High-Throughput Secure Multiparty Computation with an Honest Majority in Various Network Settings

Christopher Harth-Kitzerow
Technical University of Munich, BMW Group
christopher.harth-kitzerow@tum.de

Ajith Suresh
Technology Innovation Institute, Abu Dhabi
ajith.suresh@tii.ae

Yonqing Wang
University of Southern California, Meta AI
yongqin@usc.edu

Hossein Yalame
Bosch GmbH, Germany
hossein.yalame@de.bosch.com

Georg Carle
Technical University of Munich
carle@net.in.tum.de

Murali Annavaram
University of Southern California
annavara@usc.edu

ABSTRACT

In this work, we present novel protocols over rings for semi-honest secure three-party computation (3PC) and malicious four-party computation (4PC) with one corruption. While most existing works focus on improving total communication complexity, challenges such as network heterogeneity and computational complexity, which impact MPC performance in practice, remain underexplored.

Our protocols address these issues by tolerating multiple arbitrarily weak network links between parties without any substantial decrease in performance. Additionally, they significantly reduce computational complexity by requiring up to half the number of basic instructions per gate compared to related work. These improvements lead to up to twice the throughput of state-of-the-art protocols in homogeneous network settings and even larger performance improvements in heterogeneous settings. These advantages come at no additional cost: Our protocols maintain the best-known total communication complexity per multiplication, requiring 3 elements for 3PC and 5 elements for 4PC.

We implemented our protocols alongside several state-of-the-art protocols (Replicated 3PC, ASTRA, Fantastic Four, Tetrad) in a novel open-source C++ framework optimized for high throughput. Five out of six implemented 3PC and 4PC protocols achieve more than one billion 32-bit multiplications or over 32 billion AND gates per second using our implementation in a 25 Gbit/s LAN environment. This represents the highest throughput achieved in 3PC and 4PC so far, outperforming existing frameworks like MP-SPDZ, ABY3, MPyC, and MOTION by two to three orders of magnitude.

KEYWORDS

MPC Protocols, Honest Majority, 3PC, 4PC, Implementation

1 INTRODUCTION

Secure Multi-party Computation (MPC) enables parties to execute functions on obliviously shared inputs without revealing them [29]. Consider multiple hospitals that want to study the adverse effects of a certain medication based on their patients' data. While joining these datasets could enable more statistically significant results, hospitals might be prohibited from sharing their private patient data with each other. MPC enables these hospitals to perform this study and only reveal the final output of the function evaluated. MPC deployments are gaining prominence, such as private inventory matching to match buyers and sellers of stocks privately [42].

MPC protocols fall into two categories: high-throughput [2, 12, 13, 17] and low-latency [3, 44]. Low-latency protocols often uses garbled circuits [44], resulting in constant-round solutions. In contrast, high-throughput protocols are mainly based on secret-sharing (SS) solutions, requiring communication rounds proportional to the multiplicative depth of the circuit. However, SS-based protocols generally involve less communication than garbled circuits, allowing multiple instances of SS-based protocols to be executed in parallel, thereby achieving high throughput. For instance, Araki et al. [2] designed a high-throughput 3PC protocol capable of authenticating a login storm of 35,000 users, which requires computing a large number of parallel AES blocks. Also, several other MPC use cases such as privacy-preserving machine learning [36] with large batch sizes can benefit from implementations that achieve high throughput.

Most SS-based MPC protocols are designed to operate either over a field, or over an arbitrary ring $\mathbb{Z}_{2^\ell}$. Typical choices are the ring $\mathbb{Z}_2$ for boolean circuits and $\mathbb{Z}_{2^{64}}$ for arithmetic circuits. While boolean circuits can express comparison-based functions, arithmetic circuits can express arithmetic functions more compactly [23]. Computation over $\mathbb{Z}_{2^{64}}$ is supported by 64-bit hardware natively and thus leads to efficient implementations [40]. Multiple approaches also allow share conversion between computation domains to evaluate mixed circuits [13, 15, 34, 37].

MPC protocols can be secure against semi-honest and malicious adversaries [45]. A semi-honest adversary tries to break the privacy of the protocol. In contrast, a malicious adversary may try to break both the privacy and the correctness of the protocol by arbitrarily deviating from the protocol. Another important aspect of an MPC protocol is the maximum number of parties the adversary is allowed to corrupt. MPC protocols that can tolerate more than half of the participating parties are in the dishonest-majority class, while protocols that can tolerate less than half of the parties are in the honest-majority class. Typically, honest-majority protocols achieve significantly lower communication complexity than dishonest-majority ones.

Over the recent years, there has been a significant interest in fast and lightweight SS-based honest-majority MPC protocols for both the semi-honest [2, 12, 24] and malicious adversaries [10, 13, 17, 25, 27, 38]. In the semi-honest setting, the 3PC [2, 12, 24, 34] protocols that tolerate up to one corruption are particularly relevant due to their low bandwidth requirements. This setting allows the use of information-theoretic security techniques not applicable to two-party computation [16]. Likewise, for malicious corruption, the

4PC [10, 13, 17, 25] protocols are of particular interest as they allow exploiting the redundancy of secretly shared values to efficiently verify the correctness of exchanged messages, when compared with their 3PC counterparts. These properties of 3PC and 4PC protocols can also be utilized in the outsourced computation model [18], where three or four fixed computation nodes perform a computation for any number of input parties.

All recently proposed 3PC [2, 9, 12] and 4PC [10, 13, 17, 25] protocols share that they require at least three resp. six elements of global communication per multiplication gate. Only recently, a 4PC protocol achieved five elements of global communication per multiplication gate [27]. However, among other 4PC protocols [10, 13, 25], it is lacking an open-source implementation. Most existing honest-majority protocols work optimally in a homogeneous network setting [2, 5, 17, 24]. While a few protocols also work well in heterogeneous network settings [12, 27], they also do not provide an open-source implementation. In an orthogonal direction, a recent work proposed protocol to convert any honest-majority semi-honest protocol into a malicious one at no additional amortized communication costs [7]. However, the Distributed Zero-Knowledge Proofs (DZKP) required for this conversion come with significant computational overhead. According to a recent benchmark [17], their ring-based solution only achieves 22 multiplication gates per second. This is orders of magnitude lower than what state-of-the-art protocols achieve in practice.

Table 1: Operations and communication for multiplication

	Protocol	Operation		Communication		
		Add	Mult	Pre.[a]	On	Links [b]
3PC	Replicated [2]	12	6 (+3)[c]	0	3	0B,0L
	ASTRA [12]	11	5[d]	1	2	1B,2L
	Trio (**This work**)	11	5	1	2	1B,2L
4PC	Fantastic Four [17]	60	36	0	6	2-4B,2-4L
	Tetrad [27]	52	30	2	3	2B,5L
	Quad (**This work**)	25	12	2	3	3-4B,5L

[a] Pre. - Preprocessing
[b] B - #low-bandwidth links tolerated, L - #high-latency links tolerated. Ranges (e.g. 3-4) represent different protocol variations.
[c] Replicated 3PC additionally requires one division operation per party (+3) in the arithmetic domain.
[d] Reduced from 6 using a straightforward optimization.

The performance of MPC protocols is limited by both computational and communication complexity. Table 1 shows the number of local additions and multiplications required to compute a single multiplication gate securely using existing works in 3PC and 4PC settings. As shown in Table 1, even state-of-the-art 4PC protocols [13, 17, 27] require almost 100 local additions or multiplications for each multiplication gate on top of computing hashes and sampling shared random numbers. In §2, we quantitatively show that introducing this large computational overhead can become a serious performance hurdle for certain workloads.

Table 1 also provides the communication complexity and the number of network links that the protocol can tolerate in terms of bandwidth and latency (see Table 10 for per-party analysis). In terms of communication bottlenecks, Our quantitative analysis shows that MPC practitioners should expect significant variability in latency and bandwidth among network links, even in highly sophisticated cloud settings. Therefore, a versatile MPC protocol must address both computational complexity and network heterogeneity. Since existing related works does not address these issues, we bridge this gap by proposing an efficient protocol for semi-honest 3PC, and then a malicious 4PC protocol building on the 3PC.

Our Contributions

We approach the challenge of achieving high-throughput MPC in practical settings from two perspectives:

– **Protocol Design:** We develop new protocols that overcome existing bottlenecks in MPC workloads.

– **Framework Implementation:** We implement a C++ framework that accelerates any MPC protocol by efficiently utilizing hardware and networking resources.

This holistic, end-to-end approach allows us to maximize the potential of existing node setups and ensure that theoretical advancements in the communication and computational complexity of MPC protocols are reflected in practice.

Our novel protocols offer the following contributions:

(1) We present a semi-honest 3PC protocol, Trio (cf. §4), and a maliciously secure 4PC protocol, Quad (cf. §5). Trio requires total communication of three elements per multiplication gate, and Quad requires five, aligning with the state-of-the-art.

(2) By reducing the correlation between the shares of parties, we significantly reduce the computational complexity per gate compared to related work. Our 4PC protocol, Quad, achieves up to two times higher throughput for computationally intensive tasks like dot products.

(3) Our protocols exploit network heterogeneity by redistributing communication between parties, leveraging stronger network links and minimizing reliance on weaker ones without increasing communication complexity. Both our 3PC and 4PC protocols can tolerate all but two links having arbitrarily low bandwidth and all but one link having arbitrarily high latency, while still achieving fast runtimes.

Our novel framework offers the following contributions :

(1) Our C++ framework utilizes techniques such as Bitslicing, vectorization, message buffering, and load balancing. These implementation techniques are orthogonal to our primary protocol contributions and can accelerate existing protocols as well. By incorporating these optimizations, our framework achieves an unmatched throughput of more than 25 billion AND gates per second on a 25 Gbit/s network for each of the six currently implemented honest-majority protocols. This performance is two to three orders of magnitude higher than that of the open-source frameworks MP-SPDZ [22], ABY3 [34], MOTION [8], and MpyC [39] under the same setup.

(2) We have open-sourced our framework[1], which includes our novel protocols as well as several other state-of-the-art 3PC protocols [2, 12, 24], 4PC protocols [17, 27], and a trusted-third-party

[1] Code Repository: https://github.com/chart21/hpmpc/

Table 2: Throughput in Gates/s for Various Frameworks

Framework	Measurement	$\mathbb{Z}_{2^{32}}$ Multiplication		$\mathbb{Z}_2$ AND	
		3PC	4PC	3PC	4PC
MP-SPDZ [22]	Internal	$(1.94 \pm 0.47) \times 10^7$	$(7.07 \pm 0.22) \times 10^6$	$(4.74 \pm 0.13) \times 10^6$	$(1.84 \pm 0.03) \times 10^6$
	External	$(7.46 \pm 0.50) \times 10^6$	$(3.87 \pm 0.18) \times 10^6$	$(2.61 \pm 0.06) \times 10^6$	$(1.34 \pm 0.02) \times 10^6$
ABY3 [34]	External (Vanilla)	$(1.49 \pm 0.04) \times 10^4$	-	-	-
	External (Multithreading)	$(2.83 \pm 0.06) \times 10^5$	-	-	-
MOTION [8]	Internal	$(2.22 \pm 0.10) \times 10^3$	$(6.34 \pm 0.15) \times 10^4$	$(8.27 \pm 0.17) \times 10^4$	$(1.49 \pm 0.08) \times 10^3$
	External	$(1.15 \pm 0.19) \times 10^3$	$(4.06 \pm 0.07) \times 10^4$	$(5.28 \pm 0.17) \times 10^4$	$(8.22 \pm 0.03) \times 10^2$
MPyC [39]	Internal	$(4.41 \pm 0.50) \times 10^5$	$(4.21 \pm 0.46) \times 10^5$	$(5.82 \pm 0.01) \times 10^2$	$(5.81 \pm 0.01) \times 10^2$
	External	$(4.05 \pm 0.32) \times 10^5$	$(3.83 \pm 0.27) \times 10^5$	$(5.69 \pm 0.01) \times 10^2$	$(5.68 \pm 0.01) \times 10^2$

protocol. Some of these protocols have not been previously implemented in any open-source framework [12, 27][2]. For other protocols [2, 17], we achieve up to three orders of magnitudes higher throughput compared to their current open-source implementations in MP-SPDZ [22].

2 BOTTLENECKS OF MPC IN PRACTICE

In this work, we identify and tackle three challenges that limit the performance of MPC protocols in practice and are not related to the total communication complexity between parties. This section introduces these challenges.

Network Heterogeneity is Ubiquitous. In real-world settings, network links between distributed parties are highly heterogeneous. To quantify this heterogeneity, we set up multiple AWS C6in instances across different network settings: Same Data center (LAN), Same City (MAN), Same Continent (WAN1), Different Continents (WAN2), and Mixed Constellations (Mixed). We measured both the bandwidth and latency for each link between the parties and identified the strongest and weakest links within each node setup in terms of latency and bandwidth. Table 3 presents the measured values. We observed significant variance in both the best and worst-case measurements within the same setting, with even more dramatic differences across different settings.

Table 3: Link Heterogeneity in Different Network Settings

Setting	Latency (ms)			Bandwidth (Mbit/s)		
	Best	Worst	% Diff	Best	Worst	% Diff
LAN	0.178	0.304	70.8%	4790	4940	3.1%
MAN	0.383	0.953	148.8%	2230	2540	13.9%
WAN1	34.772	192.515	453.6%	868	142	511.3%
WAN2	91.451	276.253	202.1%	341	108	215.7%
Mixed	34.799	276.256	693.9%	108	824	663.0%

These exemplary real-world network settings would significantly benefit from a protocol that assigns all latency-critical communications to the link with the best latency between the parties, while directing bulk of the messages through links with higher available bandwidth. Most existing protocols achieve the same or similar communication complexity for each party [1, 2, 17, 19, 24], focusing on reducing the total communication complexity—for instance, from two elements per party (six in total) [24] to one element per party (three in total) [2] in the 3PC setting. However, in heterogeneous network environments like those described, designing an MPC protocol with varying round and communication complexities for each party could yield even greater performance improvements. This way, the protocol is less affected by the weaker network links in a given setting.

Computational Complexity matters. As MPC deployments span a wider range of application domains, such as privacy-preserving machine learning [36], some MPC operations are becoming computation-bound rather than communication-bound. To demonstrate this, we implemented two of the most popular 3PC [2] and 4PC protocols [17], comparing the runtime of regular multiplications with scalar products of the same output size across varying bandwidths. Figure 1 shows that while multiplications are often bottlenecked by communication in most settings, the runtime of scalar multiplications shows no significant benefit from bandwidths above 250 Mbit/s. Thus, these applications are not bounded by available bandwidth but rather are bottlenecked by local computations.

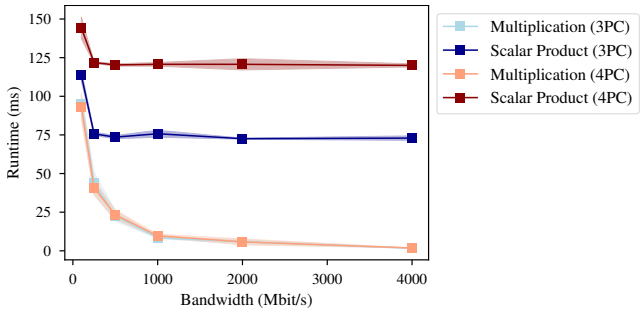


Figure 1: Comparing Runtimes for Multiplications and Scalar Product with Varying Bandwidths

Limitations of Existing Implementations. Araki et al. [2] demonstrated that implementing a semi-honest 3PC protocol in a homogeneous network setting can achieve a throughput of seven billion AND gates per second. However, the implementation of [2] has not been published, and open-source implementations do not come close to that level of throughput. For instance, on our test setup, the popular open-source library MP-SPDZ [22] achieves a throughput of less than ten million independent AND gates per second using the same 3PC protocol. To investigate whether this limitation

[2]In the meantime, ASTRA has also been implemented by [9].

also applies to other implementations, we measure the throughput of $\mathbb{Z}_{2^{32}}$ and AND gates per second for several state-of-the-art frameworks. Table 2 shows the results of our comparison.

We measured the throughput of replicated secret sharing protocols in the honest-majority setting for MP-SPDZ [22] and ABY3 [34]. MOTION [8] and MPyC [39] provide only n-party computation protocols secure against a dishonest majority, which naturally achieve lower throughput. Where available, we used each framework's concurrency features. For instance, we utilized MP-SPDZ's `@multithread` instruction and vectorized array multiplication operators. Although ABY3 does not natively support multithreading, we implemented multithreading on top of their framework for our measurements. Table 2 presents both "external" measurements obtained manually by timing the start and end of an executable, and "internal" measurements provided by most frameworks out of the box. We found that all tested existing frameworks achieve only several thousand to a few million gates per second on a 25 Gbit/s network, significantly below the theoretical capabilities expected given the available network bandwidth. Notably, the frameworks do not achieve 32 times higher throughput for AND gates compared to $\mathbb{Z}_{2^{32}}$ multiplication gates, despite the latter require 32 times more data to be sent. This indicates that existing frameworks struggle to accelerate boolean computations.

Given the analysis above, we make a case for designing efficient MPC protocols that address identified bottlenecks related to network heterogeneity and computational complexity. To demonstrate our protocols' theoretical contributions but moreover to ensure that the progress in MPC protocol design over the last years is reflected in practice, we also make a case for novel implementations that find ways to accelerate the basic building blocks required by nearly every MPC protocol.

3 PRELIMINARIES

In this section, we provide the preliminaries, including notations and sharing semantics. Our protocols are designed to operate over ℓ-bit rings of the form $\mathbb{Z}_{2^\ell}$ and also support the Boolean ring, denoted by $\mathbb{Z}_2$. Additionally, the protocols use function-dependent preprocessing [4, 21, 26] for efficiency, similar to works like ASTRA [12] and Tetrad [27].

3.1 Notations

We use $\mathcal{P}$ to denote the set of parties and P_i to denote the ith party. While $\mathcal{P} = \{P_0, P_1, P_2\}$ denote the parties in our 3PC setting, $\mathcal{P} = \{P_0, P_1, P_2, P_3\}$ denote the parties in our 4PC protocols. Moreover, $\mathcal{P}_\Phi$ denotes a subset of $\mathcal{P}$ comprising of parties in the set Φ. For instance, $\mathcal{P}_\Phi$ or simply $\mathcal{P}_{i,j}$ indicates the set of parties $\Phi = \{P_i, P_j\}$. Similarly, we use x_Φ or simply $x_{i,j}$ to denote the value x possessed by all parties in $\Phi = \{P_i, P_j\}$. We use κ to denote the computational security parameter which is set to 128 in our protocols.

Sharing Semantics. We use the *masked evaluation* technique [4, 30, 43] in our protocols, where each secret $x \in \mathbb{Z}_{2^\ell}$ is associated with a mask λ_x and the masked value m_x such that $\mathsf{m}_x = x + \lambda_x$. The mask λ_x is *input-independent* and thus all the operations involving only λ_x can be carried out in the preprocessing phase, while m_x is the *input-dependent* share. Additionally, we use $\overline{\mathsf{m}}_x = x + \lambda_x + \lambda_x^*$ to denote a doubly masked value which represents the secret x masked using two independent masks λ_x and λ_x^* (cf. §5 for details).

Secret-Sharing Schemes. We use two different sharing schemes throughout our protocols, and their overview is provided below.

(1) $[\cdot]$-sharing: A value $x \in \mathbb{Z}_{2^\ell}$ is $[\cdot]$-shared among $\mathcal{P}_\Phi$, if each $P_i \in \mathcal{P}_\Phi$ holds x^i such that $\sum_i x^i = x$.
(2) $[\![\cdot]\!]$-sharing: A value $x \in \mathbb{Z}_{2^\ell}$ is $[\![\cdot]\!]$-shared among $\mathcal{P}_\Phi$, if parties in $\mathcal{P}_\Phi$ holds m_x and $[\lambda_x]$ such that $\mathsf{m}_x = x + \lambda_x$.

The exact definitions of the above schemes differ slightly between the 3PC and 4PC protocols, as discussed in their respective sections.

Additionally, $[\![\cdot]\!]^B$ represents Boolean sharing, where addition and multiplication are replaced by XOR and AND gates, respectively. Similarly, $[\![\cdot]\!]^A$ denotes arithmetic sharing. The superscript is omitted when the type of sharing is clear from the context.

Messages and Verification. A value computed by $\mathcal{P}_\Phi$ is denoted by V_Φ and for multiple values, the jth value is denoted by $\mathsf{V}_{\Phi,j}$. Similarly, a message communicated by $\mathcal{P}_\Phi$ is denoted by M_Φ and for multiple messages, the jth message is denoted by $\mathsf{M}_{\Phi,j}$. The notation P_i^V indicates that any trailing computations or communications performed by P_i is used only for verifying messages from other parties and do not add any delays in the main protocol execution. In other words, this notation indicates that these specific computations and communications are not latency-critical and do not contribute to the round complexity of the protocol.

3.2 Generating Shared Random Values

To improve communication efficiency, our protocols utilize the $\mathcal{F}_{\mathsf{SRNG}}$ functionality (Figure 23), which allows a subset of parties $\mathcal{P}_\Phi$ to generate fresh random values without interaction. Protocol Π_{SRNG} (Figure 2) shows the instantiation of $\mathcal{F}_{\mathsf{SRNG}}$ in our protocols. We refer to this as sampling using a shared random value generator (SRNG), where random values are generated with the help of a pseudorandom function (PRF). Additionally, the protocol assumes a shared-key setup ($\mathcal{F}_{\mathsf{setup}}$) is established at the beginning, which is the case with most existing protocols [2, 12, 13, 17].

Protocol $\Pi_{\mathsf{SRNG}}(\mathcal{P}_\Phi, \mathsf{k}_\Phi, \mathsf{c}_\Phi, \ell') \rightarrow \mathsf{r}_\Phi \in \mathbb{Z}_{2^{\ell'}}$

One-time Setup: Parties in $\mathcal{P}_\Phi$ invoke $\mathcal{F}_{\mathsf{setup}}$ to obtain a unique shared key $\mathsf{k}_\Phi \in \mathbb{Z}_{2^\kappa}$. Parties initialize a counter $\mathsf{c}_\Phi \in \mathbb{N}$ to 0 and a buffer stack $\mathsf{B}_\Phi \leftarrow \varnothing$ with B^t denoting the index of top element.

