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

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

Georg Carle
Technical University of Munich

ABSTRACT

In this work, we present novel protocols over rings for semi-honest secure three-party computation (3-PC) and malicious four-party computation (4-PC) with one corruption. Compared to state-of-the-art protocols in the same setting, our protocols require fewer low-latency and high-bandwidth links between the parties to achieve high throughput. Our protocols also reduce the computational complexity by requiring up to 50 percent fewer basic instructions per gate. Further, our protocols achieve the currently best-known communication complexity (3, resp. 5 elements per multiplication gate) with an optional preprocessing phase to reduce the communication complexity of the online phase to 2 (resp. 3) elements per multiplication gate.

In homogeneous network settings, i.e. all links between the parties share similar network bandwidth and latency, our protocols achieve up to two times higher throughput than state-of-the-art protocols. In heterogeneous network settings, i.e. all links between the parties share different network bandwidth and latency, our protocols achieve even larger performance improvements.

We implemented our protocols and multiple other state-of-the-art protocols (Replicated 3-PC, Astra, Fantastic Four, Tetrad) in a novel open-source C++ framework optimized for achieving high throughput. Five out of six implemented 3-PC and 4-PC protocols achieve more than one billion 32-bit multiplication or more than 32 billion AND gates per second using our implementation in a 25 Gbit/s LAN environment. This is the highest throughput achieved in 3-PC and 4-PC so far and between two and three orders of magnitude higher than the throughput MP-SPDZ achieves in the same settings.

KEYWORDS

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

1 INTRODUCTION

Secure Multiparty Computation (MPC) enables parties to execute functions on obliviously shared inputs without revealing them [24]. 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.

A popular approach to enable MPC is designing an addition and multiplication protocol based on a secret sharing (SS) scheme. With those two protocols in place, any function can be represented as a circuit consisting of addition and multiplication gates. These operations can be performed in different computation domains. Some protocols require computation over a field, while others allow computation over any ring. 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 [20]. Computation over $\mathbb{Z}_{2^{64}}$ is supported by 64-bit hardware natively and thus leads to efficient implementations. Multiple approaches also allow share conversion between computation domains to evaluate mixed circuits [10].

To evaluate a circuit over a ring, the parties first secretly share their inputs. Then, they evaluate each gate of the circuit in topological order. Linear gates, such as additions or multiplications by constants, can typically be evaluated locally by the parties without interaction. However, multiplying two secretly shared values requires the parties to exchange messages. After evaluating the circuit, the parties reveal their final shares to obtain the result of the computation.

As parties must wait for intermediary messages to be received before they can continue evaluating the circuit, the number of communication rounds required to evaluate a circuit scales linearly with the circuit's multiplicative depth. The number of messages required to evaluate a circuit scales linearly with the number of multiplication gates. As evaluating multiple circuits in parallel does not increase the number of communication rounds, and many circuits' multiplicative depth does not increase with the number of inputs, scaling MPC for large amounts of data and complex workloads usually requires high throughput. High throughput can be achieved by reducing the computational complexity and number of elements communicated per gate.

An MPC protocol can guarantee privacy and correctness against different adversary types [32]. A protocol in the honest majority class assumes that an adversary only controls a minority of the computation parties. A protocol in the semi-honest class assumes that an adversary does not deviate from the protocol specification. Typically, honest majority protocols are significantly faster than dishonest majority protocols, while semi-honest protocols are slightly faster than malicious protocols.

The three-party setting for semi-honest SS-based protocols is particularly relevant due to its low bandwidth requirements. This setting allows the use of information-theoretic security techniques not applicable to two-party computation [9]. Likewise, the four-party setting for SS-based protocols is of particular interest for malicious security with one corruption. This setting allows exploiting the redundancy of secretly shared values to efficiently verify the correctness of exchanged messages. These properties of 3-PC and 4-PC protocols can be utilized in the client-server model [12]. Here, three or four fixed computation nodes perform a computation for any number of input parties.

Some protocols make use of a preprocessing phase. This phase is typically independent of any inputs from the parties and only requires constant communication rounds. During the online phase, the parties consume values received in the preprocessing phase to evaluate the circuit. Some honest-majority protocols do not require all parties to communicate with each other or only during the preprocessing phase. These protocols are helpful in network settings where the network connectivity between the parties differs.

1.1 Motivation

Several existing works designed protocols that achieve the same communication complexity for each party [1, 2, 11, 15, 21]. However, in practical scenarios, network latency and bandwidth between parties might differ arbitrarily. Thus, achieving high throughput in practice requires utilizing links between parties more that are well-connected and links between parties less that are not well-connected. This flexibility requires heterogeneous protocols. We refer to a heterogeneous protocol if only some links between the parties require high bandwidth and only some links between the parties require low latency to achieve high throughput. We refer to a homogeneous protocol if all the communication and round complexity of a protocol is divided evenly on all or at least most links.

Additionally, recent protocols usually do not optimize for computational complexity. Thus, even state-of-the-art 4-PC protocols [7, 11, 23] require almost 100 local additions or multiplications for each multiplication gate on top of computing hashes and sampling shared random numbers. While the performance of MPC is communication-bound in most settings, using MPC with embedded devices, in network settings with high bandwidth, or to compute functions with low communication complexity, such as dot products, can lead to a bottleneck in computation.

Araki et al. [2] demonstrated for the first time that implementing a semi-honest 3-PC protocol in a homogeneous network setting can achieve a throughput of seven billion AND gates per second. However, their implementation is not published, and open-source implementations do not come close to that throughput. For instance, on our test setup, the popular open-source library MP-SPDZ [19] achieves a throughput of less than ten million AND gates per second using the same 3-PC protocol. Moreover, several state-of-the-art protocols lack any kind of open-source implementation entirely [5–7, 22, 23, 30].

High throughput has not yet been demonstrated for heterogeneous network settings and 4-PC protocols. Due to the recent interest in the 3-PC [1, 2, 6, 8, 15, 18, 25, 29, 30] and 4-PC [5, 7, 11, 16, 22, 23] settings, as well as MPC in heterogeneous network settings [13, 17, 31], there is a need for efficient protocols and open-source implementations in these settings.

1.2 Our Contribution

In this work, we present new 3-PC and 4-PC protocols in the honest majority setting. We provide efficient constructions for both heterogeneous and homogeneous network settings. Our main contributions are that our protocols achieve less computational complexity than related work, best-known communication complexity, and tolerate a higher number of weak network links between parties. Additionally, we provide an open-source implementation [1] of our protocol along with state-of-the-art protocols [2, 6, 11, 21, 23]. Our implementation achieves a currently unmatched throughput of more than 25 billion AND gates per second on a 25 Gbit/s network for each implemented protocol. When evaluating complex circuits such as AES, this translates to four times higher throughput than the currently fastest AES implementation by [2]. More specifically, we achieve the following results:

(1) We present a semi-honest 3-PC protocol and a malicious 4-PC protocol that require three (resp. five) elements of global communication per multiplication gate. Both protocols reduce the computational complexity per gate compared to related work. Figure 10 shows that our protocols achieve up to two times higher throughput for computationally intensive tasks such as dot products.
(2) Additionally, our 4-PC protocol requires fewer high-bandwidth links between parties than related work. Figure 9 shows that even if we restrict the bandwidth between $\frac{2}{3}$ of all links arbitrarily, we still achieve a throughput of approx. 10 billion AND gates per second. Both our 3-PC and 4-PC protocols only require one low-latency and two high-bandwidth links between the parties to achieve high throughput.
(3) We implement our protocols in C++ along with several other state-of-the-art 3-PC protocols [2, 6, 21], 4-PC protocols [11, 23], and a trusted-third-party protocol in our framework. Some of these protocols have not been previously implemented in any open-source framework [6, 23]. For other protocols [2, 11], we achieve up to three orders of magnitudes higher throughput than their current open-source implementation in MP-SPDZ [19]. The results are shown in tables 3, 4, and 5.

2 MPC IN VARIOUS NETWORK SETTINGS

With our 3-PC protocol and our heterogeneous 4-PC protocol, we aim to achieve high throughput while requiring a minimal number of high-bandwidth and low-latency links between the parties. This means that certain links are only required for setting up pre-shared keys between the parties or exchanging hashes for verification at the end of the protocol but not for the bulk of the communication. Figure 1 illustrates this property. Our 3-PC protocol does not require P_0 and P_1 to communicate, while P_0 and P_2 only communicate in the preprocessing phase. In our 4-PC protocol, even more links between the parties are not required. Additionally, the online phase contains messages that only serve the purpose of verifying communication. These messages can be sent in a single communication round. As a result, only one pair of parties in our 3-PC and heterogeneous 4-PC protocol needs to share a high bandwidth link, while only two pairs of parties need to share a low-latency link.

While this property is advantageous in heterogeneous network settings where the network link properties between parties differ, we show how to apply our protocols to network settings where this property is not the primary concern. For instance, in homogeneous network settings where all parties share similar bandwidth and latency, an efficient protocol evenly divides its communication complexity on all links.

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

Every n-PC 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 novel permutation of their roles in the protocol. We refer to this technique as Split-Roles. Consider a 3-PC protocol with parties P_i, P_j, P_k. In this example, there are six unique permutations to assign the party roles P_0, P_1, P_2 to P_i, P_j, P_k.

Observe that for every protocol using l elements of global communication, the number of messages per circuit remains the same, yet all communication channels are now utilized equally. Figure 1 illustrates the resulting communication between nodes when using our 3-PC and 4-PC protocols in a heterogeneous and a homogeneous network setting.

Suppose the parties wish to evaluate a circuit that cannot be parallelized well and need to compute a function only once. In that case, we can still optimize a protocol for a given network setting at no additional communication cost. For instance, in our 3-PC and 4-PC 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. Additionally, messages sent by a party in our 4-PC protocol are usually verified by another party holding the same message. These parties can also switch roles on a per-message basis. By changing the communication per-message, parties can granularly adjust the utilization of different network links.

3 RELATED WORK

Table 1 shows the number of operations required to calculate a multiplication gate of our protocols and related work in the same setting. Especially our 4-PC protocol reduces the number of required operations significantly. Our 3-PC protocol mainly reduces the number of required operations required by the parties active in the online phase. The table also shows the total number of ring elements sent by each protocol and the number of links utilized by each party. Note that our heterogeneous 4-PC protocol requires the lowest number of low bandwidth links (2) and the lowest number of low latency links (1) between the parties. Our benchmark (cf. figure 9) shows that this property leads to significant improvements in certain network settings. The table excludes calls to shared random number generators and hash-based verification. For both metrics our protocols tie with the best state-of-the-art protocol.

3.1 Three-Party Semi-honest Computation

Most existing semi-honest 3-PC protocols work optimally in a homogeneous network setting [2, 3, 21]. Only a few protocols work well in heterogeneous network settings [6] but are missing an open-source implementation.

Compared to existing 3-PC protocols, our protocol requires the same number of overall exchanged elements per gate but slightly lower computational complexity. Some of that computation and communication can be shifted in the constant round preprocessing phase. This way, only the network link between P_2 and P_3 is utilized in the online phase. Since P_0 performs most of the computation, the relative improvement compared to related work in the online phase is larger. Hence, if only considering the online phase, our protocol reduces the total number of elementary operations per multiplication by at least 40% compared to [2, 6].

