x_0)} \rangle_h^*$, $\tilde{x}' = 2 \cdot \tilde{z} + \widetilde{x_0}$, and $\langle L'^{(x)} \rangle_h^* = 2 \cdot \langle L^{(z)} \rangle_h^* + \langle L^{(x_0)} \rangle_h^*$.

Recall $r^{(w)}, u^{(z)}$ are computed from randomly sampled $\langle r^{(w)} \rangle_h^*, \langle u^{(z)} \rangle_h^*$, and $b^{(z)}$ is sampled as random bits while emulating $\mathcal{F}_{\text{offline}}$. $\langle R^{(w)} \rangle_h^*, \langle R^{(x_0)} \rangle_h^*$ are sampled during the online phase. This is internal bookkeeping and does not affect the distribution of the adversary's view.

Hyb_{10}: When simulating each key extension for a multiplication gate from wire w to $\xi = A \cdot \widetilde{w} + B$ with the generator g, $\mathcal{S}$ does not compute the second ciphertext of the honest server $\mathsf{ct}_{h,2}$ following the protocol. Instead, it solves the constraint that $(\mathsf{ct}_{h,1})^{\widetilde{w}} \cdot \mathsf{ct}_{h,2} = g^{\langle L^{(w)} \rangle_h^*} \cdot \mathsf{exp}(\langle \xi \rangle_h)$ and $\langle \xi \rangle_h =$

$\langle A\rangle_h \cdot \widetilde{w} + \langle B\rangle_h$. This does not affect the output distribution since

$$\begin{aligned}
\mathsf{ct}_{h,2} &= g^{\langle R^{(w)}\rangle_h^*} \cdot \mathsf{exp}(\langle B\rangle_h) \\
&= g^{\langle L^{(w)}\rangle_h^* - \Delta_h \cdot \widetilde{w}} \cdot \mathsf{exp}(\langle \xi\rangle_h - \langle A\rangle_h \cdot \widetilde{w}) \\
&= g^{\langle L^{(w)}\rangle_h^*} \cdot \mathsf{exp}(\langle \xi\rangle_h) \cdot \left(g^{\Delta_h} \cdot \mathsf{exp}(\langle A\rangle_h)\right)^{-\widetilde{w}} \\
&= g^{\langle L^{(w)}\rangle_h^*} \cdot \mathsf{exp}(\langle \xi\rangle_h) \cdot (\mathsf{ct}_{h,1})^{-\widetilde{w}}
\end{aligned}$$

Similar arguments hold for simulating the key extension for truncation gates from masked value $\tilde{x}'$ to $\langle \vec{\delta}^{(z)}\rangle_h \cdot \tilde{x}' + \langle \vec{K}^{(z)}\rangle_h$. For each $j \in [\lambda']$, instead of computing $\mathsf{ct}_{h,2}[j] = g^{\langle R'^{(x)}\rangle^*} \mathsf{exp}(\langle \vec{K}^{(z)}[j]\rangle_h)$, simulate it as $\mathsf{ct}_{h,2}[j] = g^{\langle L'^{(x)}\rangle_h^*} \cdot \mathsf{exp}(\langle \vec{\xi}[j]\rangle_h) \cdot (\mathsf{ct}_{h,1}[j])^{-\tilde{x}'}$, where $\langle \vec{\xi}\rangle_h = \langle \vec{\delta}^{(z)}\rangle_h \cdot \tilde{x}' + \langle \vec{K}^{(z)}\rangle$.

Similar arguments also hold for simulating the ciphertext $\mathsf{ct}_{h,2}$ for decoding information. Instead of computing $\mathsf{ct}_{h,2} = g^{\langle R^{(y)}\rangle_h^*}$, simulate it as $\mathsf{ct}_{h,2} = g^{\langle L^{(y)}\rangle_h^*} \cdot \mathsf{ct}_{h,1}^{-\tilde{y}}$.

Hyb_{11}: When simulating each Boolean key extension for a truncation gate with output wire z, $\mathcal{S}$ computes the ciphertexts without depending on $b^{(z)}$. (We omit the superscript (z) in the following to avoid cluttering.) First, let $\mathsf{idx} = \tilde{x}' \bmod 2$, which also equals $\widetilde{x_0} \bmod 2$. Then compute

$$\begin{aligned}
\mathsf{ct}_{\mathsf{idx},h}[0] &= g^{\langle \mathsf{sk}_{\mathsf{idx}}\rangle_h^*} \cdot \mathsf{exp}(\langle \xi\rangle_h), \\
\mathsf{ct}_{1-\mathsf{idx},h}[0] &= g^{\langle \mathsf{sk}_{1-\mathsf{idx}}\rangle_h^*} \cdot \mathsf{exp}(\langle \xi\rangle_h + \Delta_h \cdot (1 - 2 \cdot \mathsf{idx})), \\
\mathsf{ct}_{\mathsf{idx},h}[1] &= g^{\langle \mathsf{sk}_{\mathsf{idx}}\rangle_h^*} \cdot \mathsf{exp}(\langle \xi'\rangle_h), \\
\mathsf{ct}_{1-\mathsf{idx},h}[1] &= g^{\langle \mathsf{sk}_{1-\mathsf{idx}}\rangle_h^*} \cdot \mathsf{exp}(\langle \xi'\rangle_h + 1 - 2 \cdot \mathsf{idx}),
\end{aligned}$$

where $\langle \mathsf{sk}_{\mathsf{idx}}\rangle_h^*$, $\langle \mathsf{sk}_{1-\mathsf{idx}}\rangle_h^*$, $\langle L^{(x_0)}\rangle_h$, $\langle \widetilde{x_0}\rangle_h$ are set as

$$\begin{aligned}
\mathsf{sk}_{\mathsf{idx}} = \mathrm{Int}((\vec{\delta} \cdot \mathsf{idx}) \oplus \vec{B}), \quad \mathsf{sk}_{1-\mathsf{idx}} &= \mathrm{Int}(\vec{\delta} \oplus (\vec{\delta} \cdot \mathsf{idx}) \oplus \vec{B}), \\
\langle \mathsf{sk}_{\mathsf{idx}}\rangle_h^* = \mathsf{sk}_{\mathsf{idx}} &- (1 - \mathsf{idx}) \cdot \mathrm{Int}(\langle \vec{B}\rangle_{\mathrm{adv}} + \langle \vec{0}\rangle_{\mathrm{adv}}) - \mathsf{idx} \cdot \mathrm{Int}(\langle \vec{\delta} \oplus \vec{B}\rangle_{\mathrm{adv}} + \langle \vec{0}\rangle_{\mathrm{adv}}) \\
\langle \mathsf{sk}_{1-\mathsf{idx}}\rangle_h^* = \mathsf{sk}_{1-\mathsf{idx}} &- \mathsf{idx} \cdot \mathrm{Int}(\langle \vec{B}\rangle_{\mathrm{adv}} + \langle \vec{0}\rangle_{\mathrm{adv}}) - (1 - \mathsf{idx}) \cdot \mathrm{Int}(\langle \vec{\delta} \oplus \vec{B}\rangle_{\mathrm{adv}} + \langle \vec{0}\rangle_{\mathrm{adv}}) \\
\langle \xi\rangle_h = \langle L^{(x_0)}\rangle_h &+ \Delta_{\mathrm{adv}} \cdot \widetilde{x_0} \\
&- \langle \Delta \cdot r^{(x_0)}\rangle_{\mathrm{adv}} - \mathsf{idx} \cdot \Delta_{\mathrm{adv}} - (1 - 2 \cdot \mathsf{idx}) \cdot \langle \Delta \cdot b^{(z)}\rangle_{\mathrm{adv}}, \\
\langle \xi'\rangle_h = \widetilde{x_0} &- \mathsf{idx} \cdot \langle 1\rangle_{\mathrm{adv}} - (1 - 2 \cdot \mathsf{idx}) \cdot \langle b^{(z)}\rangle_{\mathrm{adv}}.
\end{aligned}$$

It can be verified (by expanding our definition of $\widetilde{x_0}$ and idx) that the above simulated ciphertexts are distributed identically to those in the real execution.