Procedure: Let $PRF : \mathbb{Z}_{2^\kappa} \times \mathbb{N} \rightarrow \mathbb{Z}_{2^\kappa}$ be a pseudorandom function. To generate a new random value in $\mathbb{Z}_{2^{\ell'}}$, each $P \in \mathcal{P}_\Phi$ does the following:

(1) If $|\mathsf{B}_\Phi| < \ell'$, compute $r_\kappa = PRF(\mathsf{k}_\Phi, \mathsf{c}_\Phi)$ to obtain κ random bits and add them to the buffer B_Φ. Set $\mathsf{c}_\Phi \leftarrow \mathsf{c}_\Phi + 1$.
(2) Pop ℓ' bits $b_{\mathsf{B}^t - \ell', \ldots, \mathsf{B}^t}$ from B_Φ and compute $\mathsf{r}_\Phi = \sum_{i=0}^{\ell'} b_i \cdot 2^i$.

Figure 2: Generating shared random values

3.3 Compare-View functionality

To achieve malicious security in our 4PC protocols, each party needs to verify the correctness of the messages it receives. To enable this, the parties have access to a Compare-View functionality, similar to the joint-message passing in SWIFT [25] and the jsnd primitive in Tetrad [27]. Protocol Π_{CV} in Figure 3 instantiates this functionality.

Let $\{v_1, \ldots, v_n\}$ denote a set of values obtained by the parties in $\mathcal{P}_\Phi$ through the messages received during the protocol execution and/or local computation. Protocol Π_{CV} ensures that all parties in $\mathcal{P}_\Phi$ will either have the same set of values or will return `abort` if there is an inconsistency. To achieve this, they compare a single hash of their concatenated views of $\{v_1, \ldots, v_n\}$.

Protocol $\Pi_{\mathsf{CV}}(\mathcal{P}_\Phi, \{v_1, \ldots, v_n\}) \rightarrow$ bool

Assumptions: Each party in $\mathcal{P}_\Phi$ obtains a set of values $\{v_1, \ldots, v_n\}$ during the protocol execution. Parties in $\mathcal{P}_\Phi$ have access to a collision-resistant hash function H and $||$ denotes concatenation.

Procedure:

(1) Parties in $\mathcal{P}_\Phi$ compute and exchange $h_V = \mathsf{H}(v_1||v_2||\ldots||v_n)$.
(2) If the received hashes are consistent with the computed hash, they return true and `continue` with the protocol. Otherwise, they return false and `abort` the protocol.

Figure 3: Verifying the correctness of received values

4 Trio: 3PC PROTOCOL

This section details our 3PC protocol over rings, namely Trio, which is secure against up to one semi-honest corruption among the computing parties $\mathcal{P} = \{P_0, P_1, P_2\}$. Similar to ASTRA [12], our protocol requires the communication of one ring element in the preprocessing phase and two elements in the online phase per multiplication. If preferred, the preprocessing phase can also be executed during the online phase.

4.1 Sharing Semantics

We detail the sharing semantics of our protocol based on masked evaluation and comparing it to ASTRA in Table 4. Similar to ASTRA, in our scheme for a secret x, each online party P_1 and P_2 will possess one share of the mask λ_x. However, the parties' input-dependent shares are masked differently: In ASTRA, both parties hold the same input-dependent share m_x corresponding to x being masked by λ_x. In our scheme, P_1 holds $\mathsf{m}_{x,2}$ and P_2 holds $\mathsf{m}_{x,1}$, which correspond to x being masked with one of the shares of λ_x. Although the amount of information possessed by the parties in our protocol is the same as in ASTRA, this sharing mainly improves online computation, as discussed later in this section. Additionally, we introduce the notion of *persistent shares*, which represent the actual values each party should store in memory during the protocol execution.

Table 4: Trio: 3PC secret sharing.

	Party	ASTRA [12]	Trio (3PC)
Sharing Semantics $[\![x]\!]$	P_0	λ_x^1, λ_x^2	λ_x^1, λ_x^2
	P_1	$\mathsf{m}_x, \lambda_x^1$	$\mathsf{m}_{x,2}, \lambda_x^1$
	P_2	$\mathsf{m}_x, \lambda_x^2$	$\mathsf{m}_{x,1}, \lambda_x^2$
Persistent Shares	P_0	λ_x	λ_x^1, λ_x^2
	P_1	$\mathsf{m}_x, \lambda_x^1$	$\mathsf{m}_{x,2}, \lambda_x^1$
	P_2	$\mathsf{m}_x, \lambda_x^2$	$\mathsf{m}_{x,1}, \lambda_x^2$
Correlation		$\lambda_x = \lambda_x^1 + \lambda_x^2$ $\mathsf{m}_x = x + \lambda_x$	$\mathsf{m}_{x,1} = x + \lambda_x^1$ $\mathsf{m}_{x,2} = x + \lambda_x^2$

Note that our sharing scheme is *linear*. Thus, given public constants α, β, γ and secret-shares $[\![x]\!], [\![y]\!]$, parties can locally compute the shares of $[\![\alpha x + \beta y + \gamma]\!]$.

4.2 Input Sharing and Output Reconstruction

To allow party P_I to secret share a value $x \in \mathbb{Z}_{2^\ell}$, we modify the shared key setup functionality ($\mathcal{F}_{\mathsf{setup}}$) so that P_I receives PRF keys for both masks λ_x^1, λ_x^2. This enables P_I to compute all the shares and send them to the respective parties.

To reconstruct a secret shared value $[\![x]\!]$ towards P_O, P_2 sends $\mathsf{m}_{x,1}$ and P_0 sends λ_x^1 to P_O. Similarly, for a public reconstruction among $\mathcal{P}$, P_0 sends λ_x^1 to P_2, while P_2 sends λ_x^2 to P_1 and $\mathsf{m}_{x,1}$ to P_0. Each party then holds two of the three shares and can compute $x = \mathsf{m}_{x,1} - \lambda_x^1 = \mathsf{m}_{x,2} - \lambda_x^2$.

4.3 Multiplication

To multiply two secret-shared values $[\![a]\!]$ and $[\![b]\!]$, the 3PC protocol in ASTRA involves the online parties locally computing an additive sharing of the masked output m_c. Then, they exchange these shares to reconstruct the final result. However, we approach our protocol from a different perspective and aim to eliminate the errors in each party's local computation by using messages from other parties.

The sharing semantics of our 3PC protocols are designed so that P_1 and P_2 can communicate to obtain a masked version of the output $c = ab$ with an *input-independent* error. In the preprocessing phase, P_0 prepares a message for P_2 to correct this input-independent error. This ensures that all parties obtain valid and masked shares of c according to our sharing semantics. Protocol Π_{Mult} (Figure 4) outlines the formal steps in our multiplication protocol, detailed below.

Protocol $\Pi_{\mathsf{Mult}}([\![a]\!], [\![b]\!]) \rightarrow [\![c]\!]$

Preprocessing:

(1) Sample random values using Π_{SRNG}:

$$\mathcal{P}_{0,1} : \lambda_c^1, r_{0,1} \qquad \mathcal{P}_{0,2} : \lambda_c^2$$

(2) Locally compute:

$$P_0 : \mathsf{M}_{\{0\}} = \lambda_a^2\lambda_b^2 - (\lambda_a^1 - \lambda_a^2)(\lambda_b^1 - \lambda_b^2) + r_{0,1}$$

(3) Communicate:

$$P_0 \rightarrow P_2 : \mathsf{M}_{\{0\}}$$

Online:

(1) Locally compute:

$$P_1 : \mathsf{V}_{\{1\}} = \mathsf{m}_{a,2}\lambda_b^1 + \mathsf{m}_{b,2}\lambda_a^1 + r_{0,1} \qquad P_2 : \mathsf{V}_{\{2\}} = \mathsf{m}_{a,1}\mathsf{m}_{b,1} + \mathsf{M}_{\{0\}}$$

(2) Communicate:

$$P_1 \rightarrow P_2 : \mathsf{M}_{\{1\}} = \mathsf{V}_{\{1\}} - \lambda_c^1 \qquad P_2 \rightarrow P_1 : \mathsf{M}_{\{2\}} = \mathsf{V}_{\{2\}} + \lambda_c^2$$

(3) Locally compute:

$$P_1 : \mathsf{m}_{c,2} = \mathsf{M}_{\{2\}} - \mathsf{V}_{\{1\}} \qquad P_2 : \mathsf{m}_{c,1} = \mathsf{V}_{\{2\}} - \mathsf{M}_{\{1\}}$$

Figure 4: Trio: 3PC multiplication protocol

Given $[\![a]\!], [\![b]\!]$, note that P_2 can compute $\mathsf{m}_{a,1} \cdot \mathsf{m}_{b,1}$, which is equivalent to computing

$$\mathsf{m}_{a,1} \cdot \mathsf{m}_{b,1} = (a + \lambda_a^1)(b + \lambda_b^1) = ab + a\lambda_b^1 + b\lambda_a^1 + \lambda_a^1\lambda_b^1, \quad (1)$$

thus obtaining the output $c = ab$ with an input-dependent error $a\lambda_b^1 + b\lambda_a^1$ and an input-independent error $\lambda_a^1\lambda_b^1$. Our goal is to correct these errors using messages from P_0 and P_1 while obliviously inserting the mask λ_c^1, such that P_2 obtains $\mathsf{m}_{c,1} = ab + \lambda_c^1$. In a similar fashion, P_1 should obtain its input-dependent share $\mathsf{m}_{c,2}$.

Preprocessing Phase. Using their pre-shared keys, the parties locally sample masks for the output $(\lambda_c^1, \lambda_c^2)$ and a random pad $r_{0,1}$ to mask the intermediary message. P_0 sends the message $\mathsf{M}_{\{0\}} = \lambda_a^1\lambda_b^2 + \lambda_a^2\lambda_b^1 - \lambda_a^1\lambda_b^1 + r_{0,1}$ to P_2. This message serves the purpose of correcting the input-independent error when P_1 and P_2 communicate during the online phase. Note that the mask $r_{0,1}$ ensures that P_2 cannot infer any values from P_0's message.

Online Phase. P_2 uses $\mathsf{M}_{\{1\}} = \mathsf{m}_{a,2}\lambda_b^1 + \mathsf{m}_{b,2}\lambda_a^1 + r_{0,1} - \lambda_c^1$ from P_1 to eliminate the input-dependent error $a\lambda_b^1 + b\lambda_a^1$ from its computation. However, this results in a larger input-independent error compared to $\lambda_a^1\lambda_b^1$. Note that P_0 holds all input-independent shares and can therefore help $\mathcal{P}_{1,2}$ correcting any input-independent error. Hence, this error is removed using message $\mathsf{M}_{\{0\}}$ obtained from P_0 during the preprocessing phase. P_2 obtains:

$$\begin{aligned}\mathsf{m}_{c,1} &= \mathsf{V}_{\{2\}} - \mathsf{M}_{\{1\}} \\ &= (ab + a\lambda_b^1 + b\lambda_a^1 + \lambda_a^1\lambda_b^1 + \mathsf{M}_{\{0\}}) - (\mathsf{V}_{\{1\}} - \lambda_c^1) \\ &= (ab + \lambda_c^1) + \mathsf{M}_{\{0\}} - (\lambda_a^1\lambda_b^2 + \lambda_a^2\lambda_b^1 - \lambda_a^1\lambda_b^1 + r_{0,1}) = ab + \lambda_c^1\end{aligned}$$

The steps for P_1 are similar, except that it locally computes $\mathsf{m}_{a,2}\lambda_b^1 + \mathsf{m}_{b,2}\lambda_a^1 + r_{0,1} = a\lambda_b^1 + b\lambda_a^1 + \lambda_a^1\lambda_b^2 + \lambda_a^2\lambda_b^1 + r_{0,1}$ and uses the message from P_2 to obtain the correct share $\mathsf{m}_{c,2} = ab + \lambda_c^2$. The correctness of our protocol is detailed in §A.1.

4.4 Discussions

Our 3PC multiplication protocol Π_{Mult} (cf. Figure 4) in Trio has the same computational complexity as that of ASTRA [12]. However, as shown in Table 10 in §D, we first eliminate one multiplication in ASTRA with a straightforward optimization. Our approach lets us then shift some local computation of ASTRA from online to preprocessing, leading to computational gains in the online phase.

The security of our semi-honest 3PC, Trio, can be proven using a real-world/ideal-world simulation paradigm [20, 28], using a simplified variant of the simulation strategy discussed in §C. Due to its similarity to ASTRA, we omit a formal security proof for it.

Regarding bandwidth and latency requirements for the three network links among $\mathcal{P}_{0,1,2}$, our 3PC protocol tolerates high latency between $\mathcal{P}_{0,2}$ and $\mathcal{P}_{0,1}$ as they do not communicate in the online phase. $\mathcal{P}_{1,2}$ on the other hand need to synchronously evaluate the circuit while waiting for each other's messages after each communication round. Our protocol also tolerates low bandwidth between $\mathcal{P}_{0,1}$ as they do not communicate in both phases. (cf. Figure 7a and Figure 7b for details).

Malicious security. Instead of improving the security of Trio to tolerate malicious corruption, we chose a 4PC setting for malicious security due to the following reasons. Maliciously secure 3PC protocols over rings mainly uses one of two approaches: Distributed Zero-Knowledge Proofs (DZKP) [6] and triple sacrificing [9]. The DZKP approach achieves sublinear communication complexity at the expense of increased computational complexity, whereas the triple sacrificing approach maintains low computational complexity but necessitates a higher communication overhead. For instance, 3PC maliciously secure protocols in Replicated [1, 19], ASTRA [12] and SWIFT [25, 38] require total communications of 21, 25 and 12 ring elements per multiplication, respectively, when utilizing the triple sacrificing technique [9]. Using DZKP [6] reduces the communication requirement to 6 ring elements, but with significantly higher local computational demands.

On the other hand, maliciously secure 4PC schemes such as Fantastic Four [17] and Tetrad [27] require only 6 and 5 communication elements per multiplication, respectively, and eliminate the need for these expensive primitives. Therefore, we use our 3PC protocol Trio as the foundation and have designed a maliciously secure 4PC protocol, Quad, based on it. The details about Quad are presented next.

5 Quad: 4PC PROTOCOL

In this section, we detail our 4PC protocol over rings, namely Quad, which is secure against up to one malicious corruption among the computing parties $\mathcal{P} = \{P_0, P_1, P_2, P_3\}$. Our protocol requires communicating two ring elements in the preprocessing phase and three elements in the online phase per multiplication, similar to Tetrad [27]. The protocol builds upon the 3PC protocol Trio described in §4, incorporating the necessary redundancy to verify all messages sent between the parties. However, the core difference between our protocol and Tetrad lies in our use of two independent masks to hide the secret, which are distributed carefully across the parties, as discussed in §5.1. Regarding latency requirements of the six network links among $\mathcal{P}$, only the link between $\mathcal{P}_{1,2}$ contains latency-critical communication. During the online phase, $\mathcal{P}_{1,2}$ need to wait for each other's messages before they can synchronously proceed with the next gate in the circuit. All other links are either only utilized in the preprocessing phase, or only utilized for verification. If a party requires messages only for verification, it can delay processing of these messages to the end of the protocol without affecting the progress of the other parties as they evaluate the circuit. Regarding bandwidth requirements, our protocol tolerates three low bandwidth links between the parties as these are not utilized in both phases. Additionally, we present a variant of our protocol optimized for heterogeneous network settings. This variant increases the number of unutilized links to four and therefore performs well in settings even when the majority of parties share weak network links. The details are provided in §5.4.

5.1 Sharing Semantics

Table 5 details the sharing semantics of our 4PC protocol and compares it with Tetrad. The sharing semantics of our 4PC protocol extend those of the 3PC protocol by introducing redundancy to verify messages exchanged among the parties. At a high level, P_0 holds an input-dependent share to help verify the communication between $\mathcal{P}_{1,2}$ using messages from P_3. Additionally, P_3 helps $\mathcal{P}_{1,2}$ verify messages sent by P_0.

Tetrad uses a single mask λ_x to hide the secret x and distributes the additive shares of λ_x among the parties to create redundancy. Specifically, all the input-independent shares $\lambda_x^0, \lambda_x^1, \lambda_x^2$ in Tetrad are correlated to mask the identical input-dependent share m_x held by $\mathcal{P}_{0,1,2}$. In contrast, our protocol uses two independent masks, λ_x and λ_x^*, and correspondingly two masked values, m_x and m_x^*. This enables each party to obtain different input-dependent shares, m_x and m_x^*, similar to our 3PC scheme. This approach reduces the correlation between input-independent shares, resulting in each party needing to store fewer shares in memory (cf. persistent shares

Table 5: Quad: 4PC secret sharing.

	Party	Tetrad [27]	Quad
Sharing Semantics $[\![x]\!]$	P_0	$\mathsf{m}_x, \lambda_x^1, \lambda_x^2$	$\mathsf{m}_x^*, \lambda_x^1, \lambda_x^2$
	P_1	$\mathsf{m}_x, \lambda_x^0, \lambda_x^1$	$\mathsf{m}_x, \lambda_x^*, \lambda_x^1$
	P_2	$\mathsf{m}_x, \lambda_x^0, \lambda_x^2$	$\mathsf{m}_x, \lambda_x^*, \lambda_x^2$
	P_3	$\lambda_x^0, \lambda_x^1, \lambda_x^2$	$\lambda_x^*, \lambda_x^1, \lambda_x^2$
Persistent Shares	P_0	$\mathsf{m}_x, \lambda_x^1, \lambda_x^2$	$\mathsf{m}_x^*, \lambda_x$
	P_1	$\mathsf{m}_x, \lambda_x^0, \lambda_x^1$	$\mathsf{m}_x, \lambda_x^1$
	P_2	$\mathsf{m}_x, \lambda_x^0, \lambda_x^2$	$\mathsf{m}_x, \lambda_x^2$
	P_3	$\lambda_x^0, \lambda_x^1, \lambda_x^2$	λ_x^*, λ_x
Correlation		$\lambda_x = \lambda_x^0 + \lambda_x^1 + \lambda_x^2$ $\mathsf{m}_x = x + \lambda_x$	$\lambda_x = \lambda_x^1 + \lambda_x^2$ $\mathsf{m}_x = x + \lambda_x$ $\mathsf{m}_x^* = x + \lambda_x^*$

$*$ indicates share not correlated to $[\![\lambda_x]\!]$ and used only for verification.

in Table 5). Additionally, it allows us to reduce the number of basic instructions per multiplication gate by more than half compared to Tetrad, as shown in Table 1. Note that no single party's share reveals anything about x, but holding two distinct shares suffices to obtain x.

5.2 Input Sharing and Output Reconstruction

To securely share a value $x \in \mathbb{Z}_{2^\ell}$ in the presence of a malicious adversary, the input party P_I obtains the PRF keys corresponding to the masks λ_x^1, λ_x^2 and λ_x^*, similar to the 3PC scheme. P_I then computes and sends $\overline{\mathsf{m}}_x = x + \lambda_x^1 + \lambda_x^2 + \lambda_x^*$ to $\mathcal{P}_{0,1,2}$. The parties compare their views of $\overline{\mathsf{m}}_x$ using the compare-view functionality (cf. Π_{CV} in Figure 3) and locally convert it to their respective input-dependent share by subtracting the corresponding masks they generated together with P_I.

To reconstruct a secret $[\![x]\!]$ towards output party P_O, P_0 sends λ_x and P_2 sends m_x to P_O. P_O then uses Π_{CV} to compare their views of λ_x and m_x with P_3 and P_1, respectively. Since one party in each pair of $\{P_1, P_2\}$ and $\{P_0, P_3\}$ is honest, P_O will either obtain the correct x or `abort` the protocol.

The formal protocols for input sharing (Π_{Sh}) and output reconstruction with abort (Π_{Rec}) are provided in §B, along with a *fair* [14] reconstruction protocol (Π_{fairRec}), which guarantees that if the adversary receives an output, the honest parties do as well.

5.3 Multiplication

Protocol Π_{Mult} (Figure 5) outlines the formal steps in our multiplication protocol, detailed below. To multiply two secret-shared values, $[\![a]\!]$ and $[\![b]\!]$, the parties proceed similarly to 3PC multiplication (cf. §4.3). The goal is to eliminate errors in each party's local computation by using messages from other parties. These messages are carefully designed to ensure that if P_j computes a value v using a message sent by P_i, another party P_k also obtains v, either through local computation or with the help of a set of verified messages. This allows the use of the compare-view functionality (cf. Π_{CV} in Figure 3) to achieve malicious security with abort.

Preprocessing Phase. During this phase, parties non-interactively compute all the input-independent shares of c, specifically λ_c^1, λ_c^2, and λ_c^*, using Π_{SRNG} (cf. Figure 2). As in our 3PC, P_0 then sends a message $\mathsf{M}_{\{0\}}$ to P_2, which aims to eliminate the input-independent error during the communication between P_1 and P_2 in the online phase. This message also ensures the correct mask λ_c is inserted into their input-dependent shares. Additionally, P_0 needs to obtain an input-dependent share $\mathsf{m}_c^* = ab + \lambda_c^*$, which it uses to verify $\mathcal{P}_{1,2}$'s communication in the online phase. For this purpose, P_3 computes and sends a message $\mathsf{M}_{\{3\}}$ to eliminate the input-independent error of P_0's computation during the online phase. This way, P_0 receives an input-dependent share of c without interacting with $\mathcal{P}_{1,2}$, which is utilized for verification.