The highest reported number of evaluated gates per second for 3-PC that we are aware of is 7 billion gates per second in the semi-honest setting [2] and 1.15 billion gates per second in the malicious setting [1]. Other work reports a little over 10 million multiplications per second in the semi-honest setting and 3 million multiplications per second in the malicious setting [14]. [33] use

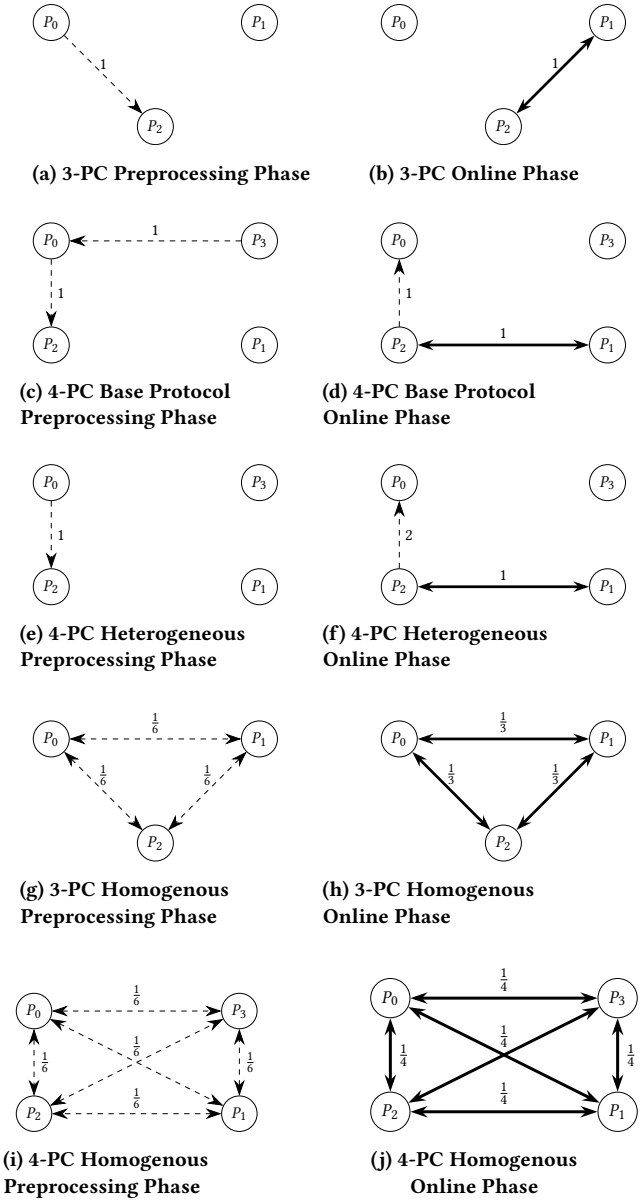

Figure 1: Number of ring elements exchanged between parties using our protocols in different network settings

Dashed arrows denote communication in constant communication rounds. Bolt arrows denote communication in linear communication rounds with respect to the circuit's multiplicative depth.

Table 1: Operations and communication for related protocols

Protocol	Party	Operation		Communication		
		Add	Mult	Off	On	Links
Replicated [2] (3-PC)	P_0	4	2 (+1)	0	1	1
	P_1	4	2 (+1)	0	1	1
	P_2	4	2 (+1)	0	1	1
	Total	12	6 (+3)	0	3	3B,3L
Astra [6] (3-PC)	P_0	2	1	1	0	1
	P_1	4	2	0	1	1
	P_2	5	3	0	1	1
	Total	11	6	1	2	2B,1L
Ours (3-PC)	P_0	4	2	1	0	1
	P_1	4	2	0	1	1
	P_2	3	1	0	1	1
	Total	11	5	1	2	2B,1L
Fantastic Four [11] (4-PC)	P_0	15	9	0	0-3	0-3
	P_1	15	9	0	0-3	0-3
	P_2	15	9	0	0-3	0-3
	P_3	15	9	0	0-3	0-3
	Total	60	36	0	6	2-4B,2-4L
Tetrad [23] (4-PC)	P_0	14	5	1	0	1
	P_1	12	8	0	1	1
	P_2	12	8	0	2	2
	P_3	14	9	1	0	1
	Total	52	30	2	3	3B,1L
Ours (4-PC)	P_0	7	3	3	2	1
	P_1	5	3	4	2	1
	P_2	6	3	3	2-3	2
	P_3	7	3	5	0-1	0-1
	Total	25	12	2	3	2-3B,1L

FPGAs to achieve 28.5 million multiplications per second. Our implementation achieves 44.49 billion AND gates or more than one billion 32-bit multiplications on a 25 Gbit/s network. Even if we consider a lower network bandwidth of 10 Gbit/s as in [2], our implementation achieves more than twice the performance.

3.2 Four-Party Malicious Computation

There are multiple 4-PC protocols that tolerate up to one corruption [5, 7, 11, 22, 23]. All of these protocols share that they require at least six elements of global communication per multiplication gate. Only recently, a protocol achieved five elements of global communication per multiplication gate [23]. However, among other 4-PC protocols [5, 7, 22], it is lacking an open-source implementation.

Our 4-PC protocol offers the following benefits over these protocols. First, it requires only five elements per multiplication gate. Second, we introduce a variation of our protocol optimized for heterogeneous network settings, requiring only one low-latency and two high-bandwidth links between the parties. As a result, out of the six total network links that exist in a 4-PC setup, four are not utilized at all, one is utilized in constant rounds only, and one is utilized with linear communication rounds. Existing work such as Trident [7] and Tetrad [23] require more than two high bandwidth links. While Fantastic Four [11] can be modified to require only two high-bandwidth links, it requires at least two low-latency links and more communication than our protocol. Figure 9 shows a setting where these properties of our protocol lead to arbitrary improvement in performance compared to existing work. Third, our protocols require storing fewer shares per party and reduce the computational complexity compared to the state-of-the-art. This includes reducing the total number of shares per value by $\frac{1}{3}$ and elementary operations per multiplication gate by more than 50% compared to [7, 11, 23]. Our benchmark demonstrates that our 4-PC protocol achieves up to 25% to 100% higher throughput than state-of-the-art protocols on the same setup.

Note that there is a recent solution that can convert any honest-majority semi-honest protocol into a malicious one at no additional amortized communication costs [4]. However, the zero-knowledge proofs required for this conversion come with significant computational overhead. According to a recent benchmark [11], 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.

The highest reported number of evaluated gates per second that we are aware of in the malicious 4-PC setting is 400,000 multiplications per second [11]. In the case of 64-bit computation, this result is communication-equivalent to 25.6 million AND gates. Our implementation achieves over 25 billion AND gates per second or more than 400 million 64-bit multiplications using the same protocol as [11], and over 600 million 64-bit multiplications using our proposed protocol.

4 PRIMITIVES

In this section, we present primitives required by our protocols to evaluate a function securely.

4.1 Generating Shared Random Numbers

Each pair of parties $\{P_i, P_j\}$ in our 3-PC protocol agrees on the same key $k_{i,j}$ at the beginning of the protocol. Using protocol Π_{SRNG}, the parties can generate new random values without interaction that are not accessible to P_k. We refer to this procedure as sampling from a shared random value generator (SRNG). For our 4-PC protocol, we assume that each set of parties $\{P_i, P_j, P_k\}$ has access to the same key $k_{i,j,k}$. Similar to our 3-PC protocol, the parties use $k_{i,j,k}$ to sample random ring elements without interaction, which are not known by P_l.

Protocol $\Pi_{\mathsf{SRNG}}(\mathsf{k_{i,j}}, \mathsf{c}) \rightarrow \mathsf{r_{i,j}}$

Setup: Let P denote the set of all parties, and $\mathbb{Z}_{2^l}$ denote the ring of integers modulo 2^l. Each subset $S \subseteq P$ exchanges a unique shared key $k_S \in \mathbb{Z}_{2^l}$ at the beginning of the protocol. Party $P_i \notin S$ does not learn k_S.

Procedure: Let $c \in \mathbb{N}$ represent a counter, and let $PRF : \mathbb{Z}_{2^l} \times \mathbb{N} \rightarrow \mathbb{Z}_{2^l}$ be a pseudorandom function.

(1) Compute $r_S = PRF(k_S, c)$ to obtain a random value $r_S \in \mathbb{Z}_{2^l}$.
(2) Upon need for a new random value, perform $c \leftarrow c + 1$ and repeat the first step.

Figure 2: Generating shared random numbers

4.2 Verifying the Correctness of Sent Messages

Our 4-PC protocol is secure against corruption of a single party. To achieve malicious security, each party needs to verify the correctness of the messages it receives. To verify the correctness of a value v obtained with the help of a message sent by a potentially malicious party P_i to P_j, the parties have access to a Compare-View functionality Π_{CV}.

If a party P_k also holds v, P_j and P_k can use the Compare-View functionality to compare their views of any number n of values $v_{1...n}$. To do so, they compare a single hash of their concatenated views of $v_{1...n}$ at the end of the protocol. Figure 3 describes this functionality.

Protocol $\Pi_{\text{CV}}(\text{v}_{\text{i...n}}, P_j, P_k) \rightarrow \text{bool}$

Assumptions: P_j obtains a set of values $\text{v}_{1...n}$ with the help of messages sent by P_i. P_k can obtain $v_{1...n}$, without interacting with P_i. P_j and P_k have access to a collision-resistant hash function H. Let $||$ denote concatenation.

Procedure:

(1) P_j and P_k compute $h_V = H(v_1||v_2||\ldots||v_n)$.
(2) They exchange and compare h_V to verify the correctness of $v_{i...n}$.
(3) If the two hash values h_V are not equal, they abort.

Figure 3: Verifying the correctness of received values

4.3 Notations

We use $P_{i,j}$ to indicate that a computation is performed by both P_i and P_j, and $r_{i,j}$ for a random value sampled by Π_{SRNG} using a key $k_{i,j}$ that is pre-shared between P_i and P_j in a setup phase. We use m^i to refer to a message that P_i sends to another party. We use v^i to refer to a term used by P_i to verify the correctness of the protocol. In some cases, parties locally compute a value x that is not finalized yet according to the protocol specification. These intermediary states of x are denoted by x'. A secret share of value x that is held by all parties according to our sharing semantics is denoted by $[x]$. Where applicable, $[x]^B$ denotes a boolean sharing of x while $[x]^A$ denotes an additive sharing of x.

5 3-PC PROTOCOL

In this section, we describe our 3-PC protocol over rings in detail. Our protocol is secure against up to one corruption. The global communication complexity is one element of communication in the preprocessing phase and two elements of communication in the online phase. To achieve high throughput, the protocol requires two high-bandwidth links and one low-latency link out of the three total links between the parties. If preferred, the preprocessing phase can also be executed within the online phase. The sharing semantics of our 3-PC protocols are designed in a way such that, P_1 and P_2 can communicate to obtain a masked version of a multiplication $c = ab$ with an input-independent error. P_0 can prepare a message for P_2 in the preprocessing phase in order to correct this input-independent error such that all parties obtain valid and masked shares of c.

5.1 Secret Sharing and Reconstruction

In order to share a value a held by input party A in our scheme, each party obtains the following shares from A.

$$P_0 : (x_1, x_2) \;\Big|\; P_1 : (x_1, a_2 = a + x_2) \;\Big|\; P_2 : (x_2, a_1 = a + x_1)$$

Note that x_1 and x_2 are input-independent values. The parties can sample x_1 using Π_{SRNG} with pre-shared key $k_{A,0,1}$, while x_2 can be sampled using pre-shared key $k_{A,0,2}$. To reconstruct a, P_0 sends x_1 to P_2 and x_2 to P_1, while P_2 sends a_1 to P_0. Each party then holds a pair (a_i, x_i) to compute $a = a_i - x_i$. Due to the linearity of the secret sharing scheme, additions and multiplications by a public value can be evaluated locally by the parties by computing the respective operations on each share.

5.2 Multiplication (AND) Gates

Let (x_1, x_2), (x_1, a_2), (x_2, a_1) be the secret sharing of a, and let (y_1, y_2), (y_1, b_2), (y_2, b_1) be the secret sharing of b. Computing $c = ab$, masked by z_i, requires parties to communicate. The intuition behind our multiplication protocol is that P_2 can locally compute $a_1b_1 = ab + ay_1 + bx_1 + x_1y_1$, thus obtaining ab with an input-dependent error: $ay_1 + bx_1$ and an input-independent error: x_1y_1. Our goal is to correct these errors using P_0's and P_2's messages while obliviously inserting the mask z_1, such that P_2 obtains $c_1 = ab + z_1$. Similarly, P_1 should obtain $c_2 = ab + z_2$. Figure 4 shows all steps required by our multiplication protocol. In the following paragraphs, we explain the formulas shown in figure 4.