Hyb_{12}: When simulating the honest server's share of target shifts $\langle V\rangle_h$ and $\langle v\rangle_h$ in each truncation gate, $\mathcal{S}$ does not follow the honest protocol. Instead, it simulates them as

$$\begin{aligned}
\langle V\rangle_h &= \langle L'^{(x)}\rangle_h - \langle L^{(x)}\rangle_h + \Delta_{\mathrm{adv}} \cdot (\tilde{x}' - \tilde{x}) \\
&\quad - 2 \cdot (\langle \Delta \cdot u^{(z)}\rangle_{\mathrm{adv}} + \langle \Delta \cdot r^{(z)}\rangle_{\mathrm{adv}}) - \langle \Delta \cdot b^{(z)}\rangle_{\mathrm{adv}} + \langle \Delta \cdot r^{(x)}\rangle_{\mathrm{adv}}, \\
\langle v\rangle_h &= \tilde{x}' - \tilde{x} - 2 \cdot (\langle u^{(z)}\rangle_{\mathrm{adv}} + \langle r^{(z)}\rangle_{\mathrm{adv}}) - \langle b^{(z)}\rangle_{\mathrm{adv}} + \langle r^{(x)}\rangle_{\mathrm{adv}}.
\end{aligned}$$

It can be verified that the above simulated honest shares of target shifts are distributed identically to those in the real execution.

Hyb_{13}: For each wire with value w in case (a) described in Hyb_9, $\mathcal{S}$ does not sample $\langle r^{(w)} \rangle_h^*$ directly when emulating $\mathcal{F}_{\text{offline}}$. Instead, it samples $\widetilde{w}_h \leftarrow [0, r_{\max})$ and lets $\langle r^{(w)} \rangle_h^* = \widetilde{w}_h - w$. Since $w \in [-\mathcal{B}, \mathcal{B}]$ and $r_{\max} > |\mathcal{C}| \cdot \mathcal{B} \cdot 2^\kappa$, the statistical distance induced by each wire is at most $\frac{1}{2^\kappa \cdot |\mathcal{C}|}$ (note that the corrupt parties' shares of $\langle r^{(w)} \rangle^*$ are chosen independently of the $\langle r^{(w)} \rangle_h^*$, so the change does not affect whether the mask range check aborts except with negligible probability). By a union bound over all output wires, we show Hyb_{13} is statistically indistinguishable from Hyb_{12}.

Hyb_{14}: For each wire with value w in case (a), $\mathcal{S}$ does not compute the honest server's share $\langle R^{(w)} \rangle^*$ following the protocol. Instead, it samples $\langle L^{(w)} \rangle_h^* \leftarrow [0, R_{\max})$ and lets $\langle R^{(w)} \rangle_h^* = \langle L^{(w)} \rangle_h^* - \Delta_h \cdot \widetilde{w}'$ where $\widetilde{w}' \in [-N^\zeta/2, N^\zeta/2)$ and $\widetilde{w}' \equiv \widetilde{w} \equiv w + r^{(w)} \mod N^\zeta$. Since $w \in [-\mathcal{B}, \mathcal{B}]$ and the offline functionality ensures $r^{(w)} \bmod N^\zeta \in [0, r_{\max}^{\text{relax}}) \cup (N^\zeta - r_{\max}^{\text{relax}}, N^\zeta)$, we have that $\widetilde{w}' \in [-r_{\max}^{\text{relax}} - \mathcal{B}, r_{\max}^{\text{relax}} + \mathcal{B})$. Given that $r_{\max}^{\text{relax}} > \mathcal{B}$, $\Delta_h \in [0, \Delta_{\max})$, and $R_{\max} = |\mathcal{C}| \cdot \Delta_{\max} \cdot r_{\max}^{\text{relax}} \cdot 2^{\kappa+1}$, the statistical distance induced by each wire is at most $\frac{1}{2^{\kappa-1} \cdot |\mathcal{C}|}$. By a union bound over all output wires, we show Hyb_{14} is statistically indistinguishable from Hyb_{13}.

Hyb_{15}: For each truncation gate in case (c), $\mathcal{S}$ does not sample $b^{(z)}$ as a random bit and $\langle u^{(z)} \rangle_h^*$ from $[0, r_{\max})$ while emulating $\mathcal{F}_{\text{offline}}$. Note that they are only used for computing $\widetilde{x_0} = x_0 + 2 \cdot u^{(z)} + b^{(z)} = x_0 + 2 \cdot (\langle u^{(z)} \rangle_h + \langle u^{(z)} \rangle_{\text{adv}}) + b^{(z)}$. We instead directly sample $v \leftarrow [0, 2 \cdot r_{\max})$ and set $\widetilde{x_0} = v^{(z)} + \langle u^{(z)} \rangle_{\text{adv}}$. Since the term $(2\langle u \rangle_h + b^{(z)})$ is distributed uniformly over $[0, 2 \cdot r_{\max})$, the change in $\widetilde{x_0}$ is statistically indistinguishable.

Hyb_{16}: For each truncation gate in case (c), $\mathcal{S}$ does not compute the honest server's share of $\langle R^{(x_0)} \rangle_h^*$ following the protocol. Instead, it samples $\langle L^{(x_0)} \rangle_h^* \leftarrow [0, R_{\max})$ and lets $\langle R^{(x_0)} \rangle_h^* = \langle L^{(x_0)} \rangle_h^* - \Delta_h \cdot \widetilde{x_0}$. Similarly to Hyb_{14}, we can show that this change is statistically indistinguishable.

Hyb_{17}: For each multiplication gate, $\mathcal{S}$ does not sample the honest server's shares of $\langle U \rangle$ and $\langle U' \rangle$ following the protocol. Instead, it directly samples $\langle \xi_x \rangle_h$ and $\langle \xi'_x \rangle_h$ from $\mathbb{Z}_{N^\zeta}$ and let

$$\langle U \rangle_h = \langle \xi_x \rangle_h - \langle S \rangle_h \cdot \widetilde{x} \quad \text{and} \quad \langle U' \rangle_h = \langle \xi'_x \rangle_h - \langle r^{(y)} \rangle_h^* \cdot \widetilde{x}$$

Since $\langle U \rangle_h$ and $\langle U' \rangle_h$ are independently and uniformly random in both cases, this change does not affect the output distribution.

