Performing machine learning (ML) computation on private data while maintaining data privacy, aka Privacy-preserving Machine Learning (PPML), is an emergent field of research. Recently, PPML has seen a visible shift towards the adoption of the Secure Outsourced Computation (SOC) paradigm due to the heavy computation that it entails. In the SOC paradigm, computation is outsourced to a set of powerful and specially equipped servers that provide service on a pay-per-use basis. In this work, we propose SWIFT, a robust PPML framework for a range of ML algorithms in SOC setting, that guarantees output delivery to the users irrespective of any adversarial behaviour. Robustness, a highly desirable feature, evokes user participation without the fear of denial of service.

At the heart of our framework lies a highly-efficient, maliciously-secure, three-party computation (3PC) over rings that provides guaranteed output delivery (GOD) in the honest-majority setting. To the best of our knowledge, SWIFT is the first robust and efficient PPML framework in the 3PC setting. SWIFT is as fast as (and is strictly better in some cases than) the best-known 3PC framework BLAZE (Patra et al. NDSS’20), which only achieves fairness. We extend our 3PC framework for four parties (4PC). In this regime, SWIFT is as fast as the best known fair 4PC framework Trident (Chaudhari et al. NDSS’20) and twice faster than the best-known robust 4PC framework FLASH (Byali et al. PETS’20).

We demonstrate our framework’s practical relevance by benchmarking popular ML algorithms such as Logistic Regression and deep Neural Networks such as VGG16 and LeNet, both over a 64-bit ring in a WAN setting. For deep NN, our results testify to our claims that we provide improved security guarantee while incurring no additional overhead for 3PC and obtaining $2 \times$ improvement for 4PC.
beforehand. To further enhance practical efficiency by leveraging CPU optimizations, recent works [7, 21, 24, 26, 28] propose MPC protocols that work over 32 or 64 bit rings. Lastly, solutions for a small number of parties have received a huge momentum due to the many cost-effective customizations that they permit, for instance, a cheaper realisation of multiplication through custom-made secret sharing schemes [3, 4, 12, 15, 16, 48].

We now motivate the need for robustness aka guaranteed output delivery (GOD) over fairness2, or even abort security3, in the domain of PPML. Robustness provides the guarantee of output delivery to all protocol participants, no matter how the adversary misbehaves. Robustness is crucial for real-world deployment and usage of PPML techniques. Consider the following scenario wherein an ML model owner wishes to provide inference service. The model owner shares the model parameters between the servers, while the end-users share their queries. A protocol that provides security with abort or fairness will not suffice as in both the cases a malicious adversary can lead to the protocol aborting, resulting in the user not obtaining the desired output. This leads to denial of service and heavy economic losses for the service provider. For data providers, as more training data leads to more accurate models, collaboratively building a model enables them to provide better ML services, and consequently, attract more clients. A robust framework encourages active involvement from multiple data providers. Hence, for the seamless adoption of PPML solutions in the real world, the robustness of the protocol is of utmost importance. Several works [15, 16, 43, 48, 56] realizing PPML via MPC settle for weaker guarantees such as abort and fairness. Achieving the strongest notion of GOD without degrading performance is an interesting goal which is the focus of this work. The hallmark result of [18] suggests that an honest-majority amongst the servers is necessary to achieve robustness. Consequent to the discussion above, we focus on the honest-majority setting with a small set of parties, especially 3 and 4 parties, both of which have drawn enormous attention recently [3, 4, 9, 10, 12, 14–16, 31, 44, 46, 48].

The 3/4-party setting enables simpler, more efficient, and customized secure protocols compared to the n-party setting. Real-world MPC applications and frameworks such as the Danish sugar beet auction [8] and Sharemind [7], have demonstrated the practicality of 3-party protocols. Additionally, in an outsourced setting, 3/4PC is useful and relevant even when there are more parties. Specifically, here the entire computation is offloaded to 3/4 hired servers, after initial sharing of inputs by the parties amongst the servers. This is precisely what we (and some existing papers [12, 42, 48]) contemplate as the setting for providing ML-as-a-service. Our protocols work over rings, are cast in the preprocessing paradigm, and achieve GOD.

Related Work We restrict the relevant work to a small number of parties and honest-majority, focusing first on MPC, followed by PPML. MPC protocols for a small population can be cast into orthogonal domains of low latency protocols [13, 14, 47], and high throughput protocols [1, 3, 4, 7, 10, 15, 17, 30, 31, 46, 48]. In the 3PC setting, [4, 15] provide efficient semi-honest protocols wherein ASTRA [15] improved upon [4] by casting the protocols in the preprocessing model and provided a fast online phase. ASTRA further provided security with fairness in the malicious setting with an improved online phase compared to [3]. Later, a maliciously-secure 3PC protocol based on distributed zero-knowledge techniques was proposed by Boneh et al. [9] providing abort security. Further, building on [9] and enhancing the security to GOD, Boyle et al. [10] proposed a concretely efficient 3PC protocol with an amortized communication cost of 3 field elements (can be extended to work over rings) per multiplication gate. Concurrently, BLAZE [48] provided a fair protocol in the preprocessing model, which required communicating 3 ring elements in each phase. However, BLAZE eliminated the reliance on the computationally intensive distributed zero-knowledge system (whose efficiency kicks in for large circuit or many multiplication gates) from the online phase and pushed it to the preprocessing phase. This resulted in a faster online phase compared to [10].

In the regime of 4PC, Gordon et al. [32] presented protocols achieving abort security and GOD. However, [32] relied on expensive public-key primitives and broadcast channels to achieve GOD. Trident [16] improved over the abort protocol of [32], providing a fast online phase achieving security with fairness, and presented a framework for mixed world computations [28]. A robust 4PC protocol was provided in FLASH [12], which requires communicating 6 ring elements, each, in the preprocessing and online phases.

In the PPML domain, MPC has been used for various ML algorithms such as Decision Trees [40], Linear Regression [29, 51], k-means clustering [11, 33], SVM Classification [55, 58], Logistic Regression [53]. In the 3PC SOC setting, the works of ABY3 [43] and SecureNN [56], provide security with abort. This was followed by ASTRA [15], which improves upon ABY3 and achieves security with fairness. ASTRA presents primitives to build protocols for Linear Regression and Logistic Regression inference. Recently, BLAZE improves over the efficiency of ASTRA and additionally tackles training for the above ML tasks, which requires building additional PPML building blocks, such as truncation and bit to arithmetic conversions. In the 4PC setting, the first robust framework for PPML was provided by FLASH [12] which proposed efficient building blocks for ML such as dot product, truncation, MSB extraction, and bit conversion. The works of [12, 15, 16, 43, 45, 48, 56] work over rings to garner practical efficiency. In terms of efficiency, BLAZE and respectively FLASH and Trident are the closest competitors of this work in 3PC and 4PC settings.

---

2This ensures either all parties or none learn the output.

3This may allow the corrupt parties alone to learn the output.
1.1 Our Contributions

We propose, **SWIFT**, a robust maliciously-secure framework for PPML in the SOC setting, with a set of 3 and 4 servers having an honest-majority. At the heart of our framework lies highly-efficient, maliciously-secure, 3PC and 4PC over rings (both \( \mathbb{Z}_2 \) and \( \mathbb{Z}_{2^l} \)) that provide GOD in the honest-majority setting. We cast our protocols in the preprocessing model, which helps obtain a fast online phase. As mentioned earlier, the input-independent (yet function-dependent) computations will be performed in the preprocessing phase.

To the best of our knowledge, SWIFT is the first robust and efficient PPML framework in the 3PC setting and is as fast as (and is strictly better in some cases than) the best known fair 3PC framework BLAZE [48]. We extend our 3PC framework for 4 servers. In this regime, SWIFT is as fast as the best known fair 4PC framework Trident [16] and twice faster than best known robust 4PC framework FLASH [12]. We detail our contributions next.

**Robust 3/4PC frameworks** The framework consists of a range of primitives realized in a privacy-preserving way which is ensured via running computation in a secret-shared fashion. We use secret-sharing over both \( \mathbb{Z}_2 \) and its special instantiation \( \mathbb{Z}_{2^l} \) and refer them as arithmetic and respectively boolean sharing. Our framework consists of realizations for all primitives needed for general MPC and PPML such as multiplication, dot-product, truncation, bit extraction (given arithmetic sharing of a value \( v \), this is used to generate boolean sharing of the most significant bit (msb) of the value), bit to arithmetic sharing conversion (converts the boolean sharing of a single bit value to its arithmetic sharing), bit injection (computes the arithmetic sharing of \( b \cdot v \), given the boolean sharing of a bit \( b \) and the arithmetic sharing of a ring element \( v \)) and above all, input sharing and output reconstruction in the SOC setting. A highlight of our 3PC framework, which, to the best of our knowledge is achieved for the first time, is a robust dot-product protocol whose (amortized) communication cost is independent of the vector size, which we obtain by extending the techniques of [9, 10]. The performance comparison in terms of concrete cost for communication and rounds, for PPML primitives in both 3PC and 4PC setting, appear in Table 1. As claimed, SWIFT is on par with BLAZE for most of the primitives (while improving security from fair to GOD) and is strictly better than BLAZE in case of dot product and dot product with truncation. For 4PC, SWIFT is on par with Trident in most cases (and is slightly better for dot product with truncation and bit injection), while it is doubly faster than FLASH. Since BLAZE outperforms the 3PC abort framework of ABY3 [43] while Trident outperforms the known 4PC with abort [32], SWIFT attains robustness with better cost than the know protocols with weaker guarantees. No performance loss coupled with the strongest security guarantee makes our robust framework an opt choice for practical applications including PPML.

**Applications and Benchmarking** We demonstrate the practicality of our protocols by benchmarking PPML, particularly, Logistic Regression (training and inference) and popular Neural Networks (inference) such as [45], LeNet [38] and VGG16 [52] having millions of parameters. The NN training requires mixed-world conversions [16, 28, 43], which we leave as future work. Our PPML blocks can be used to perform training and inference of Linear Regression, Support Vector Machines, and Binarized Neural Networks (as demonstrated in [12, 15, 16, 48]).

**New techniques and Comparisons with Prior Works** To begin with, we introduce a new primitive called Joint Message Passing (jmp) that allows two servers to relay a common message to the third server such that either the relay is successful or an honest server is identified. The identified honest party enacts the role of a trusted third party (TTP) to take the computation to completion. jmp is extremely efficient as for a message of \( \ell \) elements it only incurs the minimal communication cost of \( \ell \) elements (in an amortized sense). Without any extra cost, it allows us to replace several pivotal private communications, that may lead to abort, either because the malicious sender does not send anything or sends a wrong message. All our primitives, either for a general 3PC or a PPML task, achieve GOD relying on jmp.