Protocol $\Pi_{\mathsf{Mult}}([\![a]\!], [\![b]\!]) \to [\![c]\!]$

Preprocessing:

(1) Sample random values using Π_{SRNG}:

$\mathcal{P}_{0,1,3} : r_{0,1,3}, \lambda_c^1 \qquad \mathcal{P}_{0,2,3} : \lambda_c^2 \qquad \mathcal{P}_{1,2,3}^V : r_{1,2,3}, \lambda_c^*$

(2) Locally compute:

$P_0, P_3^V : \lambda_c = \lambda_c^1 + \lambda_c^2$

$P_0, P_3^V : \mathsf{M}_{\{0,3\}} = \lambda_c + \lambda_a\lambda_b + r_{0,1,3}$

$P_3^V : \mathsf{M}_{\{3\}} = \lambda_a(\lambda_b - \lambda_b^*) - \lambda_b\lambda_a^* - \lambda_c^* + r_{1,2,3}$

(3) Communicate:

$P_0 \to P_2 : \mathsf{M}_{\{0,3\}} \qquad P_3^V \to P_0^V : \mathsf{M}_{\{3\}}$

Online:

(1) Locally compute:

$P_0^V : \mathsf{V}_{\{0\}} = \mathsf{m}_a^*\lambda_b + \mathsf{m}_b^*\lambda_a \qquad \mathcal{P}_{1,2} : \mathsf{V}_{\{1,2\}} = \mathsf{m}_a\mathsf{m}_b$

$P_1 : \mathsf{M}_{\{1\}} = \mathsf{m}_a\lambda_b^1 + \mathsf{m}_b\lambda_a^1 + r_{0,1,3} \qquad P_2 : \mathsf{M}_{\{2\}} = \mathsf{m}_a\lambda_b^2 + \mathsf{m}_b\lambda_a^2 - \mathsf{M}_{\{0,3\}}$

$\mathcal{P}_{1,2}^V : \mathsf{M}_{\{1,2\}} = \mathsf{V}_{\{1,2\}} + r_{1,2,3}$

$P_1 : \mathsf{V}_{\{1\}} = \mathsf{V}_{\{1,2\}} - \mathsf{M}_{\{1\}} \qquad P_2 : \mathsf{V}_{\{2\}} = \mathsf{V}_{\{1,2\}} - \mathsf{M}_{\{2\}}$

(2) Communicate:

$P_1 \to P_2 : \mathsf{M}_{\{1\}} \qquad P_2 \to P_1 : \mathsf{M}_{\{2\}} \qquad P_2^V \to P_0^V : \mathsf{M}_{\{1,2\}}$

(3) Locally compute:

$P_1 : \mathsf{m}_c = \mathsf{V}_{\{1\}} - \mathsf{M}_{\{2\}} \qquad P_2 : \mathsf{m}_c = \mathsf{V}_{\{2\}} - \mathsf{M}_{\{1\}}$

$P_0^V : \mathsf{m}_c^* = \mathsf{M}_{\{1,2\}} - (\mathsf{V}_{\{0\}} + \mathsf{M}_{\{3\}})$

$\mathcal{P}_{1,2}^V : \overline{\mathsf{m}}_c = \mathsf{m}_c + \lambda_c^* \qquad P_0^V : \overline{\mathsf{m}}_c = \mathsf{m}_c^* + \lambda_c$

(4) Compare views using Π_{CV}:

$\mathcal{P}_{0,1}^V : \mathsf{M}_{\{1,2\}} \qquad \mathcal{P}_{2,3}^V : \mathsf{M}_{\{0,3\}} \qquad \mathcal{P}_{0,1,2}^V : \overline{\mathsf{m}}_c$

Figure 5: Quad: 4PC multiplication protocol

Online Phase. During the online phase, P_1 and P_2 locally compute $\mathsf{m}_a\mathsf{m}_b = ab + a\lambda_b + b\lambda_a + \lambda_a\lambda_b$. They then obtain their share $\mathsf{m}_c = ab + \lambda_c$ by removing the error terms using the messages $\mathsf{M}_{\{1\}}$ and $\mathsf{M}_{\{2\}}$ that they exchange. Note that both these messages are masked with $r_{0,1,3}$ and λ_c (contained in $\mathsf{M}_{\{0,3\}}$), to prevent leakage of information. For correctness, note that:

$$\begin{aligned}\mathsf{M}_{\{1\}} &= \mathsf{m}_a\lambda_b^1 + \mathsf{m}_b\lambda_a^1 + r_{0,1,3}\\ &= a\lambda_b^1 + b\lambda_a^1 + \lambda_a\lambda_b^1 + \lambda_a^1\lambda_b + r_{0,1,3}\\ \mathsf{M}_{\{2\}} &= \mathsf{m}_a\lambda_b^2 + \mathsf{m}_b\lambda_a^2 - \mathsf{M}_{\{0,3\}}\\ &= a\lambda_b^2 + b\lambda_a^2 + \lambda_a\lambda_b^2 + \lambda_a^2\lambda_b - \mathsf{M}_{\{0,3\}}\\ \mathsf{M}_{\{1\}} + \mathsf{M}_{\{2\}} &= a\lambda_b + b\lambda_a + 2\lambda_a\lambda_b + r_{0,1,3} - \mathsf{M}_{\{0,3\}}\end{aligned}$$

Thus, the input-dependent error $a\lambda_b + b\lambda_a$ in $\mathsf{m}_a\mathsf{m}_b$ matches the input-dependent error in $\mathsf{M}_{\{1\}} + \mathsf{M}_{\{2\}}$. Moreover, the input-independent error $\lambda_a\lambda_b - r_{0,1,3}$ can be eliminated using $\mathsf{M}_{\{0,3\}} = \lambda_c + \lambda_a\lambda_b + r_{0,1,3}$ from P_0, while simultaneously inserting the mask λ_c. Using this insight, P_1 and P_2 can obtain their share as

$$\begin{aligned}\mathsf{m}_c &= \mathsf{m}_a\mathsf{m}_b - (\mathsf{M}_{\{1\}} + \mathsf{M}_{\{2\}}) \\ &= ab - \lambda_a\lambda_b - r_{0,1,3} + \mathsf{M}_{\{0,3\}} = ab + \lambda_c\end{aligned}$$

To verify $\mathcal{P}_{1,2}$'s communication we just described, P_0 also needs to obtain an input-dependent share in our 4PC protocol. To obtain its share $\mathsf{m}_c^* = ab + \lambda_c^*$, P_0 begins by locally computing $\mathsf{V}_{\{0\}} = \mathsf{m}_a^*\lambda_b + \mathsf{m}_b^*\lambda_a$. To eliminate the errors from $\mathsf{V}_{\{0\}}$ and derive m_c^*, it uses the message $\mathsf{M}_{\{1,2\}}$ received from P_2 during the online phase and $\mathsf{M}_{\{3\}}$ from P_3 during preprocessing. The final share is $\mathsf{m}_c^* = \mathsf{M}_{\{1,2\}} - (\mathsf{V}_{\{0\}} + \mathsf{M}_{\{3\}})$. For correctness, note that:

$$\begin{aligned}\mathsf{M}_{\{1,2\}} &= \mathsf{m}_a\mathsf{m}_b + r_{1,2,3} = ab + a\lambda_b + b\lambda_a + \lambda_a\lambda_b + r_{1,2,3} \\ \mathsf{V}_{\{0\}} &= \mathsf{m}_a^*\lambda_b + \mathsf{m}_b^*\lambda_a = a\lambda_b + b\lambda_a + \lambda_a^*\lambda_b + \lambda_a\lambda_b^* \\ \mathsf{M}_{\{3\}} &= \lambda_a(\lambda_b - \lambda_b^*) - \lambda_a^*\lambda_b - \lambda_c^* + r_{1,2,3}\end{aligned}$$

Verifying Communication. To ensure the correctness of the overall protocol, parties need to ensure that they have received correct messages. Each party does this by comparing their received messages with another party who can compute the same message in a different way, using the compare-view functionality (cf. Π_{CV} in Figure 3). For instance, to verify $\mathsf{M}_{\{0,3\}}$ received from P_0 during preprocessing, P_2 compares its view of $\mathsf{M}_{\{0,3\}}$ with P_3, who can compute the same message locally. Similarly, P_0 and P_1 can jointly verify $\mathsf{M}_{\{1,2\}}$.

For the remaining messages $\mathsf{M}_{\{3\}}$, $\mathsf{M}_{\{1\}}$, and $\mathsf{M}_{\{2\}}$, we observe that a single check of $\overline{\mathsf{m}}_c = ab + \lambda_c + \lambda_c^*$ by the parties in $\mathcal{P}_{0,1,2}$ is sufficient. If P_3's message M_3 is incorrect, then P_1's and P_2's correct views will differ from P_0's corrupted view. Similarly, if either P_1's message $\mathsf{M}_{\{1\}}$ to P_2 or P_2's message $\mathsf{M}_{\{2\}}$ to P_1 is incorrect, P_0's correct view will differ from their corrupted views of $\overline{\mathsf{m}}_c$. Since our protocol tolerates up to one corrupted party, only one of these cases can occur. Therefore, the parties have successfully verified all messages exchanged during the multiplication protocol.

Since the message $\mathsf{M}_{\{1,2\}}$ from P_2 to P_0 during the online phase is used only for verification, it can be delayed for all the gates by a constant factor without affecting the protocol's throughput in the amortized sense. Consequently, $\mathcal{P}_{0,2}$ can operate over a high-latency link, even if the evaluated circuit has a high multiplicative depth. Messages used in this manner can also be considered part of a constant-round post-processing phase. In our protocol, the parties achieve low computational complexity by reusing the calculated terms across messages, verification, and obtaining shares. The correctness of our protocol and the verification of messages are detailed in §A.2.

Post-processing routine. Here, we detail the flow of the protocol when P_0 has an arbitrary delay. Consider a circuit with n consecutive multiplication gates. Let's assume that $\mathcal{P}_{0,3}$ have completed the preprocessing phase. During the online phase, P_1 and P_2 interactively execute steps (1) to (3) of Π_{Mult} (cf. Figure 5) for each multiplication gate sequentially. However, all messages from P_2 to P_0 corresponding to the n gates are delayed until the end of the protocol and sent in one round. Upon receiving the messages, P_0 performs its local computations for all n gates. Following this, P_0 compares its views with other parties using the compare-view (cf. Π_{CV} in Figure 3) functionality. Meanwhile, P_2 and P_3 execute Π_{CV} on the message $\mathsf{M}_{\{0,3\}}$ for all n gates.

While this provides the intuition, the parties are not required to execute the preprocessing, online, and postprocessing phases sequentially. Instead, they can simultaneously execute all phases. In this context, communication and computation marked by $\mathcal{P}^V$ are non-blocking, indicating they are not latency-critical. Conversely, all other computation and communication are blocking, necessitating at least one party to wait for another party's communication and computation before proceeding with the protocol. By interleaving all three phases, the total runtime of the protocol is minimized in practice. This approach is utilized in our implementation (cf. §8.1 for details) to reduce the total runtime compared to online-only protocols like Fantastic Four [17]. Since the preprocessing phase relies only on local computations, the online phase is unlikely to be blocked due to dependencies from the preprocessing phase in this interleaved processing model.

5.4 Multiplication in Heterogeneous Networks

The 4PC multiplication protocol Π_{Mult} in Figure 5 tolerates three links with low bandwidth and five links with high latency between the parties (cf. Figure 7c and Figure 7d in §7). Here, we provide a variant of the multiplication protocol, $\Pi_{\mathsf{Mult\_H}}$ (cf. Figure 6), tolerates four links with low bandwidth by shifting the communication of multiple messages to the same link. This way, this variant is resilient to network settings where the majority of the links between the parties are weak as demonstrated in figure cf. Figure 11b.

At a high level, we shift the communication in the network link between $\mathcal{P}_{0,3}$ to the already utilized link between $\mathcal{P}_{0,2}$. Specifically, we replace the message $\mathsf{M}_{\{3\}}$ from P_3 to P_0 in the preprocessing phase of Π_{Mult} with another message $\mathsf{M}_{\{1,2\},2}$ that P_2 sends to P_0 during the online phase. P_0 then verifies the communication between P_1 and P_2 using $\mathsf{M}_{\{1,2\},2}$ instead of $\mathsf{M}_{\{3\}}$. This modification has the added advantage that P_3 does not need to communicate with any other party during circuit evaluation, allowing it to have an arbitrarily weak network links to all other parties.

They key difference of $\Pi_{\mathsf{Mult\_H}}$ compared to Π_{Mult} is as follows: During the preprocessing phase, P_3 computes $\mathsf{V}_{\{0,3\}} = \lambda_a(\lambda_b - \lambda_b^*) - \lambda_b\lambda_a^* - \lambda_c + r_{1,2,3}$ but does not send it to P_0. During the online phase, P_1 and P_2 proceed similarly to Π_{Mult} to obtain m_c but they also compute two messages, $\mathsf{M}_{\{1,2\},1}$ and $\mathsf{M}_{\{1,2\},2}$, which P_2 sends to P_0. P_0 then utilizes $\mathsf{M}_{\{1,2\},1}$ to compute its share m_c^* and uses $\mathsf{M}_{\{1,2\},2}$ to verify $\mathcal{P}_{1,2}$'s communication.

Verifying Communication. To help P_1 and P_2 verify the correctness of their exchanged messages $\mathsf{M}_{\{1\}}$ and $\mathsf{M}_{\{2\}}$, P_0 proceeds as follows: P_0 computes $\mathsf{V}_{\{0,3\}} = \mathsf{M}_{\{1,2\},2} - \mathsf{V}_{\{0\}}$ and compares its view with P_3, who computed it locally during preprocessing. If the views are consistent, P_0 confirms that $\mathsf{M}_{\{1,2\},2} = \mathsf{M}_{\{1\}} + \mathsf{M}_{\{2\}} + r_{1,2,3}$, implying $\mathsf{M}_{\{1\}} + \mathsf{M}_{\{2\}}$ is correct. Given that at most one of P_1 or P_2 can be corrupt, the correctness of $\mathsf{M}_{\{1\}} + \mathsf{M}_{\{2\}}$ implies that both $\mathsf{M}_{\{1\}}$ and $\mathsf{M}_{\{2\}}$ are correct. Additionally, P_0 compares its view of $\mathsf{M}_{\{1,2\},2}$ with P_1 for consistency, while P_2 ensures the correctness of $\mathsf{M}_{\{0,3\}}$ received from P_0 by comparing it with P_3. Assuming that both

checks succeed, one can verify the correctness of $\mathsf{V}_{\{0,3\}} = \mathsf{M}_{\{1,2\},2} - \mathsf{V}_{\{0\}}$ by noting the following:

$$\mathsf{M}_{\{1,2\},2} = \mathsf{M}_{\{1\}} + \mathsf{M}_{\{2\}} + r_{1,2,3} = a\lambda_b + b\lambda_a + \lambda_a\lambda_b - \lambda_c + r_{1,2,3}$$
$$\mathsf{V}_{\{0\}} = \mathsf{m}^*_a\lambda_b + \mathsf{m}^*_b\lambda_a = a\lambda_b + b\lambda_a + \lambda_a\lambda^*_b + \lambda_b\lambda^*_a$$
$$\mathsf{V}_{\{0,3\}} = \mathsf{M}_{\{1,2\},2} - \mathsf{V}_{\{0\}} = \lambda_a\lambda_b - \lambda_a\lambda^*_b - \lambda_b\lambda^*_a - \lambda_c + r_{1,2,3}$$

Finally, to verify correctness of $\mathsf{M}_{\{1,2\},1}$ sent by P_2, P_0 performs a consistency check with P_1. This will ensure that P_0 obtained the correct share m^*_c.

The correctness of our protocol and the verification of messages are detailed in §A.3. The security of Quad is proved using real-world/ideal-world simulation paradigm [20, 28] and the details are provided in §C.

Protocol $\Pi_{\mathsf{Mult\_H}}(\llbracket a \rrbracket, \llbracket b \rrbracket) \rightarrow \llbracket c \rrbracket$

Preprocessing:

(1) Sample random values using Π_{SRNG}:

$$\mathcal{P}_{0,1,3} : r_{0,1,3}, \lambda^1_c \qquad \mathcal{P}_{0,2,3} : \lambda^2_c \qquad \mathcal{P}^V_{1,2,3} : r_{1,2,3}, \lambda^*_c$$

(2) Locally compute:

$$P_0, P^V_3 : \lambda_c = \lambda^1_c + \lambda^2_c$$
$$P_0, P^V_3 : \mathsf{M}_{\{0,3\}} = \lambda_c + \lambda_a\lambda_b + r_{0,1,3}$$
$$P^V_3 : \mathsf{V}_{\{0,3\}} = \lambda_a(\lambda_b - \lambda^*_b) - \lambda_b\lambda^*_a - \lambda_c + r_{1,2,3}$$

(3) Communicate:

$$P_0 \rightarrow P_2 : \mathsf{M}_{\{0,3\}}$$

Online:

(1) Locally compute:

$$P_1 : \mathsf{M}_{\{1\}} = \mathsf{m}_a\lambda^1_b + \mathsf{m}_b\lambda^1_a + r_{0,1,3} \qquad P_2 : \mathsf{M}_{\{2\}} = \mathsf{m}_a\lambda^2_b + \mathsf{m}_b\lambda^2_a - \mathsf{M}_{\{0,3\}}$$
$$P_1 : \mathsf{V}_{\{1\}} = \mathsf{m}_a\mathsf{m}_b - \mathsf{M}_{\{1\}} \qquad P_2 : \mathsf{V}_{\{2\}} = \mathsf{m}_a\mathsf{m}_b - \mathsf{M}_{\{2\}}$$
$$P^V_0 : \mathsf{V}_{\{0\}} = \mathsf{m}^*_a\lambda_b + \mathsf{m}^*_b\lambda_a$$

(2) Communicate:

$$P_1 \rightarrow P_2 : \mathsf{M}_{\{1\}} \qquad P_2 \rightarrow P_1 : \mathsf{M}_{\{2\}}$$

(3) Locally compute:

$$P_1 : \mathsf{m}_c = \mathsf{V}_{\{1\}} - \mathsf{M}_{\{2\}} \qquad P_2 : \mathsf{m}_c = \mathsf{V}_{\{2\}} - \mathsf{M}_{\{1\}}$$
$$\mathcal{P}^V_{1,2} : \mathsf{M}_{\{1,2\},1} = \mathsf{m}_c + \lambda^*_c \qquad \mathcal{P}^V_{1,2} : \mathsf{M}_{\{1,2\},2} = \mathsf{M}_{\{1\}} + \mathsf{M}_{\{2\}} + r_{1,2,3}$$

(4) Communicate:

$$P^V_2 \rightarrow P^V_0 : \mathsf{M}_{\{1,2\},1} \qquad P^V_2 \rightarrow P^V_0 : \mathsf{M}_{\{1,2\},2}$$

(5) Locally compute:

$$P^V_0 : \mathsf{V}_{\{0,3\}} = \mathsf{M}_{\{1,2\},2} - \mathsf{V}_{\{0\}} \qquad P^V_0 : \mathsf{m}^*_c = \mathsf{M}_{\{1,2\},1} - \lambda_c$$

(6) Compare views using Π_{CV}:

$$\mathcal{P}^V_{0,1} : \mathsf{M}_{\{1,2\},1}, \mathsf{M}_{\{1,2\},2} \qquad \mathcal{P}^V_{0,3} : \mathsf{V}_{\{0,3\}} \qquad \mathcal{P}^V_{2,3} : \mathsf{M}_{\{0,3\}}$$

Figure 6: Quad: Heterogeneous 4PC multiplication protocol

6 ADDITIONAL PROTOCOLS

So far, we have seen the details of our protocols over ℓ-bit rings. However, to use our protocols for real-world applications, it requires support for Fixed-Point Arithmetic (FPA) [11, 35] and mixed circuits [13, 34, 37]. We summarize our contributions in this direction below and the formal details are provided in §B.2. Note that these protocols use the same network links between the parties as our multiplication protocol. Therefore, they maintain their high tolerance to weak network links.

(1) Truncation (§B.2.1): Protocols using Fixed-Point Arithmetic require a truncation operation to prevent overflows during computation and maintain precision [11]. Probabilistic truncation [35] is one such method, which takes a share $\llbracket x \rrbracket = (\mathsf{m}_x, \lambda_x)$ and outputs its truncated version $(\llbracket x \rrbracket)^{pt} = \left(\lfloor \frac{\mathsf{m}_x}{2^t} \rfloor, \lfloor \frac{\lambda_x}{2^t} \rfloor\right)$. With high probability, $(\llbracket x \rrbracket)^{pt} \approx \lfloor \frac{x}{2^t} \rfloor$. Here t denotes the number of fractional bits in the FPA representation.

Similar to ASTRA [12, 41] and Tetrad [27], probabilistic truncation can be incorporated to our multiplication protocols at no additional communication cost and inherits the computational improvements from our multiplication protocols.

(2) Matrix Multiplication (§B.2.2): Similar to existing works [12, 17, 27], our multiplication protocols can be extended trivially to handle matrix multiplications, where the addition and multiplication operations are replaced with its matrix counterparts. Moreover, dot product can be viewed as a special case of matrix multiplication.

(3) Comparisons (§B.2.3): Comparisons are a key element in many applications, including non-linear activation functions like ReLU. In the context of rings, comparisons are typically performed by examining the most significant bit [12, 34]: if this bit is 1, the value is negative; if it is 0, the value is non-negative.

(4) Arithmetic to Boolean Conversion (§B.2.4): Given the arithmetic sharing of $x \in \mathbb{Z}_{2^\ell}$, this protocol generates the boolean sharing of all the ℓ bits of x.

(5) Bit to Arithmetic Conversion (§B.2.5): Given the boolean sharing of a bit a^{b}, denoted by $\llbracket a^{\mathsf{b}} \rrbracket^B$, the protocol generates the arithmetic equivalent shares $\llbracket a^{\mathsf{b}} \rrbracket^A$.

7 MPC IN VARIOUS NETWORK SETTINGS

In this section, we analyse the network links among the parties in our 3PC and 4PC protocols from the view point of latency and bandwidth. For this analysis, we distinguish three different categories of network links between the parties.