Preprocessing Phase. By using their pre-shared keys, all parties first sample values to mask intermediary messages ($r_{0,1}$) or to mask final outputs (z_1 and z_2). All input-independent shares of c: z_1,z_2 are computed non-interactively using Π_{SRNG}.

All subsequent steps in the protocol are required to let P_1 and P_2 obtain valid input-dependent shares $c_2 = ab + z_2$ and $c_1 = ab + z_1$, respectively. P_0 sends message m^0 to P_2 in the preprocessing phase that serves the purpose of correcting the input independent error when P_1 and P_2 communicate. Note that $m^0 = x_1y_2 + x_2y_1 - x_1y_1 + r_{0,1}$ without brackets. The mask $r_{0,1}$ ensures that P_2 cannot infer any values from P_0's message.

Online Phase. In Step 1 of the online phase, P_2 locally computes $c_1' = a_1b_1 + m^0 = ab + ay_1 + bx_1 + x_1y_1 + m^0$. Note that P_2 obtains ab but with an input-dependent error: $ay_1 + bx_1$ and an input-independent error: $x_1y_1 + m^0$. P_1 calculates $c_2' = a_2y_1 + b_2x_1 + r_{0,1} = ay_1 + bx_1 + x_1y_2 + x_2y_1 + r_{0,1}$. Observe that the input-dependent terms $ay_1 + bx_1$ are exactly P_1's input-dependent error. P_1 and P_2 exchange their terms in Step 2 after inserting their desired final mask z_1 or z_2 of the other party in their messages.

After exchanging their messages, P_1 and P_2 can locally compute their share of c in Step 3. P_2 obtains its local share as follows.

$$\begin{aligned} c_1 &= c_1' - m^1 = ab + ay_1 + bx_1 + x_1y_1 - m^1 + m^0 && (1)\\ &= ab + ay_1 + bx_1 + x_1y_1 && (2)\\ &\quad - (ay_1 + bx_1 + x_1y_2 + x_2y_1 + r_{0,1} - z_1) + m^0 && (3)\\ &= ab + x_1y_1 - x_1y_2 - x_2y_1 - r_{0,1} + m^0 + z_1 && (4) \end{aligned}$$

Notice that by subtracting m^1, P_2 successfully got rid of the input dependent error, at the expense of a larger input independent error:

$x_1y_1 - x_1y_2 - x_2y_1 - r_{0,1}$. Fortunately, this is exactly the term $-m^0$ of the message P_0 prepared in its message in the preprocessing phase. Thus, the final equation results in:

$$ab + x_1y_1 - x_1y_2 - x_2y_1 - r_{0,1} + m^0 + z_1 = ab + z_1 \tag{5}$$

P_1 calculates $ab + z_2$ in a similar way. In Step 2 it receives $m^2 = c_1' + z_2 = a_1b_1 + m^0 + z_2$ and subtracts its already calculated c_2 from m^2 in Step 3 to obtain $ab + z_2$. Note that both parties P_1 and P_2 achieve low computational complexity by utilizing c_1 and c_2 for both their messages and their final local computation.

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

Preprocessing:

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

$$P_{0,1} : z_1, r_{0,1} \qquad P_{0,2} : z_2$$

(2) Locally compute:

$$P_0 : m^0 = x_2y_2 - (x_1 - x_2)(y_1 - y_2) + r_{0,1}$$

(3) Communicate:

$$P_0 \rightarrow P_2 : m^0$$

Online:

(1) Locally compute:

$$P_1 : c_2' = a_2y_1 + b_2x_1 + r_{0,1} \qquad P_2 : c_1' = a_1b_1 + m^0$$

(2) Communicate:

$$P_1 \rightarrow P_2 : m^1 = c_2' - z_1 \qquad P_2 \rightarrow P_1 : m^2 = c_1' + z_2$$

(3) Locally compute:

$$P_1 : c_2 = m^2 - c_2' \qquad P_2 : c_1 = c_1' - m^1$$

Figure 4: 3-PC multiplication protocol

5.3 Security

Araki et el. [2] formulate a notion of privacy in the client-server model. Loosely speaking, a protocol is private in the presence of even a malicious adversary $\mathcal{A}$ if the view of the $\mathcal{A}$ when the input v is computationally indistinguishable from its view when the input is $v\prime$. To achieve their notion of security, a protocol needs to satisfy two conditions. First, each share sent to $\mathcal{A}$ needs to be masked by a new value obtained from correlated randomness. Second, pseudorandom values need to be keyed by a key k that A does not see. Observe that our protocols satisfy these properties: Each message m in our protocols is masked by a new call to Π_{SRNG} with a key k that is shared by all parties except the recipient of m. While this notion of privacy can be achieved even by semi-honest protocols, it does not prove correctness. In the appendix, we unfold all computations of our multiplication protocol to show that all parties obtain valid shares according to the protocol specifications.

6 4-PC PROTOCOL

Our 4-PC protocol is secure against up to one malicious corruption. The global communication complexity is two elements of communication in the preprocessing phase and three elements of communication in the online phase. We present a variation of our protocol optimized for heterogeneous network settings, which requires two high-bandwidth links and one low-latency link out of the six total links between the parties to achieve high throughput. If preferred, the preprocessing phase can also be executed within the online phase. To optimize our protocol for homogeneous network settings, we use the techniques described in section 2. The sharing semantics of our 4-PC protocol naturally extend the 3-PC protocols with the necessary redundancy to verify all messages sent between the parties. This property requires P_0 to hold an input-dependent share to verify the communication between P_1 and P_2 with the help of an additional party P_3. P_3 also assists in verifying the messages sent by P_0.

6.1 Secret Sharing and Reconstruction

In order to share a value a held by input party A in our scheme, each party obtains the following shares from A.

$$P_0 : (a_u = a + u, x_0 = x_1 + x_2) \mid P_1 : (x_1, a_0 = a + x_0)$$
$$P_2 : (x_2, a_0 = a + x_0) \mid P_3 : (u, x_0 = x_1 + x_2)$$

Note that x_1, x_2, and u are input-independent values that can be sampled with Π_{SRNG} using pre-shared keys $k_{A,0,1,3}$, $k_{A,0,2,3}$, and $k_{A,1,2,3}$ respectively. P_0 and P_3 can then locally compute $x_0 = x_1 + x_2$. It is clear that no single party's share reveals anything about a. In addition, holding two distinct shares suffices to obtain a. For instance, by exchanging their shares, P_0 and P_1 can compute $a_0 - x_0 = a$.

To securely share a value in the presence of a malicious adversary, input party A sends $a_{u,0} = a + u + x_0$ to P_0, P_1, and P_2. The parties compare their view of $a_{u,0}$ and locally convert it to their respective share by subtracting the mask they generated together with A from $a_{u,0}$. To securely reconstruct a value in the presence of a malicious adversary, P_0 sends x_0 to P_1 and P_2. P_1, P_2, and P_3 then compare their view of x_0. Due to the security assumptions, one party of the pair $\{P_0, P_3\}$ is guaranteed to be honest. Thus, P_1 and P_2 either receive a correct x_0 or they abort. P_0 and P_3 exchange their shares a_u and u without any verification. All parties locally compute and compare their view of a. As one party of the pair of $\{P_1, P_2\}$ is honest, P_0 and P_3 either also obtain a correct a or they abort. Due to the linearity of the secret sharing scheme, additions and multiplications by a public value can be evaluated locally by the parties by computing the respective operations on each share.

6.2 Multiplication (AND) Gates

Let all shares a_i be masked by x_i or u and all shares b_i masked by y_i or v. In order to compute a secret sharing of $c = ab$, each party performs a different computation on its shares to obtain c_i masked by z_i or w. Figure 5 shows all steps required by the parties.

Preprocessing Phase. Again, the parties sample random ring elements using Π_{SRNG} to mask their messages and to insert the correct mask into another party's share when communicating. All input-independent shares of c: z_1,z_2, $z_0 = z_1 + z_2$, and w are computed non-interactively using Π_{SRNG}.

All subsequent steps in the protocol are required to let the parties $P_{0,1,2}$ obtain valid input-dependent shares. Similarly to the 3-PC protocol, P_0 sends a message m^0 to P_2 that serves the purpose of eliminating the input-independent error when P_1 and P_2 communicate and inserting the correct mask z_0 in their input-dependent shares. As P_0 also needs to obtain an input-dependent share $c_w = ab + w$ in our 4-PC protocol, P_3 computes message m^3 that serves the purpose of eliminating the input-independent error of P_0's computation in the online phase.

Online Phase. In the online phase, P_1 and P_2 exchange messages m^1 and $m^{2,0}$ to compute $c_0 = ab + z_0$. In Step 1 of the online phase, P_1 calculates m^1, and P_2 calculates $m^{2,0}$. Both messages are masked with $r_{0,1,3}$ and z_0 (contained in m^0) respectively. Both parties locally compute $c'_0 = a_0 b_0$ and subtract their locally computed messages from c'_0. After exchanging m^1 and $m^{2,0}$ in Step 2, both parties also subtract their received message to compute c_0 in Step 3.

Observe that the following equation holds:

$$\begin{aligned} m^1 &= a_0 y_1 + b_0 x_1 + r_{0,1,3} && (6)\\ &= a y_1 + b x_1 + x_0 y_1 + x_1 y_0 + r_{0,1,3} && (7)\\ m^{2,0} &= a_0 y_2 + b_0 x_2 - m^0 && (8)\\ &= a y_2 + b x_2 + x_0 y_2 + x_2 y_0 - m^0 && (9)\\ m^1 + m^{2,0} &= a y_0 + b x_0 + 2 x_0 y_0 + r_{0,1,3} - m^0 && (10)\\ a_0 b_0 &= ab + a y_0 + b x_0 + x_0 y_0 && (11) \end{aligned}$$

The equation shows that the input-dependent error $ay_0 + bx_0$ when computing $a_0 b_0$ matches the input-dependent term when computing $m^1 + m^{2,0}$. Using this insight, the parties can calculate $c_0 = a_0 b_0 - m^1 - m^{2,0} = ab - x_0 y_0 - r_{0,1,3} + m^0$. Notice that $m^0 = z_0 + x_0 y_0 + r_{0,1,3}$ was constructed by P_0 to eliminate the remaining input-independent error of $x_0 y_0 - r_{0,1,3}$ and obliviously insert the mask z_0 in c_0 such that both parties obtain $ab + z_0$.