Second, instead of using the multiplication of [10] (which has the same overall communication cost as that of our online phase), we build a new protocol. This is because the former involves distributed zero-knowledge protocols. The cost of this heavy machinery gets amortized only for large circuits having millions of gates, which is very unlikely for inference and moderately heavy training tasks in PPML. As in BLAZE [48], we follow a similar structure for our multiplication protocol but differ considerably in techniques as our goal is to obtain GOD. Our approach is to manipulate and transform some of the protocol steps so that two other servers can locally compute the information required by a server in a round. However, this transformation is not straightforward since BLAZE was constructed with a focus towards providing only fairness (details appear in §3). The multiplication protocol forms a technical basis for our dot product protocol and other PPML building blocks. We emphasise again that the (amortized) cost of our dot product protocol is independent of the vector size.

Third, extending to 4PC brings several performance improvements over 3PC. Most prominent of all is a conceptually simple jmp instantiation, which forgoes the broadcast channel while retaining the same communication cost; and a dot product with cost independent of vector size sans the 3PC amortization technique.

Fourth, we provide robust protocols for input sharing and output reconstruction phase in the SOC setting, wherein a user shares its input with the servers, and the output is reconstructed towards a user. The need for robustness and commu-
nication efficiency together makes these tasks slightly non-trivial. As a highlight, we introduce a super-fast online phase for the reconstruction protocol, which gives $4 \times$ improvement in terms of rounds (apart from improvement in the communication) compared to BLAZE. Although we aim for GOD, we ensure that an end-user is never part of broadcast which is relatively expensive than atomic point-to-point communication.

As a final remark, we note that the recent work of [23] proposes a variant of GOD in the 4PC setting, which is termed as private robustness. The authors of [23] state that private robustness is a variant of GOD which guarantees that the correct output is produced in the end, but without relying on an honest party learning the user’s private inputs. Thus, departing from the approach of employing a TTP to complete the computation when malicious behaviour is detected, [23] attains GOD by eliminating a potentially corrupt party, and repeating the secure computation with fewer number of parties which are deemed to be honest. We point out a few concerns on this work. Firstly, as mentioned earlier, the goal of private robustness is to prevent an honest party learning the user’s input, thereby preventing it from misusing this private information (user’s input) in the future if it goes rogue. We note, however, that in the private robustness setting, although an honest party does not learn a user’s input as a part of the protocol, nothing prevents an adversary from revealing its view to an honest party. In such a scenario, if the honest party goes rogue in the future, it can use its view together with the view received from the adversary to obtain a user’s input. Hence, we believe that the attacks that can be launched in our variant for achieving GOD, can also be launched in the private robustness variant, making the two equivalent. Secondly and importantly, a formal treatment of private robustness is missing in [23] which makes it unclear what additional security is achieved on top of traditional GOD security. Here, we additionally note that the notion of private robustness does not comply with the recently introduced notion of FaF security [2] in [23]. Lastly, we emphasize that the approach of eliminating a potentially corrupt party, and re-running the computation results in doubling or tripling the communication cost, thereby undermining the efficiency gains.

### 1.2 Organisation of the paper

The rest of the paper is organized as follows. In §2 we describe the system model, preliminaries and notations used. §3 and §4 detail our constructs in the 3PC and 4PC setting respectively.

---

**Table 1: 3PC and 4PC: Comparison of SWIFT with its closest competitors in terms of Communication and Round Complexity**

<table>
<thead>
<tr>
<th>Building Blocks</th>
<th>3PC</th>
<th>4PC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Multiplication</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ref.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[9]</td>
<td>[10]</td>
<td></td>
</tr>
<tr>
<td>Pre.</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Online</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Comm. (ℓ)</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Rounds</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comet. (ℓ)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Security</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abort</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GOD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flash</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SWIFT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trident</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Fair</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GOD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SWIFT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trinity</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>Fair</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GOD</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>SWIFT</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Fair</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>GOD</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>SWIFT</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Security</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abort</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GOD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SWIFT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trinity</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>Fair</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GOD</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>SWIFT</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Fair</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>GOD</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>SWIFT</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Security</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abort</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GOD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SWIFT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trinity</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>Fair</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GOD</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>SWIFT</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Fair</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>GOD</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>SWIFT</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

---

-Notations: ℓ - size of ring in bits, n - size of vectors for dot product.
We consider a set of three servers $\mathcal{P} = \{P_0, P_1, P_2\}$ that are connected by pair-wise private and authentic channels in a synchronous network, and a static, malicious adversary that can corrupt at most one server. We use a broadcast channel for 3PC alone, which is inevitable [19]. For ML training, several data-owners who wish to jointly train a model, secret share (using the sharing semantics that will appear later) their data among the servers. For ML inference, a model-owner and client secret share the model and the query, respectively, among the servers. Once the inputs are available in the shared format, the servers perform computations and obtain the output in the shared form. In the case of training, the output model is reconstructed towards the data-owners, whereas for inference, the prediction result is reconstructed towards the client. We assume that an arbitrary number of data-owners may collude with a corrupt server for training, whereas for the case of prediction, we assume that either the model-owner or the client can collude with a corrupt server. We prove the security of our protocols using a standard real-world / ideal-world paradigm. We also explore the above model for the four server setting with $\mathcal{P} = \{P_0, P_1, P_2, P_3\}$. The aforementioned setting has been explored extensively [12, 15, 16, 43, 45, 48].

Our constructions achieve the strongest security guarantee of GOD. A protocol is said to be robust or achieve GOD if all parties obtain the output of the protocol regardless of how the adversary behaves. In our model, this translates to all the data owners obtaining the trained model for the case of ML training, while the client obtaining the query output for ML inference. All our protocols are cast into: input-independent preprocessing phase and input-dependent online phase.

For 3/4PC, the function to be computed is expressed as a circuit ckt, whose topology is public, and is evaluated over an arithmetic ring $\mathbb{Z}_2^\ell$ or boolean ring $\mathbb{Z}_2$. For PPML, we consider computation over the same algebraic structure. To deal with floating-point values, we use Fixed-Point Arithmetic (FPA) [12, 15, 16, 43, 45, 48] representation in which a decimal value is represented as an $FPA$ [12, 15, 16, 43, 45, 48] representation in which a $\ell$-bit integer is treated as an $\ell$-bit integer in signed 2’s complement representation. The most significant bit (MSB) represents the sign bit, and $x$ least significant bits are reserved for the fractional part. The $\ell$-bit integer is then treated as an element of $\mathbb{Z}_2^\ell$, and operations are performed modulo $2^\ell$. We set $\ell = 64$, $x = 13$, leaving $\ell - x - 1$ bits for the integer part.

The servers use a one-time key setup, modelled as a functionality $J_{\text{setup}}$ (Fig. 27), to establish pre-shared random keys for pseudo-random functions (PRF) between them. A similar setup is used in [3, 10, 15, 31, 43, 48, 50] for three server case and in [12, 16] for four server setting. The key-setup can be instantiated using any standard MPC protocol in the respective setting. Further, our protocols make use of a collision-resistant hash function, denoted by $H()$, and a commitment scheme, denoted by $\text{Com}()$. The formal details of key setup, hash function, and commitment scheme are deferred to §A.

**Notation 2.1.** The $i^{th}$ element of a vector $\mathbf{x}$ is denoted as $x_i$. The dot product of two $n$ length vectors, $\mathbf{x}$ and $\mathbf{y}$, is computed as $\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^{n} x_i y_i$. For two matrices $\mathbf{X}, \mathbf{Y}$, the operation $\mathbf{X} \circ \mathbf{Y}$ denotes the matrix multiplication. The bit in the $i^{th}$ position of an $\ell$-bit value $v$ is denoted by $v[i]$.

**Notation 2.2.** For a bit $b \in \{0, 1\}$, we use $b^R$ to denote the equivalent value of $b$ over the ring $\mathbb{Z}_2$. $b^R$ will have its least significant bit set to $b$, while all other bits will be set to zero.

### 3 Robust 3PC and PPML

In this section, we first introduce the sharing semantics for three servers. Then, we introduce our new Joint Message Passing (jmp) primitive, which plays a crucial role in obtaining the strongest security guarantee of GOD, followed by our protocols in the three server setting.

#### Secret Sharing Semantics

We use the following secret-sharing semantics.

- $[\cdot]$-sharing: A value $v \in \mathbb{Z}_2^\ell$ is $[\cdot]$-shared among $P_1, P_2$, if $P_3$ for $s \in \{1, 2\}$ holds $[v]_s \in \mathbb{Z}_2^\ell$ such that $v = [v]_1 + [v]_2$.
- $\langle \cdot \rangle$-sharing: A value $v \in \mathbb{Z}_2^\ell$ is $\langle \cdot \rangle$-shared among $\mathcal{P}$, if there exists $v_0, v_1, v_2 \in \mathbb{Z}_2^\ell$ such that $v = v_0 + v_1 + v_2$.
- $P_s$ holds $[v_i, v_{(s+1) \% 3}]$ for $s \in \{0, 1, 2\}$.
- $[\cdot]$-sharing: A value $v \in \mathbb{Z}_2^\ell$ is $[\cdot]$-shared among $\mathcal{P}$, if there exists $\alpha_v \in \mathbb{Z}_2^\ell$ that is $[\cdot]$-shared among $P_1, P_2$.
- There exists $\beta_v, \gamma_v \in \mathbb{Z}_2^\ell$ such that $\beta_v = v + \alpha_v$ and $P_0$ holds $([\alpha_v]_1, [\alpha_v]_2, \beta_v, \gamma_v)$ while $P_s$ for $s \in \{1, 2\}$ holds $([\alpha_v]_s, \beta_v, \gamma_v)$.

#### Arithmetic and Boolean Sharing

**Arithmetic sharing** refers to sharing over $\mathbb{Z}_2^\ell$ while **boolean sharing**, denoted as $[\cdot]^B$, refers to sharing over $\mathbb{Z}_2$.

**Linearity of the Secret Sharing Scheme**

Given $[\cdot]$-shares of $v_1, v_2$, and public constants $c_1, c_2$, servers can locally compute $[\cdot]$-share of $c_1 v_1 + c_2 v_2$ as $[c_1 [v_1] + c_2 [v_2]]$. It is trivial to see that linearity property is satisfied by $\langle \cdot \rangle$ and $[\cdot]^B$ sharings.