Hyb_{18}: For each multiplication gate, $\mathcal{S}$ does not compute the target $\langle \xi_y \rangle_h = \langle T \rangle_h \cdot \widetilde{y} + \langle V \rangle_h$ and $\langle \xi_y \rangle'_h = \langle r^{(x)} \rangle_h^* \cdot \widetilde{y} + \langle V' \rangle_h$ following the protocol. Instead, it calculates

$$\begin{aligned}
\langle \xi_y \rangle_h &= \widetilde{x} \cdot \langle L^{(y)} \rangle_h^* - \widetilde{x} \cdot \langle \Delta \cdot r^{(y)} \rangle_{\text{adv}} - \widetilde{y} \cdot \langle \Delta \cdot r^{(x)} \rangle_{\text{adv}} + (\widetilde{x} \cdot \widetilde{y} - \widetilde{z}) \cdot \langle \Delta \rangle_{\text{adv}}^* \\
&\quad + \langle \Delta \cdot r^{(x)} \cdot r^{(y)} \rangle_{\text{adv}} + \langle \Delta \cdot r^{(z)} \rangle_{\text{adv}} - \langle L^{(z)} \rangle_h^* - \langle \xi_x \rangle_h \pmod{N^\zeta} \\
\langle \xi'_y \rangle_h &= \widetilde{x} \cdot \widetilde{y} - \widetilde{x} \cdot \langle r^{(y)} \rangle_{\text{adv}}^* - \widetilde{y} \cdot \langle r^{(x)} \rangle_{\text{adv}}^* - \widetilde{z} + \langle r^{(x)} \cdot r^{(y)} \rangle_{\text{adv}} + \langle r^{(z)} \rangle_{\text{adv}}^* \\
&\quad - \langle \xi'_x \rangle_h \pmod{N^\zeta}
\end{aligned}$$

where $\langle X \rangle_{\text{adv}} = \sum_{i \neq h} \langle X \rangle_i$ for any shared value $\langle X \rangle$. The correctness of the above calculation can be verified by expanding the terms. We verify this via symbolic computation (e.g., using Wolfram

Mathematica) using the following relations:

$$\begin{aligned}
\langle R^{(w)}\rangle_h^* &= \langle L^{(w)}\rangle_h^* - \widetilde{w}\cdot\Delta_h \quad \forall w\in\{x,y,z\}\\
\langle \Delta\cdot r^{(w)}\rangle_h &= (\Delta_h + \langle\Delta\rangle_{\text{adv}}^*)\cdot(\widetilde{w}-w) - \langle\Delta\cdot r^{(w)}\rangle_{\text{adv}} \quad \forall w\in\{x,y,z\}\\
\langle \Delta\cdot r^{(x)}\cdot r^{(y)}\rangle_h &= (\Delta_h + \langle\Delta\rangle_{\text{adv}}^*)\cdot(\widetilde{x}-x)\cdot(\widetilde{y}-y) - \langle\Delta\cdot r^{(x)}\cdot r^{(y)}\rangle_{\text{adv}}\\
\langle S\rangle_h &= \langle\Delta\cdot r^{(y)}\rangle_h + \langle R^{(y)}\rangle_h^*, \quad \langle T\rangle_h = \langle\Delta\cdot r^{(x)}\rangle_h\\
\langle U\rangle_h &= \langle\xi_x\rangle_h - \langle S\rangle_h\cdot\widetilde{x}\\
\langle V\rangle_h &= -\langle U\rangle_h - \langle\Delta\cdot r^{(x)}\cdot r^{(y)}\rangle_h - \langle\Delta\cdot r^{(z)}\rangle_h - \langle R^{(z)}\rangle_h^*\\
\langle\xi_y\rangle_h &= \langle T\rangle_h\cdot\widetilde{y} + \langle V\rangle_h
\end{aligned}$$

and similarly for $\langle\xi'_y\rangle_h$:

$$\begin{aligned}
\langle r^{(w)}\rangle_h^* &= (\widetilde{w}-w) - \langle r^{(w)}\rangle_{\text{adv}}^* \quad \forall w\in\{x,y,z\}\\
\langle r^{(x)}\cdot r^{(y)}\rangle_h &= (\widetilde{x}-x)\cdot(\widetilde{y}-y) - \langle r^{(x)}\cdot r^{(y)}\rangle_{\text{adv}}^*\\
\langle U'\rangle_h &= \langle\xi'_x\rangle_h - \langle r^{(y)}\rangle_h^*\cdot\widetilde{x}\\
\langle V'\rangle_h &= -\langle U'\rangle_h - \langle r^{(x)}\cdot r^{(y)}\rangle_h^* - \langle r^{(z)}\rangle_h^*\\
\langle\xi'_y\rangle_h &= \langle r^{(x)}\rangle_h^*\cdot\widetilde{y} + \langle V'\rangle_h
\end{aligned}$$

To this point, the honest server's shares of $\langle S\rangle, \langle T\rangle, \langle U\rangle, \langle U'\rangle, \langle V\rangle, \langle V'\rangle$ are not used in the simulation, so $\mathcal{S}$ no longer computes them.

Hyb_{19}: When simulating the key extension for a truncation gate, instead of using $\langle\vec{\xi}\rangle_h = \langle\vec{\delta}^{(z)}\rangle_h\cdot\tilde{x}' + \langle\vec{K}^{(z)}\rangle_h$, set

$$\langle\vec{\xi}\rangle_h = \underbrace{\vec{\delta}^{(z)}\cdot\tilde{x}' + 2\cdot\vec{U}^{(z)} + \vec{B}^{(z)}}_{\vec{\xi}} - \underbrace{\langle\vec{\delta}^{(z)}\rangle_{\text{adv}}\cdot\tilde{x}' - 2\cdot\langle\vec{U}^{(z)}\rangle_{\text{adv}} - \langle\vec{B}^{(z)}\rangle_{\text{adv}}}_{\langle\vec{\xi}\rangle_{\text{adv}}},$$

It can be verified that this change is an equivalent way of computing $\langle\vec{\xi}\rangle_h$. Further substituting $\tilde{x}' = \widetilde{x_0} + 2\cdot\tilde{z}$, and $\mathsf{idx} = \widetilde{x_0} \bmod 2$, we have

$$\vec{\xi} = (\vec{\delta}^{(z)}\cdot\mathsf{idx})\oplus\vec{B}^{(z)} + 2\cdot(\vec{U}^{(z)} + (\tilde{z} + \lfloor\widetilde{x_0}/2\rfloor)\cdot\vec{\delta}^{(z)})$$

Since $\langle\vec{U}^{(z)}\rangle_h$ is sampled at random from a range much larger than $\tilde{z}$, $\widetilde{x_0}$ and $\vec{\delta}^{(z)}$, we can instead set $\vec{\xi} = (\vec{\delta}^{(z)}\cdot\mathsf{idx})\oplus\vec{B}^{(z)} + 2\cdot\vec{U}^{(z)}$ with negligible statistical distance. Furthermore, by the randomness of $\vec{B}^{(z)}$, we can instead set $\vec{\xi} = \vec{B}^{(z)} + 2\cdot\vec{U}^{(z)}$, $\mathsf{sk}_{\mathsf{idx}} = \vec{B}^{(z)}$, $\mathsf{sk}_{1-\mathsf{idx}} = \vec{\delta}^{(z)}\oplus\vec{B}^{(z)}$, without affecting the distribution.