In our 4-PC protocol, P_0 also needs to obtain the input-dependent share $c_w = ab + w$. In Step 1 of the online phase, P_0 locally computes $c'_w = a_u y_0 + b_v x_0$. P_1 and P_2 compute $m^{2,1} = c'_0 + r_{1,2,3}$. In step 2 of the online phase, P_2 sends $m^{2,1}$ to P_0. This message is used by P_0 in Step 3 to compute c_w locally. Observe that the following equation holds:

$$\begin{aligned} c'_w &= a_u y_0 + b_v x_0 = a y_0 + b x_0 + u y_0 + v x_0 && (12)\\ m^{2,1} &= a_0 b_0 + r_{1,2,3} = ab + a y_0 + b x_0 + x_0 y_0 + r_{1,2,3} && (13)\\ m^3 &= x_0 (y_0 - v) - y_0 u - w + r_{1,2,3} && (14)\\ c_w &= m^{2,1} - (c'_w + m^3) && (15)\\ &= a_0 b_0 - a y_0 - b y_0 - x_0 y_0 + w = ab + w && (16) \end{aligned}$$

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

Preprocessing:

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

$P_{0,1,3} : r_{0,1,3}, z_1$ $\quad$ $P_{0,2,3} : z_2$ $\quad$ $P_{1,2,3} : r_{1,2,3}, w$

(2) Locally compute:

$P_0, P_3 : z_0 = z_1 + z_2$ $\quad$ $P_0, P_3 : m^0 = z_0 + x_0 y_0 + r_{0,1,3}$

$P_3 : m^3 = x_0 (y_0 - v) - y_0 u - w + r_{1,2,3}$

(3) Communicate:

Online: $\quad P_0 \to P_2 : m^0$ $\quad$ $P_3 \to P_0 : m^3$

(1) Locally compute:

$P_0 : c'_w = a_u y_0 + b_v x_0$ $\quad$ $P_{1,2} : c'_0 = a_0 b_0$

$P_1 : m^1 = a_0 y_1 + b_0 x_1 + r_{0,1,3}$ $\quad$ $P_2 : m^{2,0} = a_0 y_2 + b_0 x_2 - m^0$

$P_{1,2} : m^{2,1} = c'_0 + r_{1,2,3}$

$P_1 : c''_0 = c'_0 - m^1$ $\quad$ $P_2 : c'''_0 = c'_0 - m^{2,0}$

(2) Communicate:

$P_1 \to P_2 : m^1$ $\quad$ $P_2 \to P_1 : m^{2,0}$ $\quad$ $P_2 \to P_0 : m^{2,1}$

(3) Locally compute:

$P_1 : c_0 = c''_0 - m^{2,0}$ $\quad$ $P_2 : c_0 = c'''_0 - m^1$

$P_0 : c_w = m^{2,1} - (c'_w + m^3)$

$P_{1,2} : c_{w,0} = c_0 + w$ $\quad$ $P_0 : c_{w,0} = c_w + z_0$

(4) Compare views using Π_{CV}:

$P_{0,1} : m^{2,1}$ $\quad$ $P_{2,3} : m^0$ $\quad$ $P_{0,1,2} : c_{w,0}$

Figure 5: 4-PC multiplication protocol

Verifying Communication. To ensure that their local share of c is valid, all parties that receive a message need to verify its correctness. To verify message m^0 sent by P_0 to P_2 in the preprocessing phase, P_2 and P_3 compare their views of m^0. Note that P_3 holds all values to compute m^0 locally. Similarly, P_1 can compute $m^{2,1}$ locally from its shares and compare its view with P_0.

The remaining messages are m^3 sent by P_3, and the messages m^1 and $m^{2,0}$ exchanged between P_1 and P_2. All these messages can be verified with a single check by parties $P_{0,1,2}$ comparing their view of $ab + c_0 + w$. If P_3's message m^3 is incorrect, P_1's and P_2's correct view will differ from P_0's corrupted view. If P_1's or P_2's message is incorrect, P_0's correct view will differ from their corrupted views of $ab + c_0 + w$. As our protocol tolerates up to one corrupted party, only one of these cases can occur. Therefore, the parties successfully verified all messages exchanged during the multiplication protocol.

Note that message $m^{2,1}$ is only needed to let P_0 verify the messages exchanged between P_1 and P_2. Delaying all messages for all gates by a constant factor does not affect the protocol's throughput in the amortized sense. For this reason, P_0 and P_1 can share a high-latency link even if the evaluated circuit has a high multiplicative depth. Messages utilized that way can also be viewed as part of a constant-round post-processing phase. Note that the parties achieve low computational complexity by reusing calculated terms across messages, verification, and obtaining shares.

6.3 Multiplication in Heterogeneous Network settings

The previously introduced multiplication protocol requires three high-bandwidth and one low-latency link between all parties. For heterogeneous network settings, we can reduce the number of

required high-bandwidth links further to two. While difficult to prove, this property seems optimal for high-throughput secret-sharing-based schemes in terms of required low-latency and high-bandwidth links between the parties. A lower number of low-latency links would imply that the parties can compute any circuit in constant rounds. A lower number of required high-bandwidth links would imply that two parties can efficiently evaluate a non-linear gate without obtaining any messages from a third party.

In order to shift all the communication from our multiplication protocol to two links, we replace the need for m^3 that P_3 sends to P_0 in protocol Π_{Mult} with a message $m^{2,2}$ that P_2 sends to P_0. P_0 then verifies P_1's and P_2's communication with the help of this new message $m^{2,2}$. Our modification has the additional advantage that P_3 now does not need to communicate to any other party when evaluating a multiplication gate. Thus, P_3 can have an arbitrarily weak network link to all other parties. Figure 6 shows all steps required by the parties.

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

Preprocessing:

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

$P_{0,1,3} : r_{0,1,3}, z_1 \qquad P_{0,2,3} : z_2 \qquad P_{1,2,3} : r_{1,2,3}, w$

(2) Locally compute:

$P_0, P_3 : z_0 = z_1 + z_2 \qquad P_0, P_3 : m^0 = z_0 + x_0 y_0 + r_{0,1,3}$

$P_3 : v^3 = x_0(y_0 - v) - y_0 u - z_0 + r_{1,2,3}$

(3) Communicate:

$P_0 \rightarrow P_2 : m^0$

Online:

(1) Locally compute:

$P_1 : m^1 = a_0 y_1 + b_0 x_1 + r_{0,1,3} \qquad P_2 : m^{2,0} = a_0 y_2 + b_0 x_2 - m^0$

$P_1 : c_0' = a_0 b_0 - m^1 \qquad P_2 : c_0'' = a_0 b_0 - m^{2,0}$

$P_0 : c_w' = a_u y_0 + b_v x_0$

(2) Communicate:

$P_1 \rightarrow P_2 : m^1, m^{2,0}$

(3) Locally compute:

$P_1 : c_0 = c_0' - m^{2,0} \qquad P_2 : c_0 = c_0'' - m^1$

$P_1, P_2 : m^{2,1} = c_0 + w \qquad P_1, P_2 : m^{2,2} = m^1 + m^{2,0} + r_{1,2,3}$

(4) Communicate:

$P_2 \rightarrow P_0 : m^{2,1} \qquad P_2 \rightarrow P_0 : m^{2,2}$

(5) Locally compute:

$P_0 : v^3 = m^{2,2} - c_w' \qquad P_0 : c_w = m^{2,1} - z_0$

(6) Compare views using Π_{CV}:

$P_0, P_1 : m^{2,1}, m^{2,2} \qquad P_0, P_3 : v^3 \qquad P_2, P_3 : m^0$

Figure 6: Heterogeneous 4-PC multiplication protocol

The key modification in the preprocessing phase is that P_3 now calculates v^3 (m^3 in protocol Π_{Mult}). P_3 does not send v^3 to P_0. Similarly to protocol Π_{Mult}, P_1 and P_2 use Steps 1-3 of the online phase to compute their share c_0. However, in protocol $\Pi_{\text{Mult\_H}}$ they define $m^{2,1} = c_0 + w$ and $m^{2,2} = m^1 + m^2$ in Step 3. P_2 then sends these messages to P_0. In Step 5, P_0 uses message $m^{2,1}$ to locally compute its share c_w without requiring interaction with P_3.

To help P_1 and P_2 verify the correctness of their exchanged messages m^1 and $m^{2,0}$, P_0 uses $m^{2,2}$ to compute $v^3 = m^{2,2} - c_w'$ and compares its view of v^3 with P_3. The following equations show why the messages can be verified this way.

$$m^{2,2} = m^1 + m^{2,0} + r_{1,2,3} = ay_0 + bx_0 + x_0 y_0 - z_0 + r_{1,2,3} \quad (17)$$

$$c_w' = a_u y_0 + b_v x_0 = ay_0 + bx_0 + uy_0 + vx_0 \quad (18)$$

$$v^3 = m^{2,2} - c_w' = (m^1 + m^{2,0}) - a_u y_0 + b_v x_0 \quad (19)$$

$$= x_0 y_0 - uy_0 - vx_0 - z_0 + r_{1,2,3} \quad (20)$$

The equations show that P_0 obtains the same v^3 that P_3 computes locally in the preprocessing phase only if m^1 and $m^{2,0}$ are correct. To verify whether P_2 also sent $m^{2,2}$ that is consistent with P_1's view of $m^{2,2}$, P_0 and P_1 compare their views of $m^{2,2}$. Finally, P_0 needs to verify $m^{2,1}$ sent by P_2. To do so, P_0 simply compares its view of $m^{2,1}$ with P_1.

Similarly to the basic variant of the 4-PC multiplication protocol, P_2's messages $m^{2,1}$ and $m^{2,2}$ can be received by P_0 with arbitrary delay and can be regarded as part of a constant round post-processing phase. Therefore, the multiplication protocol requires only one communication round and one low latency link between P_1 and P_2.

To ensure that all parties can detect corrupted messages, we consider all possible scenarios:

(1) P_0 is corrupted. P_0 can send a corrupted m^0 to P_2. In this case, P_2's and P_3's views of m^0 differ.
(2) P_1 is corrupted. P_1 can send a corrupted m^1 to P_2. In this case, P_0's and P_3's views of v^3 differ.
(3) P_2 is corrupted. P_2 can choose the same error e in all its messages. In this case, P_0's and P_3's views of v^3 differ. As soon as P_2 chooses different errors in its messages P_1's view of $m^{2,1}$ or $m^{2,2}$ will differ from P_0's.
(4) P_3 is corrupted. P_3 does not send any messages but only compares its views with other parties at the end of the protocol. Any corrupted messages in this phase will lead to an abort of the protocol.

6.4 Security

The earlier defined notion of privacy in the presence of a malicious adversary $\mathcal{A}$ also holds for our 4-PC protocols: Each message m in our protocol is masked by a new call to Π_{SRNG} with a key k that is shared by all parties except the recipient of m. Similarly to [11], we show in the appendix that each message a party sends is verified by a set of other parties using Π_{CV}.

7 ADDITIONAL PROTOCOLS

Using our protocols for real-world applications requires support for fixed point arithmetic and mixed circuits. In the appendix, we provide additional protocols for these settings. Truncation is required after multiplying two fixed point shares. Similiar to Tetrad [23], probabilistic truncation as proposed by ABY3 [28] can be built into our 3-PC and 4-PC multiplication protocols at no additional

communication costs. We are not aware of any existing 3-PC protocol that implements a probabilistic truncation protocol that is fused with the multiplication protocol. As in our multiplication protocol we improve over Tetrad in computational complexity and higher tolerance to weak network links.

To evaluate a comparison $[a] > [b]$ we use the following established sequence proposed by [28]. First, the parties calculate $[c] = [b] - [a]$. Note that the sign bit of c is 1 if $a > b$, and 0 otherwise. By converting $[c]$ 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.

Share conversion from the arithmetic to the boolean domain requires one ring element of total communication in our 3-PC scheme and two elements of communication in our 4-PC scheme, followed by a boolean addition. Similarly, converting a shared bit to an arithmetic share requires one ring element of total communication in our 3-PC scheme and two elements of communication in our 4-PC scheme, followed by an XOR in the arithmetic domain. In both schemes, converting from the arithmetic to the boolean domain is part of non-latency critical communication and thus does not add to the round complexity. Table 2 shows the number of ring elements exchanged for the different protocols. All additional protocols only utilize the same network links between the parties that are already utilized in our multiplication protocol. Thus, our protocols maintain their high tolerance to weak network links. Similar to our multiplication protocols, each message is masked by Π_{SRNG}, while each message is verified using Π_{CV}.

Table 2: Communication complexity of additional protocols

Primitive	Scheme	Off	On	Rounds
Multiplication + Truncation	3-PC	1	2	1
	4-PC	2	3	1
Arithmetic to Boolean (ex. Boolean Addition)	3-PC	1	0	0
	4-PC	1	1	0
Bit to Arithmetic (ex. Arithmetic XOR)	3-PC	1	0	0
	4-PC	1	1	0

8 IMPLEMENTATION

We implemented our protocols and related state-of-the-art ones in C++. Our implementation supports Bitslicing, Vectorization, multiprocessing, hardware instructions such as VAES and SHA, and adjustable message buffering.

All results in this section and section 9 are based on a test setup of 3-4 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.

8.1 Accelerating Basic Instructions

To implement the protocols efficiently, we first need to accelerate the operations required by all protocols. We use Single Instruction Multiple Data (SIMD) approaches to accelerate these operations using wider register sizes. For example, we can use the AVX-512 instruction set to perform eight 64-bit additions, 512 1-bit logic gates, or four 128-bit AES rounds in parallel using a single instruction on a 512-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.