### 3.1 Joint Message Passing primitive

The jmp primitive allows two servers to relay a common message to the third server such that either the relay is successful or an honest server (or a conflicting pair) is identified. The striking feature of jmp is that it offers a rate-1 communication i.e. for a message of $\ell$ elements, it only incurs a communication of $\ell$ elements (in an amortized sense). The task of jmp is
captured in an ideal functionality (Fig. 1) and the protocol for the same appears in Fig. 2. Next, we give an overview.

Given two servers \( P_i, P_j \) possessing a common value \( v \in \mathbb{Z}_{2^t} \), protocol \( \Pi_{jmp} \) proceeds as follows. First, \( P_i \) sends \( v \) to \( P_k \) while \( P_j \) sends a hash of \( v \) to \( P_k \). The communication of the hash is done once and for all from \( P_j \) to \( P_k \). In the simplest case, \( P_k \) receives a consistent (value, hash) pair, and the protocol terminates. In all other cases, a TTP is identified as follows without having to communicate \( v \) again. Importantly, the following part can be run once and for all instances of \( \Pi_{jmp} \) with \( P_i, P_j, P_k \) in the same roles, invoked in the final 3PC protocol. Consequently, the cost relevant to this part vanishes in an amortized sense, making the construction rate-1.

**Functionality \( \mathcal{F}_{jmp} \)**

\( \mathcal{F}_{jmp} \) interacts with the servers in \( \mathcal{P} \) and the adversary \( S \).

**Step 1:** \( \mathcal{F}_{jmp} \) receives (Input, \( v_i \)) from \( P_i \) for \( i \in \{i, j, k\} \), while it receives (Select, tftp) from \( S \). Here \( ttp \) denotes the server that \( S \) wants to choose as the TTP. Let \( P^* \in \mathcal{P} \) denote the server corrupted by \( S \).

**Step 2:** If \( v_i = v_j \) and tftp = \( \perp \), then set \( msg_i = msg_j = \perp, msg_k = v_i \) and go to **Step 5**.

**Step 3:** If tftp \( \in \mathcal{P} \setminus \{P^*\} \), then set \( msg_i = msg_j = msg_k = \text{tftp} \).

**Step 4:** Else, \( \mathcal{F}_{jmp} \) is set to be the honest server with smallest index. Set \( msg_i = msg_j = msg_k = \text{TTP} \).

**Step 5:** Send (Output, \( msg_s \)) to \( P_s \) for \( s \in \{0, 1, 2\} \).

---

**Notation 3.1.** We say that \( P_i, P_j \) jmp-send \( v \) to \( P_k \) when they invoke \( \Pi_{jmp}(P_i, P_j, P_k, v) \).

---

**Protocol \( \Pi_{jmp}(P_i, P_j, P_k, v) \)**

Each server \( P_s \) for \( s \in \{i, j, k\} \) initializes bit \( b_s = 0 \).

- **Send Phase:** \( P_i \) sends \( v \) to \( P_k \).
- **Verify Phase:** \( P_j \) sends \( H(v) \) to \( P_k \).
  - \( P_k \) broadcasts \( (\text{accuse}, P_i) \)∗, if \( P_i \) is silent and \( \text{TTP} = P_j \).
  - Analogously for \( P_j \). If \( P_j \) accuses both \( P_i, P_j \), then \( \text{TTP} = P_i \).
  - Otherwise, \( P_k \) receives some \( v \) and either sets \( b_2 = 0 \) when the value and the hash are consistent or sets \( b_k = 1 \). \( P_k \) then sends \( b_k \) to \( P_i, P_j \) and terminates if \( b_2 = 0 \).
  - If \( P_i \) does not receive a bit from \( P_k \), it broadcasts \( (\text{accuse}, P_k) \)∗ and \( \text{TTP} = P_i \).
  - Analogously for \( P_j \). If both \( P_i, P_j \) accuse \( P_k \), then \( \text{TTP} = P_i \).
  - Otherwise, \( P_i \) for \( s \in \{i, j\} \) sets \( b_s = b_i \).
  - \( P_i, P_j \) exchange their bits to each other. If \( P_i \) does not receive \( b_j \) from \( P_j \), it broadcasts \( (\text{accuse}, P_j) \)∗ and \( \text{TTP} = P_k \).
  - Analogously for \( P_j \). Otherwise, \( P_i \) resets its bit to \( b_i \lor b_j \) and likewise \( P_j \) resets its bit to \( b_i \lor b_j \).
  - \( P_i \) for \( s \in \{i, j, k\} \) broadcasts \( H_s = H(v^s) \) if \( b_s = 1 \), where \( v^s = v \) for \( s \in \{i, j\} \) and \( v^s = \perp \) otherwise. If \( P_i \) does not broadcast, terminate. If either \( P_i \) or \( P_j \) does not broadcast, then \( \text{TTP} = P_k \).
- Otherwise:
  - If \( H_i \neq H_j \): \( \text{TTP} = P_k \).
  - Else if \( H_i \neq H_k \): \( \text{TTP} = P_j \).
  - Else if \( H_i = H_j = H_k \): \( \text{TTP} = P_i \).

---

**Figure 2:** 3PC: Joint Message Passing Protocol

**Using jmp in protocols.** As mentioned in the introduction, the jmp protocol needs to be viewed as consisting of two phases (send, verify), where send phase consists of \( P_i \) sending \( v \) to \( P_k \) and the rest goes to verify phase. Looking ahead, most of our protocols use jmp, and consequently, our final construction, either of general MPC or any PPML task, will have several calls to jmp. To leverage amortization, the send phase will be executed in all protocols invoking jmp on the flow, while the verify for a fixed ordered pair of senders will be executed once and for all in the end. The verify phase will determine if all the sends were correct. If not, a TTP is identified, as explained, and the computation completes with the help of TTP, just as in the ideal-world.

**Lemma 3.2 (Communication).** Protocol \( \Pi_{jmp} \) (Fig. 2) requires 1 round and an amortized communication of \( \ell \) bits overall.
**Proof.** Server $P_i$ sends value $v$ to $P_j$ while $P_j$ sends hash of the same to $P_k$. This accounts for one round and communication of $\ell$ bits. $P_k$ then sends back its inconsistency bit to $P_i, P_j$, who then exchange it; this takes another two rounds. This is followed by servers broadcasting hashes on their values and selecting a TTP based on it, which takes one more round. All except the first round can be combined for several instances of $\Pi_{\text{jmp}}$ protocol and hence the cost gets amortized. \hfill \Box

### 3.2 3PC Protocols

We now describe the protocols for 3 parties/servers and defer the security proofs to §C.1.

**Sharing Protocol** Protocol $\Pi_{\text{sh}}$ (Fig. 3) allows a server $P_i$ to generate $[\cdot]$-shares of a value $v \in \mathbb{Z}_{2^\ell}$. In the preprocessing phase, $P_0, P_j$ for $j \in \{1, 2\}$ along with $P_i$ sample a random $[\alpha_1] \in \mathbb{Z}_{2^\ell}$, while $P_j, P_2, P_i$ sample random $\gamma_j \in \mathbb{Z}_{2^\ell}$. This allows $P_i$ to know both $\alpha_j$ and $\gamma_j$ in clear. During the online phase, if $P_i = P_0$, then $P_i$ sends $\gamma_j = v + \alpha_j$ to $P_0, P_j, P_0$ then jmp-send $\gamma_j$ to $P_2$ to complete the secret sharing. If $P_i = P_1$, $P_i$ sends $\gamma_j = v + \alpha_j$ to $P_2$. Then $P_i, P_2$ jmp-send $\gamma_j$ to $P_0$. The case for $P_i = P_2$ proceeds similar to that of $P_1$. The correctness of the shares held by each server is assured by the guarantees of $\Pi_{\text{jmp}}$.

**Protocol $\Pi_{\text{sh}}(P_i, v)$**

**Preprocessing:**
- If $P_i = P_0 \neq P_0, P_j$, for $j \in \{1, 2\}$, together sample random $[\alpha_1] \in \mathbb{Z}_{2^\ell}$, while $P_i$ together sample random $\gamma_j \in \mathbb{Z}_{2^\ell}$.
- If $P_i = P_1 \neq P_0, P_j$, together sample random $[\alpha_1] \in \mathbb{Z}_{2^\ell}$, while $P_i$ together sample a random $[\alpha_1] \in \mathbb{Z}_{2^\ell}$. Also, $P_i, P_2$ together sample random $\gamma_j \in \mathbb{Z}_{2^\ell}$.
- If $P_i = P_2$: Symmetric to the case when $P_i = P_1$.

**Online:**
- If $P_i = P_0$, $P_i$ computes $\gamma_j = v + \alpha_j$ and sends $\gamma_j$ to $P_1, P_2$.
- If $P_i = P_2$, for $j \in \{1, 2\}$: $P_j$ computes $\gamma_j = v + \alpha_j$, sends $\gamma_j$ to $P_{3-j}$, $P_j, P_2$ jmp-send $\gamma_j$ to $P_0$.

Figure 3: 3PC: Generating $[\cdot]$-shares by server $P_i$

**Lemma 3.3** (Communication). Protocol $\Pi_{\text{sh}}$ (Fig. 3) is non-interactive in the preprocessing phase and requires 2 rounds and an amortized communication of $2\ell$ bits in the online phase.

**Proof.** During the preprocessing phase, servers non-interactively sample the $[\cdot]$-shares of $\alpha_j$ and $\gamma_j$ values using the shared key setup. In the online phase, when $P_i = P_0$, it computes $\gamma_j$ and sends it to $P_1$, resulting in one round and $\ell$ bits communicated. They then jmp-send $\gamma_j$ to $P_2$, which requires additional one round in an amortized sense, and $\ell$ bits to be communicated. For the case when $P_i = P_1$, it sends $\gamma_j$ to $P_2$, resulting in one round and a communication of $\ell$ bits. Then, $P_1, P_2$ jmp-send $\gamma_j$ to $P_0$. This again requires an additional one round and $\ell$ bits. The analysis is similar in the case of $P_i = P_2$. \hfill \Box

**Joint Sharing Protocol** Protocol $\Pi_{\text{sh}}$ (Fig. 4) allows two servers $P_i, P_j$ to jointly generate a $[\cdot]$-sharing of a value $v \in \mathbb{Z}_{2^\ell}$ that is known to both. Towards this, servers execute the preprocessing of $\Pi_{\text{sh}}$ (Fig. 3) to generate $[\alpha_1]$ and $\gamma_j$. If $(P_i, P_j) = (P_1, P_0)$, then $P_0, P_0$ jmp-send $\gamma_j = v + \alpha_j$ to $P_0$. The case when $(P_i, P_j) = (P_j, P_0)$ proceeds similarly. The case for $(P_i, P_j) = (P_1, P_2)$ is optimized further as follows: servers locally set $[\alpha_1] = [\alpha_2] = 0$. $P_1, P_2$ together sample random $\gamma_j \in \mathbb{Z}_{2^\ell}$, set $\gamma_j = v$ and jmp-send $\gamma_j$ to $P_0$.