Hyb_{20}: We now switch the ciphertexts depending on the secret keys Δ_h, and $\vec{\delta}^{(z)}$ to encryptions of random values. They include

1. Key extension ciphertexts for multiplication gates: they have the form $\mathsf{ct}_{h,1} = g^{\Delta_h}\cdot\mathsf{exp}(\langle S\rangle_h)$, or $\mathsf{ct}_{h,1} = g^{\Delta_h}\cdot\mathsf{exp}(\langle T\rangle_h)$, where g are freshly sampled generators for different ciphertexts, and

   $$\begin{aligned}
   \langle S\rangle_h &= \langle\Delta\cdot r^{(y)} + R^y\rangle_h = (\Delta_h + \langle\Delta\rangle_{\text{adv}}^*)\cdot r^{(y)} - \langle\Delta\cdot r^{(y)}\rangle_{\text{adv}} + \langle L^{(y)}\rangle_h^* - \Delta_h\cdot\widetilde{y}\\
   \langle T\rangle_h &= \langle\Delta\cdot r^{(x)}\rangle_h = (\Delta_h + \langle\Delta\rangle_{\text{adv}}^*)\cdot r^{(x)} - \langle\Delta\cdot r^{(x)}\rangle_{\text{adv}}
   \end{aligned}$$

   We note that both of them are affine functions of Δ_h.

2. Key extension ciphertexts for truncation gates: they have the form $\mathsf{ct}_{h,1}[j] = g^{\Delta_h} \cdot \mathsf{exp}(\langle \vec{\delta}^{(z)}[j] \rangle_h)$, where
$$\langle \vec{\delta}^{(z)} \rangle_h = \vec{\delta}^{(z)} - \langle \vec{\delta}^{(z)} \rangle_{\mathrm{adv}},$$
which is an affine function of $\vec{\delta}^{(z)}$.
3. Boolean ciphertext for truncation gates: they have the form
$$\begin{aligned}\mathsf{ct}_{1-\mathsf{idx}} &= g^{\langle \mathsf{sk}^{(z)}_{1-\mathsf{idx}} \rangle^*_h} \cdot \mathsf{exp}(\langle \xi \rangle_h + \Delta_h \cdot (1 - 2 \cdot \mathsf{idx})) \\ &= \underbrace{g^{\mathsf{sk}^{(z)}_{1-\mathsf{idx}}} \cdot \mathsf{exp}(\langle \xi \rangle_h + \Delta_h \cdot (1 - 2 \cdot \mathsf{idx}))}_{\mathsf{ct}'_{1-\mathsf{idx}}} \cdot g^{\langle \mathsf{sk}^{(z)}_{1-\mathsf{idx}} \rangle^*_{\mathrm{adv}}},\end{aligned}$$
where the term $\langle \mathsf{sk}^{(z)}_{1-\mathsf{idx}} \rangle^*_{\mathrm{adv}}$ is independent of $\vec{\delta}^{(z)}$. We focus on switching the term $\mathsf{ct}'_{1-\mathsf{idx}}$, as we can simply add back $g^{\langle \mathsf{sk}^{(z)}_{1-\mathsf{idx}} \rangle^*_{\mathrm{adv}}}$ after switching. The secret key $\mathsf{sk}^{(z)}_{1-\mathsf{idx}}$ is computed as $\mathsf{sk}^{(z)}_{1-\mathsf{idx}} = \mathrm{Int}(\vec{\delta}^{(z)} \oplus \vec{B}^{(z)})$.
4. Decoding information ciphertexts: they have the form $\mathsf{ct}_{h,1} = g^{\Delta_h}$.

Before applying the KDM security assumption, we perform a further change by sampling each $\mathsf{sk}^{(z)}_{1-\mathsf{idx}} \leftarrow [2^{\lambda'}]$ at random instead, and set the message $\langle \vec{\delta}^{(z)} \rangle_h$ to the above key extension ciphertext for truncation gates as
$$\langle \vec{\delta}^{(z)} \rangle_h = (\mathrm{BitsOf}(\mathsf{sk}^{(z)}_{1-\mathsf{idx}}) \oplus \vec{B}) - \langle \vec{\delta}^{(z)} \rangle_{\mathrm{adv}}.$$
Such a message can be viewed as affine functions over each individual bit of the secret key $\mathsf{sk}^{(z)}_{1-\mathsf{idx}}$. Now our ciphertexts to switch match exactly the KDM oracle interfaces, and hence can be simulated with the oracle $\mathcal{O}_{\mathrm{real}}$ in Lemma B.13. By assumption, it can be replaced with $\mathcal{O}_{\mathrm{random}}$, which switches the encrypted messages to random. This switch is computationally indistinguishable by Lemma B.13.

We also note that if the circuit has no truncation gates, then we only need to switch type 1 and 4 ciphertexts from the above. In this case, they match the simpler KDM oracle interface $\mathcal{O}_0^{\mathsf{KDM},\Delta_h}$ of Lemma 3.2. By assumption, it can be replaced with $\mathcal{O}_1^{\mathsf{KDM},\Delta_h}$, which switches the encrypted messages to random. This switch is computationally indistinguishable under the standard DCR assumption by Lemma 3.2.

To this point, $\mathcal{S}$ no longer needs to compute $\langle r^{(w)} \rangle^*_h$ or the value of any wire w that is not an input/output wire associated with a corrupt client.

Hyb_{21}: Finally, we switch from following the output verification step for output wires y associated with an honest client C to following the step in the simulator description. When following the honest verification step, $\mathcal{S}$ receives $\mathsf{ct}, \mathsf{ct}_{\mathrm{adv},1}, \mathsf{ct}_{\mathrm{adv},2}$, and a masked value $\tilde{y}^*$, and checks
$$\begin{aligned}\mathsf{ct} &\stackrel{?}{=} (\mathsf{ct}_{\mathrm{adv},1} \cdot \mathsf{ct}_{h,1})^{\tilde{y}^*} \cdot (\mathsf{ct}_{\mathrm{adv},2} \cdot \mathsf{ct}_{h,2}) \\ &= (\mathsf{ct}_{\mathrm{adv},1} \cdot g^{\Delta_h} \cdot \mathsf{exp}(m))^{\tilde{y}^*} \cdot \mathsf{ct}_{\mathrm{adv},2} \cdot g^{\langle L^{(y)} \rangle^*_h} / (g^{\Delta_h} \cdot \mathsf{exp}(m))^{\tilde{y}},\end{aligned}$$
where m is a random message returned by the KDM oracle $\mathcal{O}_{\mathrm{random}}$.

If $\tilde{y}^*$ does not equal the simulated $\tilde{y}$, i.e. $\epsilon = \tilde{y}^* - \tilde{y} \neq 0$, then the check can be re-written as
$$\mathsf{ct} / (\mathsf{ct}_{\mathrm{adv},1})^{\tilde{y}^*} / \mathsf{ct}_{\mathrm{adv},2} \stackrel{?}{=} g^{\langle L^{(y)} \rangle^*_h} \cdot g^{\Delta_h \cdot \epsilon} \cdot \mathsf{exp}(m \cdot \epsilon).$$

Since $\epsilon \neq 0$, and by the range check on $\tilde{y}^*$, we have $\epsilon < n \cdot r_{\max} < N^\zeta/(|C| \cdot 2^\kappa)$. This means there are at least $N^\zeta/\epsilon > |C| \cdot 2^\kappa$ number of possible values for $m \cdot \epsilon$. Over the randomness of m, the honest verification fails except with $< 1/(|C| \cdot 2^\kappa)$ probability. This means the honest verification check is equivalent, except with negligible probability, to checking $\epsilon = 0$ and $\mathsf{ct}/(\mathsf{ct}_{\mathrm{adv},1})^{\tilde{y}^*}/\mathsf{ct}_{\mathrm{adv},2} \stackrel{?}{=} g^{\langle L^{(y)} \rangle_h^*}$, as in the simulator description.