8.2 Bitslicing and Vectorization

The key idea of Bitslicing is that computing a bit-wise logical operation on an m-bit register effectively works like m parallel boolean conjunctions, each processing a single bit [26]. Thus, Bitslicing can accelerate single-bit operations such as AND or XOR.

Vectorization is a technique to perform multiple operations in parallel by packing multiple values next to each other in a single vector. Modern X_{86} processors provide hardware instructions to perform operations such as additions and multiplications on a vector of packed values using a single instruction. To efficiently switch from a bit-sliced representation to a vectorized representation, we utilize the matrix transposition techniques implemented by the Usuba Bitslicing compiler [27].

Figure 7 shows the throughput of the protocols when utilizing Bitslicing. 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 boolean variable and performing one instruction for each input.

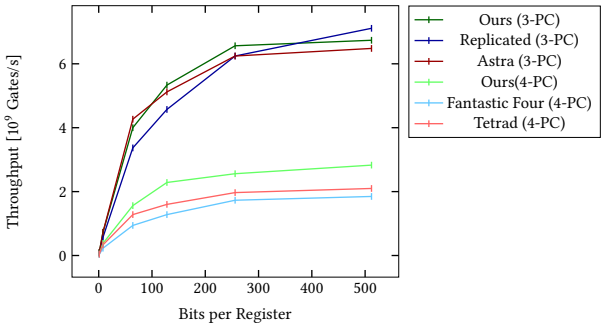

Figure 7: Throughput in billion AND gates per second when utilizing Bitslicing

8.3 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 $3MB$ of messages lead to the highest throughput.

8.4 Multiprocessing

Figure 8 shows that when combining the Bitslicing with multiprocessing, our implementations achieves a throughput of more than 20 billion AND gates per second for all protocols except Tetrad.

These results are within 80% – 95% 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 explained by the networking overhead when sending and receiving messages with multiple threads using conventional sockets.

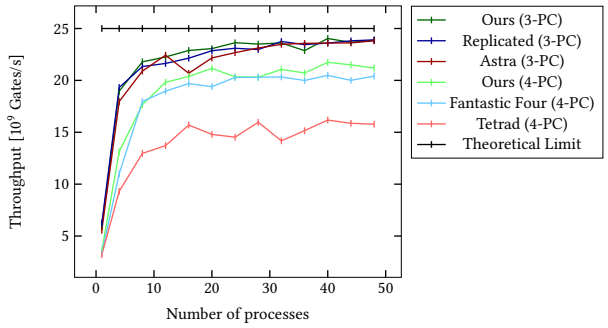

Figure 8: Throughput in billion AND gates per second when utilizing Multiprocessing

8.5 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 3-PC protocol that requires three elements of global communication. We can increase the throughput further by executing a 3-PC protocol with four parties, essentially creating a 4-PC 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 4 shows the throughput of the implemented protocols when utilizing Split-Roles along with our other tweaks.

8.6 Online Phase

Most protocols we implemented offer a preprocessing phase that can be detached from the online phase. Table 4 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 (c.f. Table 5). The table shows that the throughput when utilizing a TTP is less than one order of magnitude higher than the secure alternatives when utilizing all aforementioned tweaks.

9 BENCHMARKS

In this section, we present the results of our benchmarks. We implemented two other state-of-the-art protocols for each category, namely the 3-PC protocols Astra [6] and Replicated [2] and the 4-PC protocols Fantastic Four [11] and Tetrad [23]. One protocol in each category offers function-dependent preprocessing (Astra, Tetrad), while the other does not (Replicated, Fantastic Four). All benchmarks were performed on the aforementioned test setup with AMD EPYC nodes on a 25 Gbit/s network and 0.3ms latency between the nodes. We start by benchmarking the throughput of independent *AND* and multiplication gates, as accelerating these basic operations in MPC also benefits all higher-level functions. Apart from our benchmarks, we also tested the correctness of all implemented secret sharing and multiplication protocols, which should give more confidence in the protocols that have been lacking an open-source implementation so far [6, 23].

9.1 MP-SPDZ

MP-SPDZ [19] also implements the Replicated 3-PC and the Fantastic Four 4-PC protocols. Table 3 shows the throughput of the two protocols on our test setup. Observe also that MP-SPDZ's throughput of *AND* gates (ring size 2) does not differ significantly from its throughput of 64-bit multiplication gates (ring size 2^{64}), even though the latter requires sending 64 times the data between the parties. This shows that MP-SPDZ does not utilize the whole network bandwidth but is either CPU- or memory-bottlenecked. Overall, MP-SPDZ achieves a throughput of 2.8 million to 9.7 million gates per second on our test setup in the different settings. While MP-SPDZ also offers multithreading functionality with so-called tapes, utilizing multiple tapes reduced the throughput in our tests.

Table 3: MP-SPDZ - Throughput in million gates per second

Protocol	Gate	Throughput
Replicated	Mult	9.7
	AND	7.5
Fantastic-Four	Mult	2.8
	AND	6.4

9.2 Our Framework

We move on to the results of our implementation. Table 5 and table 4 show the throughput in billion AND gates per second of our protocol and related state-of-the-art ones. We combine all optimizations introduced earlier to achieve the highest throughput for each protocol. We also implemented a Trusted Third Party (TTP) protocol that performs the computation on a single node in the clear. The column stating "Performance Improvement" shows the percentage difference in throughput that our protocol in the same category achieves compared to that specific protocol. The column stating "Theoretical Limit" shows the throughput that our implementation achieves compared to the theoretical optimum that our given network could achieve if we only consider the communication complexity of the protocol. Note that the theoretical limit increases if we utilize Split-Roles. The suffix "(Online)" of a protocol name indicates that we measured only the time of the online phase.

Table 5 and 4 show, that our protocols achieve at least 69% network utilization in all setups and provide higher throughput than state-of-the-art alternatives. Our implementation of the Replicated and Fantastic Four protocols achieves more than three orders of magnitudes higher throughput of *AND* gates than their MP-SPDZ implementation. Notably, the throughput of *AND* gates per second on the same hardware when utilizing a Trusted Third Party (TTP) instead of a secure protocol is only around one order of magnitude

Table 4: Throughput for the implemented protocols when using Split-Roles

Category	Protocol	Billion Gates/s	Performance Improvement (%)	Theoretical Limit (%)
3-PC Semi-Honest	Replicated	40.16	10.55%	80.31%
	Astra	44.14	0.58%	88.28%
	Ours	44.39	-	88.79%
	Astra (Online)	68.25	1.05%	91.00%
	Ours (Online)	71.84	-	95.79%
4-PC Semi-Honest	Replicated	61.85	15.06%	61.85%
	Astra	67.77	5.02%	67.77%
	Ours	71.17	-	71.17%
	Astra (Online)	126.69	3.24%	84.46%
	Ours (Online)	130.79	-	87.19%
4-PC Malicious	Fantastic Four	26.48	56.76%	44.14%
	Tetrad	33.03	25.68%	55.05%
	Ours	41.51	-	69.19%
	Tetrad (Online)	42.22	67.68%	42.22%
	Ours(Online)	70.80	-	70.80%

Table 5: Throughput for the implemented protocols without using Split-Roles

Protocol	Billion Gates/s	Performance Improvement (%)	Theoretical Limit (%)
Replicated	24.30	-0.16%	97.22%
Astra	24.10	0.68%	96.40%
Ours (3-PC)	24.27	-	97.06%
Fantastic Four	20.46	10.83%	81.85%
Tetrad	15.78	43.70%	63.12%
Ours (4-PC)	22.68	-	90.71%
Trusted Third Party	512	-	-

higher. Due to Bitslicing, our implementation's throughput in Gbit/s does not differ significantly when calculating *AND* gates, 32-bit, or 64-bit multiplications. Thus, the throughput in multiplications per second can be roughly calculated as the reported number of *AND* gates per second divided by the integer bitlength.

9.2.1 3-PC protocols. Table 5 shows that without utilizing Split-Roles, all 3-PC protocols achieve similar runtimes and over 95% network utilization. This result suggests that the available bandwidth restricts the protocols' throughput. Once we utilize Split-Roles, we need more computing power to saturate the network bandwidth. In this case, we start to see a noticeable difference in throughput between the protocols (cf. Table 4). This difference increases when using 3-PC protocols in a 4-PC setting, where we can distribute the communication complexity of the protocol on more links between the parties. Overall, our 3-PC protocol achieves up to 5% higher throughput than the best state-of-the-art 3-PC protocol.

9.2.2 4-PC protocols. In contrast to the 3-PC protocols, the 4-PC protocols already show computation bottlenecks without using Split-roles. This is expected as they require significantly more local computation. As a result, Tetrad only achieves 63.12% network saturation (cf. Table 5). Since in the 4-PC setting, the improvement in computational complexity and memory complexity between our protocol and the state-of-the-art protocols is higher than in the 3-PC setting, our protocol still achieves 90.71% network saturation in this setting. Overall, our 4-PC protocol achieves up to 25.68% higher throughput than the best state-of-the-art protocol in the same category. Notably, when considering only the online phase, all our protocols achieve more than 70 billion AND gates or more than two billion 32-bit multiplications per second.

9.3 Sweetspots

Our 4-PC protocols excel especially in two scenarios: Heterogeneous network settings, and computational extensive tasks.

9.3.1 Heterogeneous Network Settings. We simulate heterogeneous network settings by using Linux traffic control (tc) to restrict the bandwidth between the nodes. In these cases, we did not use Split-roles to benefit from the heterogeneous properties of our protocols. When we restrict the bandwidth between $\frac{1}{3}$ of the nodes in our setup, our 3-PC 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 fact that the link between P_0 and P_1 is not utilized at all in the multiplication protocol.

Figure 9 shows that even if we restrict the bandwidth of $\frac{2}{3}$ of the links in our setup, our 4-PC variation optimized for heterogeneous settings still achieves a throughput of approx. 10 billion *AND* gates per second. In this setting, we cannot achieve close to 25 billion *AND* gates per second because to divide five elements of communication on two links, one link necessarily has to transmit two elements per *AND* gate in the same direction. The figure shows that while the bandwidth restriction affects all other protocols, it does not affect the throughput of our heterogeneous protocol.

9.3.2 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: The communication complexity to evaluate

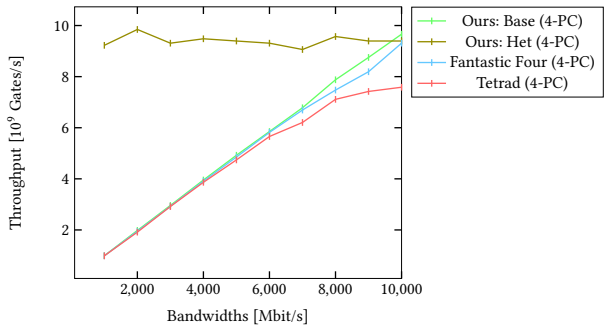

Figure 9: Throughput when restricting the bandwidths between $\frac{2}{3}$ of all links

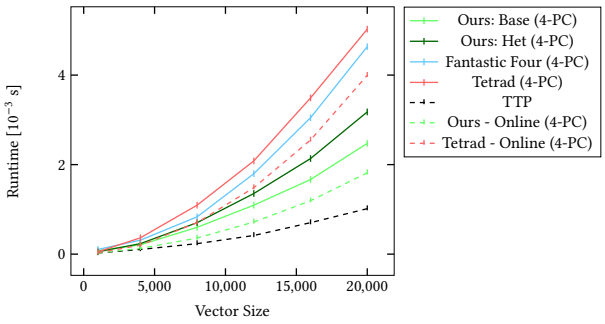

Figure 10: Runtimes of 4-PC protocols to evaluate a Vector-Matrix-Product

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 10 shows that our 4-PC protocol is two times faster when evaluating large dot products than Tetrad and Fantastic Four. Furthermore, our Trusted-Third-Party implementation is less than three times faster than our 4-PC protocol on the same hardware.