(1) *Category I:* This category includes links used only for setting up pre-shared keys between parties or exchanging hashes for verification at protocol's end, not for the bulk of communication; e.g., link between $\mathcal{P}_{0,1}$ in our protocols. Low bandwidth or high latency on these links does not significantly impact performance.

(2) *Category II:* This category involves links among parties that depend on each other's messages to jointly evaluate a circuit. An example is the network link between $\mathcal{P}_{1,2}$. These parties must interactively evaluate each layer of the circuit and repeatedly wait for each other's incoming communication. Hence, a circuit with a multiplicative depth of k needs at least k sequential chunks of messages exchanged between these parties. Low bandwidth or high latency on these links significantly reduces performance.

(3) *Category III:* This category covers links only required during a constant-round preprocessing phase or verification. An example is the link between $\mathcal{P}_{0,2}$. P_0 only sends messages to P_2 in the preprocessing phase. These messages can be received in a single chunk, and delaying them only adds a constant overhead to the

protocol's execution time irrespective of the circuit's multiplicative depth. In Quad, P_2 also sends messages to P_0 for verification purposes. In contrast to the link between $\mathcal{P}_{1,2}$, there is no bidirectional dependency associated with P_0 receiving P_2's messages: P_0 can compute all its messages to other parties based solely on its input-independent shares during the preprocessing. Therefore, low bandwidth on these links reduces performance noticeably, but high latency does not.

Figure 7 illustrates the resulting link requirements between the parties following these three different cases, where links of Category II are denoted by bold arrows (⟶), while links of Category III are denoted by dotted arrows (⇢). The remaining links between the parties are of Category I and therefore mainly underutilized.

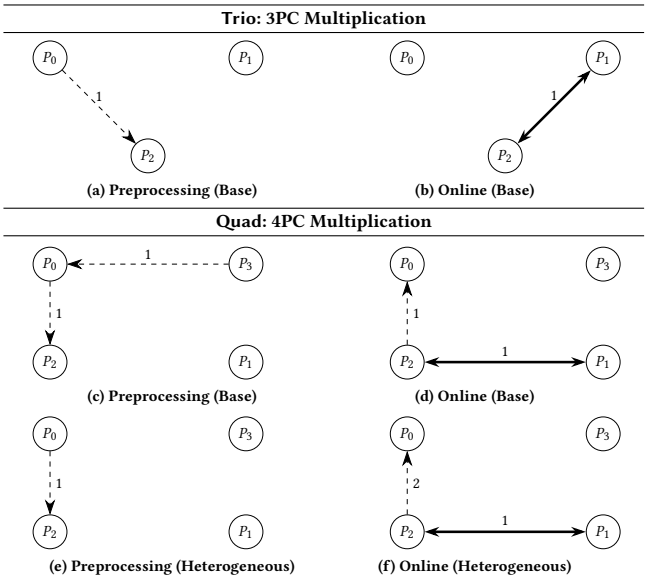


Figure 7: Communication pattern during multiplication

– (Bandwidth-critical) Dashed arrows denote constant communication rounds.
– (Bandwidth & Latency-critical) Bold arrows denote communication rounds linear in the circuit's multiplicative depth.

Heterogeneous Network Settings. When designing a protocol for heterogeneous network settings, our goal is to maximize the use of Category I links while minimizing the use of Category II links. Consequently, in our 3PC and heterogeneous 4PC protocols, only one link between the parties is latency-critical, and only two links are bandwidth-critical.

In our heterogeneous 4PC protocol, only P_3 needs to use Category I links that are not performance-critical. This means that any network configuration that performs well with our semi-honest 3PC protocol will also perform well in a malicious setting, as long as the parties can agree on a fourth participant. However, this protocol variant requires P_2 to send two elements of communication per multiplication gate in one direction, which effectively throttles communication on that link by a factor of two. Therefore, this variant is most suitable for scenarios where the available bandwidth between parties differs by at least a factor of two.

Homogeneous Network Settings. While minimizing link dependencies is beneficial in heterogeneous network settings where network link properties between parties differ, we show how our protocols can be adapted to network settings where this is not a primary concern. In homogeneous network settings, where all parties have similar bandwidth and latency, an efficient protocol distributes its communication complexity evenly across all links. This approach has been previously used for load-balancing MPC protocols [27, 31].

Any n-party protocol can be converted into a protocol optimized for homogeneous network settings by running $n!$ circuits in parallel. In each evaluation, the parties select a unique permutation of their roles in the protocol. For example, consider a 3PC protocol with parties P_i, P_j, and P_k. There are six unique permutations to assign the roles of $\mathcal{P}_{0,1,2}$ to P_i, P_j, and P_k.

For every protocol using l elements of global communication, the number of messages per circuit remains the same, but all communication channels are now utilized equally. We refer to the technique of switching player assignments during joint evaluation of MPC workloads as *Split Roles*. Figure 8 illustrates the resulting communication between nodes when using our 3PC and 4PC protocols in a homogeneous network setting when utilizing this technique. To optimize their communication pattern for a given network setting, the parties can also customize the set of permutations depending on which links should be utilized more and which ones less.

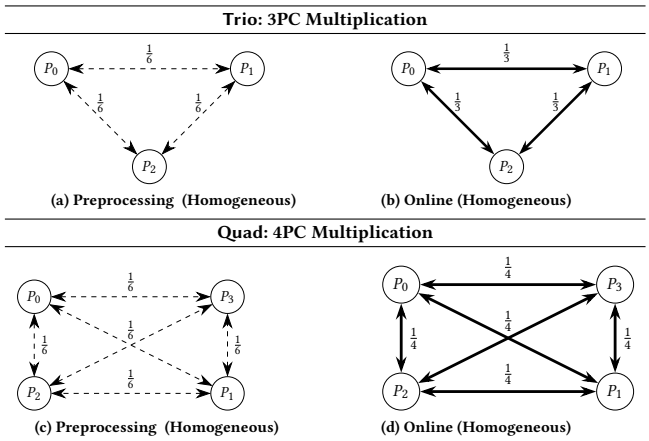


Figure 8: Communication Pattern using Split Roles

– (Bandwidth-critical) Dashed arrows denote constant communication rounds.
– (Bandwidth & Latency-critical) Bold arrows denote communication rounds linear in the circuit's multiplicative depth.

Other Network Settings. Our protocols offer high flexibility in optimizing communication patterns for different network settings, thanks to their high tolerance for weak network links and their use of replicated sharing semantics. The parties can utilize their replicated secret sharing to adjust their communication pattern on a per-message basis.

When parties hold one input-dependent and one input-independent share, any outgoing or incoming message can be shifted between parties without introducing any additional communication costs. For instance, in our 3PC and 4PC protocols, P_1's and P_2's computation and communication pattern can be easily adjusted such that P_0 sends its message in the preprocessing phase to P_1 instead of P_2. By selecting the percentage of messages to send to a specific party, P_0 can granularly adjust the utilization of different network links based on available bandwidth.

8 IMPLEMENTATION & BENCHMARKING

We implemented our protocols and related state-of-the-art ones in C++. Specifically, we implemented two other state-of-the-art protocols for each category, namely the 3PC protocols ASTRA [12] and Replicated [2] and the 4PC protocols Fantastic Four [17] and Tetrad [27]. One protocol in each category offers function-dependent preprocessing (ASTRA, Tetrad), while the other does not (Replicated, Fantastic Four).

All results in this section are based on a test setup of 3 to 4 nodes. Each node is connected with a 25 Gbit/s duplex link to each other node and equipped with a 32-core AMD EPYC 7543 processor. If not stated otherwise, we do not use a separate preprocessing phase but perform all preprocessing operations during the online phase.

This section is divided into three main parts:

(1) In §8.1, we demonstrate how to scale MPC protocols to achieve a throughput of billions of gates per second under standard conditions. We propose a series of optimizations that can be universally applied to any MPC protocol.

(2) §8.2 highlights the improvements of our protocols in practical scenarios, including heterogeneous and bandwidth-restricted settings. This part addresses and overcomes the practical bottlenecks of MPC discussed in §2.

(3) Finally, we present the benchmarking results for AES circuit evaluations using our protocols in §8.3.

8.1 Scaling MPC to Billions of Gates

Our framework integrates a comprehensive set of universal optimizations: Bitslicing, Vectorization, multiprocessing, hardware instructions like `VAES` and `SHA`, and adjustable message buffering. In this section, we show how stacking up these optimizations allow us to scale MPC workloads to billions of gates per second. Additionally, our results confirms our finding that existing open-source frameworks struggle to achieve a throughput of more than a few million gates per second (cf. §2).

Different MPC protocols typically require at least some of the following components to evaluate a non-linear gate: Encrypted communication channels, cryptographically secure random number generators and hash functions, as well as multiple elementary operations. Additionally, MPC workloads are highly sensitive to network bandwidth and latency. Therefore, implementations benefit significantly from accelerating these basic instructions and optimizing network communication. We focus on these aspects next.

Accelerating Basic Instructions. As MPC protocols require the same kind of instructions repeatedly for each gate, we use the Single Instruction Multiple Data (SIMD) approach to accelerate them by using wider register sizes. For example, we can use the `AVX-2` instruction set to perform eight 32-bit additions, 256 1-bit logic gates, or two 128-bit AES rounds in parallel using a single instruction on a 256-bit register. For SSL-encrypted communication, we rely on the OpenSSL library. Notably, the throughput of the cryptographic instructions is, on average, only 5-10 times slower than the throughput of the non-cryptographic instructions thanks to available hardware instructions such as `VAES` and `SHA`.

Bitslicing and Vectorization. The key idea of Bitslicing is that computing a bitwise logical operation on an m-bit register effectively operates like m parallel boolean conjunctions, each processing a single bit [32]. Thus, Bitslicing can accelerate single-bit operations such as AND or XOR, provided that the bits to be operated on are packed properly. For example, instead of performing a single 1-bit XOR operation across two bits, we could perform 32 XOR operations on 32-bit data if the bits are packed into a single 32-bit variable. Furthermore, one can exploit hardware instruction sets such as `AVX-2` to pack 256 bits and compute 256 XOR operations in parallel. To support Bitslicing on various CPU architectures, we utilize the header files provided by the Usuba Bitslicing compiler [33].

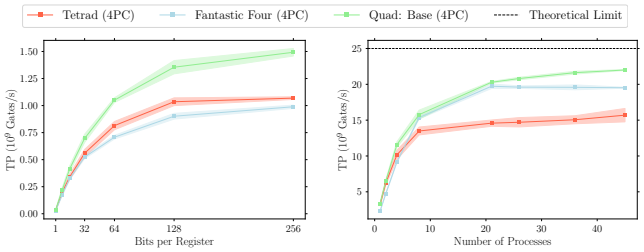


Figure 9: Throughput when utilizing Bitslicing and Multiprocessing

Figure 9 (left) shows the throughput of the 4PC protocols when utilizing Bitslicing with increasing bit widths to evaluate AND gates. As we increase the bit width, we effectively utilize hardware parallelism on a single core to process a batch of 256 boolean operations using a single instruction. Consequently, the throughput measured on our test setup increases over 100 times when performing 256 AND gates in parallel on an `AVX-2` register, compared to using a single boolean variable and performing one instruction for each input.

Table 6: Throughput for protocols without using Split-Roles

Setting	Protocol	Billion Gates/s ($\sigma \pm \mu$)	Theoretical Limit (%)
3PC	Replicated	24.57 ± 0.13	98.28 ± 0.51
	ASTRA	23.91 ± 0.12	95.64 ± 0.50
	Trio	23.81 ± 0.38	95.24 ± 1.50
4PC	Fantastic Four	18.93 ± 0.44	75.71 ± 1.75
	Tetrad	15.18 ± 0.27	60.72 ± 1.10
	Quad	22.04 ± 0.05	88.16 ± 0.20

Various MPC workloads, such as privacy-preserving machine learning, or user authentication as motivated by [2], operates on batches of data and can benefit significantly from utilizing multiple cores. Figure 9 (right) and Table 6 show that when combining Bitslicing with multiprocessing, our implementations achieve a throughput of more than 15 billion AND gates per second across all protocols. These results are within 61% – 98% of the theoretical limit of 25 billion AND gates per second that we can achieve on a 25-Gbit/s network without using Split-roles. The remaining gap in throughput is likely due to networking overhead incurred when sending and receiving messages with multiple threads using conventional sockets.

Optimizing Network Communication. Optimizing local computations for MPC workloads is only effective until the network bottlenecks further performance improvements. In the following lines, we present techniques to fully utilize the available network bandwidth between parties.

Buffering. When evaluating a circuit, the parties must exchange a certain number of messages in each communication round. A party can either send each message as soon as it is computed, or it can buffer a set of messages and send them all at once. Our measurements showed a 50 times difference in throughput between an ideal and worst-case buffer size. On our test setup, buffering between 0.3MB and 3*MB* of messages lead to the highest throughput.

Split-Roles. To achieve more than 25 billion AND gates per second on our network, we need to utilize Split-Roles. This way, all messages are equally distributed between the parties, and the available network bandwidth is fully utilized. For instance, on a 25-Gbit/s network, we can theoretically achieve a throughput of 50 billion AND gates per second by utilizing Split-Roles with a 3PC protocol that requires three elements of global communication. We can increase the throughput further by executing a 3PC protocol with four parties, essentially creating a 4PC protocol. This way, we can achieve a theoretical throughput of 100 billion AND gates per second on a 25-Gbit/s network as the total number of links between the parties doubles. Table 7 shows the throughput of the implemented protocols when utilizing Split-Roles along with our other tweaks.

Table 7: Throughput for protocols when using Split-Roles

Setting	Protocol	Billion Gates/s ($\sigma \pm \mu$)	Theoretical Limit (%)
3PC SH	Replicated	36.16 ± 0.35	72.32 ± 0.70
	ASTRA	45.81 ± 0.47	91.62 ± 0.94
	Trio	44.04 ± 0.71	88.08 ± 1.42
	ASTRA (Online)	58.16 ± 4.07	77.55 ± 5.43
	Trio (Online)	61.95 ± 2.71	82.60 ± 3.61
4*PC SH[a]	Replicated	64.50 ± 0.72	64.50 ± 0.72
	ASTRA	73.81 ± 1.37	73.81 ± 1.37
	Quad	70.96 ± 1.42	70.96 ± 1.42
	ASTRA (Online)	90.93 ± 3.92	60.62 ± 2.61
	Quad (Online)	88.21 ± 7.22	58.81 ± 4.81
4PC MAL	Fantastic Four	26.92 ± 0.07	44.87 ± 0.11
	Tetrad	32.30 ± 0.97	43.07 ± 1.29
	Quad	38.29 ± 0.81	51.05 ± 1.08
	Tetrad (Online)	47.05 ± 1.44	47.05 ± 1.44
	Quad (Online)	59.39 ± 4.92	59.39 ± 4.92
TTP	Trusted Third Party	540.99 ± 20.57	-

SH refers to semihonest, MAL refers to malicious
[a] 4*PC SH refers to executing a 3PC protocol on four nodes

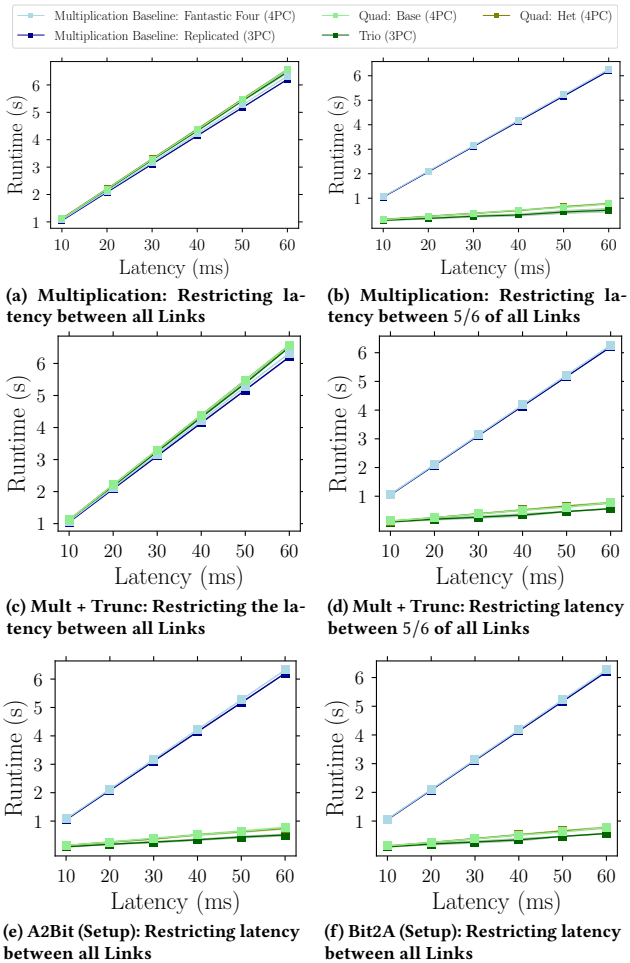
Online Phase. Most protocols we implemented offer a preprocessing phase that can be detached from the online phase. Table 7 shows the throughput of the implemented protocols when considering both phases and when only considering the Online Phase. We additionally compare the throughput of the Online Phase to the throughput a Trusted Third Party (TTP) can achieve on the same hardware. Notably, the throughput when utilizing a TTP is less than one order of magnitude higher than the secure alternatives when utilizing all aforementioned optimizations.

8.2 Overcoming MPC Bottlenecks

Our protocols excel especially in two scenarios: Heterogeneous network settings, and computational extensive tasks. We simulate heterogeneous network settings by using Linux traffic control (tc) to restrict the bandwidth between the nodes.

Latency Restricted Settings. To demonstrate the utility of our protocols under latency-restricted settings, we first compare 50 sequential evaluations of our multiplication and multiplication + truncation protocols against a baseline of the Fantastic Four and Replicated multiplication protocols which are one-round protocols as well. Figures 10a and 10c show that if we restrict the latency between all links in our setup, our protocols perform almost identically to the baseline. When we however simply allow one link between the parties to be unaffected by the link throttling, figures 10b and 10d show, that our protocols are only marginally affected by the restriction compared to the baseline.

Our A2Bit and Bit2A setup protocols do not require any latency-critical communication between the parties and are thus classified by us as 0-round protocols. Figures 10e and 10f show that even if we restrict the latency of all links between parties, these are marginally affected compared to the one-round baseline. Note that these are only the setup phases required before performing an arithmetic XOR or boolean addition to complete the share conversion.



(a) Multiplication: Restricting latency between all Links

(b) Multiplication: Restricting latency between 5/6 of all Links

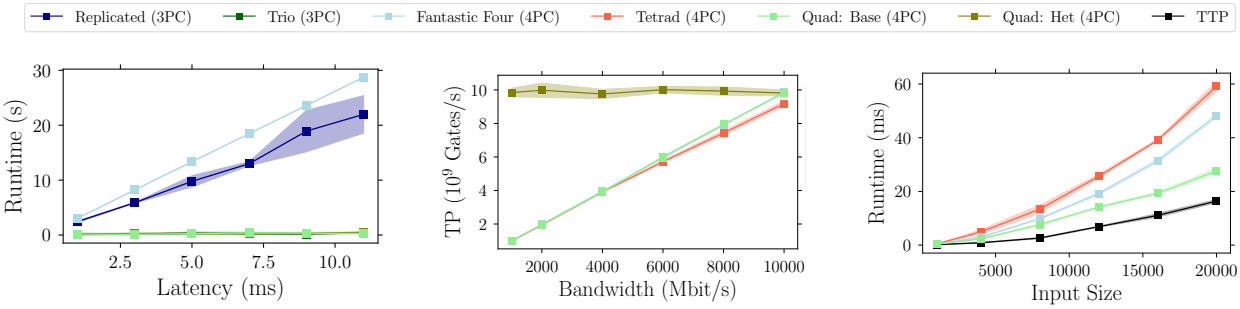
(c) Mult + Trunc: Restricting the latency between all Links

(d) Mult + Trunc: Restricting latency between 5/6 of all Links

(e) A2Bit (Setup): Restricting latency between all Links

(f) Bit2A (Setup): Restricting latency between all Links

Figure 10: Runtimes of 50 sequential operations under various latency restrictions



(a) AES Runtime: Restricting latency between $\frac{5}{6}$ of all links

(b) Multiplication Throughput (TP): Restricting bandwidths between $\frac{2}{3}$ of all links

(c) Vector-Matrix-Product Runtimes against TTP Baseline

Figure 11: Sweetspots of our 4PC protocols

Beyond these basic primitives, we also show the advantage of relying only on a few links for the more complex AES circuit. Figure 11a shows that our protocols are significantly faster than Replicated 3PC and Fantastic Four in latency-restricted settings when the network latency between the parties differs.

Bandwidth restricted Settings. When we restrict the bandwidth between $\frac{1}{3}$ of the nodes in our setup, our 3PC protocol still achieves the same throughput of approx. 24 billion *AND* gates per second, as measured in the unrestricted setting. This is due to the link between $\mathcal{P}_{0,1}$ not being utilized at all in the multiplication protocol. Figure 11b shows that even if we restrict the bandwidth of $\frac{2}{3}$ of the links in our setup, our 4PC variation optimized for heterogeneous settings still achieves a throughput of approx. 10 billion *AND* gates per second. The figure shows that while the bandwidth restriction affects all other protocols, it does not affect the throughput of our heterogeneous protocol.

Computationally Expensive Tasks. Our protocols excel at computationally demanding tasks due to their reduced number of basic instructions compared to related work. Dot products are one example of these tasks: ABY3 [34] first discovered that the communication complexity to evaluate a dot product of size l is that of a single multiplication. Thus, sufficiently large dot products become inevitably computation-bound. To benchmark the performance of dot products, we compute the product of a vector of size n with a matrix of size $n \times n$, resulting in n dot products of size $l = n$. A vector-matrix product is, for instance, required in privacy-preserving machine learning when evaluating fully connected layers. Figure 11c shows that our 4PC protocol is two times faster when evaluating large dot products than Tetrad and Fantastic Four. Furthermore, our Trusted-Third-Party implementation is less than two times faster than our 4PC protocol on the same hardware.