**Protocol $\Pi_{\text{sh}}(P_i, P_j, v)$**

**Preprocessing:**
- If $(P_i, P_j) = (P_1, P_0)$: Servers execute the preprocessing of $\Pi_{\text{sh}}(P_1, v)$ and then locally set $\gamma_j = 0$.
- If $(P_i, P_j) = (P_2, P_0)$: Similar to the case above.
- If $(P_i, P_j) = (P_2, P_2)$: $P_1, P_2$ together sample random $\gamma_j \in \mathbb{Z}_{2^\ell}$. Servers locally set $[\alpha_1] = [\alpha_2] = 0$.

**Online:**
- If $(P_i, P_j) = (P_1, P_0)$: $P_0, P_i$ compute $\gamma_j = v + [\alpha_1] + [\alpha_2]$ and sends $\gamma_j$ to $P_0, P_j$ jmp-send $\gamma_j$ to $P_j$.
- If $(P_i, P_j) = (P_2, P_1)$: Similar to the case above.
- If $(P_i, P_j) = (P_2, P_2)$: $P_1, P_2$ locally set $\gamma_j = v$. $P_1, P_2$ jmp-send $\gamma_j$ to $P_0$.

Figure 4: 3PC: $[\cdot]$-sharing of a value $v \in \mathbb{Z}_{2^\ell}$ jointly by $P_i, P_j$

When the value $v$ is available to both $P_i, P_j$ in the preprocessing phase, protocol $\Pi_{\text{sh}}$ can be made non-interactive in the following way: $P_i$ sample a random $r \in \mathbb{Z}_{2^\ell}$ and locally set their share according to Table 2.

<table>
<thead>
<tr>
<th>$(P_i, P_j)$</th>
<th>$(P_i, P_j)$</th>
<th>$(P_i, P_j)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(P_1, P_0)$</td>
<td>$[\alpha_1] = 0, [\alpha_2] = 0$</td>
<td>$[\alpha_1] = 0, [\alpha_2] = 0$</td>
</tr>
<tr>
<td></td>
<td>$[\beta_1] = v, [\beta_2] = r - v$</td>
<td>$[\beta_1] = v, [\beta_2] = r - v$</td>
</tr>
<tr>
<td>$P_0$</td>
<td>$(0, 0, r)$</td>
<td>$(-v, 0, r)$</td>
</tr>
<tr>
<td>$P_1$</td>
<td>$(0, 0, r - v)$</td>
<td>$(0, 0, r)$</td>
</tr>
<tr>
<td>$P_2$</td>
<td>$(0, 0, r - v)$</td>
<td>$(0, 0, r)$</td>
</tr>
</tbody>
</table>

Table 2: The columns depict the three distinct possibility of input contributing pairs. The first row shows the assignment to various components of the sharing. The last row, along with three sub-rows, specify the shares held by the three servers.

**Lemma 3.4** (Communication). Protocol $\Pi_{\text{sh}}$ (Fig. 4) is non-interactive in the preprocessing phase and requires 1 round and an amortized communication of $\ell$ bits in the online phase.

**Proof.** In this protocol, servers execute $\Pi_{\text{jmp}}$ protocol once. Hence the overall cost follows from that of an instance of the $\Pi_{\text{jmp}}$ protocol (Lemma 3.2). \hfill \Box
**Addition Protocol** Given \([-\cdot]-\)shares on input wires \(x, y\), servers can use linearity property of the sharing scheme to locally compute \([-\cdot]-\)shares of the output of addition gate, \(z = x + y\) as \([z] = [x] + [y]\).

**Multiplication Protocol** Protocol \(\Pi_{\text{mult}}(P, [x], [y])\) (Fig. 5) enables the servers in \(P\) to compute \([-\cdot]\)-sharing of \(z = xy\), given the \([-\cdot]\)-sharing of \(x\) and \(y\). We build on the protocol of BLAZE [48] and discuss along the way the differences and resemblances. We begin with a protocol for the semi-honest setting, which is also the starting point of BLAZE. During the preprocessing phase, \(P_0, P_j\) for \(j \in \{1, 2\}\) sample random \([\alpha_j]_j \in \mathbb{Z}_{2^j}\), while \(P_1, P_2\) sample random \([\gamma_j]_j \in \mathbb{Z}_{2^j}\). In addition, \(P_0\) locally computes \(\Gamma_{xy} = \alpha_x \alpha_y\) and generates \([-\cdot]\)-sharing of the same between \(P_1, P_2\). Since,

\[
\beta_z = z + \alpha_x = xy + \alpha_x = (\beta_x + \alpha_x)(\beta_y + \alpha_y) + \alpha_x
\]

\[
= \beta_x \beta_y + \beta_x \alpha_y + \beta_y \alpha_x + \Gamma_{xy} + \alpha_x
\]

(1)

servers \(P_1, P_2\) locally compute \([\beta_z]_j = (j-1)\beta_x \beta_y - \beta_x [\alpha_j]_j - \beta_y [\alpha_j]_j + [\Gamma_{xy}]_j + [\alpha_x]_j\) during the online phase and mutually exchange their shares to reconstruct \(\beta_z\). \(P_1\) then sends \(\beta_z + \gamma_z\) to \(P_0\), completing the semi-honest protocol. The correctness that asserts \(z = xy\) in other words \(\beta_z = \alpha_x = xy\) holds due to Eq. 1.

The following issues arise in the above protocol when a malicious adversary is considered:

1) When \(P_1\) is corrupt, the \([-\cdot]\)-sharing of \(\Gamma_{xy}\) performed by \(P_0\) might not be correct, i.e. \(\Gamma_{xy} \neq \alpha_x \alpha_y\).

2) When \(P_1\) (or \(P_2\)) is corrupt, \([-\cdot]\)-share of \(\beta_z\) handed over to the fellow honest evaluator during the online phase might not be correct, causing reconstruction of an incorrect \(\beta_z\).

3) When \(P_1\) is corrupt, the value \(\beta_z + \gamma_z\) that is sent to \(P_0\) during the online phase may not be correct.

All the three issues are common with BLAZE (copied verbatim), but we differ from BLAZE in handling them. We begin with solving the last issue first. We simply make \(P_1, P_2\) jmp-send \(\beta_z + \gamma_z\) to \(P_0\) (after \(\beta_z\) is computed). This either leads to success or a TTP selection. Due to jmp’s rate-1 communication, \(P_1\) alone sending the value to \(P_0\) remains as costly as using jmp in amortized sense. While in BLAZE, the malicious version simply makes \(P_2\) to send a hash of \(\beta_z + \gamma_z\) to \(P_0\) (in addition to \(P_1\)’s communication of \(\beta_z + \gamma_z\) to \(P_0\)), which aborts if the received values are inconsistent.

For the remaining two issues, similar to BLAZE, we reduce both to a multiplication (on values unrelated to inputs) in the preprocessing phase. However, our method leads to either success or TTP selection, with no additional cost.

We start with the second issue. To solve it, where a corrupt \(P_1\) (or \(P_2\)) sends an incorrect \([-\cdot]\)-share of \(\beta_z\), BLAZE makes use of server \(P_0\) to compute a version of \(\beta_z\) for verification, based on \(\beta_x, \beta_y, \alpha_x, \alpha_y, \alpha_z\) and \(\Gamma_{xy}\). \(P_0\) computes:

\[
\beta_z^* = -(\beta_x + \gamma_x)\alpha_y - (\beta_y + \gamma_y)\alpha_x + 2\Gamma_{xy} + \alpha_z
\]

\[
= (-\beta_x \alpha_y - \beta_y \alpha_x + \Gamma_{xy} + \alpha_z) - (\gamma_x \alpha_y + \gamma_y \alpha_x - \Gamma_{xy})
\]

by Eq. 1

\[
= (\beta_x - \beta_y)\beta_y - (\gamma_x \alpha_y + \gamma_y \alpha_x) - \Gamma_{xy}[\beta_z - \beta_z^*] - \chi
\]

[where \(\chi = \gamma_x \alpha_y + \gamma_y \alpha_x - \Gamma_{xy}\)]

Now if \(\chi\) can be made available to \(P_0\), it can send \(\beta_z^* + \chi\) to \(P_1\) and \(P_2\) who using the knowledge of \(\beta_x, \beta_y, \gamma\), can verify the correctness of \(\beta_z\) by computing \(\beta_z - \beta_z^* \psi\) and checking against the value \(\beta_z^* + \psi\) received from \(P_0\). However, disclosing \(\chi\) on clear to \(P_0\) will cause a privacy issue when \(P_0\) is corrupt, because one degree of freedom on the pair \((\gamma_x, \gamma_y)\) is lost and the same impact percolates down to \((\beta_x, \beta_y)\) and further to the actual values \((v_x, v_y)\) on the wires \(x, y\). This is resolved through a random value \(\psi \in \mathbb{Z}_{2^j}\), sampled together by \(P_1\) and \(P_2\). Now, \(\chi\) and \(\beta_z^*\) are set to \(\gamma_x \alpha_y + \gamma_y \alpha_x - \Gamma_{xy} + \psi\), \(\beta_z - \beta_z^* \psi\), and the check by \(P_1, P_2\) involves computing \(\beta_z - \beta_z^* \psi + \psi\). The rest of the logic in BLAZE goes on to discuss how to enforce \(P_0\) (a) to compute a correct \(\chi\) (when honest), and (b) to share correct \(\Gamma_{xy}\) (when corrupt). Tying the ends together, they identify the precise shared multiplication triple and map its components to \(\chi\) and \(\Gamma_{xy}\) so that these values are correct by virtue of the correctness of the product relation. This reduces ensuring the correctness of these values to doing a single multiplication of two values in the preprocessing phase.
share of $\beta^*_2$ to the third server which computes $\beta^*_3$. Due to the presence of one honest server in each team, this $\beta^*_3$ is correct and $P_1, P_2$ directly use it to compute $\beta_4$, with the knowledge of $\psi, \beta_2, \beta_3$. The outcome of our approach is a win-win situation i.e. either success or TTP selection. Our approach of computing $\beta_4$ from $\beta^*_3$ is a departure from BLAZE, where the latter suggests computing $\beta_4$ from the exchange $P_1, P_2$’s respective share of $\beta_4$ (as in the semi-honest construction) and use $\beta^*_3$ for verification. Our new $\beta^*_3$ and $\chi$ are:

$$\chi = \gamma_x \alpha_y + \gamma_y \alpha_x + \Gamma_{xy} - \psi$$

and

$$\beta^*_3 = -(\beta_x + \gamma_x) \alpha_y - (\beta_y + \gamma_y) \alpha_x + \alpha_z + \chi$$