Hyb_{22}: $\mathcal{S}$ no longer uses the honest clients' input values or evaluates the circuit by itself. Instead, it sends the corrupt clients' inputs (extracted in Hyb_2) to $\mathcal{F}_{\mathrm{main}}$ and receives the corrupt clients' output values. Since the circuit's evaluation remains the same, Hyb_{22} is identical to Hyb_{21}.

We note that Hyb_{22} is the same as the ideal-world execution with $\mathcal{S}$. Thus, the real-world execution is computationally indistinguishable from the ideal world.

Finally, we show that the protocol does not abort when all parties follow the protocol honestly. We analyze the conditions under which the main protocol might abort when all parties follow the protocol honestly:

- While emulating the offline phase, the functionality $\mathcal{F}_{\mathrm{offline}}$ aborts only if there exists a mask $r^{(w)}$ such that $r^{(w)} \bmod N^\zeta \in [r_{\max}^{\mathrm{relax}}, N^\zeta - r_{\max}^{\mathrm{relax}}]$. Since honest shares $\langle r^{(w)} \rangle_i^*$ are sampled from $[0, r_{\max})$, the sum $r^{(w)} < n \cdot r_{\max} < r_{\max}^{\mathrm{relax}}$, so it is impossible for $r^{(w)}$ to fall into the abort range.

- The servers abort if the received masked input $\widetilde{x}$ is not in $[0, n \cdot r_{\max})$. We show that this is impossible if all the parties are honest. Recall that the client computes $\widetilde{x} = x + r^{(x)}$ where x is a binary input and $r^{(x)} = \sum_{i=1}^n \langle r^{(x)} \rangle_i^*$. Since each honest share $\langle r^{(x)} \rangle_i^*$ is sampled from $[0, r_{\max})$, we have $0 \leq \widetilde{x} \leq 1 + n \cdot (r_{\max} - 1) < n \cdot r_{\max}$.

- The servers abort during the batch verification step if any opened value during the multiplications is inconsistent or if any $[\![z]\!]$ does not open to 0. Since all parties are honest, all opened values are consistent, and for each input wire x, we have $x \cdot (x-1) = 0$ since x is a bit. Thus, all $[\![z]\!]$ open to 0.

- During the output phase, the client aborts the protocol if $\prod_{i=1}^n G_i \neq 1$, or it receives an inconsistent masked value from the servers, or the reconstructed output is not in $[-B, B]$. To see why all checks pass when all parties are honest, we consider the ideal world execution, where we have $G_i' = G_i$ for every $i \in [n]$ and $G_h = (\prod_{i \neq h} G_i)^{-1}$. Therefore, $\mathcal{S}$ never aborts when checking $\prod_{i=1}^n G_i' = 1$. Similarly, we have $(\widetilde{y_j})_i = (\widetilde{y_j})_h$ for all $i \neq h$, so $\mathcal{S}$ never aborts. By the indistinguishability between the real and ideal executions, the same check also passes (with overwhelming probability) in the real execution when all parties are honest.

To summarize, the protocol only aborts with negligible probability when all parties are honest. □

B.4 DCR Assumption and Proof of Lemma 3.2

First, we define the DCR assumption [Pai99], which has been widely used in the design of homomorphic encryption schemes and related cryptographic primitives. Below, we adopt a variant that requires the prime factors to be safe primes, i.e., $p = 2p' + 1$ and $q = 2q' + 1$ for some primes p', q'. This variant is commonly used in previous literature [CS02, LLPT23, ADOS22].

Definition B.15 (Decision Composite Residuosity Assumption (DCR), [Pai99])**.** Let Gen be a polynomial-time algorithm which, on input a security parameter λ, outputs (N, p, q) where p and

q are λ-bit safe primes and $N = pq$. Let λ be a security parameter. We say that the Decision Composite Residuosity (DCR) problem is hard relative to modulus-sampling algorithm Gen if

$$\left\{(N,x) : \begin{matrix}(N,p,q) \leftarrow \mathsf{Gen}(1^\lambda) \\ x \leftarrow \mathbb{Z}_{N^2}^\times\end{matrix}\right\} \approx_c \left\{(N, x^N \bmod N^2) : \begin{matrix}(N,p,q) \leftarrow \mathsf{Gen}(1^\lambda) \\ x \leftarrow \mathbb{Z}_{N^2}^\times\end{matrix}\right\}.$$

From now on, when we say p, q are random λ-bit safe primes, we mean that they are generated by the above Gen algorithm. We also use p', q' to denote the corresponding Sophie Germain primes, i.e., $p = 2p' + 1$ and $q = 2q' + 1$.

Now, we introduce the following lemma about the subgroup structure of $\mathbb{Z}_{N^{\zeta+1}}^\times$ that will be useful in the proof.

Lemma B.16 (Subset Membership Hardness [CS02,LLPT23])**.** *Let $N = p \cdot q$ where p, q are distinct random λ-bit safe primes, $X = \{a \mid a \in \mathbb{Z}_{N^{\zeta+1}}^\times, (\frac{a}{N}) = 1\}$ be the subgroup of $\mathbb{Z}_{N^{\zeta+1}}^\times$ with Jacobi symbol 1 modulo N, and $Y = \{a^{N^\zeta} \mid a \in X\}$ be the subgroup of N^ζ-th powers in X. Then, we have the following:*

1. *The subgroups X and Y are cyclic, with order $2N^\zeta p'q'$ and $2p'q'$ respectively. Knowing only N, one can efficiently sample (statistically close to uniform) a generator of Y.*
2. *X is the direct product $Y \times H$, where $H = \{\mathsf{exp}(m) \mid m \in \mathbb{Z}_{N^\zeta}\}$.*
3. *Assuming DCR, the uniform distribution over X is computationally indistinguishable from the uniform distribution over Y.*

Next, we present the proof of Lemma 3.2, which follows a similar approach to prior works [LLPT23, BMO+26].

Lemma 3.2 (KDM Security of the DCR-based LHE)**.** *Under the DCR assumption, the LHE scheme in Figure 3 is nonce-based and KDM secure with respect to the class of affine functions $f(x) = a \cdot x + b$ (Definition 3.1), where a, b are constants in $\mathbb{Z}_{N^\zeta}$.*

Proof. Let $Y = \{a^{N^\zeta} \mid a \in X\}$ be the subgroup of N^ζ-th powers in X. We define the following hybrids. Since the encryption algorithm already outputs its randomness g, we omit the same randomness in the oracle responses for brevity.

Hyb_0: Let $\Delta \xleftarrow{\$} [0, \lfloor N/2 \rfloor)$ (we omit it for Hyb_1 - Hyb_4). The oracle samples $g \leftarrow X$ and returns $\big(g,\ g^\Delta \cdot \mathsf{exp}(a \cdot \Delta + b)\big)$. This is identical to $\mathcal{O}_0^{\mathsf{KDM},\Delta}(a, b)$.