8.3 Benchmarking AES Circuits

AES is a common benchmark for assessing the performance of MPC frameworks and protocols. Araki et al. [2] have achieved the highest AES throughput so far, with 1.3 million 128-bit AES blocks per second. To test whether our tweaks on the throughput of raw *AND* and multiplication gates translate to more complex circuits, we benchmark the throughput of 128-bit AES blocks using the implemented protocols. As the basis for the AES circuit, we utilize the optimized AES circuit proposed by USUBA [33]. We perform over 90 million AES blocks in parallel using all optimizations introduced in §8.1.

Table 8: Throughput in million AES blocks per second

Setting	Protocol	Million Blocks/s ($\sigma \pm \mu$)	Theoretical Limit (%)
3PC	Replicated	5.18 ± 0.06	54.71 ± 0.59
	ASTRA	6.03 ± 0.32	63.69 ± 3.35
	Trio	5.13 ± 0.06	54.16 ± 0.68
	ASTRA (Online)	6.69 ± 0.21	47.12 ± 1.46
	Trio (Online)	6.69 ± 0.11	47.07 ± 0.77
4PC	Fantastic Four	2.11 ± 0.06	18.73 ± 0.53
	Tetrad	2.57 ± 0.03	22.81 ± 0.23
	Quad	3.63 ± 0.06	32.25 ± 0.53
	Tetrad (Online)	2.86 ± 0.04	15.34 ± 0.22
	Quad (Online)	4.15 ± 0.07	22.23 ± 0.36
	Trusted Third Party	17.21 ± 0.05	-

Table 8 shows the throughput in AES blocks per second. While our protocols cannot saturate the network to the same degree as for raw *AND* gates, we can still achieve four times higher throughput than previous work using the same 3PC protocol as [2]. The 4PC protocols achieve around half the performance of the 3PC protocols in our implementation. Depending on the computational complexity of the underlying protocol, we are able to saturate between 15% and 64% of our 25 Gbit/s connection.

9 CONCLUSION

In this work, we introduced novel three-party and four-party computation protocols designed to achieve high throughput across various network settings. Leveraging outsourced MPC computations, our protocols enable efficient MPC for any number of input parties. Our open-source implementation demonstrates the capability to evaluate billions of gates per second, even in scenarios where inter-party links exhibit high latency and limited bandwidth. This underscores MPC's ability to manage intensive workloads across real-world settings characterized by varied computational node capabilities. Our benchmarks highlight the need for optimizing both communication and computational aspects of MPC protocols as well as enhancements in MPC implementations to bridge the performance gap with Trusted Third Party setups.

Looking forward, an interesting direction for future research involves exploring optimizations tailored for heterogeneous network conditions in different honest-majority scenarios. Moreover, given the high throughput our implementation achieves while processing boolean and arithmetic gates, future research could extend these techniques to handle more complex tasks, such as privacy-preserving machine learning applications that involve large-scale dot product evaluations and non-linear activation functions.

Acknowledgements

The authors Yongqin Wang and Murali Annavaram would like to acknowledge the support by Defense Advanced Research Projects Agency (DARPA) under Contract Nos. HR001120C0088, NSF award number 2224319, REAL@USC-Meta center, and VMware gift. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

REFERENCES

[1] Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, and Or Weinstein. 2017. Optimized Honest-Majority MPC for Malicious Adversaries - Breaking the 1 Billion-Gate Per Second Barrier. In *IEEE S&P*. 3, 6

[2] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. 2016. High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority. In *CCS*. 1, 2, 3, 4, 11, 13, 23

[3] Donald Beaver, Silvio Micali, and Phillip Rogaway. 1990. The Round Complexity of Secure Protocols (Extended Abstract). In *STOC*. 1

[4] Aner Ben-Efraim, Michael Nielsen, and Eran Omri. 2019. Turbospeedz: Double Your Online SPDZ! Improving SPDZ Using Function Dependent Preprocessing. In *ACNS*. 4

[5] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. 1988. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract). In *STOC*. 1–10. 2

[6] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. 2019. Zero-Knowledge Proofs on Secret-Shared Data via Fully Linear PCPs. In *CRYPTO*. 6

[7] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. 2019. Practical Fully Secure Three-Party Computation via Sublinear Distributed Zero-Knowledge Proofs. In *CCS*. 2

[8] Lennart Braun, Daniel Demmler, Thomas Schneider, and Oleksandr Tkachenko. 2022. MOTION - A Framework for Mixed-Protocol Multi-Party Computation. *ACM Trans. Priv. Secur.* (2022). 2, 3, 4

[9] Andreas Brüggemann, Oliver Schick, Thomas Schneider, Ajith Suresh, and Hossein Yalame. 2024. Don't Eject the Impostor: Fast Three-Party Computation With a Known Cheater. In *IEEE S&P*. 2, 3, 6, 18

[10] Megha Byali, Harsh Chaudhari, Arpita Patra, and Ajith Suresh. 2020. FLASH: Fast and Robust Framework for Privacy-preserving Machine Learning. *PETS* (2020). 1, 2

[11] Octavian Catrina and Amitabh Saxena. 2010. Secure Computation with Fixed-Point Numbers. In *FC*. 9

[12] Harsh Chaudhari, Ashish Choudhury, Arpita Patra, and Ajith Suresh. 2019. ASTRA: High Throughput 3PC over Rings with Application to Secure Prediction. In *CCSW@CCS*. 1, 2, 3, 4, 5, 6, 9, 11, 23

[13] Harsh Chaudhari, Rahul Rachuri, and Ajith Suresh. 2020. Trident: Efficient 4PC Framework for Privacy Preserving Machine Learning. In *NDSS*. 1, 2, 4, 9

[14] Ran Cohen and Yehuda Lindell. 2017. Fairness Versus Guaranteed Output Delivery in Secure Multiparty Computation. *Journal of Cryptology* (2017). 7

[15] Ronald Cramer, Ivan Damgård, and Yuval Ishai. 2005. Share Conversion, Pseudorandom Secret-Sharing and Applications to Secure Computation. In *TCC*. 1

[16] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. 2015. *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press. 1

[17] Anders P. K. Dalskov, Daniel Escudero, and Marcel Keller. 2021. Fantastic Four: Honest-Majority Four-Party Secure Computation With Malicious Security. In *USENIX Security*. 1, 2, 3, 4, 6, 8, 9, 11, 23

[18] Ivan Damgård, Kasper Damgård, Kurt Nielsen, Peter Sebastian Nordholt, and Tomas Toft. 2016. Confidential Benchmarking Based on Multiparty Computation. In *FC*. 2

[19] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. 2017. High-Throughput Secure Three-Party Computation for Malicious Adversaries and an Honest Majority. In *EUROCRYPT*. 3, 6

[20] Oded Goldreich. 2004. *The Foundations of Cryptography - Volume 2: Basic Applications*. Cambridge University Press. 6, 9, 20

[21] S. Dov Gordon, Samuel Ranellucci, and Xiao Wang. 2018. Secure Computation with Low Communication from Cross-Checking. In *ASIACRYPT*. 4

[22] Marcel Keller. 2020. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In *ACM CCS*. 2, 3, 4

[23] Marcel Keller, Emmanuela Orsini, and Peter Scholl. 2016. MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer. In *CCS*. 1

[24] Liisi Kerik, Peeter Laud, and Jaak Randmets. 2016. Optimizing MPC for Robust and Scalable Integer and Floating-Point Arithmetic. In *FC*. 1, 2, 3

[25] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. 2021. SWIFT: Super-fast and Robust Privacy-Preserving Machine Learning. In *USENIX Security*. 1, 2, 4, 6

[26] Nishat Koti, Shravani Mahesh Patil, Arpita Patra, and Ajith Suresh. 2023. MPClan: Protocol Suite for Privacy-Conscious Computations. *Journal of Cryptology* (2023). 4

[27] Nishat Koti, Arpita Patra, Rahul Rachuri, and Ajith Suresh. 2022. Tetrad: Actively Secure 4PC for Secure Training and Inference. In *NDSS*. 1, 2, 3, 4, 6, 7, 9, 10, 11, 17, 23

[28] Yehuda Lindell. 2016. How To Simulate It - A Tutorial on the Simulation Proof Technique. Cryptology ePrint Archive, Report 2016/046. https://eprint.iacr.org/2016/046. 6, 9, 20

[29] Yehuda Lindell. 2021. Secure multiparty computation. *Commun. ACM* (2021). 1

[30] Yehuda Lindell, Benny Pinkas, Nigel P. Smart, and Avishay Yanai. 2015. Efficient Constant Round Multi-party Computation Combining BMR and SPDZ. In *CRYPTO*. 4

[31] Yibiao Lu, Bingsheng Zhang, and Kui Ren. 2023. Load-Balanced Server-Aided MPC in Heterogeneous Computing. *ePrint Archive* (2023). https://eprint.iacr.org/2023/1826 10

[32] Darius Mercadier and Pierre-Évariste Dagand. 2019. Usuba: high-throughput and constant-time ciphers, by construction. In *ACM PLDI*. 11

[33] Darius Mercadier, Pierre-Évariste Dagand, Lionel Lacassagne, and Gilles Muller. 2018. Usuba: Optimizing & Trustworthy Bitslicing Compiler. In *WPMVP@PPoPP*. 11, 13

[34] Payman Mohassel and Peter Rindal. 2018. ABY^3: A Mixed Protocol Framework for Machine Learning. In *CCS*. 1, 2, 3, 4, 9, 13, 18

[35] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *IEEE S&P*. 9, 17, 18

[36] Lucien KL Ng and Sherman SM Chow. 2023. SoK: Cryptographic Neural-Network Computation. In *IEEE S&P*. 1, 3

[37] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. 2021. ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation. In *USENIX Security*. 1, 9

[38] Arpita Patra and Ajith Suresh. 2020. BLAZE: Blazing Fast Privacy-Preserving Machine Learning. In *NDSS*. 1, 6, 18

[39] Berry Schoenmakers. 2018. MPyC—Python package for secure multiparty computation. In *TPMPC*. https://github.com/lschoe/mpyc. 2, 3, 4

[40] Shreya Sharma, Chaoping Xing, and Yang Liu. 2019. Privacy-Preserving Deep Learning with SPDZ. In *PPAI@AAAI*. 1

[41] Ajith Suresh. 2021. *MPCLeague: Robust MPC Platform for Privacy-Preserving Machine Learning*. Ph. D. Dissertation. Indian Institute of Science (IISc), Bangalore. https://arxiv.org/abs/2112.13338. 9

[42] Berkeley University of California. 2024. MPC Deployments. https://mpc.cs.berkeley.edu/ Accessed: 2024-06-04. 1

[43] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. 2017. Authenticated Garbling and Efficient Maliciously Secure Two-Party Computation. In *ACM CCS*. 4

[44] Andrew Chi-Chih Yao. 1986. How to Generate and Exchange Secrets (Extended Abstract). In *FOCS*. 1

[45] Chuan Zhao, Shengnan Zhao, Minghao Zhao, Zhenxiang Chen, Chong-Zhi Gao, Hongwei Li, and Yu-an Tan. 2019. Secure Multi-Party Computation: Theory, practice and applications. *Inf. Sci.* 476 (2019), 357–372. 1

A CORRECTNESS

In this section, we explain the correctness of our 3PC (cf. §4) and 4PC (cf. §5) multiplication protocols by unfolding the computations, showing that each party receives a valid share of the output $c = ab$. Additionally, for our 4PC protocol, which provides malicious security with abort against an adversary $\mathcal{A}$ corrupting at most one party, we show that each message sent by $\mathcal{A}$ can be verified by a set of honest parties using Π_{CV} (cf. Figure 3), ensuring that any inconsistency will be detected.

A.1 Trio: 3PC Protocol

In our 3PC multiplication protocol Π_{Mult} (cf. Figure 4), P_0 along with P_1/P_2 locally samples their share of mask λ_c during the preprocessing by invoking Π_{SRNG} (cf. Figure 2) using the shared PRF keys. To see the correctness of P_1's computation during the online phase, note that:

Party P_1 in Π_{Mult} (cf. Figure 4) :

$$\begin{aligned}
\mathsf{m}_{c,2} &= \mathsf{M}_{\{2\}} - \mathsf{V}_{\{1\}} \\
&= \mathsf{m}_{a,1}\mathsf{m}_{b,1} + \mathsf{M}_{\{0\}} + \lambda_c^2 - (\mathsf{m}_{a,2}\lambda_b^1 + \mathsf{m}_{b,2}\lambda_a^1 + r_{0,1}) \\
&= ab + a\lambda_b^1 + b\lambda_a^1 + \lambda_a^1\lambda_b^1 + \lambda_c^2 + \mathsf{M}_{\{0\}} \\
&\quad - a\lambda_b^1 - b\lambda_a^1 - \lambda_a^1\lambda_b^2 - \lambda_a^2\lambda_b^1 - r_{0,1} \\
&= ab + \lambda_a^1\lambda_b^1 - \lambda_a^1\lambda_b^2 - \lambda_a^2\lambda_b^1 - r_{0,1} + \lambda_c^2 + \mathsf{M}_{\{0\}} \\
&= ab + \lambda_a^1\lambda_b^1 - \lambda_a^1\lambda_b^2 - \lambda_a^2\lambda_b^1 - r_{0,1} + \lambda_c^2 + \lambda_a^2\lambda_b^2 \\
&\quad - (\lambda_a^1\lambda_b^1 - \lambda_a^2\lambda_b^1 - \lambda_a^1\lambda_b^2 + \lambda_a^2\lambda_b^2) + r_{0,1} \\
&= ab + \lambda_c^2
\end{aligned}$$

Similarly, P_2 computes the following:

Party P_2 in Π_{Mult} (cf. Figure 4) :

$$\begin{aligned}
\mathsf{m}_{c,1} &= \mathsf{V}_{\{2\}} - \mathsf{M}_{\{1\}} \\
&= \mathsf{m}_{a,1}\mathsf{m}_{b,1} + \mathsf{M}_{\{0\}} - (\mathsf{m}_{a,2}\lambda_b^1 + \mathsf{m}_{b,2}\lambda_a^1 + r_{0,1} - \lambda_c^1) \\
&= ab + a\lambda_b^1 + b\lambda_a^1 + \lambda_a^1\lambda_b^1 + \lambda_c^1 + \mathsf{M}_{\{0\}} \\
&\quad - a\lambda_b^1 - b\lambda_a^1 - \lambda_a^1\lambda_b^2 - \lambda_a^2\lambda_b^1 - r_{0,1} \\
&= ab + \lambda_a^1\lambda_b^1 - \lambda_a^1\lambda_b^2 - \lambda_a^2\lambda_b^1 - r_{0,1} + \lambda_c^1 + \mathsf{M}_{\{0\}} \\
&= ab + \lambda_a^1\lambda_b^1 - \lambda_a^1\lambda_b^2 - \lambda_a^2\lambda_b^1 - r_{0,1} + \lambda_c^1 + \lambda_a^2\lambda_b^2 \\
&\quad - (\lambda_a^1\lambda_b^1 - \lambda_a^2\lambda_b^1 - \lambda_a^1\lambda_b^2 + \lambda_a^2\lambda_b^2) + r_{0,1} \\
&= ab + \lambda_c^1
\end{aligned}$$

A.2 Quad: 4PC Protocol

Similar to the 3PC protocol, parties locally sample the shares of masks λ_c and λ_c^* during the preprocessing of Π_{Mult} (cf. Figure 5). Note that P_3 has no local computation in the online phase except the execution of compare-view functionality with P_2. The computation of $\mathcal{P}_{0,1,2}$ during the online phase is as follows:

Party P_0 in Π_{Mult} (cf. Figure 5) :

$$\begin{aligned}
\mathsf{m}_c^* &= \mathsf{M}_{\{1,2\}} - (\mathsf{V}_{\{0\}} + \mathsf{M}_{\{3\}}) \\
&= \mathsf{V}_{\{1,2\}} + r_{1,2,3} - (\mathsf{m}_a^*\lambda_b + \mathsf{m}_b^*\lambda_a) - \mathsf{M}_{\{3\}} \\
&= \mathsf{m}_a\mathsf{m}_b - (\mathsf{m}_a^*\lambda_b + \mathsf{m}_b^*\lambda_a) - \mathsf{M}_{\{3\}} + r_{1,2,3} \\
&= ab + a\lambda_b + b\lambda_a + \lambda_a\lambda_b - a\lambda_b - b\lambda_a \\
&\quad - \lambda_a^*\lambda_b - \lambda_b^*\lambda_a - \mathsf{M}_{\{3\}} + r_{1,2,3} \\
&= ab + \lambda_a\lambda_b - \lambda_a^*\lambda_b - \lambda_b^*\lambda_a - \mathsf{M}_{\{3\}} + r_{1,2,3} \\
&= ab + \lambda_a\lambda_b - \lambda_a^*\lambda_b - \lambda_b^*\lambda_a - \lambda_a(\lambda_b - \lambda_b^*) + \lambda_b\lambda_a^* + \lambda_c^* \\
&= ab + \lambda_c^*
\end{aligned}$$

Party P_1 in Π_{Mult} (cf. Figure 5) :

$$\begin{aligned}
\mathsf{m}_c &= \mathsf{V}_{\{1\}} - \mathsf{M}_{\{2\}} = \mathsf{m}_a\mathsf{m}_b - \mathsf{M}_{\{1\}} - \mathsf{M}_{\{2\}} \\
&= \mathsf{m}_a\mathsf{m}_b - (\mathsf{m}_a\lambda_b^1 + \mathsf{m}_b\lambda_a^1 + r_{0,1,3}) \\
&\quad - (\mathsf{m}_a\lambda_b^2 + \mathsf{m}_b\lambda_a^2 - \mathsf{M}_{\{0,3\}}) \\
&= \mathsf{m}_a\mathsf{m}_b - (\mathsf{m}_a\lambda_b + \mathsf{m}_b\lambda_a + r_{0,1,3} - \mathsf{M}_{\{0,3\}}) \\
&= ab + a\lambda_b + b\lambda_a + \lambda_a\lambda_b - a\lambda_b - b\lambda_a \\
&\quad - \lambda_a\lambda_b - \lambda_a\lambda_b - r_{0,1,3} + \mathsf{M}_{\{0,3\}} \\
&= ab - \lambda_a\lambda_b - r_{0,1,3} + \mathsf{M}_{\{0,3\}} = ab + \lambda_c
\end{aligned}$$

Party P_2 in Π_{Mult} (cf. Figure 5) :

$$\mathsf{m}_c = \mathsf{V}_{\{2\}} - \mathsf{M}_{\{1\}} = \mathsf{m}_a\mathsf{m}_b - \mathsf{M}_{\{1\}} - \mathsf{M}_{\{2\}} = ab + \lambda_c$$

Correctness of messages. While the equations above showed that the parties correctly compute their shares, the correctness is conditioned on obtaining the correct messages during the protocol execution. To show the correctness of the messages communicated during the protocol (denoted by $\mathsf{M}_{\{\cdot\}}$ in Π_{Mult}), we reduce each such message to an instance of compare-view functionality instantiated using Π_{CV} protocol. We use $\mathsf{M}^{\mathsf{e}}_{\{\cdot\}}$ to denote a potentially corrupted message with an error e.

We consider the cases of adversary $\mathcal{A}$ corrupting each parties separately and the reductions are shown below:

Case: $\mathcal{A} = P_0$

Corrupted message:	$\mathsf{M}^{\mathsf{e}}_{\{0,3\}} = \mathsf{M}_{\{0,3\}} + \mathsf{e}$
Reduction:	$\Pi_{\mathsf{CV}}(\mathcal{P}^V_{2,3}, \mathsf{M}_{\{0,3\}})$

Case: $\mathcal{A} = P_1$

Corrupted message:	$\mathsf{M}^{\mathsf{e}}_{\{1\}} = \mathsf{M}_{\{1\}} + \mathsf{e}$
Reduction:	$\Pi_{\mathsf{CV}}(\mathcal{P}^V_{0,1,2}, \overline{\mathsf{m}}_c)$

If $e \neq 0$, P_2 obtains $\overline{\mathsf{m}}_c - \mathsf{e}$ and $\Pi_{\mathsf{CV}}(\mathcal{P}^V_{0,1,2}, \overline{\mathsf{m}}_c)$ fails.

Case: $\mathcal{A} = P_2$

Corrupted message:	$\mathsf{M}^{\mathsf{e}_1}_{\{2\}} = \mathsf{M}_{\{2\}} + \mathsf{e}_1$, $\mathsf{M}^{\mathsf{e}_2}_{\{1,2\}} = \mathsf{M}_{\{1,2\}} + \mathsf{e}_2$
Reduction:	$\Pi_{\mathsf{CV}}(\mathcal{P}^V_{0,1,2}, \overline{\mathsf{m}}_c)$, $\Pi_{\mathsf{CV}}(\mathcal{P}^V_{0,1}, \mathsf{M}_{\{1,2\}})$

If $\mathsf{e}_2 \neq 0$, $\Pi_{\mathsf{CV}}(\mathsf{M}_{\{1,2\}}, \mathcal{P}^V_{0,1})$ fails. If $\mathsf{e}_2 = 0$ but $\mathsf{e}_1 \neq 0$, P_1 obtains $\overline{\mathsf{m}}_c - \mathsf{e}_1$ and $\Pi_{\mathsf{CV}}(\overline{\mathsf{m}}_c, \mathcal{P}^V_{0,1,2})$ fails. Hence, any errors $\mathsf{e}_1 \neq 0 \vee \mathsf{e}_2 \neq 0$ leads to `abort`.

Case: $\mathcal{A} = P_3$

Corrupted message: $\mathsf{M}^{\mathsf{e}}_{\{3\}} = \mathsf{M}_{\{3\}} + \mathsf{e}$
Reduction: $\Pi_{\mathsf{CV}}(\mathcal{P}^V_{0,1,2}, \overline{\mathsf{m}}_c)$

If $e \neq 0$, P_0 obtains $\overline{\mathsf{m}}_c - \mathsf{e}$ and $\Pi_{\mathsf{CV}}(\mathcal{P}^V_{0,1,2}, \overline{\mathsf{m}}_c)$ fails.