Clearly, both $P_0$ and $P_1$ can compute $[\beta^*_3]_i = -(\beta_x + \gamma_x) [\alpha_y_i] - (\beta_y + \gamma_y) [\alpha_x_i] + [\chi_i]$, given $[\chi_i]$. The rest of our discussion explains how (a) $i$th share of $[\chi]$ can be made available to $\{P_0, P_1\}$ and (b) $\psi$ can be derived by $P_1, P_2$, from a multiplication triple. Similar to BLAZE, yet for a different triple, we observe that $(d, e, f)$ is a multiplication triple, where $d = (\gamma_x + \alpha_x), e = (\gamma_y + \alpha_y), f = (\gamma_x \gamma_y + \psi) + \chi$ if and only if $\chi$ and $\Gamma_{xy}$ are correct. Indeed,

$$de = (\gamma_x + \alpha_x)(\gamma_y + \alpha_y) = \gamma_x \gamma_y + \gamma_y \alpha_x + \gamma_x \alpha_y + \Gamma_{xy}$$

$$= (\gamma_x \gamma_y + \psi) + (\gamma_y \alpha_x + \gamma_x \alpha_y + \Gamma_{xy} - \psi)$$

Based on this observation, we compute the above multiplication triple using a multiplication protocol and extract out the values for $\psi$ and $\chi$ from the shares of $f$ which are bound to be correct. This can be executed entirely in the preprocessing phase. Specifically, the servers (a) locally obtain $\langle \cdot \rangle$-shares of $d, e$ as in Table 3, (b) compute $\langle \cdot \rangle$-shares of $f (= de)$, say denoted by $f_0, f_1, f_2$, using an efficient, robust 3-party multiplication protocol, say $\Pi_{mulPre}$ (abstracted in a functionality Fig. 6) and finally (c) extract out the required preprocessing data locally as in Eq. 2. We switch to $\langle \cdot \rangle$-sharing in this part to be able to use the best robust multiplication protocol of [10] that supports this form of secret sharing and requires communication of just 3 elements. Fortunately, the switch does not cost anything, as both the step (a) and (c) (as above) involve local computation and the cost simply reduces to a single run of a multiplication protocol.

<table>
<thead>
<tr>
<th>$\langle v \rangle$</th>
<th>$P_0$ $(v_0, v_1)$</th>
<th>$P_1$ $(v_1, v_2)$</th>
<th>$P_2$ $(v_2, v_0)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\langle d \rangle$</td>
<td>$([\alpha_x], [\alpha_y])$</td>
<td>$([\alpha_y], [\gamma_y])$</td>
<td>$([\gamma_x], [\alpha_x])$</td>
</tr>
<tr>
<td>$\langle e \rangle$</td>
<td>$([\alpha_x], [\alpha_y])$</td>
<td>$([\alpha_y], [\gamma_y])$</td>
<td>$([\gamma_x], [\alpha_x])$</td>
</tr>
</tbody>
</table>

Table 3: The $\langle \cdot \rangle$-sharing of values $d$ and $e$

According to $\langle \cdot \rangle$-sharing, both $P_0$ and $P_1$ obtain $f_1$ and hence obtain $[\chi]_1$. Similarly, $P_0, P_2$ obtain $f_0$ and hence $[\chi]_2$.

Finally, $P_1, P_2$ obtain $f_2$ from which they compute $\psi = f_2 - \gamma_x \gamma_y$. This completes the informal discussion.

To leverage amortization, the send phase of jmp-send alone is executed on the fly and verify is performed once for multiple instances of jmp-send. Further, observe that $P_1, P_2$ possess the required shares in the online phase to compute the entire circuit. Hence, $P_0$ can come in only during verify of jmp-send towards $P_1, P_2$, which can be deferred towards the end. Hence, the jmp-send of $\beta_x + \gamma_x$ to $P_0$ (enabling computation of the verification information) can be performed once, towards the end, thereby requiring a single round for sending $\beta_x + \gamma_x$ to $P_0$ for multiple instances. Following this, the verify of jmp-send towards $P_0$ is performed first, followed by performing the verify of jmp-send towards $P_1, P_2$ in parallel.

We note that to facilitate a fast online phase for multiplication, our preprocessing phase leverages a robust multiplication protocol [10] in a black-box manner to derive the necessary preprocessing information. A similar black-box approach is also taken for the dot product protocol in the preprocessing phase. This leaves room for further improvements in the communication cost, which can be obtained by instantiating the black-box with an efficient, robust protocol coupled with the fast online phase.

**Functionality $\muPre$**

$\muPre$ interacts with the servers in $\mathcal{P}$ and the adversary $S$. $\muPre$ receives $\langle \cdot \rangle$-shares of $d, e$ from the servers where $P_i$, for $s \in \{0, 1, 2\}$, holds $(d)_s = (d_s, d_{s+1} \% 3)$ and $(e)_s = (e_s, e_{s+1} \% 3)$ such that $d = d_0 + d_1 + d_2$ and $e = e_0 + e_1 + e_2$. Let $P_i$ denotes the server corrupted by $S$. $\muPre$ receives $(f)_s = (f_s, f_{(s+1) \% 3})$ from $S$ where $f = de$. $\muPre$ proceeds as follows:

- Reconstructs $d, e$ using the shares received from honest servers and compute $f = de$.
- Compute $f_{(s+1) \% 3} = f - f_{(s+1) \% 3}$ and set the output shares as $(f_0) = (f_0, f_1), (f_1) = (f_1, f_2), (f_2) = (f_2, f_0)$.
- Send $\langle Output, (f)_s \rangle$ to server $P_s \in \mathcal{P}$.

**Figure 6:** 3PC: Ideal functionality for $\muPre$ protocol

**Lemma 3.5 (Communication).** Protocol $\Pi_{mul}$ (Fig. 5) requires an amortized cost of $3\ell$ bits in the preprocessing phase, and 1 round and amortized cost of $3\ell$ bits in the online phase.

**Proof.** In the preprocessing phase, generation of $\alpha_x$ and $\gamma_x$ are non-interactive. This is followed by one execution of $\Pi_{mulPre}$, which requires an amortized communication cost of $3\ell$ bits. During the online phase, $P_0, P_1$ jmp-send $[\beta^*_2]_1$ to $P_2$, while $P_0, P_2$ jmp-send $[\beta^*_2]_2$ to $P_1$. This requires one round and a communication of $2\ell$ bits. Following this, $P_1, P_2$ jmp-send $\beta_x + \gamma_x$ to $P_0$, which requires one round and a communication of $\ell$ bits. However, jmp-send of $\beta_x + \gamma_x$ can be delayed till the end of the protocol, and will require only one round for the entire circuit and can be amortized. □
Reconstruction Protocol Protocol $\Pi_{\text{rec}}$ (Fig. 7) allows servers to robustly reconstruct value $v \in \mathbb{Z}_2^\ell$ from its $||v||$ shares. Note that each server misses one share of $v$ which is held by the other two servers. Consider the case of $P_0$ who requires $\gamma_v$ to compute $v$. During the preprocessing, $P_1, P_2$ compute a commitment of $\gamma_v$, denoted by $\text{Com}(\gamma_v)$ and $\text{jmp}-\text{send}$ the same to $P_0$. Similar steps are performed for the values $[\alpha_{v}]_2$ and $[\alpha_v]_1$ that are required by servers $P_1$ and $P_2$ respectively. During the online phase, servers open their commitments to the intended server who accepts the opening that is consistent with the agreed upon commitment.

![Protocol $\Pi_{\text{rec}}(P, [v])$](image)

Preprocessing:
- $P_0, P_j$, for $j \in \{1, 2\}$, compute $\text{Com}([\alpha_{v}]_j)$, while $P_1, P_2$ compute $\text{Com}(\gamma_v)$.
- $P_1, P_2$ $\text{jmp}-\text{send}$ $\text{Com}(\gamma_v)$ to $P_0$, while $P_0, P_1$ $\text{jmp}-\text{send}$ $\text{Com}([\alpha_v]_1)$ to $P_2$, and $P_0, P_2$ $\text{jmp}-\text{send}$ $\text{Com}([\alpha_v]_2)$ to $P_1$.

Online:
- $P_0, P_i$ open $\text{Com}([\alpha_{v}]_1)$ to $P_2$. $P_0, P_2$ open $\text{Com}([\alpha_v]_2)$ to $P_1$.
- $P_1, P_2$ open $\text{Com}(\gamma_v)$ to $P_0$.
- Each server accepts the opening that is consistent with the agreed upon commitment. $P_1, P_2$ compute $v = [\beta_v] - [\alpha_v]_1 - [\alpha_v]_2$, while $P_0$ computes $v = ([\beta_v] + \gamma_v) - [\alpha_v]_1 - [\alpha_v]_2 - \gamma_v$.

Figure 7: 3PC: Reconstruction of $v$ among the servers

Lemma 3.6 (Communication). Protocol $\Pi_{\text{rec}}$ (Fig. 7) requires 1 round and a communication of $6\ell$ bits in the online phase.

Proof. The preprocessing phase consists of communication of commitment values using the $\Pi_{\text{jmp}}$ protocol. The hash-based commitment scheme allows generation of a single commitment for several values and hence the cost gets amortised away for multiple instances. During the online phase, each server receives an opening for the commitment from other two servers, which requires one round and an overall communication of $6\ell$ bits.