9.4 AES

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 [27]. We perform over 90 million AES blocks in parallel using all tweaks introduced in section 8.

Table 6: Throughput in million AES blocks per second

Protocol	Million Blocks/s	Performance Improvement (%)	Theoretical Limit (%)
Replicated	5.59	2.58%	59.01%
Astra	6.27	-8.62%	66.24%
Ours (3-PC)	5.73	-	60.54%
Astra (Online)	6.44	11.23%	45.36%
Ours (3-PC, Online)	7.17	-	50.46%
Fantastic Four	2.43	62.18%	25.64%
Tetrad	2.25	74.81%	19.82%
Ours (4-PC)	3.94	-	34.65%
Tetrad (Online)	2.63	80.78%	13.87%
Ours (4-PC, Online)	4.75	-	25.07%
Trusted Third Party	19.16	-	-

Table 6 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 more than four times higher throughput than previous work using the same 3-PC protocol. Again, especially our 4-PC protocol improves performance compared to other protocols significantly. Our 3-PC protocol mainly shows improvements in the online phase, as most computation in the protocol is performed by P_0 and can thus be shifted to the preprocessing phase. In the appendix, we also evaluate the RAM usage when running the AES benchmark.

10 CONCLUSION

In this work, we proposed novel honest-majority three-party and four-party computation protocols optimized for achieving high throughput in various network settings. By utilizing the client-server model, our results can be used to enable efficient MPC for any number of input parties. Our open-source implementation demonstrates that our protocols can evaluate billions of gates per second, even if most of the links between the parties have high latency and low bandwidth. This result shows that MPC can handle demanding workloads in diverse real-world settings where computation nodes may have varying bandwidth and latency. Finally, our benchmarks suggest that bridging the runtime difference between MPC and a TTP, needs optimizations both in the communication and computational aspects of MPC protocols, as well as enhancements in MPC implementations.

An interesting direction for future work is to investigate which other honest-majority settings can be optimized for heterogeneous network settings. Due to the high throughput our implementation achieves when evaluating boolean and arithmetic gates, another direction for future work is to apply the techniques we presented in this work to high-level MPC frameworks and use cases. Additionally, demanding workloads that require evaluating large dot products, such as privacy-preserving machine learning, can particularly benefit from our novel protocols.

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 *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 843–862.

[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 *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 805–817.

[3] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. 2019. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*. 351–371.

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

[5] Megha Byali, Harsh Chaudhari, Arpita Patra, and Ajith Suresh. 2019. FLASH: Fast and robust framework for privacy-preserving machine learning. *Cryptology ePrint Archive* (2019).

[6] Harsh Chaudhari, Ashish Choudhury, Arpita Patra, and Ajith Suresh. 2019. ASTRA: high throughput 3pc over rings with application to secure prediction. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*. 81–92.

[7] Harsh Chaudhari, Rahul Rachuri, and Ajith Suresh. 2019. Trident: Efficient 4pc framework for privacy preserving machine learning. *arXiv preprint arXiv:1912.02631* (2019).

[8] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. 2018. Fast large-scale honest-majority MPC for malicious adversaries. In *Advances in Cryptology–CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part III 38*. Springer, 34–64.

[9] Ronald Cramer, Ivan Damgard, and Jesper Buus Nielsen. 2012. Secure multiparty computation and secret sharing-an information theoretic appoach. *Book draft* (2012).

[10] Ronald Cramer, Ivan Damgård, and Yuval Ishai. 2005. Share Conversion, Pseudorandom Secret-Sharing and Applications to Secure Computation. *Lecture Notes in Computer Science* 3378, 342–362. https://doi.org/10.1007/978-3-540-30576-7_19

[11] Anders PK Dalskov, Daniel Escudero, and Marcel Keller. 2021. Fantastic Four: Honest-Majority Four-Party Secure Computation With Malicious Security.. In *USENIX Security Symposium*. 2183–2200.

[12] Ivan Damgård, Kasper Damgård, Kurt Nielsen, Peter Sebastian Nordholt, and Tomas Toft. 2016. Confidential benchmarking based on multiparty computation. In *International Conference on Financial Cryptography and Data Security*. Springer, 169–187.

[13] Mentari Djatmiko, Mathieu Cunche, Roksana Boreli, and Aruna Seneviratne. 2012. Heterogeneous secure multi-party computation. In *11th International Networking Conference (NETWORKING)*. Springer, 198–210.

[14] Hendrik Eerikson, Marcel Keller, Claudio Orlandi, Pille Pullonen, Joonas Puura, and Mark Simkin. 2019. Use your brain! Arithmetic 3PC for any modulus with active security. *Cryptology ePrint Archive* (2019).

[15] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. 2017. High-throughput secure three-party computation for malicious adversaries and an honest majority. In *Advances in Cryptology–EUROCRYPT 2017: 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30–May 4, 2017, Proceedings, Part II 36*. Springer, 225–255.

[16] S Dov Gordon, Samuel Ranellucci, and Xiao Wang. 2018. Secure computation with low communication from cross-checking. In *Advances in Cryptology–ASIACRYPT 2018: 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2–6, 2018, Proceedings, Part III 24*. Springer, 59–85.

[17] Vipul Goyal, Chen-Da Liu-Zhang, and Rafail Ostrovsky. 2023. Asymmetric Multi-Party Computation. Cryptology ePrint Archive, Paper 2023/704. https://eprint.iacr.org/2023/704 https://eprint.iacr.org/2023/704.

[18] Yuval Ishai, Ranjit Kumaresan, Eyal Kushilevitz, and Anat Paskin-Cherniavsky. 2015. Secure computation with minimal interaction, revisited. In *Advances in Cryptology–CRYPTO 2015: 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II 35*. Springer, 359–378.

[19] Marcel Keller. 2020. MP-SPDZ: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*. 1575–1590.

[20] Marcel Keller, Emmanuela Orsini, and Peter Scholl. 2016. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 830–842.

[21] Liisi Kerik, Peeter Laud, and Jaak Randmets. 2016. Optimizing MPC for robust and scalable integer and floating-point arithmetic. In *International Conference on Financial Cryptography and Data Security*. Springer, 271–287.

[22] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. 2021. SWIFT: Super-fast and Robust Privacy-Preserving Machine Learning.. In *USENIX Security Symposium*. 2651–2668.

[23] Nishat Koti, Arpita Patra, Rahul Rachuri, and Ajith Suresh. 2021. Tetrad: Actively secure 4pc for secure training and inference. *arXiv preprint arXiv:2106.02850* (2021).

[24] Yehuda Lindell. 2020. Secure Multiparty Computation (MPC). *IACR Cryptol. ePrint Arch.* 2020 (2020), 300.

[25] Yehuda Lindell and Ariel Nof. 2017. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 259–276.

[26] Darius Mercadier and Pierre-Évariste Dagand. 2019. Usuba: high-throughput and constant-time ciphers, by construction. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 157–173.

[27] Darius Mercadier, Pierre-Évariste Dagand, Lionel Lacassagne, and Gilles Muller. 2018. Usuba: optimizing & trustworthy bitslicing compiler. In *Proceedings of the 2018 4th Workshop on Programming Models for SIMD/Vector Processing*. 1–8.

[28] Payman Mohassel and Peter Rindal. 2018. ABY3: A mixed protocol framework for machine learning. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 35–52.

[29] Arpita Patra and Divya Ravi. 2018. On the exact round complexity of secure three-party computation. In *Advances in Cryptology–CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part II 38*. Springer, 425–458.

[30] Arpita Patra and Ajith Suresh. 2020. BLAZE: blazing fast privacy-preserving machine learning. *arXiv preprint arXiv:2005.09042* (2020).

[31] Mariana Raykova. 2012. *Secure Computation in Heterogeneous Environments: How to Bring Multiparty Computation Closer to Practice?* Columbia University.

[32] 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. *Information Sciences* 476 (2019), 357–372.

[33] Xing Zhou, Zhilei Xu, Cong Wang, and Mingyu Gao. 2022. PPMLAC: high performance chipset architecture for secure multi-party computation. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 87–101.

A CORRECTNESS

We unfold all computations of our 3-PC and 4-PC multiplication protocols to show that each party obtains a valid share. Our 4-PC protocols achieve security with abort against an adversary $\mathcal{A}$ corrupting a single party. Thus, we also show that each message sent by $\mathcal{A}$ can be verified by a set of honest parties using Π_{CV}.

A.1 3-PC Protocol

We unfold all computations in Π_{Mult} and show that parties obtain the correct shares of the product $c = ab$.

$$P_1 : \tag{21}$$

$$c_2 = m^2 - c_2' \tag{22}$$

$$= a_1 b_1 + m^0 + z_2 - (a_2 y_1 + b_2 x_1 + r_{0,1}) \tag{23}$$

$$= ab + a y_1 + b x_1 + x_1 y_1 + z_2 \tag{24}$$

$$+ m^0 - a y_1 - b x_1 - x_1 y_2 - x_2 y_1 - r_{0,1} \tag{25}$$

$$= ab + x_1 y_1 - x_1 y_2 - x_2 y_1 - r_{0,1} + z_2 + m^0 \tag{26}$$

$$= ab + x_1 y_1 - x_1 y_2 - x_2 y_1 - r_{0,1} + z_2 \tag{27}$$

$$+ x_2 y_2 - (x_1 y_1 - x_2 y_1 - x_1 y_2 + x_2 y_2) + r_{0,1} \tag{28}$$

$$= ab + z_2 \tag{29}$$

$$P_2: \tag{30}$$
$$c_1 = c_1' - m^1 \tag{31}$$
$$= a_1b_1 + m^0 - (a_2y_1 + b_2x_1 + r_{0,1} - z_1) \tag{32}$$
$$= ab + ay_1 + bx_1 + x_1y_1 + z_1 \tag{33}$$
$$+ m^0 - ay_1 - bx_1 - x_1y_2 - x_2y_1 - r_{0,1} \tag{34}$$
$$= ab + x_1y_1 - x_1y_2 - x_2y_1 - r_{0,1} + z_1 + m^0 \tag{35}$$
$$= ab + x_1y_1 - x_1y_2 - x_2y_1 - r_{0,1} + z_1 \tag{36}$$
$$+ x_2y_2 - (x_1y_1 - x_2y_1 - x_1y_2 + x_2y_2) + r_{0,1} \tag{37}$$
$$= ab + z_1 \tag{38}$$

A.2 4-PC Protocol

We unfold all computations in Π_{Mult} and show that parties obtain the correct shares of the product $c = ab$.

$$P_0: \tag{39}$$
$$c_w = m^{2,1} - (c_w' + m^3) \tag{40}$$
$$= c_0' + r_{1,2,3} - (a_uy_0 + b_vx_0) - m^3 \tag{41}$$
$$= a_0b_0 - (a_uy_0 + b_vx_0) - m^3 + r_{1,2,3} \tag{42}$$
$$= ab + ay_0 + bx_0 + x_0y_0 \tag{43}$$
$$- ay_0 - bx_0 - uy_0 - vx_0 - m^3 + r_{1,2,3} \tag{44}$$
$$= ab + x_0y_0 - uy_0 - vx_0 - m^3 + r_{1,2,3} \tag{45}$$
$$= ab + x_0y_0 - uy_0 - vx_0 - x_0(y_0 - v) + y_0u + w \tag{46}$$
$$= ab + w \tag{47}$$
$$P_1: \tag{48}$$
$$c_0 = c_0'' - m^{2,0} = a_0b_0 - m^1 - m^{2,0} \tag{49}$$
$$= a_0b_0 - (a_0y_1 + b_0x_1 + r_{0,1,3}) - (a_0y_2 + b_0x_2 - m^0) \tag{50}$$
$$= a_0b_0 - (a_0y_0 + b_0x_0 + r_{0,1,3} - m^0) \tag{51}$$
$$= ab + ay_0 + bx_0 + x_0y_0 \tag{52}$$
$$- ay_0 - bx_0 - x_0y_0 - x_0y_0 - r_{0,1,3} + m^0 \tag{53}$$
$$= ab - x_0y_0 - r_{0,1,3} + m^0 = ab + z_0 \tag{54}$$
$$P_2: \tag{55}$$
$$c_0 = c_0''' - m^1 = a_0b_0 - m^1 - m^{2,0} = ab + z_0 \tag{56}$$

For all cases where $\mathcal{A}$ violates the protocol specifications, we show that proving the correctness of functionality Π_{Mult} reduces to proving the correctness of functionality Π_{CV}.