A.3 Heterogeneous 4PC Protocol

Similar to analysis of Π_{Mult} in §A.2, parties locally sample the shares of λ_c and λ^*_c during the preprocessing of $\Pi_{\mathsf{Mult\_H}}$ (cf. Figure 6). Also, P_3 has no local computation in the online phase except invocations of Π_{CV} with P_0 and P_2. The computation of $\mathcal{P}_{0,1,2}$ during the online phase is as follows:

$$P_0 : \mathsf{m}^*_c = \mathsf{M}_{\{1,2\},1} - \lambda_c = \mathsf{m}_c + \lambda^*_c - \lambda_c = ab + \lambda^*_c$$
$$P_1 : \mathsf{m}_c = \mathsf{V}_{\{1,2\}} - \mathsf{M}_{\{2\}} = \mathsf{m}_a\mathsf{m}_b - \mathsf{M}_{\{1\}} - \mathsf{M}_{\{2\}} = ab + \lambda_c$$
$$P_2 : \mathsf{m}_c = \mathsf{V}_{\{1\}} - \mathsf{M}_{\{1\}} = \mathsf{m}_a\mathsf{m}_b - \mathsf{M}_{\{1\}} - \mathsf{M}_{\{2\}} = ab + \lambda_c$$

Correctness of messages. To ensure that all corrupted messages $\mathsf{M}^{\mathsf{e}}_{\{\cdot\}}$ are caught during verification, we reduce the correctness of messages to an instance of Π_{CV}, similar to Π_{Mult} in §A.2.

Case: $\mathcal{A} = P_0$

Corrupted message: $\mathsf{M}^{\mathsf{e}}_{\{0,3\}} = \mathsf{M}_{\{0,3\}} + \mathsf{e}$
Reduction: $\Pi_{\mathsf{CV}}(\mathcal{P}^V_{2,3}, \mathsf{M}_{\{0,3\}})$

Case: $\mathcal{A} = P_1$

Corrupted message: $\mathsf{M}^{\mathsf{e}}_{\{1\}} = \mathsf{M}_{\{1\}} + \mathsf{e}$
Reduction: $\Pi_{\mathsf{CV}}(\mathcal{P}^V_{0,3}, \mathsf{V}_{\{0,3\}})$

If $\mathsf{e} \neq 0$, P_2 obtains $\mathsf{M}^{\mathsf{e}}_{\{1,2\},2} = \mathsf{M}_{\{1,2\},2} + \mathsf{e}$ and sends it to P_0, who computes $\mathsf{V}^{\mathsf{e}}_{\{0,3\}}$ based on $\mathsf{M}^{\mathsf{e}}_{\{1,2\},2}$. The following equation shows that $\Pi_{\mathsf{CV}}(\mathcal{P}^V_{0,3}, \mathsf{V}_{\{0,3\}})$ fails in this case.

$$\begin{aligned}\mathsf{V}^{\mathsf{e}}_{\{0,3\}} &= \mathsf{M}^{\mathsf{e}}_{\{1,2\},2} - \mathsf{V}_{\{0\}} \\ &= a\lambda_b + b\lambda_a + \lambda_a\lambda_b - \lambda_c + r_{1,2,3} + \mathsf{e} - \mathsf{m}^*_a\lambda_b - \mathsf{m}^*_b\lambda_a \\ &= \lambda_a\lambda_b - \lambda^*_a\lambda_b - \lambda^*_b\lambda_a + r_{1,2,3} - \lambda_c + \mathsf{e} = \mathsf{V}_{\{0,3\}} + \mathsf{e}\end{aligned}$$

Case: $\mathcal{A} = P_2$

Corrupted message: $\mathsf{M}^{\mathsf{e}_1}_{\{2\}} = \mathsf{M}_{\{2\}} + \mathsf{e}_1$
$\mathsf{M}^{\mathsf{e}_2}_{\{1,2\},1} = \mathsf{M}_{\{1,2\},1} + \mathsf{e}_2$
$\mathsf{M}^{\mathsf{e}_3}_{\{1,2\},2} = \mathsf{M}_{\{1,2\},2} + \mathsf{e}_3$
Reduction: $\Pi_{\mathsf{CV}}(\mathcal{P}^V_{0,3}, \mathsf{V}_{\{0,3\}})$
$\Pi_{\mathsf{CV}}(\mathcal{P}^V_{0,1}, \mathsf{M}_{\{1,2\},1})$, $\Pi_{\mathsf{CV}}(\mathcal{P}^V_{0,1}, \mathsf{M}_{\{1,2\},2})$

If $\mathsf{e}_1 \neq 0$, honest P_1 obtains $\mathsf{M}_{\{1,2\},1} - \mathsf{e}_1$ and $\mathsf{M}_{\{1,2\},2} - \mathsf{e}_1$. In that case, any assignment other than $\mathsf{e}_1 = \mathsf{e}_2 = \mathsf{e}_3$ leads to abort due to $\Pi_{\mathsf{CV}}(\mathcal{P}^V_{0,1}, \mathsf{M}_{\{1,2\},1})$ or $\Pi_{\mathsf{CV}}(\mathcal{P}^V_{0,1}, \mathsf{M}_{\{1,2\},2})$. Assigning $\mathsf{e}_1 = \mathsf{e}_2 = \mathsf{e}_3 \neq 0$ leads to P_0 obtaining a corrupted $\mathsf{V}_{\{0,3\}}$. Hence, $\Pi_{\mathsf{CV}}(\mathcal{P}^V_{0,3}, \mathsf{V}_{\{0,3\}})$ fails.
If $\mathsf{e}_1 = 0$, P_1 obtains $\mathsf{M}_{\{1,2\},1}$ and $\mathsf{M}_{\{1,2\},2}$. Then, $\mathsf{e}_2 \neq 0$ or $\mathsf{e}_3 \neq 0$ leads to abort due to $\Pi_{\mathsf{CV}}(\mathcal{P}^V_{0,1}, \mathsf{M}_{\{1,2\},1})$ or $\Pi_{\mathsf{CV}}(\mathcal{P}^V_{0,1}, \mathsf{M}_{\{1,2\},2})$.

Case: $\mathcal{A} = P_3$

P_3 does not communicate any message of the form $\mathsf{M}^{\mathsf{e}}_{\{\cdot\}}$.

B ADDITIONAL DETAILS

This section provides the formal details for the protocols discussed in §5 and §6. We begin with the protocols for input sharing and output reconstruction in our 4PC protocol Quad. Next, we show how to extend the properties of our protocols—such as high tolerance to weak network links, low computational complexity, and best-known communication complexity—to other protocols like multiplication with truncation, comparison, and bit conversions, as discussed in §6.

B.1 Quad: Handling Inputs and Outputs in 4PC

Protocol Π_{Sh} in Figure 12 provides the details for input sharing in our 4PC protocol Quad, where P_I holds the secret input $x \in \mathbb{Z}_{2^\ell}$ to be shared among $\mathcal{P}_{0,1,2,3}$.

Protocol $\Pi_{\mathsf{Sh}}(\mathcal{P}_I, x) \rightarrow [\![a]\!]$

Setup:
(1) Invoke $\mathcal{F}_{\mathsf{setup}}$ to obtain the following PRF keys:
$\mathcal{P}_{I,0,1,3} : \mathsf{k}_{I,0,1,3}$ $\mathcal{P}_{I,0,2,3} : \mathsf{k}_{I,0,2,3}$ $\mathcal{P}_{I,1,2,3} : \mathsf{k}_{I,1,2,3}$

Preprocessing:
(1) Sample random values using Π_{SRNG} (Figure 2):
$\mathcal{P}^V_{I,1,2,3} : \lambda^*_x$ $\mathcal{P}_{I,0,1,3} : \lambda^1_x$ $\mathcal{P}_{I,0,2,3} : \lambda^2_x$
(2) Locally compute:
$\mathcal{P}^V_{I,0,3} : \lambda_x = \lambda^1_x + \lambda^2_x$

Online:
(1) Communicate:
$\mathcal{P}_I \rightarrow \mathcal{P}_{0,1,2} : \overline{\mathsf{m}}_x = x + \lambda_x + \lambda^*_x$
(2) Compare views using Π_{CV} (Figure 3):
$\mathcal{P}^V_{0,1,2} : \overline{\mathsf{m}}_x$
(3) Locally compute:
$P^V_0 : \mathsf{m}^*_x = \overline{\mathsf{m}}_x - \lambda_x$ $\mathcal{P}_{1,2} : \mathsf{m}_x = \overline{\mathsf{m}}_x - \lambda^*_x$

$\mathcal{P}_I$ denotes the input party holding secret $x \in \mathbb{Z}_{2^\ell}$.

Figure 12: Quad: 4PC Secret Sharing

Protocol Π_{Rec} in Figure 13 allows the parties in $\mathcal{P}_{0,1,2,3}$ to reconstruct a shared secret $[\![x]\!]$ towards a designated party P_O. This protocol trivially extends to a public reconstruction among $\mathcal{P}_{0,1,2,3}$, where each party in $\mathcal{P}_{0,1,2,3}$ is assigned the role of P_O.

Protocol $\Pi_{\mathsf{Rec}}(\mathcal{P}_O, [\![x]\!]) \rightarrow x$

Postprocessing:
(1) Communicate:
$P_2 \rightarrow P_O : \mathsf{m}_x$ $P_0 \rightarrow P_O : \lambda_x$
(2) Locally compute:
$P_O : x = \mathsf{m}_x - \lambda_x$
(3) Compare views using Π_{CV} (Figure 3):
$\mathcal{P}^V_{O,1} : \mathsf{m}_x$ $\mathcal{P}^V_{O,3} : \lambda_x$

P_O denotes the output party to obtain the secret x.

Figure 13: Quad: 4PC Reconstruction

Table 9: Communication complexity of additional protocols

Protocol	Scheme	Preprocessing	Online	Rounds
Multiplication + Truncation	3PC	1	2	1
	4PC	2	3	1
Arithmetic to Boolean (Setup) + Ripple Carry Adder[a]	3PC	$1 + (\ell - 1)$	$0 + (2\ell - 1)$	$0 + (\ell - 1)$
	4PC	$1 + 2(\ell - 1)$	$1 + (3\ell - 1)$	$0 + (\ell - 1)$
Arithmetic to Boolean (Setup) + Parallel Prefix Adder[a]	3PC	$1 + O(\ell \cdot \log_2 \ell)$	$0 + 2 \cdot O(\ell \cdot \log_2 \ell)$	$0 + (\log_2 \ell + 1)$
	4PC	$1 + 2 \cdot O(\ell \cdot \log_2 \ell)$	$1 + 3 \cdot O(\ell \cdot \log_2 \ell)$	$0 + (\log_2 \ell + 1)$
Bit to Arithmetic (Setup) + Arithmetic XOR[a]	3PC	$1 + 1$	$0 + 2$	$0 + 1$
	4PC	$1 + 2$	$1 + 2$	$0 + 1$

[a] Setup protocols (c.f. §B) + High-level circuit implemented with elementary gates.

In the reconstruction protocol Π_{Rec} (cf. Figure 13) discussed above, an adversary $\mathcal{A}$ can selectively abort for some P_O by sending incorrect messages during Π_{CV} within Π_{Rec}. To ensure fairness during a public reconstruction, we provide a fair reconstruction variant Π_{fairRec} in Figure 14, similar to Tetrad [27].

In Π_{fairRec}, we let the parties $\mathcal{P}_{0,1,2,3}$ store the actual secret-shares for all the output wires, instead of storing only the persistent shares (cf. Table 5). This modification provides the advantage that each of the following values—$\overline{\mathsf{m}}_x$, λ_x^*, λ_x^1, λ_x^2—will be held by three out of the four parties. Given that there is at most one corruption, this guarantees a honest majority of each of these values among the parties and we leverage the same.

Similar to Tetrad, each party maintains an *aliveness* bit indicating whether some of the verification prior to reconstruction failed or not. Parties mutually exchange this bit and accepts the majority. If the agreement is continue, parties proceed with sending the values to P_O who computes output as $x = \overline{\mathsf{m}}_x - \lambda_x^* - \lambda_x^1 - \lambda_x^2$.

Protocol $\Pi_{\text{fairRec}}(\mathcal{P}_O, [\![x]\!]) \to x$

Postprocessing:

(1) $\mathcal{P}_{0,1,2,3}$ sets their aliveness bit b = 0, if any of their verification during Π_{CV} (Figure 3) fails. Else, set b = 1.
(2) $\mathcal{P}_{0,1,2,3}$ mutually exchanges their bit b and continue, if there is a majority with b = 1. Else, abort.
(3) Communicate:

$$\mathcal{P}_{0,1,2} \to P_O : \overline{\mathsf{m}}_x \qquad \mathcal{P}_{1,2,3} \to P_O : \lambda_x^*$$
$$\mathcal{P}_{0,1,3} \to P_O : \lambda_x^1 \qquad \mathcal{P}_{0,2,3} \to P_O : \lambda_x^2$$

(4) Locally compute using the majority values:

$$P_O : x = \overline{\mathsf{m}}_x - \lambda_x^* - \lambda_x^1 - \lambda_x^2$$

P_O denotes the output party to obtain the secret x.

Figure 14: Quad: 4PC Fair Reconstruction

B.2 Other Protocols

This section covers the formal details for the protocols discussed in §6. The costs for additional protocols are shown in table 9.

B.2.1 Truncation. Let $(x)^t$ denote $\lfloor \frac{x}{2^t} \rfloor$. Let $(\mathsf{m}_x)^{pt}$ denote probabilistic truncation as defined by [35], To truncate-multiply two values a and b in our schemes scheme, $\mathcal{P}$ need to obtain a share of $(\mathsf{m}_c)^t = (ab + \lambda_c)^t$ and $(\lambda_c)^t$ for a random mask λ_c. $(\mathsf{m}_c)^t$ and $(\lambda_c)^t$ then form a truncation pair where $(\mathsf{m}_c)^t - (\lambda_c)^t = (c)^{pt} \approx (c)^t$ with high probability.

3PC. In the following explanation, we use three sequential steps to explain the intuition of our 3PC truncation protocol shown in w 15: First, P_0 calculates an input-independent error, similar to our general multiplication protocol. P_0 then locally truncates the input-dependent error to obtain $(\lambda_c)^t$. Then, P_0 shares $(\lambda_c)^t$ among $\mathcal{P}$. Second, $\mathcal{P}_{1,2}$ exchange messages to obtain $ab + \lambda_c$ in the clear. They proceed to locally truncate $ab + \lambda_x$. To share $(ab + \lambda_c)^t$ among $\mathcal{P}$, all parties simply set their input-independent shares to 0. As the parties now hold both shares, they perform the final step by subtracting their local shares to obtain $[\![(ab)^{pt}]\!] = [\![(ab + \lambda_c)^t]\!] - [\![(\lambda_c)^t]\!]$. While this three-step procedure explains the intuition of our formal protocol in Figure 15, the actual protocol does not create the two shares explicitly but fuses them into a single one to achieve a more efficient construction.

Protocol $\Pi_{\text{Mult3T}}([\![a]\!], [\![b]\!]) \to [\![(c)^{pt}]\!]$

Preprocessing:

(1) Sample random values using Π_{SRNG}:

$$P_{0,1} : r_{0,1}, \lambda_c^1 \qquad P_{0,2} : r_{0,2}$$

(2) Locally compute:

$$P_0 : \lambda_c^2 = \mathsf{M}_{\{0\}} = \left((\lambda_a^1 - \lambda_a^2)(\lambda_b^1 - \lambda_b^2) - \lambda_a^2 \lambda_b^2 + r_{0,1} + r_{0,2}\right)^t - \lambda_c^1$$

(3) Communicate:

$$P_0 \to P_2 : \mathsf{M}_{\{0\}}$$

Online:

(1) Locally compute:

$$P_1 : \mathsf{M}_{\{1\}} = \mathsf{m}_{a,2}\lambda_b^1 + \mathsf{m}_{b,2}\lambda_a^1 - r_{0,1} \qquad P_2 : \mathsf{M}_{\{2\}} = \mathsf{m}_{a,1}\mathsf{m}_{b,1} + r_{0,2}$$

(2) Communicate:

$$P_1 \to P_2 : \mathsf{M}_{\{1\}} \qquad P_2 \to P_1 : \mathsf{M}_{\{2\}}$$

(3) Locally compute:

$$P_1 : \mathsf{m}_{c,2} = (\mathsf{M}_{\{2\}} - \mathsf{M}_{\{1\}})^t - \lambda_c^1 \qquad P_2 : \mathsf{m}_{c,1} = (\mathsf{M}_{\{2\}} - \mathsf{M}_{\{1\}})^t - \mathsf{M}_{\{0\}}$$
$$P_2 : \lambda_c^2 = \mathsf{M}_{\{0\}}$$

Figure 15: Trio: 3PC multiplication with truncation

Nevertheless, one can see that step 2-3 in the preprocessing phase are mainly responsible for sharing the locally-truncated input-independent error among $\mathcal{P}_{1,2}$. Steps 1 and 2 in the online phase are responsible for letting $\mathcal{P}_{1,2}$ obtain $ab + \lambda_c$ in the clear. Finally, in step

3, $\mathcal{P}_{1,2}$ parties locally truncate $ab + \lambda_c$ and perform the subtraction using their fused shares.

4PC. Truncation also comes for free in our 4PC schemes. To truncate-multiply a value in our 4PC schemes, P_0 shares the locally truncated $(\lambda_c)^t$ with P_1 and P_2 similarly to our 3PC protocol. P_3 verifies this message.

As in our 3PC protocol, P_1 and P_2 obtain $ab + \lambda_c$ in the clear. To do so, they exchange $\mathsf{M}_{\{1\}} = \mathsf{m}_a\lambda_b^1 + \mathsf{m}_b\lambda_a^1 - r_{0,1,3}$ and $\mathsf{M}_{\{1,2\}} = \mathsf{m}_a\lambda_b^2 + \mathsf{m}_b\lambda_a^2 - r_{0,2,3}$. They locally compute $\mathsf{m}_a\mathsf{m}_b - \mathsf{M}_{\{1\}} - \mathsf{M}_{\{1,2\}}$ to obtain $\mathsf{m}_c = ab - \lambda_a\lambda_b + r_{0,1,3} + r_{0,2,3} = ab + \lambda_c$. They then locally truncate m_c. In our 4PC protocol, P_0 also needs to obtain an input-dependent share for verification purposes.Thus, $\mathcal{P}_2$ sends a masked message $\mathsf{M}_{\{1,2\}}$ ($\mathsf{M}_{\{1,2\},1}$ in the heterogeneous variant) to P_0. Observe that $\mathsf{M}_{\{1,2\}}$ already contains both truncation pairs.

The verification of exchanged messages is handled analogously to our general multiplication protocol. Figures 16 and 17 implement the presented intuition in a computationally efficient way.