The Complete 3PC For the sake of completeness and to demonstrate how GOD is achieved, we show how to compile the above primitives for a general 3PC. A similar approach will be taken for 4PC and each PPML task, and we will avoid repetition. In order to compute an arithmetic circuit over $\mathbb{Z}_2^\ell$, we first invoke the key-setup functionality $\mathcal{F}_{\text{setup}}$ (Fig. 27) for key distribution and preprocessing of $\Pi_{\text{sh}}, \Pi_{\text{mult}}$ and $\Pi_{\text{rec}}$, as per the given circuit. During the online phase, $P_i \in \mathcal{P}$ shares its input $x_i$ by executing online steps of $\Pi_{\text{sh}}$ (Fig. 3). This is followed by the circuit evaluation phase, where servers evaluate the gates in the circuit in the topological order, with addition gates (and multiplication-by-a-constant gates) being computed locally, and multiplication gates being computed via online of $\Pi_{\text{mult}}$ (Fig. 5). Finally, servers run the online steps of $\Pi_{\text{rec}}$ (Fig. 7) on the output wires to reconstruct the function output. To leverage amortization, only $\text{send}$ phases of all the $\text{jmp}$ are run on the flow. At the end of preprocessing, the $\text{verify}$ phase for all possible ordered pair of senders are run. We carry on computation in the online phase only when the $\text{verify}$ phases in the preprocessing are successful. Otherwise, the servers simply send their inputs to the elected TTP, who computes the function and returns the result to all the servers. Similarly, depending on the output of the $\text{verify}$ at the end of the online phase, either the reconstruction is carried out or a TTP is identified. In the latter case, computation completes as mentioned before.

![Protocol 3PC](image)

Let ckt denote the circuit representation of the function to be evaluated. Servers invoke $\mathcal{F}_{\text{setup}}$ to establish common keys among themselves.

- **Preprocessing:** Preprocessing phase of $\Pi_{\text{sh}}, \Pi_{\text{mult}}, \Pi_{\text{rec}}$ is executed depending on ckt topology. For every pair of servers, $P_i, P_j \in \mathcal{P}$, the verification $\text{jmp}$ is executed once, for all the $\text{jmp}$ instances with $P_i, P_j$ as the senders.
- **Online phase:** If TTP is identified in the preprocessing phase, servers send their inputs to the TTP, which evaluates the function on the clear inputs, and sends the result back to the servers. Else,
  - **Input sharing:** The online phase of $\Pi_{\text{sh}}$ is executed.
  - **Evaluation phase:**
    - Linear gates such as addition and multiplication-by-a-constant are evaluated locally.
    - For multiplication gates, the online phase of $\Pi_{\text{mult}}$ is executed with only the $\text{send}$ phase of $\text{jmp}$ being executed for $\text{jmp}$ towards $P_i, P_j$. All communication towards $P_0$ is carried out in the verification phase.
    - **Verification phase:** With respect to the multiplication gates, the communication towards $P_0$ is carried out by invoking $\text{jmp}$ (both $\text{send}$ and $\text{verify}$ phase). This is followed by the $\text{verify}$ phase of $\text{jmp}$ for all $\text{jmp}$ instances with $P_i, P_j$ as the receivers.
      - If TTP is identified at any point during verification, servers send their inputs to the TTP, which evaluates the function on the clear inputs, and sends the result back to the servers.
      - **Output reconstruction:** If verification succeeds without TTP identification, servers execute online phase of $\Pi_{\text{rec}}$.

Figure 8: Complete 3PC

On the security of our framework: We emphasize that we follow the standard traditional (real-world / ideal-world based) security definition of MPC, according to which, in the 4-party setting with 1 corruption, exactly 1 party is assumed to be corrupt, and rest are honest. As per this definition, disclosing the honest party’s inputs to a selected honest party is not a breach of security. Indeed, in our framework, the data sharing and the computation on the shared data is done in a way that any malicious behaviour leads to establishment of a TTP who is enabled to receive all the inputs and compute the output on the clear. There has been a recent study on the additional
3.3 Building Blocks for PPML using 3PC

This section provides details on robust realizations of the following building blocks for PPML in 3-server setting—i) Dot Product, ii) Truncation, iii) Dot Product with Truncation, iv) Secure Comparison, and v) Non-linear Activation functions—Sigmoid and ReLU. We provide the security proofs in §C.1.

We begin by providing details of input sharing and reconstruction in the SOC setting.

<table>
<thead>
<tr>
<th>Protocol $\Pi_{SOC}^{sh}(U,v)$ and $\Pi_{SOC}^{rec}(U,[v])$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input Sharing:</strong></td>
</tr>
<tr>
<td>- $P_0,P_1$, for $s \in {1,2}$, together sample random $[\alpha_s]<em>s \in \mathbb{Z}</em>{2^l}$, while $P_0,P_1$ together sample random $[\gamma]<em>y \in \mathbb{Z}</em>{2^l}$.</td>
</tr>
<tr>
<td>- $P_0,P_1$ jump-send $\text{Com}(\alpha_1)$ to $P_2$, while $P_0,P_2$ jump-send $\text{Com}(\alpha_2)$ to $P_1$, and $P_1,P_2$ jump-send $\text{Com}(\gamma)$ to $P_0$.</td>
</tr>
<tr>
<td>- Each server sends $\text{Com}(\alpha_1),\text{Com}(\alpha_2),\text{Com}(\gamma)$ to $U$ who accepts the shares that form majority. Also, $P_0,P_2$, for $s \in {1,2}$, open $[\alpha_s]$ towards $U$ while $P_1,P_2$ open $[\gamma]$ towards $U$.</td>
</tr>
<tr>
<td>- $U$ accepts the consistent opening, recovers $[\alpha_1],[\alpha_2],[\gamma]$, and sends $\beta_i = [\alpha_i]_1 + [\alpha_i]_2 + [\gamma]_y$ to all three servers.</td>
</tr>
<tr>
<td>- Servers broadcast the received value and accept the majority value if it exists, and a default value, otherwise. $P_1,P_2$ locally compute $\beta_i$ from $\beta_i + [\gamma]_y$ to complete the sharing of $v$.</td>
</tr>
</tbody>
</table>

| **Output Reconstruction:**                                |
| - Servers execute the preprocessing of $\Pi_{rec}(P,[v])$ to agree upon commitments of $[\alpha_1], [\alpha_2]$ and $[\gamma]_y$. |
| - Each server sends $\beta_i + [\gamma]_y$ and commitments on $[\alpha_1], [\alpha_2]$ and $[\gamma]_y$ to $U$, who accepts the values forming majority. |
| - $P_0,P_i$ for $i \in \{1,2\}$ open $[\alpha_i]$ to $U$, while $P_1,P_2$ open $[\gamma]_y$ to $U$. |
| - $U$ accepts the consistent opening and computes $v = (\beta_i + [\gamma]_y) - [\alpha_1]_1 - [\alpha_2]_2 - [\gamma]_y$. |

Figure 9: 3PC: Input Sharing and Output Reconstruction

Input Sharing and Output Reconstruction in the SOC Setting Protocol $\Pi_{SOC}^{sh}(\text{Fig. 9})$ extends input sharing to the SOC setting and allows a user $U$ to generate the $\lfloor \cdot \rfloor$-shares of its input $v$ among the three servers. Note that the necessary commitments to facilitate the sharing are generated in the preprocessing phase by the servers which are then communicated to $U$, along with the opening, in the online phase. $U$ selects the commitment forming the majority (for each share) owing to the presence of an honest majority among the servers, and accepts the corresponding shares. Analogously, protocol $\Pi_{SOC}^{rec}$ (Fig. 9) allows the servers to reconstruct a value $v$ towards user $U$. In either of the protocols, if at any point, a TTP is identified, then servers signal the TTP’s identity to $U$. $U$ selects the TTP as the one forming a majority and sends its input in the clear to the TTP, who computes the function output and sends it back to $U$.

**MSB Extraction** Protocol $\Pi_{bitext}$ allows servers to compute the boolean sharing of the most significant bit (msb) of a value $v$ given its arithmetic sharing $[v]$. To compute the msb, we use the optimized 2-input Parallel Prefix Adder (PPA) Boolean circuit proposed by ABY [43]. The PPA circuit consists of $2l-2$ AND gates and has a multiplicative depth of $\log \ell$.

<table>
<thead>
<tr>
<th>$P_0$</th>
<th>$P_1$</th>
<th>$P_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[v_0[i]]^B$</td>
<td>$(0,0,0)$</td>
<td>$(0,v_0[i],v_0[i])$</td>
</tr>
<tr>
<td>$[v_1[i]]^B$</td>
<td>$(v_1[i],0,0)$</td>
<td>$(v_1[i],0,0)$</td>
</tr>
<tr>
<td>$[v_2[i]]^B$</td>
<td>$(0,0,0)$</td>
<td>$(0,v_2[i],0)$</td>
</tr>
</tbody>
</table>

Table 4: The $\lfloor i \rfloor^B$-sharing corresponding to the $i^{th}$ bit of $v_0 = [\alpha_1], v_1 = [\alpha_2], v_2 = [\alpha_3]$. Here $i \in \{0,\ldots,\ell - 1\}$. Let $v_0 = [\alpha_1], v_1 = [\alpha_2], v_2 = [\alpha_3]$. Then $v = v_0 + v_1 + v_2$. Servers first locally compute the boolean shares corresponding to each bit of the values $v_0,v_1$ and $v_2$ according to Table 4. It has been shown in ABY3 that $v = v_0 + v_1 + v_2$ can also be expressed as $v = c + s$ where $FA(v_0[i],v_1[i],v_2[i]) \rightarrow (c[i],s[i])$ for $i \in \{0,\ldots,\ell - 1\}$. Here FA denotes a Full Adder circuit while $s$ and $c$ denote the sum and carry bits respectively.

To summarize, servers execute $\ell$ instances of FA in parallel to compute $\lfloor c \rfloor^B + \lfloor s \rfloor^B$. The FA’s are executed independently and require one round of communication. The final result is then computed as $\text{msb}(2[\lfloor c \rfloor^B + \lfloor s \rfloor^B])$ using the optimized PPA circuit.

**Lemma 3.7** (Communication). Protocol $\Pi_{bitext}$ requires a communication cost of $9\ell - 6$ bits in the preprocessing phase and require $\log \ell + 1$ rounds and an amortized communication of $9\ell - 6$ bits in the online phase.

**Proof.** In $\Pi_{bitext}$, first round comprises of $\ell$ Full Adder (FA) circuits executing in parallel, each comprising of single AND gate. This is followed by the execution of the optimized PPA circuit of ABY3 [43], which comprises of $2\ell - 2$ AND gates and has a multiplicative depth of $\log \ell$. Hence the communication cost follows from the multiplication for $3\ell - 2$ AND gates.