Case: $\mathcal{A} = P_0$

Corrupted message:	$m^{0\prime} = m^0 + e$
Reduction:	$\Pi_{\mathsf{CV}}(m^0, P_2, P_3)$

Case: $\mathcal{A} = P_1$

Corrupted message:	$m^{1\prime} = m^1 + e$
Reduction:	$\Pi_{\mathsf{CV}}(c_{w,0}, P_0, P_1, P_2)$

If $e \neq 0$, P_2 obtains $c_{w,0} + e$ and $\Pi_{\mathsf{CV}}(c_{w,0}, P_0, P_1, P_2)$ fails.

Case: $\mathcal{A} = P_2$

Corrupted message:	$m^{2,0\prime} = m^{2,0} + e^1$
	$m^{2,1\prime} = m^{2,1} + e^2$
Reduction:	$\Pi_{\mathsf{CV}}(c_{w,0}, P_0, P_1, P_2)$
	$\Pi_{\mathsf{CV}}(m^{2,1}, P_0, P_1)$

If $e^1 \neq 0$, P_1 obtains $c_{w,0} + e^1$ and $\Pi_{\mathsf{CV}}(c_{w,0}, P_0, P_1, P_2)$ fails. If $e^2 \neq e^1$, $\Pi_{\mathsf{CV}}(m^{2,1}, P_0, P_1)$ fails. Hence, any assignment $e^1 \neq 0 \vee e^2 \neq 0$ leads to abort.

Case: $\mathcal{A} = P_3$

Corrupted message:	$m^{3\prime} = m^3 + e$
Reduction:	$\Pi_{\mathsf{CV}}(c_{w,0}, P_0, P_1, P_2)$

If $e \neq 0$, P_0 obtains $c_{w,0} + e$ and $\Pi_{\mathsf{CV}}(c_{w,0}, P_0, P_1, P_2)$ fails.

A.3 Heterogeneous 4-PC Protocol

We unfold all computations in $\Pi_{\mathsf{Mult\_H}}$ and show that parties obtain the correct shares of the product $c = ab$.

$$P_0: c_w = m^{2,1} - z_0 = c_0 + w - z_0 = ab + w \tag{57}$$
$$P_1: c_0 = c_0' - m^{2,0} = a_0b_0 - m^1 - m^{2,0} = ab + z_0 \tag{58}$$
$$P_2: c_0 = c_0'' - m^1 = a_0b_0 - m^1 - m^{2,0} = ab + z_0 \tag{59}$$

For all cases where $\mathcal{A}$ violates the protocol specifications, we show that proofing correctness of functionality $\Pi_{\mathsf{Mult\_H}}$ reduces to proofing correctness of Π_{CV}.

Case: $\mathcal{A} = P_0$

Corrupted message:	$m^{0\prime} = m^0 + e$
Reduction:	$\Pi_{\mathsf{CV}}(m^0, P_2, P_3)$

Case: $\mathcal{A} = P_1$

Corrupted message:	$m^{1\prime} = m^1 + e$
Reduction:	$\Pi_{\mathsf{CV}}(v_3, P_0, P_3)$

If $e \neq 0$, P_2 obtains $m^{2,2\prime} = m^{2,2} + e$ and sends it to P_0. P_0 computes $v^{3\prime}$ based on $m^{2,2\prime}$. The following equation shows that $\Pi_{\mathsf{CV}}(v_3, P_0, P_3)$ fails.

$$v^{3\prime} = m^{2,2\prime} - c_w' = ay_0 + bx_0 - x_0y_0 - z_0 \tag{60}$$
$$+ r_{1,2,3} + e - a_uy_0 - b_vx_0 \tag{61}$$
$$= x_0y_0 - uy_0 - vx_0 + r_{1,2,3} - z_0 + e = v_3 + e \tag{62}$$

Case: $\mathcal{A} = P_2$

Corrupted message:	$m^{2,0\prime} = m^{2,0} + e^1$
	$m^{2,1\prime} = m^{2,1} + e^2$
	$m^{2,2\prime} = m^{2,2} + e^3$
Reduction:	$\Pi_{\mathsf{CV}}(v_3, P_0, P_3)$
	$\Pi_{\mathsf{CV}}(m^{2,1}, P_0, P_1)$
	$\Pi_{\mathsf{CV}}(m^{2,2}, P_0, P_1)$

If $e^1 \neq 0$, P_1 obtains $m^{2,1\prime} = m^{2,1} + e^1$ and $m^{2,2\prime} = m^{2,2} + e^1$. In that case, any other assignment than $e^1 = e^2 = e^3$ leads to abort due to $\Pi_{\mathsf{CV}}(m^{2,1}, P_0, P_1)$ or $\Pi_{\mathsf{CV}}(m^{2,2}, P_0, P_1)$. Assigning $e^1 = e^2 = e^3 \neq 0$ leads to P_0 obtaining a corrupted $v_3\prime$. Hence, $\Pi_{\mathsf{CV}}(v_3, P_0, P_3)$ fails.

If $e^1 = 0$, P_1 obtains $m^{2,1}$ and $m^{2,2}$. Hence, assigning $e^2 \neq 0$ or $e^3 \neq 0$ leads to abort due to $\Pi_{\mathsf{CV}}(m^{2,1}, P_0, P_1)$ or $\Pi_{\mathsf{CV}}(m^{2,2}, P_0, P_1)$.

Case: $\mathcal{A} = P_3$

Corrupted message:	-

B ADDITIONAL PROTOCOLS

The properties of our protocols, i.e. high tolerance to weak network links, low computational complexity, and best known communication complexity can be extended to other protocols as well.

B.1 Truncation

Similar to Tetrad [23], probabilistic truncation as proposed by ABY3 [28] can be built into our 3-PC and 4-PC multiplication protocols at no additional communication cost. We improve on state-of-the-art by requiring fewer local operations per party. Probabilistic truncation takes a value $v = x - r$ and a value r and outputs their correct truncated versions v^t and r^t with a high probability. We combine truncation with our multiplication protocols to obtain more efficient constructions.

B.1.1 3-PC. To truncate-multiply two values a and b in our 3-PC scheme, all parties need to obtain a share of $ab + e$ and e, where e is a masked error term.

Let x^t denote $\lfloor \frac{x}{2^t} \rfloor$. First, the parties obtain a share of e^t. To do so, P_0 computes $e = (x_1 - x_2)(y_1 - y_2) - x_2y_2 + r_{0,1} + r_{0,2}$ and locally truncates the result to obtain e^t. $P_{1,2}$ then sample z_1 and P_0 sends $e^t - z_1$ to P_2. The parties now hold the following shares of e: $c_1 = m^0 = e^t - z_1, x_1 = -z_1, a_2 = z_1, x_2 = -(e^t - z_1)$.

Now, the parties calculate a share of $ab + e$. P_2 calculates $m_2 = a_1b_1 + r_{0,2}$, while P_1 calculates $m_1 = a_2y_1 + b_2x_1 - r_{0,1}$. The parties exchange m_1 and m_2 to obtain $ab + e = m_2 - m_1$. Notice that $e = x_1y_2 + x_2y_1 - x_1y_1 + r_{0,1} + r_{0,2}$. P_1 and P_2 can locally truncate $ab + e$ to obtain $(ab + e)^t$. The parties' shares are now defined as $a_1 = (ab + e)^t, a_2 = (ab + e)^t, x_1 = 0, x_2 = 0$.

Finally, the parties locally subtract their shares to compute $(ab + e)^t - e^t$ and e^t. The parties obtain $a_1 = (ab + e)^t - m^0 = ab - e^t + z_1, x_1 = z_1, a_2 = (ab+e)^t - z_1, x_2 = m^0 = -e^t - z_1$. Note that adding $a_i + x_i$ results in $(ab+e)^t - e^t$ which is what we intended. Figure 11 implements the presented intuition in a computationally efficient way.

Protocol $\Pi_{\text{Mult}3T}([\text{a}], [\text{b}]) \rightarrow [c]$

Preprocessing:

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

$P_{0,1} : r_{0,1}, z_1$ $\quad$ $P_{0,2} : r_{0,2}$

(2) Locally compute:

$P_0 : z_2 = m^0 = [(x_1 - x_2)(y_1 - y_2) - x_2y_2 + r_{0,1} + r_{0,2}]^t - z_1$

(3) Communicate:

$P_0 \rightarrow P_2 : m^0$

Online:

(1) Locally compute:

$P_1 : m_1 = a_2y_1 + b_2x_1 - r_{0,1}$ $\quad$ $P_2 : m_2 = a_1b_1 + r_{0,2}$

(2) Communicate:

$P_1 \rightarrow P_2 : m^1$ $\quad$ $P_2 \rightarrow P_1 : m^2$

(3) Locally compute:

$P_1 : c_2 = [m^2 - m^1]^t - z_1$ $\quad$ $P_2 : c_1 = [m^2 - m^1]^t - m_0$

$P_2 : z_2 = m^0$

Figure 11: 3-PC multiplication protocol with truncation

B.1.2 4-PC. Truncation also comes for free in our 4-PC schemes. To truncate-multiply a value in our 4-PC schemes, P_1 and P_2 obtain a share of $ab + e$. To do so, they exchange $m^1 = a_0y_1 + b_0x_1 - r_{0,1,3}$ and $m^2 = a_0y_2 + b_0x_2 - r_{0,2,3}$. They locally compute $a_0b_0 - m^1 - m^2$ to obtain $ab - x_0y_0 + r_{0,1,3} + r_{0,2,3}$. They then locally truncate $ab + e$. The parties then define their shares of $[ab + e]^t$ as follows: $z_0 = 0, c_0 = [ab + e]^t, z_1 = 0, z_2 = 0$.

P_0 shares e^t with P_1 and P_2 similarly to our 3-PC protocol. P_0 sets $m_0 = z_1 - e^t$ and sends it to P_2. The parties then define their shares of e^t as follows: $z_0 = -e^t, c_0 = 0, z_1 = -z_1, z_2 = z_1 - e^t$. The parties can now set their shares of ab: $z_0 = e^t, c_0 = [ab + e]^t, z_1 = z_1, z_2 = e^t - z_1$.

P_0 also needs to obtain a share of ab. Thus, P_2 masks c_0 with w and sends it to P_0. All parties now hold their valid share. Figures 12 and 13 implement the presented intuition in a computationally efficient way.