Protocol $\Pi_{\mathsf{Mult}4T}(\llbracket \mathrm{a} \rrbracket, \llbracket \mathrm{b} \rrbracket) \to \llbracket (c)^{pt} \rrbracket$

Preprocessing:

(1) Sample random values using Π_{SRNG}:

$$P_{0,1,3} : r_{0,1,3}, \lambda_c^1 \qquad P_{0,2,3} : r_{0,2,3} \qquad P_{1,2,3}^V : r_{1,2,3}, \lambda_c^*$$

(2) Locally compute:

$$P_0, P_3^V : \lambda_c = \left(r_{0,1,3} + r_{0,2,3} - \lambda_a\lambda_b\right)^t$$
$$P_0, P_3^V : \mathsf{M}_{\{0,3\}} = \lambda_c - \lambda_c^1$$
$$P_3^V : \mathsf{M}_{\{3\}} = \lambda_a(\lambda_b - \lambda_b^*) - \lambda_b\lambda_a^* - r_{0,1,2} - r_{0,2,3} + r_{1,2,3}$$

(3) Communicate:

$$P_0 \to P_2 : \mathsf{M}_{\{0,3\}}$$
$$P_3^V \to P_0^V : \mathsf{M}_{\{3\}}$$

Online:

(1) Locally compute:

$$P_1 : \mathsf{M}_{\{1\}} = \mathsf{m}_a\lambda_b^1 + \mathsf{m}_b\lambda_a^1 - r_{0,1,3} \qquad \mathcal{P}_{1,2} : \mathsf{V}_{\{1,2\},1} = \mathsf{m}_a\mathsf{m}_b$$
$$P_2 : \mathsf{M}_{\{2\}} = \mathsf{m}_a\lambda_b^2 + \mathsf{m}_b\lambda_a^2 - r_{0,2,3} \qquad P_2 : \lambda_c^2 = \mathsf{M}_{\{0,3\}}$$
$$P_0^V : \mathsf{V}_{\{0,1,2\}} = \mathsf{m}_a^*\lambda_b + \mathsf{m}_b^*\lambda_a + \lambda_a\lambda_b + \mathsf{M}_{\{3\}}$$

(2) Communicate:

$$P_1 \to P_2 : \mathsf{M}_{\{1\}} \qquad P_2 \to P_1 : \mathsf{M}_{\{2\}}$$

(3) Locally compute:

$$\mathcal{P}_{1,2} : \mathsf{V}_{\{1,2\},2} = \mathsf{M}_{\{1\}} + \mathsf{M}_{\{2\}} \qquad \mathcal{P}_{1,2} : \mathsf{m}_c = \left(\mathsf{V}_{\{1,2\},1} - \mathsf{V}_{\{1,2\},2}\right)^t$$
$$\mathcal{P}_{1,2}^V : \mathsf{M}_{\{1,2\}} = \mathsf{m}_c + \lambda_c^* \qquad \mathcal{P}_{1,2}^V : \mathsf{V}_{\{0,1,2\}} = \mathsf{V}_{\{1,2\},2} + r_{1,2,3}$$

(4) Communicate:

$$P_2^V \to P_0^V : \mathsf{M}_{\{1,2\}}$$

(5) Locally compute:

$$P_0^V : \mathsf{m}_c^* = \mathsf{M}_{\{1,2\}} - \lambda_c$$

(6) Compare views using Π_{CV}:

$$\mathcal{P}_{0,1}^V : \mathsf{M}_{\{1,2\}} \qquad \mathcal{P}_{0,1,2}^V : \mathsf{V}_{\{0,1,2\}} \qquad \mathcal{P}_{2,3}^V : \mathsf{M}_{\{0,3\}}$$

Figure 16: Quad: 4PC multiplication with truncation

Protocol $\Pi_{\mathsf{Mult}4HT}(\llbracket \mathrm{a} \rrbracket, \llbracket \mathrm{b} \rrbracket) \to \llbracket (c)^{pt} \rrbracket$

Preprocessing:

(1) Sample random values using Π_{SRNG}:

$$P_{0,1,3} : r_{0,1,3}, \lambda_c^1 \qquad P_{0,2,3} : r_{0,2,3} \qquad P_{1,2,3}^V : r_{1,2,3}, \lambda_c^*$$

(2) Locally compute:

$$P_0, P_3^V : \lambda_c = \left(r_{0,1,3} + r_{0,2,3} - \lambda_a\lambda_b\right)^t$$
$$P_0, P_3^V : \mathsf{M}_{\{0,3\}} = \lambda_c - \lambda_c^1$$
$$P_3^V : \mathsf{V}_{\{0,3\}} = \lambda_a(\lambda_b - \lambda_b^*) - \lambda_b\lambda_a^* - r_{0,1,2} - r_{0,2,3} + r_{1,2,3}$$

(3) Communicate:

$$P_0 \to P_2 : \mathsf{M}_{\{0,3\}}$$

Online:

(1) Locally compute:

$$P_1 : \mathsf{M}_{\{1\}} = \mathsf{m}_a\lambda_b^1 + \mathsf{m}_b\lambda_a^1 - r_{0,1,3} \qquad P_2 : \mathsf{M}_{\{2\}} = \mathsf{m}_a\lambda_b^2 + \mathsf{m}_b\lambda_a^2 - r_{0,2,3}$$
$$P_1 : \mathsf{V}_{\{1\}} = \mathsf{m}_a\mathsf{m}_b - \mathsf{M}_{\{1\}} \qquad P_2 : \mathsf{V}_{\{2\}} = \mathsf{m}_a\mathsf{m}_b - \mathsf{M}_{\{1,2\}}$$
$$P_0^V : \mathsf{V}_{\{0\}} = \mathsf{m}_a^*\lambda_b + \mathsf{m}_b^*\lambda_a + \lambda_a\lambda_b \qquad P_2 : \lambda_c^2 = \mathsf{M}_{\{0,3\}}$$

(2) Communicate:

$$P_1 \to P_2 : \mathsf{M}_{\{1\}} \qquad P_2 \to P_1 : \mathsf{M}_{\{2\}}$$

(3) Locally compute:

$$P_1 : \mathsf{m}_c = \left(\mathsf{V}_{\{1\}} - \mathsf{M}_{\{2\}}\right)^t \qquad P_2 : \mathsf{m}_c = \left(\mathsf{V}_{\{2\}} - \mathsf{M}_{\{1\}}\right)^t$$
$$\mathcal{P}_{1,2}^V : \mathsf{M}_{\{1,2\},1} = \mathsf{m}_c + \lambda_c^* \qquad \mathcal{P}_{1,2}^V : \mathsf{M}_{\{1,2\},2} = \mathsf{M}_{\{1\}} + \mathsf{M}_{\{2\}} + r_{1,2,3}$$

(4) Communicate:

$$P_2^V \to P_0^V : \mathsf{M}_{\{1,2\},1} \qquad P_2^V \to P_0^V : \mathsf{M}_{\{1,2\},2}$$

(5) Locally compute:

$$P_0^V : \mathsf{V}_{\{0,3\}} = \mathsf{M}_{\{1,2\},2} - \mathsf{V}_{\{0\}} \qquad P_0^V : \mathsf{m}_c^* = \mathsf{M}_{\{1,2\},1} - \lambda_c$$

(6) Compare views using Π_{CV}:

$$\mathcal{P}_{0,1}^V : \mathsf{M}_{\{1,2\},1}, \mathsf{M}_{\{1,2\},2} \qquad \mathcal{P}_{0,3}^V : \mathsf{V}_{\{0,3\}} \qquad \mathcal{P}_{2,3}^V : \mathsf{M}_{\{0,3\}}$$

Figure 17: Quad: 4PC heterogeneous multiplication protocol with truncation

B.2.2 Matrix Multiplication. As observed in several works [9, 34, 35, 38], a matrix multiplication $C = A \cdot B$ can be trivially computed with an existing MPC multiplication protocol that operates correctly on single values. Suppose the parties hold $\llbracket A \rrbracket$ and $\llbracket B \rrbracket$. In that case, they execute the protocol regularly but compute a local matrix addition for each addition operation and a local matrix multiplication for each multiplication operation. Let $|C|$ be the total number of elements in the output matrix C. For each call to Π_{SRNG} the parties need to sample $|C|$ elements to ensure unique masks for every value. Every message in the protocol also becomes of size $|C|$. Since scalar products have an output size of 1, the cost of a scalar multiplication is the same as the cost of a regular multiplication.

B.2.3 Comparisons. To evaluate a comparison $\llbracket a \rrbracket > \llbracket b \rrbracket$ we use the following established sequence proposed by [34]. First, the parties calculate $\llbracket z \rrbracket = \llbracket b \rrbracket - \llbracket a \rrbracket$. Note that the sign bit of c is 1 if $a > b$, and 0 otherwise. By converting $\llbracket c \rrbracket$ into a boolean share, the parties can extract a share of its sign bit. Afterward, the parties can convert the share of the sign bit back to an arithmetic share to use the result of the comparison.

Both conversions require a setup phase followed by a high-level circuit that can be implemented using multiplication and addition

gates in Z_2 or Z_ℓ. The setup phases convert a single share from either the boolean or the arithmetic domain to two separate shares in the target computation domain. These two separate shares then get combined into a single share of the target domain with the respective high-level circuit. Following this procedure, each setup phase requires an additional primitive. In both our 3PC and 4PC schemes, the setup phases for switching computation domains are part of non-latency critical communication and thus do not add to the round complexity.

B.2.4 Arithmetic to Boolean. To convert an arithmetic share $[\![a]\!]^A$ to a boolean share $[\![a]\!]^B$, the parties compute boolean shares of $b = [\![a + \lambda_a]\!]^B$ and $c = [\![-\lambda_a]\!]^B$. The parties then use a boolean adder to compute $[\![b]\!]^B + [\![c]\!]^B = [\![a + \lambda_a]\!]^B + [\![-\lambda_a]^B]\!]$ to receive an XOR-sharing of $[\![a]\!]^B$. If the parties only wish to obtain a single bit of $[\![a]\!]$, they can use a carry-out adder instead of a boolean adder. Figures 18 and 19 show the formal protocols for our 3PC and 4PC protocols respectively.

3PC. To compute a share of $b^B = [\![-\lambda_a]\!]^B$, $\mathcal{P}_{0,1}$ sample $r_{0,1}$ and P_0 sends $\mathsf{M}_{\{0\}} = [\![-\lambda_a]\!]^B \oplus r_{0,1}$ to P_2 in the preprocessing phase. The parties then define their shares as shown in figure 18. In the online phase, each party locally computes a share of $b = [\![a + \lambda_a]\!]^B$. The parties proceed to compute $[\![a]\!]^B = [\![a + \lambda_a]\!]^B + [\![-\lambda_a]\!]^B$ using a Boolean adder. To verify the correctness, observe that $[\![-\lambda_a]\!] = \mathsf{m}_{c,1} \oplus \lambda_c^1$ and $[\![-\lambda_a]\!] = \mathsf{m}_{c,2} \oplus \lambda_c^2$. Likewise, $[\![a + \lambda_a]\!] = \mathsf{m}_{b,1} \oplus \lambda_b^1$ and $[\![a + \lambda_a]\!] = \mathsf{m}_{b,2} \oplus \lambda_b^2$. Hence, the parties successfully created two boolean shares $[\![b]\!]$ and $[\![c]\!]$ such that $[\![a]\!]^B = [\![b]\!]^B + [\![c]\!]^B$.

Protocol $\Pi_{\text{A2B},3PC}([\![\mathrm{a}]\!]^{\mathrm{A}}) \rightarrow [\![a]\!]^B$

Preprocessing:

(1) Sample random values using Π_{SRNG}:

$$\mathcal{P}_{0,1} : r_{0,1}$$

(2) Communicate:

$$P_0 \rightarrow P_2 : \mathsf{M}_{\{0\}} = (-\lambda_a) \oplus r_{0,1}$$

(3) Locally compute:

$$\mathcal{P}_{0,1} : \lambda_b^1 = 0 \quad \mathcal{P}_{0,2} : \lambda_b^2 = 0 \quad \mathcal{P}_{0,1} : \lambda_c^1 = r_{0,1} \quad \mathcal{P}_{0,2} : \lambda_c^2 = \mathsf{M}_{\{0\}}$$

Online:

(1) Locally compute:

$$P_1 : \mathsf{m}_{b,2} = \mathsf{m}_{a,2} + \lambda_a^1 \qquad P_2 : \mathsf{m}_{b,1} = \mathsf{m}_{a,1} + \lambda_a^2$$
$$P_1 : \mathsf{m}_{c,2} = \lambda_c^1 \qquad P_2 : \mathsf{m}_{c,1} = \lambda_c^2$$

(2) Jointly compute:

$$\mathcal{P} : [\![a]\!]^B = [\![b]\!]^B + [\![c]\!]^B$$

Figure 18: Trio: 3PC Arithmetic to Binary Conversion

4PC. Each party first obtains a share of $c = [\![-\lambda_a]\!]^B$. $\mathcal{P}_{0,1,3}$ sample $r_{0,1,3}$ and P_0 sends $\mathsf{M}_{\{0,3\}} = (-\lambda_a) \oplus r_{0,1,3}$ to P_2 in the preprocessing phase. P_3 compares its view of $\mathsf{M}_{\{0,3\}}$ with P_2. The parties then define their shares of b as shown in figure 21. $\mathcal{P}_{1,2}$ locally compute a share of $b = (a + \lambda_a)^B$. P_2 sends $\mathsf{M}_{\{1,2\}} = (a + \lambda_a) \oplus r_{1,2,3}$ to P_0. P_0 and P_1 compare their views of $\mathsf{M}_{\{1,2\}}$. The parties then define their shares as shown in figure 21. The parties proceed to compute $[\![a]\!]^B = [\![b]\!]^B + [\![c]\!]^B$ using a Boolean adder. To verify the correctness, observe that $[\![-\lambda_a]\!] = \mathsf{m}_c \oplus \lambda_c = \mathsf{m}_c^* \oplus \lambda_c^*$. Likewise, $[\![a+\lambda_a]\!] = \mathsf{m}_b \oplus \lambda_b = \mathsf{m}_b^* \oplus \lambda_b^*$. Hence, the parties successfully created two boolean shares $[\![b]\!]$ and $[\![c]\!]$ such that $[\![a]\!]^B = [\![b]\!]^B + [\![c]\!]^B$.

Protocol $\Pi_{\text{A2B},4PC}([\![\mathrm{a}]\!]^{\mathrm{A}}) \rightarrow [\![a]\!]^B$

Preprocessing:

(1) Sample random values using Π_{SRNG}:

$$\mathcal{P}_{0,1,3} : r_{0,1,3} \qquad \mathcal{P}_{1,2,3}^V : r_{1,2,3}$$

(2) Locally compute:

$$P_0, P_3^V : \mathsf{M}_{\{0,3\}} = -\lambda_a \oplus r_{0,1,3}$$

(3) Communicate:

$$P_0 \rightarrow P_2 : \mathsf{M}_{\{0,3\}}$$

(4) Locally compute:

$$P_1 : \lambda_b^1 = 0 \qquad P_2 : \lambda_b^2 = 0 \qquad P_0, P_3^V : \lambda_b = 0$$
$$P_1 : \lambda_c^1 = r_{0,1,3} \qquad P_2 : \lambda_c^2 = \mathsf{M}_{\{0,3\}} \qquad P_0, P_3^V : \lambda_c = (-\lambda_a)$$
$$P_3^V : \lambda_b^* = r_{1,2,3} \qquad P_3^V : \lambda_c^* = 0 \qquad \mathcal{P}_{1,2}^V : \mathsf{M}_{\{1,2\}} = \mathsf{m}_a \oplus r_{1,2,3}$$

Online:

(1) Communicate:

$$P_2^V \rightarrow P_0^V : \mathsf{M}_{\{1,2\}}$$

(2) Locally compute:

$$\mathcal{P}_{1,2} : \mathsf{m}_b = \mathsf{m}_a \quad P_0 : \mathsf{m}_b^* = \mathsf{M}_{\{1,2\}} \quad \mathcal{P}_{1,2} : \mathsf{m}_c = 0 \quad P_0 : \mathsf{m}_c^* = \lambda_c$$

(3) Compare views using Π_{CV}:

$$\mathcal{P}_{2,3}^V : \mathsf{M}_{\{0,3\}} \qquad \mathcal{P}_{0,1}^V : \mathsf{M}_{\{1,2\}}$$

(4) Jointly compute:

$$\mathcal{P} : [\![a]\!]^B = [\![b]\!]^B + [\![c]\!]^B$$

Figure 19: Quad: 4PC Arithmetic to Binary Conversion

B.2.5 Bit to Arithmetic Conversion. To promote a shared bit $[\![a^{\mathrm{b}}]\!] = \mathsf{m}_a \oplus \lambda_a$ in the boolean domain to a shared bit $[\![a^b]\!]^A = \mathsf{m}_c + \lambda_c$ in the arithmetic domain, the parties construct an *XOR*-sharing of $[\![b]\!]^A = a \oplus \lambda_a$ and $[\![c]\!]^A = \lambda_a$. This way, $[\![a]\!]^A = [\![b]\!]^A \oplus [\![c]\!]^A$ Then, they perform a private *XOR* of the resulting shares in the arithmetic domain. Note that $\mathsf{m}_a \oplus \lambda_a = \mathsf{m}_a + \lambda_a - 2\mathsf{m}_a\lambda_a$. Figures 20 and 21 show the formal protocols for our 3PC and 4PC protocols respectively.

3PC. Each party first obtains a share of $c = [\![\lambda_a]\!]^A$. $\mathcal{P}_{0,1}$ sample $r_{0,1}$ and P_0 sends $\mathsf{M}_{\{0\}} = \lambda_a + r_{0,1}$ to P_2 in the preprocessing phase. The parties then define their shares of b as shown in figure 20. All parties locally compute $[\![b]\!] = (a \oplus \lambda_a)^A$. By computing an *XOR* of $[\![b]\!]^A$ and $[\![c]\!]^A$ in the arithmetic domain, the parties obtain an arithmetic share of a. To verify the correctness, observe that $[\![\lambda_a]\!] = \mathsf{m}_{c,1} - \lambda_c^1 = \mathsf{m}_{c,2} - \lambda_c^2$. Likewise, $[\![a \oplus \lambda_a]\!] = \mathsf{m}_{b,1} - \lambda_b^1 = \mathsf{m}_{b,2} - \lambda_b^2$. Hence, the parties successfully created two arithmetic shares $[\![b]\!]^A$ and $[\![c]\!]^A$ such that $[\![a]\!]^A = [\![b]\!]^A \oplus [\![c]\!]^A$.

4PC. Each party first obtains a share of $c = [\![\lambda_a]\!]^A$. $\mathcal{P}_{0,1,3}$ sample $r_{0,1,3}$ and P_0 sends $\mathsf{M}_{\{0,3\}} = \lambda_a + r_{0,1,3}$ to P_2 in the preprocessing phase. P_3 compares its view of $\mathsf{M}_{\{0,3\}}$ with P_2. The parties then define their shares of c as shown in figure 21. $\mathcal{P}_{1,2}$ locally compute a share of $b = [a \oplus \lambda_a]^A$. P_2 sends $\mathsf{M}_{\{1,2\}} = a \oplus \lambda_a + r_{1,2,3}$ to P_0. P_0 and P_2 compare their views of $\mathsf{M}_{\{1,2\}}$. The parties then define their shares of b and c as shown in figure 21. By computing an

XOR of $[\![b]\!]^A$ and $[\![c]\!]^A$ in the arithmetic domain, the parties obtain an arithmetic share of a. To verify the correctness, observe that $[\![\lambda_a]\!] = \mathsf{m}_c - \lambda_c$ and $[\![\lambda_a]\!] = \mathsf{m}_c^* - \lambda_c^*$. Likewise, $[\![a \oplus \lambda_a]\!] = \mathsf{m}_b - \lambda_b$ and $[\![a \oplus \lambda_a]\!] = \mathsf{m}_b^* - \lambda_b^*$. Hence, the parties successfully created two arithmetic shares $[\![b]\!]$ and $[\![c]\!]$ such that $[\![a]\!]^A = [\![b]\!]^A \oplus [\![c]\!]^A$.

Protocol $\Pi_{\text{Bit2A},3PC}([\![\mathsf{a}^\mathsf{b}]\!]^B) \rightarrow [\![a^b]\!]^A$

Preprocessing:

(1) Sample random values using Π_{SRNG}:

$$\mathcal{P}_{0,1} : r_{0,1}$$

(2) Communicate:

$$P_0 \rightarrow P_2 : \mathsf{M}_{\{0\}} = \lambda_a + r_{0,1}$$

(3) Locally compute:

$$\mathcal{P}_{0,1} : \lambda_b^1 = 0 \quad \mathcal{P}_{0,2} : \lambda_b^2 = 0 \quad \mathcal{P}_{0,1} : \lambda_c^1 = r_{0,1} \quad \lambda_c^2 = -\mathsf{M}_{\{0\}}$$

Online:

(1) Locally compute:

$$P_1 : \mathsf{m}_{b,2} = \mathsf{m}_{a,2} \oplus \lambda_a^1 \qquad P_2 : \mathsf{m}_{b,1} = \mathsf{m}_{a,1} \oplus \lambda_a^2$$
$$P_1 : \mathsf{m}_{c,2} = -r_{0,1} \qquad P_2 : \mathsf{m}_{c,1} = \mathsf{M}_{\{0\}}$$

(2) Jointly compute:

$$\mathcal{P} : [\![a]\!]^A = [\![b]\!]^A \oplus [\![c]\!]^A$$

Figure 20: Trio: 3PC Bit to Arithmetic

Protocol $\Pi_{\text{Bit2A},4PC}([\![\mathsf{a}^\mathsf{b}]\!]^B) \rightarrow [\![a^b]\!]^A$

Preprocessing:

(1) Sample random values using Π_{SRNG}:

$$\mathcal{P}_{0,1,3} : r_{0,1,3} \qquad \mathcal{P}_{1,2,3}^V : r_{1,2,3}$$

(2) Locally Compute:

$$P_0, P_3^V : \mathsf{M}_{\{0,3\}} = \lambda_a + r_{0,1,3}$$

(3) Communicate:

$$P_0 \rightarrow P_2 : \mathsf{M}_{\{0,3\}}$$

(4) Locally compute:

$$P_1 : \lambda_b^1 = 0 \qquad P_2 : \lambda_b^2 = 0 \qquad P_0 : \lambda_b = 0$$
$$P_1 : \lambda_c^1 = r_{0,1,3} \qquad P_2 : \lambda_c^2 = -\mathsf{M}_{\{0,3\}} \qquad \mathcal{P}_0, P_3^V : \lambda_c = -\lambda_a$$
$$P_3^V : \lambda_b^* = r_{1,2,3} \qquad P_3^V : \lambda_c^* = 0 \qquad \mathcal{P}_{1,2}^V : \mathsf{M}_{\{1,2\}} = \mathsf{m}_a + r_{1,2,3}$$

Online:

(1) Communicate:

$$P_2^V \rightarrow P_0^V : \mathsf{M}_{\{1,2\}}$$

(2) Locally compute:

$$\mathcal{P}_{1,2} : \mathsf{m}_b = \mathsf{m}_a \quad P_0 : \mathsf{m}_b^* = \mathsf{M}_{\{1,2\}} \quad \mathcal{P}_{1,2} : \mathsf{m}_c = 0 \quad P_0 : \mathsf{m}_c^* = \lambda_a$$

(3) Compare views using Π_{CV}:

$$\mathcal{P}_{2,3}^V : \mathsf{M}_{\{0,3\}} \qquad \mathcal{P}_{0,1}^V : \mathsf{M}_{\{1,2\}}$$

(4) Jointly compute:

$$\mathcal{P} : [\![a]\!]^A = [\![b]\!]^A \oplus [\![c]\!]^A$$

Figure 21: Quad: 4PC Bit to Arithmetic

C Quad: SECURITY PROOF FOR 4PC

This section provides the security proof for our 4PC protocol Quad described in §5. We consider a circuit ckt representing a function $f(\cdot)$ to be evaluated. We establish security using the real-world/ideal-world simulation paradigm [20, 28].

The security is evaluated by comparing an adversary's capabilities in the real-world execution of the protocol with those in an ideal-world execution involving a trusted third party. In the ideal world, parties send inputs to the trusted third party via secure channels, the trusted third party performs the computation, and returns the output to the parties. A protocol is considered secure if an adversary in the real world can only do what it could also do in the ideal world.

Functionality $\mathcal{F}_{\text{abort}}^{\text{4pc}}$

$\mathcal{F}_{\text{abort}}^{\text{4pc}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{A}$. $\mathcal{F}_{\text{abort}}^{\text{4pc}}$ is parameterized by a 4-input function $f(\cdot)$, represented by a publicly known circuit ckt. Every honest party $P_i \in \mathcal{P}$ sends its input x_i to the functionality. Corrupt parties may send arbitrary inputs as instructed by $\mathcal{A}$. While sending the inputs, $\mathcal{A}$ is also allowed to send a special `abort` command.

Input: On message (Input, x_i) from P_i, do the following: if $(\text{Input}, *)$ already received from P_i, then ignore the current message. Otherwise, record $x_i' = x_i$ internally. If x_i is outside P_i's input domain, consider $x_i' = \texttt{abort}$.