Hyb_1: The oracle samples $v \leftarrow Y$, $m \leftarrow [0,\ N^\zeta)$ and returns

$$\big(v \cdot \mathsf{exp}(m),\ v^\Delta \cdot \mathsf{exp}((a+m) \cdot \Delta + b)\big).$$

Hyb_2: The oracle samples $v \leftarrow Y$, $m \leftarrow [0,\ N^\zeta)$ and returns

$$\big(v \cdot \mathsf{exp}(m-a),\ v^\Delta \cdot \mathsf{exp}(m \cdot \Delta + b)\big).$$

Hyb_3: The oracle samples $g \leftarrow X$ and returns $\big(g \cdot \mathsf{exp}(-a),\ g^\Delta \cdot \mathsf{exp}(b)\big)$.

Hyb_4: The oracle samples $g \leftarrow Y$ and returns $\big(g \cdot \mathsf{exp}(-a),\ g^\Delta \cdot \mathsf{exp}(b)\big)$.

Hyb_5: Let $\Delta \xleftarrow{\$} [0,\, \lfloor N/2 \rfloor \cdot N^\zeta)$. The oracle samples $g \leftarrow Y$ and returns

$$\left(g \cdot \mathsf{exp}(-a),\, g^\Delta \cdot \mathsf{exp}(b)\right).$$

Hyb_6: Let $\Delta \xleftarrow{\$} [0,\, \lfloor N/2 \rfloor \cdot N^\zeta)$. The oracle samples $g \leftarrow X$ and returns

$$\left(g \cdot \mathsf{exp}(-a),\, g^\Delta \cdot \mathsf{exp}(b)\right).$$

Hyb_7: Let $\Delta \xleftarrow{\$} [0,\, 2p'q' \cdot N^\zeta)$. The oracle samples $g \leftarrow X$ and returns

$$\left(g \cdot \mathsf{exp}(-a),\, g^\Delta \cdot \mathsf{exp}(b)\right).$$

Hyb_8: Let $\Delta \xleftarrow{\$} [0,\, 2p'q' \cdot N^\zeta)$. The oracle samples $v \leftarrow Y$, $m \leftarrow [0,\, N^\zeta)$ and returns

$$\left(v \cdot \mathsf{exp}(m - a),\, v^\Delta \cdot \mathsf{exp}(m \cdot \Delta + b)\right).$$

Hyb_9: Let $\Delta_1 \xleftarrow{\$} [0,\, 2p'q')$, $\Delta_2 \xleftarrow{\$} [0,\, N^\zeta)$. The oracle samples $v \leftarrow Y$, $m \leftarrow \mathbb{Z}_{N^\zeta}$ and returns

$$\left(v \cdot \mathsf{exp}(m - a),\, v^{\Delta_1} \cdot \mathsf{exp}(m \cdot \Delta_2 + b)\right).$$

Hyb_{10}: Let $\Delta \xleftarrow{\$} [0,\, 2p'q')$. The oracle samples $v \leftarrow Y$, $m \leftarrow \mathbb{Z}_{N^\zeta}$, $z \leftarrow \mathbb{Z}_{N^\zeta}$ and returns

$$\left(v \cdot \mathsf{exp}(m),\, v^\Delta \cdot \mathsf{exp}(m \cdot \Delta + z)\right).$$

Hyb_{11}: Let $\Delta \xleftarrow{\$} [0,\, 2p'q')$. The oracle samples $g \leftarrow X$, $z \leftarrow \mathbb{Z}_{N^\zeta}$ and returns $\left(g,\, g^\Delta \cdot \mathsf{exp}(z)\right)$.

Hyb_{12}: Let $\Delta \xleftarrow{\$} [0,\, \lfloor N/2 \rfloor)$. The oracle samples $g \leftarrow X$, $z \leftarrow \mathbb{Z}_{N^\zeta}$ and returns $\left(g,\, g^\Delta \cdot \mathsf{exp}(z)\right)$. This is identical to $\mathcal{O}_1^{\mathsf{KDM},\Delta}(a, b)$.

Next we show that the hybrids are computationally indistinguishable assuming DCR.

- $\mathsf{Hyb}_0 = \mathsf{Hyb}_1$. By property 2 of Lemma B.16, the two hybrids are identical.
- $\mathsf{Hyb}_1 = \mathsf{Hyb}_2$. We can obtain Hyb_2 from Hyb_1 by substituting m with $m - a$.
- $\mathsf{Hyb}_2 = \mathsf{Hyb}_3$. Again, by property 2 of Lemma B.16, the hybrids are identical.
- $\mathsf{Hyb}_3 \approx_c \mathsf{Hyb}_4$. By property 3 of Lemma B.16, the two hybrids are computationally indistinguishable.
- $\mathsf{Hyb}_4 \approx_s \mathsf{Hyb}_5$. Recall that $p = 2p' + 1$ and $q = 2q' + 1$, so the difference between the range size $\lfloor N/2 \rfloor$ and $2p'q'$ is approximately $(p + q)/2 \approx \sqrt{N}$, which is negligible relative to N. Thus, we can switch the range of Δ from $[0, \lfloor N/2 \rfloor)$ to $[0, 2p'q')$ with negligible statistical distance. Next, since $g \in Y$ and the exponent of group Y is $2p'q'$, the value g^Δ depends only on Δ mod $2p'q'$. Therefore, changing the range to $[0, 2p'q' \cdot N^\zeta)$ does not alter the distribution of g^Δ. Finally, we switch the range of Δ again from $[0, 2p'q' \cdot N^\zeta)$ to $[0, \lfloor N/2 \rfloor \cdot N^\zeta)$, which is statistically indistinguishable for the same reason as the first step.

- $\mathsf{Hyb}_5 \approx_c \mathsf{Hyb}_6$. By property 3 of Lemma B.16, the two hybrids are computationally indistinguishable.

- $\mathsf{Hyb}_6 \approx_s \mathsf{Hyb}_7$. Similar to the transition from Hyb_4 to Hyb_5, we change the range of Δ from $[0, \lfloor N/2 \rfloor \cdot N^\zeta)$ to $[0, 2p'q' \cdot N^\zeta)$. The statistical distance is negligible because the difference in range sizes is negligible relative to the size of the range.

- $\mathsf{Hyb}_7 = \mathsf{Hyb}_8$. By property 2 of Lemma B.16, we can uniquely decompose any $g \in X$ as $g = v \cdot \mathsf{exp}(m)$ where $v \in Y$ and $m \in \mathbb{Z}_{N^\zeta}$. Thus, sampling $g \leftarrow X$ is equivalent to sampling $v \leftarrow Y$ and $m \leftarrow \mathbb{Z}_{N^\zeta}$ and computing g as above. Plugging this decomposition into the oracle's output shows that the two hybrids are identical.

- $\mathsf{Hyb}_8 = \mathsf{Hyb}_9$. Since $\gcd(2p'q', N^\zeta) = 1$, by the Chinese Remainder Theorem, the map $\Delta \mapsto (\Delta \bmod 2p'q', \Delta \bmod N^\zeta)$ is a bijection from $\mathbb{Z}_{2p'q' \cdot N^\zeta}$ to $\mathbb{Z}_{2p'q'} \times \mathbb{Z}_{N^\zeta}$. Thus, sampling Δ uniformly from $[0, 2p'q' \cdot N^\zeta)$ is equivalent to sampling $\Delta_1 \leftarrow [0, 2p'q')$ and $\Delta_2 \leftarrow [0, N^\zeta)$ independently. We then replace v^Δ with v^{Δ_1} (since the order of v divides $2p'q'$) and $\mathsf{exp}(m \cdot \Delta)$ with $\mathsf{exp}(m \cdot \Delta_2)$.