Protocol $\Pi_{\text{Mult}4T}([\text{a}], [\text{b}]) \rightarrow [c]$

Preprocessing:

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

$P_{0,1,3} : r_{0,1,3}, z_1$ $\quad$ $P_{0,2,3} : r_{0,2,3}$ $\quad$ $P_{1,2,3} : r_{1,2,3}, w$

(2) Locally compute:

$P_0, P_3 : z_0 = [r_{0,1,3} + r_{0,2,3} - x_0y_0]^t$ $\quad$ $P_0, P_3 : m^0 = z_0 - z_1$

$P_3 : m^3 = x_0(y_0 - v) - y_0u - r_{0,1,2} - r_{0,2,3} + r_{1,2,3}$

(3) Communicate:

$P_0 \rightarrow P_2 : m^0$

$P_3 \rightarrow P_0 : m^3$

Online:

(1) Locally compute:

$P_1 : m^1 = a_0y_1 + b_0x_1 - r_{0,1,3}$ $\quad$ $P_2 : m^{2,0} = a_0y_2 + b_0x_2 - r_{0,2,3}$

$P_{1,2} : c_0' = a_0b_0$ $\quad$ $P_0 : v'^{1,2} = a_uy_0 + b_vx_0 + x_0y_0 + m^3$

$P_2 : z_2 = m^0$

(2) Communicate:

$P_1 \rightarrow P_2 : m^1, m^{2,0}$

(3) Locally compute:

$P_{1,2} : v'^{1,2} = m^1 + m^{2,0}$ $\quad$ $P_{1,2} : [c_0]^t = [c_0' - v'^{1,2}]^t$

$P_1, P_2 : m^{2,1} = [c_0]^t + w$ $\quad$ $P_1, P_2 : v^{1,2} = v'^{1,2} + r_{1,2,3}$

(4) Communicate:

$P_2 \rightarrow P_0 : m^{2,1}$

(5) Locally compute:

$P_0 : c_w = m^{2,1} - z_0$

(6) Compare views using Π_{CV}:

$P_0, P_1 : m^{2,1}$ $\quad$ $P_0, P_1, P_2 : v^{1,2}$ $\quad$ $P_2, P_3 : m^0$

Figure 12: 4-PC multiplication protocol with truncation

Protocol $\Pi_{\text{Mult}4HT}([\text{a}], [\text{b}]) \rightarrow [c]$

Preprocessing:

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

$$P_{0,1,3} : r_{0,1,3}, z_1 \qquad P_{0,2,3} : r_{0,2,3} \qquad P_{1,2,3} : r_{1,2,3}, w$$

(2) Locally compute:

$$P_0, P_3 : z_0 = [r_{0,1,3} + r_{0,2,3} - x_0 y_0]^t \qquad P_0, P_3 : m^0 = z_0 - z_1$$

$$P_3 : v^3 = x_0(y_0 - v) - y_0 u - r_{0,1,2} - r_{0,2,3} + r_{1,2,3}$$

(3) Communicate:

$$P_0 \to P_2 : m^0$$

Online:

(1) Locally compute:

$$P_1 : m^1 = a_0 y_1 + b_0 x_1 - r_{0,1,3} \qquad P_2 : m^{2,0} = a_0 y_2 + b_0 x_2 - r_{0,2,3}$$

$$P_1 : c_0' = a_0 b_0 - m^1 \qquad P_2 : c_0'' = a_0 b_0 - m^{2,0}$$

$$P_0 : c_w' = a_u y_0 + b_v x_0 + x_0 y_0 \qquad P_2 : z_2 = m^0$$

(2) Communicate:

$$P_1 \to P_2 : m^1 \qquad P_2 \to P_1 : m^{2,0}$$

(3) Locally compute:

$$P_1 : [c_0]^t = [c_0' - m^{2,0}]^t \qquad P_2 : [c_0]^t = [c_0'' - m^1]^t$$

$$P_1, P_2 : m^{2,1} = [c_0]^t + w \qquad P_1, P_2 : m^{2,2} = m^1 + m^{2,0} + r_{1,2,3}$$

(4) Communicate:

$$P_2 \to P_0 : m^{2,1} \qquad P_2 \to P_0 : m^{2,2}$$

(5) Locally compute:

$$P_0 : v^3 = m^{2,2} - c_w' \qquad P_0 : c_w = m^{2,1} - z_0$$

(6) Compare views using Π_{CV}:

$$P_0, P_1 : m^{2,1}, m^{2,2} \qquad P_0, P_3 : v^3 \qquad P_2, P_3 : m^0$$

Figure 13: 4-PC heterogeneous multiplication protocol with truncation

B.2 Arithmetic to Binary

To convert an arithmetic share $[a]^A$ to a boolean share $[c]^B$, the parties compute boolean shares of $[a+x_0]^B$ and $[-x_0]^B$. The parties then use a boolean adder to compute $[a+x_0]^B + [-x_0]^B$ to receive an XOR-sharing of $[c]^B$.

B.2.1 3-PC. To compute a share of $b = [-x_0]^B$, $P_{0,1}$ sample $r_{0,1}$ and P_0 sends $m^0 = [-x]^B \oplus r_{0,1}$ to P_2 in the preprocessing phase. The parties then define their shares as shown in figure 14. Each party locally computes a share of $c' = [a+x_0]^B$. The parties proceed to compute $[c]^B = [a+x_0]^B + [-x_0]^B$ using a Boolean adder.

Protocol $\Pi_{\text{A2B},3PC}([\text{a}]^{\text{A}}) \to [c]^B$

Preprocessing:

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

$$P_{0,1} : r_{0,1}$$

(2) Communicate:

$$P_0 \to P_2 : m^0 = (-x_0) \oplus r_{0,1}$$

(3) Locally compute:

$$P_{0,1} : y_1 = 0 \qquad P_{0,2} : y_2 = 0 \qquad P_{0,1} : z_1' = r_{0,1} \qquad P_{0,2} : z_2' = m^0$$

Online:

(1) Locally compute:

$$P_1 : b_2 = a_2 + x_1 \qquad P_2 : b_1 = a_2 + x_2 \qquad P_1 : c_2' = z_1 \qquad P_2 : c_1' = z_2$$

$$P_{0,1,2,3} : [c]^B = [b]^B + [c']^B$$

Figure 14: 3-PC Arithmetic to Binary Conversion

B.2.2 4-PC. Each party first obtains a share of $b = [-x_0]^B$. $P_{0,1,3}$ sample $r_{0,1,3}$ and P_0 sends $m^0 = (-x_0) \oplus r_{0,1,3}$ to P_2 in the preprocessing phase. P_3 compares its view of m^0 with P_2. The parties then define their shares of b as shown in figure 17. $P_{1,2}$ locally compute a share of $c' = [a + x_0]^A$. P_2 sends $m^2 = a + x_0 \oplus r_{1,2,3}$ to P_0. P_0 and P_2 compare their views of m^2. The parties then define their shares as shown in figure 17. The parties proceed to compute $[c^b] = [b]^b + [c']^b$ using a Boolean adder.

Protocol $\Pi_{\text{A2B},4PC}([\text{a}]^{\text{A}}) \to [c]^B$

Preprocessing:

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

$$P_{0,1,3} : r_{0,1,3} \qquad P_{1,2,3} : r_{1,2,3}$$

(2) Communicate:

$$P_0 \to P_2 : m^0 = (-x_0) \oplus r_{0,1,3}$$

(3) Locally compute:

$$P_1 : y_1 = 0 \qquad P_2 : y_2 = 0 \qquad P_0 : y_0 = 0 \qquad P_3 : v = r_{1,2,3}$$

$$P_1 : z_1' = r_{0,1,3} \qquad P_2 : z_2' = m^0 \qquad P_{0,3} : z_0 = (-x_0) \qquad P_3 : w = 0$$

Online:

(1) Communicate:

(2) Locally compute: $P_2 \to P_0 : m^2 = a_0 \oplus r_{1,2,3}$

$$P_{1,2} : b_0 = a_0 \qquad P_0 : b_u = m^2 \qquad P_{1,2} : c_0' = 0 P_0 : c_u' = z_0$$

$$P_{0,1,2,3} : [c]^B = [b]^B + [c']^B$$

(3) Compare views using Π_{CV}:

$$P_2, P_3 : m^0 \qquad P_0, P_1 : m^2$$

Figure 15: 4-PC Arithmetic to Binary Conversion

B.3 Bit to Arithmetic

To promote a shared bit $[a]^B = a_0 \oplus x_0$ in the boolean domain to a shared bit $[c]^A = c_0 + z_0$ in the arithmetic domain, the parties first locally construct an *XOR*-sharing of $[a]^A$ and $[b]^A$ with $[c]^A = [a]^A \oplus [b]^A$ in the arithmetic domain. Then, they perform a private *XOR* of the resulting shares in the arithmetic domain. Note that $c_0 \oplus z_0 = c_0 + z_0 - 2c_0 z_0$.

B.3.1 3-PC. Each party first obtains a share of $b = [x_0]^A$. $P_{0,1}$ sample $r_{0,1}$ and P_0 sends $m^0 = x_0 + r_{0,1}$ to P_2 in the preprocessing phase. The parties then define their shares of b as shown in figure 16. All parties locally compute a share of $c' = [a \oplus x_0]^A$ as shown in

figure 17. By computing an XOR of $[b]^A$ and $[c']^A$ in the arithmetic domain, the parties obtain an arithmetic share of a.

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

Preprocessing:

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

$$P_{0,1} : r_{0,1}$$

(2) Communicate:

$$P_0 \rightarrow P_2 : m^0 = x_0 + r_{0,1}$$

(3) Locally compute:

$$P_{0,1} : y_1 = 0 \quad P_{0,2} : y_2 = 0 \quad P_{0,1} : z_1' = r_{0,1} \quad z_2' = -m^0$$

Online:

(1) Locally compute:

$$P_1 : b_2 = a_2 \oplus x_1 \quad P_2 : b_1 = a_1 \oplus x_2 \quad P_1 : c_2' = -r0, 1 \quad P_2 : c_1' = m^0$$

$$P_{0,1,2,3} : [c]^A = [b]^A + [c']^A - 2[b]^A[c']^A$$

Figure 16: 3-PC Bit to Arithmetic

4-PC. Each party first obtains a share of $b = [x_0]^A$. $P_{0,1,3}$ sample $r_{0,1,3}$ and P_0 sends $m^0 = x_0 + r_{0,1,3}$ to P_2 in the preprocessing phase. P_3 compares its view of m^0 with P_2. The parties then define their shares of b as shown in figure 17. $P_{1,2}$ locally compute a share of $c' = [a \oplus x_0]^A$. P_2 sends $m^2 = a \oplus x_0 + r_{1,2,3}$ to P_0. P_0 and P_2 compare their views of m^2. The parties then define their shares of c as shown in figure 17. By computing an XOR of $[b]^A$ and $[c]^A$ in the arithmetic domain, the parties obtain an arithmetic share of a.

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

Preprocessing:

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

$$P_{0,1,3} : r_{0,1,3} \qquad P_{1,2,3} : r_{1,2,3}$$

(2) Communicate:

$$P_0 \rightarrow P_2 : m^0 = x_0 + r_{0,1,3}$$

(3) Locally compute:

$$P_1 : y_1 = 0 \quad P_2 : y_2 = 0 \quad P_0 : y_0 = 0 \quad P_3 : v = r_{1,2,3}$$
$$P_1 : z_1' = r_{0,1,3} \quad P_2 : z_2' = -m^0 \quad P_{0,3} : z_0' = -x_0 \quad P_3 : w' = 0$$

Online:

(1) Communicate:

(2) Locally compute: $P_2 \rightarrow P_0 : m^2 = a_0 + r_{1,2,3}$

$$P_{1,2} : b_0 = a_0 \quad P_0 : b_u = m^2 \quad P_{1,2} : c_0' = 0 P_0 : c_u = x_0$$

(3) Compare views using Π_{CV}:

$$P_2, P_3 : m^0 \qquad P_0, P_1 : m^2$$

$$P_{0,1,2,3} : [c]^A = [b]^A + [c']^A - 2[b]^A[c']^A$$

Figure 17: 4-PC Bit to Arithmetic

C RAM UTILIZATION

Our implementation provides the option to perform the preprocessing phase within the online phase. This has the advantage that a party does not need to receive all offline material at once but can do so in chunks as it evaluates the circuit. Also, this processing model interleaves the communication and computation of the preprocessing and the online phase, which leads to faster total runtimes than executing both phases sequentially. In our AES benchmark, we evaluate more than 90 million AES blocks and measure the peak RAM utilization of each node. Table 7 shows the measurement results. As expected, the RAM utilization increases when using a separate preprocessing phase, as all offline material is located in memory when the online phase starts. Also, the protocols not using preprocessing [2, 11] show better RAM utilization as the parties evaluate the circuit synchronously and do not buffer as many messages. Nevertheless, we see the advantage of our 4-PC protocol, which stores fewer shares per party than related work.

Table 7: Peak RAM Utilization in MB

Category	Protocols	RAM	RAM (Off → On)
3-PC	Replicated	636	-
	Astra	1880	2249
	Ours	1259	2251
4-PC	Fantastic Four	1853	-
	Tetrad	2980	3746
	Ours	1562	4258
TTP	Trusted Third Party	316	-