**Bit to Arithmetic Conversion** Given the boolean sharing of a bit $b$, denoted as $[b]^B$, protocol $\Pi_{bit2a}$ (Fig. 10) allows servers to compute the arithmetic sharing $[b]^R$. Here $b^R$ denotes the equivalent value of $b$ over ring $\mathbb{Z}_{2^l}$ (see Notation 2.2). As pointed out in BLAZE, $b^R = ([b]_0 \oplus a_0)^R = \ldots$
During the online phase, this is followed by an execution of the entire multiplication protocol. During the preprocessing phase, \(P_0, P_j\) for \(j \in \{1, 2\}\) execute \(\Pi_{\text{sh}}\) on \([\alpha_0]_j^R\) to generate \(\llbracket \alpha_0 \rrbracket^R\). Servers then execute \(\Pi_{\text{mul}}\) on \(\llbracket \alpha_0 \rrbracket^R\) and \(\llbracket \alpha_0 \rrbracket^R\) to generate \(\llbracket \alpha_0 \rrbracket_1^R\) and \(\llbracket \alpha_0 \rrbracket_2^R\) followed by locally computing \(\llbracket \alpha_0 \rrbracket^B\). During the online phase, \(P_1, P_2\) execute \(\Pi_{\text{sh}}\) on \([\alpha_0]_j^R\) to jointly generate \(\llbracket \beta_0^R\). Servers then execute \(\Pi_{\text{mul}}\) protocol on \([\beta_0]_j^R\) and \([\alpha_0]_j^R\) to compute \(\llbracket \beta_0^R \alpha_0^R\rrbracket^B\) followed by locally computing \(\llbracket \beta_0^B\).

**Lemma 3.8 (Communication).** Protocol \(\Pi_{\text{bit2A}}\) (Fig. 10) requires an amortized communication cost of 9\(\ell\) bits in the preprocessing phase and requires 1 round and an amortized communication of 4\(\ell\) bits in the online phase.

**Proof.** In the preprocessing phase, servers run two instances of \(\Pi_{\text{sh}}\), which can be done non-interactively (ref. Table 2). This is followed by an execution of entire multiplication protocol, which requires 6\(\ell\) bits to be communicated (Lemma 3.5). Parallelly, the servers execute the preprocessing phase of \(\Pi_{\text{mul}}\), resulting in an additional 3\(\ell\) bits of communication (Lemma 3.5). During the online phase, \(P_1, P_2\) execute \(\Pi_{\text{sh}}\) once, which requires one round and \(4\ell\) bits to be communicated. In \(\Pi_{\text{sh}}\), the communication towards \(P_1\) can be deferred till the end, thereby requiring a single round for multiple instances. This is followed by an execution of the online phase of \(\Pi_{\text{mul}}\), which requires one round and a communication of 3\(\ell\) bits.

**Protocol \(\Pi_{\text{bit2A}}(\mathcal{P}, [b]^B)\)**

**Preprocessing:**
- \(P_0, P_j\) for \(j \in \{1, 2\}\) execute \(\Pi_{\text{sh}}\) on \([\alpha_0]_j^R\) to generate \(\llbracket \alpha_0 \rrbracket^R\).
- Servers execute \(\Pi_{\text{mul}}(\mathcal{P}, [\alpha_0]_j^R, [\alpha_0]_j^R)\) to generate \([u]\) where \(u = [\alpha_0]_j^R, [\alpha_0]_j^R\), followed by locally computing \(\llbracket \alpha_0 \rrbracket^B = \llbracket [\alpha_0]_j^R \rrbracket + \llbracket [\alpha_0]_j^R \rrbracket - 2[u]\).
- Servers execute the preprocessing phase of \(\Pi_{\text{mul}}(\mathcal{P}, [\beta_0]_j^R, [\alpha_0]_j^R)\) for \(v = [\beta_0]_j^R, [\alpha_0]_j^R\).

**Online:**
- \(P_1, P_2\) execute \(\Pi_{\text{sh}}(P_1, P_2, [\beta_0]_j^R)\) to generate \(\llbracket \beta_0 \rrbracket^R\).
- Servers execute online phase of \(\Pi_{\text{mul}}(\mathcal{P}, [\beta_0]_j^R, [\alpha_0]_j^R)\) to generate \([v]\) where \(v = [\beta_0]_j^R, [\alpha_0]_j^R\), followed by locally computing \([b]^B\) = \(\llbracket \beta_0 \rrbracket^B + \llbracket \alpha_0 \rrbracket^B - 2[v]\).

![Figure 10: 3PC: Bit2A Protocol](image)

**Bit Injection** Given the binary sharing of a bit \(b\), denoted as \([b]^B\), and the arithmetic sharing of \(v \in \mathbb{Z}_2^f\), protocol \(\Pi_{\text{bit2A}}\) computes \(\llbracket b \rrbracket\)-sharing of \(bv\). Towards this, servers first execute \(\Pi_{\text{bit2A}}\) on \([b]^B\) to generate \([b]\). This is followed by servers computing \([bv]\) by executing \(\Pi_{\text{mul}}\) protocol on \([b]\) and \([v]\).

**Lemma 3.9 (Communication).** Protocol \(\Pi_{\text{bit2A}}\) requires an amortized communication cost of 12\(\ell\) bits in the preprocessing phase and requires 2 rounds and an amortized communication of 7\(\ell\) bits in the online phase.

**Proof.** Protocol \(\Pi_{\text{bit2A}}\) is essentially an execution of \(\Pi_{\text{bit2A}}\) (Lemma 3.8) followed by one invocation of \(\Pi_{\text{mul}}\) (Lemma 3.5) and the costs follow.

**Dot Product** Given the \(\llbracket \cdot \rrbracket\)-sharing of vectors \(\vec{x}\) and \(\vec{y}\), protocol \(\Pi_{\text{dotp}}\) (Fig. 11) allows servers to generate \(\llbracket \cdot \rrbracket\)-sharing of \(z = \vec{x} \cdot \vec{y}\) robustly. \(\llbracket \cdot \rrbracket\)-sharing of a vector \(\vec{x}\) of size \(n\), means that each element \(x_i \in \mathbb{Z}_2^n\) of \(\vec{x}\), for \(i \in [n]\), is \(\llbracket \cdot \rrbracket\)-shared. We borrow ideas from BLAZE for obtaining an online communication cost independent of \(n\) and use jmp primitive to ensure either success or TTP selection. Analogous to our multiplication protocol, our dot product protocol loads one call to a robust dot product protocol to the preprocessing. By extending techniques of [9, 10], we give an instantiation for the dot product protocol used in our preprocessing whose (amortized) communication cost is constant, thereby making our preprocessing cost also independent of \(n\).

To begin with, \(z = \vec{x} \cdot \vec{y}\) can be viewed as \(n\) parallel multiplication instances of the form \(z_i = x_i y_i\), for \(i \in [n]\), followed by adding up the results. Let \(\beta_i^* = \sum_{i=1}^{n} [\beta_i]_1^R\). Then,

\[
\beta_i^* = -\sum_{i=1}^{n} (\alpha_i + \gamma_i) \alpha_i - \sum_{i=1}^{n} (\beta_i + \gamma_i) \alpha_i + \alpha_i + \chi
\]

where \(\chi = \sum_{i=1}^{n} \Gamma_{x_i y_i - \psi_i}\).

Apart from the aforementioned modification, the online phase for dot product proceeds similar to that of multiplication protocol. \(P_1, P_1\) locally compute \([\beta_i^*]_1^R\) as per Eq. 3 and jmp-send \([\beta_i^*]_1^R\) to \(P_2\). \(P_1\) obtains \([\beta_i^*]_2^R\) in a similar fashion. \(P_1, P_2\) reconstruct \(\beta_i^* = [\beta_i^*]_1^R + [\beta_i^*]_2^R\) and compute \(\beta_i = [\beta_i^*]_1^R + \sum_{i=1}^{n} [\beta_i]_1^R [\alpha_i] + [\psi_i] - [\beta_i]_1^R [\alpha_i]_1^R\).

Finally, \(P_1, P_2\) jmp-send \(\beta_i^*\) to \(P_0\).

We now provide the details for preprocessing phase that enable servers to obtain the required values (\(\chi, \psi\)) with the invocation of a dot product protocol in a black-box way. Towards this, let \(\vec{d} = [d_1, \ldots, d_n]\) and \(\vec{e} = [e_1, \ldots, e_n]\), where \(d_i = \gamma_i + \alpha_x\) and \(e_i = \gamma_i + \alpha_y\) for \(i \in [n]\), as in the case of multiplication. Then for \(f = \vec{d} \odot \vec{e}\),

\[
f = \vec{d} \odot \vec{e} = \sum_{i=1}^{n} d_i e_i = \sum_{i=1}^{n} (\gamma_i + \alpha_x) (\gamma_i + \alpha_y)
\]

\[
= \sum_{i=1}^{n} (\gamma_i \gamma_i + \psi_i) + \sum_{i=1}^{n} \chi_i = \sum_{i=1}^{n} (\gamma_i \gamma_i + \psi_i) + \chi
\]

\[
= \sum_{i=1}^{n} (\gamma_i \gamma_i + \psi_i) + [\chi_1]_2 + [\chi_2]_1 = f_2 + f_1 + f_0,
\]

where \(f_2 = \sum_{i=1}^{n} (\gamma_i \gamma_i + \psi_i), f_1 = [\chi_1]_2\) and \(f_0 = [\chi_2]_1\).

Using the above relation, the preprocessing phase proceeds as follows: \(P_0, P_j\) for \(j \in \{1, 2\}\) sample a random
with $\mathbf{d}, \mathbf{e}$ similar to that of multiplication protocol. Servers locally prepare $\mathbf{d}, \mathbf{e}$ as input and compute $f$ given $\omega_i = \gamma_i + \alpha_i$, for $i \in [n]$ as follows (cf. Eq. 4):

$$\psi = f_2 - \sum_{i=1}^{n} \gamma_i \gamma_i, \quad |\chi_1| = f_1, \quad |\chi_2| = f_0,$$

(4)

It is easy to see from the semantics of $\langle \cdot \rangle$-sharing that both $P_1, P_2$ obtain $f_2$ and hence $\psi$. Similarly, both $P_0, P_1$ obtain $f_1$ and hence $|\chi_1|$, while $P_0, P_2$ obtain $|\chi_2|$.