- $\mathsf{Hyb}_9 \approx_c \mathsf{Hyb}_{10}$. First, since m is chosen uniformly from $[0, N^\zeta)$, it is co-prime to N with overwhelming probability. When $\gcd(m, N) = 1$, m is invertible modulo N^ζ. Since Δ_2 is uniform in $\mathbb{Z}_{N^\zeta}$ and independent of m, the value $y = m \cdot \Delta_2 + b \bmod N^\zeta$ is uniformly distributed and independent of m. Consequently, in Hyb_9, the pair $(m - a, m \cdot \Delta_2 + b)$ is statistically close to uniform over $(\mathbb{Z}_{N^\zeta})^2$ in each oracle call. On the other hand, in hybrid Hyb_{10}, the oracle returns $\big(v \cdot \mathsf{exp}(m), v^\Delta \cdot \mathsf{exp}(m \cdot \Delta + z)\big)$. Since z is uniform, $(m, m \cdot \Delta + z)$ is also uniform over $(\mathbb{Z}_{N^\zeta})^2$. Thus, the distributions in the two hybrids are computationally indistinguishable.

- $\mathsf{Hyb}_{10} = \mathsf{Hyb}_{11}$. This step reverses the decomposition of g and Δ (Property 2 of Lemma B.16 and CRT), effectively recomposing the terms into $g^\Delta \mathsf{exp}(z)$.

- $\mathsf{Hyb}_{11} \approx_s \mathsf{Hyb}_{12}$. We switch the range of Δ back to $[0, \lfloor N/2 \rfloor)$. This is statistically indistinguishable for the same reasons as before: the range size difference is negligible, and $g^\Delta \mathsf{exp}(z)$ distribution is statistically close. This matches the distribution of $\mathcal{O}_1^{\mathsf{KDM},\Delta}$.

□

C Removing KDM Assumption for Truncation Gates

Recall that due to the use of a global key Δ, we require the underlying LHE scheme to be KDM secure when the circuit contains truncation gates. In this section, we outline the changes needed to remove this KDM assumption and achieve our result in Theorem 1.2.

Level of wire. We define the level of a wire w in the circuit $\mathcal{C}$ as follows:

- For an input wire, its level is 0.

- For a linear gate or multiplication gate with output wire z and input wires x and y, the level of the output wire z is $\max(\text{level}(x), \text{level}(y))$.

- For a truncation gate $z = \lfloor x/2 \rfloor$, the level of the output wire z is $\text{level}(x) + 1$.

Per-level keys. To avoid the circular dependencies of secret keys, the parties sample an independent key $k^{[l]} \leftarrow \mathsf{LHE.KeyGen}(1^\lambda)$ for the wires at each level l. To achieve malicious security, the parties also need to hold the MAC share $\Delta \cdot k^{(l)}$, where Δ is the global MAC key in the SPDZ framework and no longer serves as an encryption key. The shares can be generated in the preprocessing phase using the $\mathcal{F}^{N,\zeta}_{\text{SPDZprep}}$ functionality.

Shares of random masks. Correspondingly, the parties also need to hold shares of random masks multiplied by the level-wise keys, i.e., $r^{(w)} \cdot k^{(l_w)}$ for each wire w at level l_w. These shares can be generated using the standard SPDZ multiplication, given that the parties already hold SPDZ shares of $r^{(w)}$ and $k^{(l_w)}$.

Linear gates. When the two input wires x and y are not at the same level, we no longer support free linear combination by simply adding the labels. Without loss of generality, let's assume $\text{level}(x) < \text{level}(y)$. Then, the parties will use key-extension to convert the label of x into a new label under the key of $\text{level}(y)$, and then perform the linear combination to get the label of z. The parties can still set $r^{(z)} = \mu \cdot r^{(x)} + \nu \cdot r^{(y)}$ to maintain the invariant that $\widetilde{z} = z + r^{(z)}$.

Multiplication gates. Again, let's consider a multiplication gate with input wires x and y and output wire z, assuming $\text{level}(x) < \text{level}(y) = l$. Note that since the output wire is also at level l, the evaluator can still compute $L^{(z)} = \widetilde{x} \cdot L^{(y)} - \xi_x - \xi_y$, but the correction terms become

$$\xi_x = (k^{(l)} \cdot r^{(y)} + R^{(y)}) \cdot \widetilde{x} + U$$
$$\xi_y = (k^{(l)} \cdot r^{(x)}) \cdot \widetilde{y} - (U + k^{(l)} \cdot r^{(x)} \cdot r^{(y)} + k^{(l)} \cdot r^{(z)} + R^{(z)})$$

where U is a random mask.

Consequently, to prepare for the key-extension coefficients, the parties need to run the SPDZ multiplication protocol to compute shares of both $k^{(l)} \cdot r^{(x)}$ and $k^{(l)} \cdot r^{(x)} \cdot r^{(y)}$ during the offline phase. The coefficients for the masked values remain unchanged.

Truncation gates. The protocol for truncation gates remains mostly unchanged, except that (1) when we convert from the Boolean label to the arithmetic label $L^{(x_0)}$, the latter should be under the key of the output wire level, and (2) we need to extend the input wire label $L^{(x)}$ itself to use the key of the next level for the final subtraction. This is the same as the linear gate.

Input and output. During the offline phase, the parties can reveal the mask $r^{(w)}$ of each input or output wire w to the client in the same way as before. For the input phase, the parties will send to the evaluator their shares of the input labels under the key of level 0. For the output phase, the parties will send to the clients the decoding information and encoded output under the key of the maximum level.

Security proof. The security proof mostly follows the same structure as before. The main difference is that, instead of replacing all the LHE ciphertexts in one shot, we now replace the ciphertexts level by level. Namely, we first replace all the ciphertexts encrypted under the input-level key $k^{(0)}$, then $k^{(1)}$, and so forth. For each level of replacement, we can prove computational indistinguishability using Lemma 3.2 since (1) the level-wise key $k^{(l)}$ no longer appears in the message of any other level lower than l as we have replaced them, (2) $k^{(l)}$ cannot appear in the message of any other level higher than l due to our definition of the wire level, and (3) the messages of the ciphertexts at level l are either independent of $k^{(l)}$ or an affine function of $k^{(l)}$.

Summary. With the above changes, we can remove the KDM assumption. When the circuit $\mathcal{C}$ is a layered circuit, each linear gate $z = \mu \cdot x + \nu \cdot y$ still has no communication. Note that since x, y, z are at the same level l, no key-extension is needed. Moreover, the linear gate does not incur extra SPDZ multiplication or preprocessing, as $k^{(l)} \cdot r^{(z)}$ can be computed as $\mu \cdot (k^{(l)} \cdot r^{(x)}) + \nu \cdot (k^{(l)} \cdot r^{(y)})$, and the maximum level of the circuit only depends on truncation gates. For general circuits, the bandwidth cost of each linear gate becomes $O(n \cdot (\lambda + \log \mathcal{B}))$, same as the multiplication gate. The bandwidth cost of multiplication gates also increases by a constant factor due to the additional SPDZ multiplications.