Output: If there exists an $i \in \{0,1,2,3\}$ such that $x_i' = \texttt{abort}$, send $(\text{Output}, \perp)$ to $\mathcal{P}$. Otherwise, compute $y = f(x_0', x_1', x_2', x_3')$ and send (Output, y) to $\mathcal{A}$. Receive $(\text{Select}, \mathcal{P}_\Phi)$ from $\mathcal{A}$, where $\mathcal{P}_\Phi$ denotes a subset of honest parties. Send $(\text{Output}, \perp)$ to every honest party $P_i \in \mathcal{P}_\Phi$. Send (Output, y) to all the remaining parties.

Figure 22: Ideal 4PC functionality for computing $f(\cdot)$ with abort.

Let $\mathcal{A}$ be a probabilistic polynomial time (PPT) real-world adversary corrupting at most one party in $\mathcal{P}$, $\mathcal{S}$ be the corresponding ideal world adversary, and $\mathcal{F}$ be the ideal functionality. Let $\text{IDEAL}_{\mathcal{F},\mathcal{S}}(1^k, z)$ be the joint output of the honest parties and $\mathcal{S}$ from the ideal execution with security parameter κ and auxiliary input z. Similarly, $\text{REAL}_{\Pi,\mathcal{S}}(1^\kappa, z)$ be the joint output of the honest parties and $\mathcal{A}$ from the real world execution. A protocol Π securely realizes $\mathcal{F}$ if for every PPT adversary $\mathcal{A}$, there is an ideal world adversary $\mathcal{S}$ corrupting the same parties such that $\text{IDEAL}_{\mathcal{F},\mathcal{S}}(1^k, z)$ and $\text{REAL}_{\Pi,\mathcal{S}}(1^\kappa, z)$ are computationally indistinguishable. Figure 22 shows the ideal functionality $\mathcal{F}_{\text{abort}}^{\text{4pc}}$ for computing a function $f(\cdot)$ in the 4PC setting with abort security.

Functionality $\mathcal{F}_{\text{SRNG}}(\mathcal{P}_\Phi, \ell') \rightarrow \mathsf{r}_\Phi \in \mathbb{Z}_{2^{\ell'}}$

$\mathcal{F}_{\text{SRNG}}$ interacts with the parties in $\mathcal{P}_\Phi$ and the adversary $\mathcal{A}$.

Procedure: $\mathcal{F}_{\text{SRNG}}$ samples a random value $\mathsf{r}_\Phi \in \mathbb{Z}_{2^{\ell'}}$ and sends the message $(\text{Output}, \mathsf{r}_\Phi)$ to all the parties in $\mathcal{P}_\Phi$.

Figure 23: Shared random value generator (SRNG) functionality

We now provide the details for our simulation. For simplicity, we focus on the circuit evaluation process that includes input sharing, linear operations, multiplications and output reconstruction. Moreover, we provide the simulation for each of these stages separately

and carrying out these simulations in the topological order of the circuit provides the simulation for the entire computation. We provide proofs in the $\mathcal{F}_{\mathsf{setup}}$, $\mathcal{F}_{\mathsf{SRNG}}$-hybrid model, where $\mathcal{F}_{\mathsf{setup}}$ and $\mathcal{F}_{\mathsf{SRNG}}$ (Figure 23) denote the ideal functionalities for shared-key setup and Π_{SRNG} (Figure 2) protocol respectively.

Simulation strategy. The strategy for simulating the computation of the function $f(\cdot)$ (represented by the circuit ckt) is as follows: The simulation starts with the simulator emulating the shared-key setup ($\mathcal{F}_{\mathsf{setup}}$) functionality and distributing the corresponding keys to the adversary. Next is the input sharing phase, where the simulator $\mathcal{S}$ computes the input for the adversary $\mathcal{A}$ using the known keys and the messages received from $\mathcal{A}$, and sets the inputs of the honest parties to 0 for the simulation. $\mathcal{S}$ then invokes the ideal functionality $\mathcal{F}^{\mathsf{4pc}}_{\mathsf{abort}}$ on behalf of $\mathcal{A}$ with the extracted input and receives the output y. Note that with the inputs of $\mathcal{A}$ now known, $\mathcal{S}$ can compute all the intermediate values for each building block. Finally, $\mathcal{S}$ simulates each of the building blocks in topological order. To keep track of honest parties that abort during the simulation due to incorrect messages by $\mathcal{A}$, $\mathcal{S}$ maintains a list of parties denoted as $\mathcal{P}_\Phi$, initially set to $\emptyset$.

Input Sharing. To simulate the input sharing protocol Π_{Sh} (cf. Figure 12), the ideal world adversary $\mathcal{S}_{\Pi_{\mathsf{Sh}}}$ emulates $\mathcal{F}_{\mathsf{setup}}$ and obtains all the PRF keys. It then gives the respective PRF keys to the adversary $\mathcal{A}$. We consider the case of corrupt P_0 and P_3 and the simulation for corrupt P_1 and P_2 follows similar to P_0. To avoid repetition, we omit the details of $\mathcal{F}_{\mathsf{setup}}$ in the simulation steps.

Simulator $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Sh}}}$

Preprocessing:

(1) *Case I:* $P_I = P_0$. $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Sh}}}$ executes the steps of Π_{SRNG} using the respective PRF keys shared with $\mathcal{A}$ to obtain λ^1_x, λ^2_x, and λ^*_x.
(2) *Case II:* $P_I \neq P_0$. $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Sh}}}$ executes the steps of Π_{SRNG} using the respective PRF keys shared with $\mathcal{A}$ to obtain λ^1_x, and λ^2_x. It samples λ^*_x randomly from $\mathbb{Z}_{2^\ell}$.

Online:

(1) *Case I:* $P_I = P_0$.
  (a) $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Sh}}}$ receives $\overline{\mathsf{m}}_x$ from P_0 on behalf of P_1 and P_2
  (b) If the received values are consistent, $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Sh}}}$ computes input of P_0 (= $\mathcal{A}$) as $x = \overline{\mathsf{m}}_x - \lambda^1_x - \lambda^2_x - \lambda^*_x$. Else, it sets x = abort.
  (c) $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Sh}}}$ honestly executes the steps of Π_{CV} using the messages it received from P_0. If $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Sh}}}$ receives inconsistent hash from P_0 on behalf of $P_j \in \mathcal{P}_{1,2}$, it adds P_j to $\mathcal{P}_\Phi$.
(2) *Case II:* $P_I \neq P_0$.
  (a) $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Sh}}}$ sets $x = 0$ and honestly executes the steps of Π_{Sh} on behalf of P_1 and P_2. If $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Sh}}}$ receives inconsistent hash from P_0 on behalf of $P_j \in \mathcal{P}_{1,2}$, it adds P_j to $\mathcal{P}_\Phi$.

$\mathcal{P}_I$ denotes the input party with the secret $x \in \mathbb{Z}_{2^\ell}$.

Figure 24: Simulator $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Sh}}}$ for corrupt P_0

Shares unknown to $\mathcal{A}$ are randomly sampled in the simulation, whereas in the real protocol, they are sampled using the pseudo-random function (PRF). Therefore, the indistinguishability of the simulation is guaranteed by reducing it to the security of the PRF.

For the case when $P_I \neq P_0$ in Figure 24, $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Sh}}}$ sets the input of the honest parties $\mathcal{P}_{1,2,3}$ as $x = 0$. Given that $\mathcal{A}$ (who is P_0) only receives $\overline{\mathsf{m}}_x$ in the online phase but does not possess λ^*_x, the shares of $x = 0$ are indistinguishable to $\mathcal{A}$ from the actual shares of the honest parties in the real protocol.

Simulator $\mathcal{S}^{P_3}_{\Pi_{\mathsf{Sh}}}$

Preprocessing:

(1) $\mathcal{S}^{P_3}_{\Pi_{\mathsf{Sh}}}$ executes the steps of Π_{SRNG} using the respective PRF keys shared with $\mathcal{A}$ to obtain λ^1_x, λ^2_x, and λ^*_x.

Online:

(1) *Case I:* $P_I = P_3$.
  (a) $\mathcal{S}^{P_3}_{\Pi_{\mathsf{Sh}}}$ receives $\overline{\mathsf{m}}_x$ from P_3 on behalf of P_0, P_1 and P_2
  (b) If the received values are consistent, $\mathcal{S}^{P_3}_{\Pi_{\mathsf{Sh}}}$ computes input of P_3 (= $\mathcal{A}$) as $x = \overline{\mathsf{m}}_x - \lambda^1_x - \lambda^2_x - \lambda^*_x$. Else, it sets x = abort.
(2) *Case II:* $P_I \neq P_3$. There is nothing to simulate as P_3 does not receive any value during the online phase.

$\mathcal{P}_I$ denotes the input party with the secret $x \in \mathbb{Z}_{2^\ell}$.

Figure 25: Simulator $\mathcal{S}^{P_3}_{\Pi_{\mathsf{Sh}}}$ for corrupt P_3

Linear Operations. Since linear operations are local, they does not require communications to be simulated. The simulator $\mathcal{S}$ carries out the local operations on behalf of the honest parties.

Multiplication. To simulate protocol Π_{Mult} (cf. Figure 5), $\mathcal{S}_{\Pi_{\mathsf{Mult}}}$ uses the PRF keys obtained during the emulation of $\mathcal{F}_{\mathsf{setup}}$ and executes the Π_{SRNG} protocol. Once the random values are obtained, $\mathcal{S}_{\Pi_{\mathsf{Mult}}}$ executes the remaining steps of Π_{Mult} honestly. For this, $\mathcal{S}_{\Pi_{\mathsf{Mult}}}$ uses the inputs extracted during the simulation of Π_{Sh} and thereby learns all the intermediate values of the circuit. While simulating the steps of Π_{CV}, $\mathcal{S}_{\Pi_{\mathsf{Mult}}}$ keeps track of the honest parties who receive incorrect messages from $\mathcal{A}$ and adds them to $\mathcal{P}_\Phi$.

Simulator $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Mult}}}$

Preprocessing:

(1) $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Mult}}}$ executes the steps of Π_{SRNG} using the respective PRF keys shared with $\mathcal{A}$ to obtain λ^1_c, λ^2_c, and $r_{0,1,3}$. It samples λ^*_c and $r_{1,2,3}$ randomly from $\mathbb{Z}_{2^\ell}$.
(2) $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Mult}}}$ computes the values λ_c, $\mathsf{M}_{\{0,3\}}$ and $\mathsf{M}_{\{3\}}$ honestly.
(3) $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Mult}}}$ receives $\mathsf{M}_{\{0,3\}}$ from P_0 on behalf of P_2. If the received value is inconsistent with its locally computed version, $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Mult}}}$ sets the input of party P_0 as x = abort.
(4) $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Mult}}}$ sends $\mathsf{M}_{\{3\}}$ to P_0 on behalf of P_3.

Online:

(1) $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Mult}}}$ executes the steps honestly on behalf of P_1 and P_2.
(2) $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Mult}}}$ sends $\mathsf{M}_{\{1,2\}}$ to P_0 on behalf of P_2.
(3) $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Mult}}}$ interacts with P_0 honestly for Π_{CV} instances corresponding to $\mathsf{M}_{\{1,2\}}$, and $\overline{\mathsf{m}}_c$. If it receives an inconsistent value from P_0 in any of the two instances, $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Mult}}}$ adds the respective honest parties to $\mathcal{P}_\Phi$.

Figure 26: Simulator $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Mult}}}$ for corrupt P_0

We now provide the simulation for corrupt P_0 and P_2. The case for P_1 is similar to P_2, and for P_3, the role is minimal. Hence, we avoid providing simulation details for those cases. We begin with the case of corrupt P_0.

The simulation discussed above captures two scenarios. If $\mathcal{A}$ sends an incorrect message to any of the honest parties during the protocol, it is equivalent to $\mathcal{A}$ inputting `abort` to the ideal functionality $\mathcal{F}^{\text{4pc}}_{\texttt{abort}}$. In this case, $\mathcal{S}^{P_0}_{\Pi_{\text{Mult}}}$ simulates this behavior by setting $\mathcal{A}$'s input to `abort`.

In the other scenario, where $\mathcal{A}$ provides inconsistent values to honest parties during Π_{CV}, this is equivalent to $\mathcal{A}$ choosing those honest parties to receive `abort` in the ideal world. $\mathcal{S}^{P_0}_{\Pi_{\text{Mult}}}$ simulates this by identifying such instances and adding those honest parties to the list $\mathcal{P}_\Phi$ that it maintains. After simulating all the individual protocols, $\mathcal{S}^{P_0}_{\Pi_{\text{Mult}}}$ will input this list to $\mathcal{F}^{\text{4pc}}_{\texttt{abort}}$ on behalf of $\mathcal{A}$.

Simulator $\mathcal{S}^{P_2}_{\Pi_{\text{Mult}}}$

Preprocessing:

(1) $\mathcal{S}^{P_2}_{\Pi_{\text{Mult}}}$ executes the steps of Π_{SRNG} using the respective PRF keys shared with $\mathcal{A}$ to obtain λ^2_c, λ^*_c, and $r_{1,2,3}$. It samples λ^1_c and $r_{0,1,3}$ randomly from $\mathbb{Z}_{2^\ell}$.
(2) $\mathcal{S}^{P_2}_{\Pi_{\text{Mult}}}$ sends $\mathsf{M}_{\{0,3\}}$ to P_2 on behalf of P_0.

Online:

(1) $\mathcal{S}^{P_2}_{\Pi_{\text{Mult}}}$ executes the steps honestly on behalf of P_0 and P_1.
(2) $\mathcal{S}^{P_2}_{\Pi_{\text{Mult}}}$ sends $\mathsf{M}_{\{1\}}$ to P_2 on behalf of P_1.
(3) $\mathcal{S}^{P_2}_{\Pi_{\text{Mult}}}$ receives $\mathsf{M}_{\{2\}}$ and $\mathsf{M}_{\{1,2\}}$ from P_2 on behalf of P_1 and P_0 respectively. If any of the received values are inconsistent with their locally computed version, $\mathcal{S}^{P_2}_{\Pi_{\text{Mult}}}$ sets the input of party P_2 as $x =$ `abort`.
(4) $\mathcal{S}^{P_2}_{\Pi_{\text{Mult}}}$ interacts with P_2 honestly for Π_{CV} instances corresponding to $\mathsf{M}_{\{0,3\}}$, and $\overline{\mathsf{m}}_c$. If it receives an inconsistent value from P_2 in any of the two instances, $\mathcal{S}^{P_2}_{\Pi_{\text{Mult}}}$ adds the respective honest parties to $\mathcal{P}_\Phi$.

Figure 27: Simulator $\mathcal{S}^{P_2}_{\Pi_{\text{Mult}}}$ for corrupt P_2

Heterogeneous Multiplication. The simulation steps for the 4PC multiplication protocol in the heterogeneous setting, $\Pi_{\text{Mult\_H}}$ (see Figure 6), are very similar to those of Π_{Mult} discussed in this section. The primary difference from a simulation perspective is one communication message $\mathsf{M}_{\{3\}}$ in Π_{Mult}, which is replaced with a message from P_2 to P_0 in $\Pi_{\text{Mult\_H}}$. Therefore, we omit the formal details for $\Pi_{\text{Mult\_H}}$.

Output Reconstruction. To simulate the output reconstruction protocol Π_{Rec} (cf. Figure 13), $\mathcal{S}_{\Pi_{\text{Rec}}}$ utilizes the information about $\mathcal{A}$'s inputs that the ideal world adversary extracted during the simulation of the input sharing protocol. Additionally, it uses the list $\mathcal{P}_\Phi$, which it maintained to track the honest parties who were aborted during the simulation. We consider the case of corrupt P_0 and P_3 and the simulation for other parties are similar.

Simulator $\mathcal{S}^{P_0}_{\Pi_{\text{Rec}}}$

Postprocessing:

(1) $\mathcal{S}^{P_0}_{\Pi_{\text{Rec}}}$ invokes $\mathcal{F}^{\text{4pc}}_{\texttt{abort}}$ on (Input, x) on behalf of $\mathcal{A}$ to obtain the function output y. Here, x denotes the extracted inputs of P_0 during simulation of input sharing.
(2) *Case I:* $P_O = P_0$.
  (a) If $y \neq \perp$, $\mathcal{S}^{P_0}_{\Pi_{\text{Rec}}}$ computes and sends $\mathsf{m}_y = y + \lambda^1_y + \lambda^2_y$ to P_0 on behalf of P_2. Else, terminate.
  (b) $\mathcal{S}^{P_0}_{\Pi_{\text{Rec}}}$ honestly executes the steps of Π_{CV} for m_y and λ_y on behalf of P_1 and P_3 respectively. If $\mathcal{S}^{P_0}_{\Pi_{\text{Rec}}}$ receives inconsistent hash from P_0 on behalf of $P_j \in \mathcal{P}_{1,3}$, it adds P_j to $\mathcal{P}_\Phi$.
(3) *Case II:* $P_O \neq P_0$.
  (a) $\mathcal{S}^{P_0}_{\Pi_{\text{Rec}}}$ receives λ_y from P_0 on behalf of P_O. If the received value is not consistent with the value it holds (obtained as part of simulating $\mathcal{F}_{\text{setup}}$), it adds P_O and P_3 to $\mathcal{P}_\Phi$.
(4) $\mathcal{S}^{P_0}_{\Pi_{\text{Rec}}}$ sends $(\text{Select}, \mathcal{P}_\Phi)$ to $\mathcal{F}^{\text{4pc}}_{\texttt{abort}}$ on behalf of $\mathcal{A}$ and terminates.

$\mathcal{P}_O$ denotes the output party.

Figure 28: Simulator $\mathcal{S}^{P_0}_{\Pi_{\text{Rec}}}$ for corrupt P_0

Simulator $\mathcal{S}^{P_3}_{\Pi_{\text{Rec}}}$

Postprocessing:

(1) $\mathcal{S}^{P_3}_{\Pi_{\text{Rec}}}$ invokes $\mathcal{F}^{\text{4pc}}_{\texttt{abort}}$ on (Input, x) on behalf of $\mathcal{A}$ to obtain the function output y. Here, x denotes the extracted inputs of P_3 during simulation of input sharing.
(2) *Case I:* $P_O = P_3$.
  (a) If $y \neq \perp$, $\mathcal{S}^{P_3}_{\Pi_{\text{Rec}}}$ computes and sends $\mathsf{m}_y = y + \lambda^1_y + \lambda^2_y$ to P_3 on behalf of P_2. Else, terminate.
  (b) $\mathcal{S}^{P_3}_{\Pi_{\text{Rec}}}$ sends λ_y to P_3 on behalf of P_0.
  (c) $\mathcal{S}^{P_3}_{\Pi_{\text{Rec}}}$ honestly executes the steps of Π_{CV} for m_y on behalf of P_1. If $\mathcal{S}^{P_3}_{\Pi_{\text{Rec}}}$ receives inconsistent hash from P_3 on behalf of P_1, it adds P_1 to $\mathcal{P}_\Phi$.
(3) *Case II:* $P_O \neq P_3$.
  (a) $\mathcal{S}^{P_3}_{\Pi_{\text{Rec}}}$ honestly executes the steps of Π_{CV} for λ_y (obtained as part of simulating $\mathcal{F}_{\text{setup}}$) on behalf of P_O. If $\mathcal{S}^{P_3}_{\Pi_{\text{Rec}}}$ receives inconsistent hash from P_3 on behalf of P_O, it adds P_O to $\mathcal{P}_\Phi$.
(4) $\mathcal{S}^{P_3}_{\Pi_{\text{Rec}}}$ sends $(\text{Select}, \mathcal{P}_\Phi)$ to $\mathcal{F}^{\text{4pc}}_{\texttt{abort}}$ on behalf of $\mathcal{A}$ and terminates.

$\mathcal{P}_O$ denotes the output party.

Figure 29: Simulator $\mathcal{S}^{P_3}_{\Pi_{\text{Rec}}}$ for corrupt P_3

D COMPUTATIONAL COMPLEXITY

Table 10 extends Table 1 in §1 by showing the specific number of operations each party needs to perform locally in our protocols and related ones. While our 3PC protocol performs similarly to related work, we note that we shift a large amount of that computation to P_0 which is only active in the preprocessing phase. The online phase is generally considered a bottleneck for MPC deployments as it is latency-critical. Hence shifting computation and communication to the offline phase may become more attractive for deployment. Our 4PC protocol reduces the computational complexity significantly with similar considerations: The majority of the computation is assigned to $\mathcal{P}_{0,3}$ who only carry out non-latency critical operations.

Table 10: Operations and communication for multiplications (Extended)

Setting	Protocol	Party	Operation		Com	
			Add	Mult[a]	Pre.[b]	On
3PC	Replicated [2]	P_0	4	2 (+1)	0	1
		P_1	4	2 (+1)	0	1
		P_2	4	2 (+1)	0	1
		Total	12	6 (+3)	0	3
	ASTRA [12]	P_0	2	1	1	0
		P_1	4	2	0	1
		P_2	5	2[c]	0	1
		Total	**11**	5	1	**2**
	Trio (**This work**)	P_0	4	2	1	0
		P_1	4	2	0	1
		P_2	3	1	0	1
		Total	**11**	**5**	1	**2**
4PC	Fantastic Four [17]	P_0	15	9	0	0-3
		P_1	15	9	0	0-3
		P_2	15	9	0	0-3
		P_3	15	9	0	0-3
		Total	60	36	0	6
	Tetrad [27]	P_0	14	5	1	0
		P_1	12	8	0	1
		P_2	12	8	0	2
		P_3	14	9	1	0
		Total	52	30	2	**3**
	Quad (**This work**)	P_0	7	3	3	2
		P_1	5	3	4	2
		P_2	6	3	3	2-3
		P_3	7	3	5	0-1
		Total	**25**	**12**	2	**3**

[a] Replicated 3PC additionally requires a division operation (+1) per party in the arithmetic domain.
[b] Pre. - Preprocessing
[c] Reduced from 3 to 2 using a straightforward optimization.