D Detailed Cost Analysis

We analyze the concrete bandwidth cost of each phase for the main protocol, assuming the use of the global key Δ. Let m_{in}, m_{out}, m_{mul}, and m_{div} denote the number of input wires, output wires, multiplication gates, and truncation gates in $\mathcal{C}$, respectively. For simplicity, we use N and ζ in our analysis below, while their concrete values depend on the wire bound $\mathcal{B}$ and the security parameters, as defined in $\mathcal{F}_{\text{setup}}$ in Section A.2. We note that when $\mathcal{B}$ is sufficiently large, we have $(\zeta + 1) \log N \approx \zeta \log N \approx \log \mathcal{B}$. In the analysis below, we assume $|\mathcal{C}| \in \mathsf{poly} \log N$ and omit lower-order terms in $\mathcal{C} \log N$. For simplicity, we also assume the statistical security parameter $\kappa \ll \log N$ and omit it in the expressions.

Preprocessing Phase. The bandwidth cost of Π_{prep} comes from the following components:

- Based on prior works [GLM+24, RS22], the functionality $\mathcal{F}^{p}_{\text{SPDZprep}}$ can be implemented with a bandwidth cost of
$$\left(\left(12m^{(p)}_{\text{trip}} + 16m_{\text{bits}}\right)(n-1) + O\left(n^2 \log |\mathcal{C}|\right)\right) \log p$$
where $m^{(p)}_{\text{trip}}$ is the number of triples mod p, m_{bits} is the number of random bits, and the additive n^2 dependency is due to the pairwise Vector Oblivious Linear Evaluation (VOLE) calls. For our protocol, we have $m^{p}_{\text{trip}} = m_{\text{div}} \log N + \kappa\zeta \log N$ and $m_{\text{bits}} = 2m^{p}_{\text{trip}}$, and $\log p < \log N$.
- Similarly, the functionality $\mathcal{F}^{N^\zeta}_{\text{SPDZprep}}$ can be implemented with a bandwidth cost of
$$\left(12m^{(N^\zeta)}_{\text{trip}}(n-1) + O\left(n^2 \log |\mathcal{C}|\right)\right) \zeta \log N$$
where $m^{(N^\zeta)}_{\text{trip}}$ is the number of triples mod N^ζ. For our protocol, we have $m^{(N^\zeta)}_{\text{trip}} = m_{\text{mul}} + 3m_{\text{in}} + 2m_{\text{out}} + m_{\text{div}} \log N$.
- m_{bits} random bits are converted from mod p to integer or mod N^ζ shares, each costing $2(n-1) \log p$ bits of bandwidth, where $\log p < \log N$.
- $m_{\text{mul}} + m_{\text{in}} + 2m_{\text{div}}$ bounded random values are generated for our wire masks, each costing $2(n-1)\zeta \log N$ bits from the opening in Π_{auth}. There is also an additive cost of $2\kappa(n-1)\zeta \log N$ bits for opening the sum of the masks in the randomized range check.

Assuming the statistical security parameter $\kappa \in o(|\mathcal{C}|)$, then the total bandwidth cost of the preprocessing phase is approximately
$$(n-1)\left((14m_{\text{mul}} + 38m_{\text{in}} + 24m_{\text{out}})\zeta \log N + (12\zeta + 48)m_{\text{div}} \log^2 N\right) + O(n^2 \zeta \log |\mathcal{C}| \log N)$$

Offline Phase. The bandwidth cost of Π_{offline} comes from the following components:

- We exclude the cost of $\mathcal{F}_{\text{setup}}$, $\mathcal{F}_{\text{coin}}$, and $\mathcal{F}_{\text{commit}}$ since each is only invoked a constant number of times.
- Around $2\frac{\zeta+1}{\zeta}(4m_{\text{mul}}+m_{\text{out}}+m_{\text{div}}\log N)$ random values in $\mathbb{Z}_{N^\zeta}$ are opened during the sampling of the random generators, each costing around $2(n-1)\zeta\log N$ bits[11].
- $m_{\text{mul}}+m_{\text{in}}+m_{\text{out}}+m_{\text{div}}\log N$ SPDZ multiplications are performed to prepare for the key-extension coefficients, compute shares of Boolean labels, and authenticate the random masks of input and output wires to the clients. Each multiplication costs around $4(n-1)\zeta\log N$ bits from opening two values in Π_{mul}.
- $5(m_{\text{in}}+m_{\text{out}})$ values are opened to the clients, each costing $n\zeta\log N$ bits.

Thus, the total bandwidth cost of the offline phase is

$$(n-1)\left(((20\zeta+16)m_{\text{mul}}+9\zeta m_{\text{in}}+(13\zeta+4)m_{\text{out}})\log N+(8\zeta+4)m_{\text{div}}\log^2 N\right)$$

Online Phase. The bandwidth cost of Π_{main} comes from the following components:

- m_{in} masked values are sent from clients to servers, each costing at most $n\zeta\log N$ bits.
- m_{in} SPDZ multiplications are performed for range-checking the clients' input, each costing around $4(n-1)\zeta\log N$ bits.
- m_{in} input labels are reconstructed, each costing $(n-1)\zeta\log N$ bits.
- $4m_{\text{mul}}+2\cdot m_{\text{div}}\log N$ arithmetic key-extension operations are performed. Each requires sending 2 LHE ciphertexts and costs $2(n-1)(\zeta+1)\log N$ bits.
- $2m_{\text{div}}$ differences of random masks are sent from the servers to the evaluator for mask switching, each costing at most $n\zeta\log N$ bits.
- $2m_{\text{div}}$ garbled truth tables are needed for converting Boolean labels to arithmetic labels. Each requires sending 2 LHE ciphertexts and costs $2(n-1)(\zeta+1)\log N$ bits.
- $2m_{\text{out}}$ LHE ciphertexts are sent from each server to the clients in the decoding information, each costing $n(\zeta+1)\log N$ bits.
- $2m_{\text{out}}$ LHE ciphertexts and masked output values are sent from the evaluator to the clients in the encoded outputs, each costing $(\zeta+1)\log N$ bits.

Thus, the total bandwidth cost of the online phase is

$$\begin{aligned}&((8(n-1)m_{\text{mul}}+(2n+1)m_{\text{out}})(\zeta+1)+(6n-5)\zeta m_{\text{in}})\log N\\&\quad+4(n-1)(\zeta+1)m_{\text{div}}\log^2 N\end{aligned}$$

[11] We do not include the cost of sampling generators in Section 6, as our practical instantiation assumes they are sampled via a random oracle.

Total Cost. Finally, we sum up the costs from all three phases, and split the cost into the terms with different dependencies on m_{in}, m_{mul}, m_{out}, and m_{div}:

$$\begin{aligned}
\text{Cost per multiplication:} &\quad (n-1)(42\zeta+24)\log N \\
\text{Cost per input wire:} &\quad (53n-52)\zeta\log N \\
\text{Cost per output wire:} &\quad (39n\zeta-36\zeta+6n)\log N \\
\text{Cost per truncation:} &\quad (n-1)(24\zeta+56)\log^2 N \\
\text{Additive cost:} &\quad O(n^2\zeta\log|\mathcal{C}|\log N)
\end{aligned}$$

Hence, for $\mathcal{B} \in O(\log N)$ and $|\mathcal{C}| \geq n \cdot \log|\mathcal{C}|$, our MPC protocol for $\mathcal{B}$-bounded arithmetic circuits achieves a constant rate per party.

Comparison with [HY24]. Based on their cost analysis, Hazay and Yang [HY24]'s two-party protocol requires a total bandwidth of $(164\zeta+32)\log N$ bits per addition or multiplication gate. Hence, we achieve at least $3.9\times$ reduction in the concrete cost at $n=2$, assuming a sufficiently large bound $\mathcal{B}$.