Protocol $\Pi_{\text{dotp}}(\mathcal{P}, \{[\chi_i], [\gamma_i]\}_{i \in [n]})$

**Preprocessing:**
- $P_0, P_j$, for $j \in \{1, 2\}$, together sample random $[\alpha_2]_j \in Z_{2^m}$, while $P_1, P_2$ sample random $\gamma_2 \in Z_{2^m}$.
- Servers locally compute $\langle \cdot \rangle$-sharing of $\mathbf{d}, \mathbf{e}$ with $d_i = \gamma_i + \alpha_i$ and $e_i = \gamma_i + \alpha_i$, for $i \in [n]$ as follows:
  $$\langle d_i \rangle_0 = (\alpha_{x_i}, \alpha_{x_i}), \langle d_i \rangle_1 = (\alpha_{y_i}, \gamma_{x_i}), \langle d_i \rangle_2 = (\gamma_{y_i}, \alpha_{x_i}), \langle e_i \rangle_0 = (\gamma_{y_i}, \gamma_{x_i}), \langle e_i \rangle_1 = (\gamma_{y_i}, \gamma_{x_i})$$
- Servers execute $\Pi_{\text{dotpPre}}(\mathcal{P}, (\mathbf{d}, \mathbf{e}))$ to generate $f = (\mathbf{d} \odot \mathbf{e})$.
- $P_0, P_i$ locally set $|\chi_1| = f_1$, while $P_0, P_2$ locally set $|\chi_2| = f_0$. $P_1, P_2$ locally compute $\psi = f_2 - \sum_{i=1}^{n} \gamma_i \gamma_i$.

**Online:**
- $P_0, P_j$, for $j \in \{1, 2\}$, compute $[\beta_2]_j = -\sum_{i=1}^{n} (\beta_i + \gamma_i)[\alpha_2]_j + (\gamma_i + \gamma_i)[\alpha_2]_j + [\alpha_2]_j + |\chi_1|$.
- $P_0, P_j$ jmp-send $[\beta_2]_j$ to $P_2$ and $P_0, P_2$ jmp-send $[\beta_2]_j$ to $P_1$.
- $P_1, P_2$ locally compute $\beta_2 = [\beta_2]_1 + [\beta_2]_2$ and set $\beta_2 = \beta_2 + \sum_{i=1}^{n} (\beta_i \beta_i) + \psi$.
- $P_1, P_2$ jmp-send $\beta_2 + \gamma_2$ to $P_0$.

The ideal world functionality for realizing $\Pi_{\text{dotpPre}}$ is presented in Fig. 12. A trivial way to instantiate $\Pi_{\text{dotpPre}}$ is to treat a dot product operation as $n$ multiplications. However, this results in a communication cost that is linearly dependent on the feature size. Instead, we instantiate $\Pi_{\text{dotpPre}}$ by a semi-honest dot product protocol followed by a verification phase to check the correctness. For the verification phase, we extend the techniques of [9, 10] to provide support for verification of dot product tuples. Setting the verification phase parameters appropriately gives a $\Pi_{\text{dotpPre}}$ whose (amortized) communication cost is independent of the feature size. Details appear in §B.

**Lemma 3.10 (Communication).** Protocol $\Pi_{\text{dotp}}$ (Fig. 11) requires an amortized communication of $3 \ell$ bits in the preprocessing phase and requires 1 round and an amortized communication of $3 \ell$ bits in the online phase.

**Proof.** During the preprocessing phase, servers execute $\Pi_{\text{dotpPre}}$. This requires communicating $3 \ell$ bits for a single semi-honest dot product protocol and $O\left(\sqrt{\frac{n}{m}}\right)$ extended ring elements for its verification. By appropriately setting the values of $n, m$, the cost of communicating $O\left(\sqrt{\frac{n}{m}}\right)$ elements can be amortized away, thereby resulting in an amortized communication cost of $3 \ell$ bits in the preprocessing phase. The online phase follows similarly to that of $\Pi_{\text{mul}}$, the only difference being that servers combine their shares corresponding to all the $n$ multiplications into one and then exchange. This requires one round and an amortized communication of $3 \ell$ bits.

**Truncation** Working over fixed-point values, repeated multiplications using FPA arithmetic can lead to an overflow resulting in loss of significant bits of information. This put forth the need for truncation [12, 15, 43, 45, 48] that re-adjusts the shares after multiplication so that FPA semantics are maintained. As shown in SecureML [45], the method of truncation would result in loss of information on the least significant bits and affect the accuracy by a very minimal amount.

For truncation, servers execute $\Pi_{\text{trgen}}$ (Fig. 13) to generate $(\langle r \rangle, \langle r^{fd} \rangle)$-pair, where $r$ is a random ring element, and $r^{fd}$ is the truncated value of $r$, i.e., the value $r$ right-shifted by $d$ bit positions. Recall that $d$ denotes the number of bits allocated for the fractional part in the FPA representation. Given $(r, r^{fd})$, the truncated value of $v$, denoted as $v^{fd}$, is computed as $v^{fd} = (v - r^{fd}) + r^{fd}$. The correctness and accuracy of this method was shown in ABY3 [43].

Protocol $\Pi_{\text{trgen}}$ is inspired from [16, 43] and proceeds as follows to generate $(\langle r \rangle, \langle r^{fd} \rangle)$. Analogous to the approach of ABY3 [43], servers generate a boolean sharing of an $\ell$-bit value $r = r_1 \oplus r_2$, non-interactively. Each server truncates its share of $r$ locally to obtain a boolean sharing of $r^{fd}$ by removing the lower $d$ bits. To obtain the arithmetic shares of $(r, r^{fd})$ from their boolean sharing, we do not, however, rely.
on the approach of ABY3 as it requires more rounds. Instead, we implicitly perform a boolean to arithmetic conversion, as was proposed in Trident [16], to obtain the arithmetic shares of \((r, r^d)\). This entails performing two dot product operations and constitutes the cost for \(\Pi_{\text{trgen}}\).

**Protocol \(\Pi_{\text{trgen}}(\mathcal{P})\)**

- To generate each bit \(r[i]\) of \(r\) for \(i \in \{0, \ldots, \ell - 1\}\), \(P_0, P_j\) for \(j \in \{1, 2\}\) sample random \(r[i]\in \mathbb{Z}_d\) and define \(r[i] = r_1[i] \oplus r_2[i]\).
- Servers generate \(\cdot\)-shares of \((r_1[i])^R\) for \(i \in \{0, \ldots, \ell - 1\}, j \in \{1, 2\}\) non-interactively following Table 2.
- Define \(\mathbf{x}\) and \(\mathbf{y}\) such that \(x = 2^{i-d+1}(r_1[i])^R\) and \(y_j = (r_2[i])^R\), respectively, for \(i \in \{d, \ldots, \ell - 1\}\). Define \(\mathbf{p}\) and \(\mathbf{q}\) such that \(p_i = 2^{i-1}(r_1[i])^R\) and \(q_i = (r_2[i])^R\), respectively, for \(i \in \{0, \ldots, \ell - 1\}\). Servers execute \(\Pi_{\text{dotpt}}(\cdot)\) to compute \(\cdot\)-shares of \(A = \mathbf{x} \cdot \mathbf{y}\) and \(B = \mathbf{p} \cdot \mathbf{q}\).
- Servers locally compute \(r^d = \sum_{i=d}^{\ell-1} x_i \cdot (r_1[i])^R + (r_2[i])^R - [A]\), and \([r]\) = \(\sum_{i=0}^{\ell-1} 2^{i}((r_1[i])^R + (r_2[i])^R)\) - \([B]\).
- \(P_0\) locally computes \(\beta_i = r + \alpha_i\), \(P_0, P_i\) set \([r]\) = \(\beta_i - \alpha_i\).

Figure 13: 3PC: Generating Random Truncated Pair \((r, r^d)\)

We now give details for generating \((\langle r_1, r^d \rangle)\). For this, servers proceed as follows: \(P_0, P_j\) for \(j \in \{1, 2\}\) sample random \(r_1[i] \in \mathbb{Z}_d\). Recall that the bit at \(\ell\)th position in \(r\) is denoted as \(r_1[i]\). Define \(r_2[i] = r_1[i] \oplus r_2[i]\) for \(i \in \{0, \ldots, \ell - 1\}\). For \(r\) defined as above, we have \(r^d[i] = r_1[i] + d \oplus r_2[i] + d\) for each \(i \in \{0, \ldots, \ell - d - 1\}\). Further,

\[
r = \sum_{i=0}^{\ell-1} 2^i r[i] = \sum_{i=0}^{\ell-1} 2^i r_1[i] \oplus r_2[i]
= \sum_{i=0}^{\ell-1} 2^i ((r_1[i])^R + (r_2[i])^R - 2(r_1[i])^R \cdot (r_2[i])^R)
= \sum_{i=0}^{\ell-1} 2^i (r_1(i)])^R + (r_2(i)])^R - \sum_{i=0}^{\ell-1} 2^i (1 \cdot i) \cdot (r_2(i)])^R \tag{5}
\]

Similarly, for \(r^d\) we have the following,

\[
r^d = \sum_{i=0}^{\ell-1} 2^{i-d} ((r_1[i])^R + (r_2[i])^R) \sum_{i=0}^{\ell-1} 2^{i-d+1} (r_1[i])^R \cdot (r_2[i])^R \tag{6}
\]

The servers non-interactively generate \([\cdot]\)-shares (arithmetic shares) for each bit of \(r_1\) and \(r_2\) as in Table 2. Given their \([\cdot]\)-shares, the servers execute \(\Pi_{\text{dotpt}}(\cdot)\) twice to compute \([\cdot]\)-share of \(A = \sum_{i=0}^{\ell-1} 2^{i-d+1} (r_1[i])^R \cdot (r_2[i])^R\), and \(B = \sum_{i=0}^{\ell-1} 2^{i-d+1} (r_1[i])^R \cdot (r_2[i])^R\). Using these values, the servers can locally compute the \([\cdot]\)-shares for \((r, r^d)\) pair following Equation 5 and 6, respectively. Note that servers need \([\cdot]\)-shares of \(r\) and \(r^d\) for \(\cdot\)-shares of \(r\) and \(\cdot\)-shares of \(r\) respectively. The \([\cdot]\)-shares can be computed from the \([\cdot]\)-shares locally as follows. Let \((\alpha, \beta, \gamma)\) be the values corresponding to the \([\cdot]\)-shares of \(r\). Since \(P_0\) knows the entire value \(r\) in clear, and it knows \(\alpha\), it can locally compute \(\beta\). Now, the servers set \([\cdot]\)-shares as: \([r]\) = \(\beta - \alpha\) and \([r]\) = \(\beta - \alpha\). The protocol appears in Fig. 13.

**Lemma 3.11** (Communication). Protocol \(\Pi_{\text{trgen}}(\cdot)\) requires an amortized communication of 12\ell bits.

**Proof.** All the operations in \(\Pi_{\text{trgen}}\) are non-interactive except for the two dot product calls required to compute \(A, B\). The cost thus follows from Lemma 3.10.

**Dot Product with Truncation** Given the \([\cdot]\)-sharing of vectors \(x\) and \(y\), protocol \(\Pi_{\text{dotpt}}(\cdot)\) (Fig. 14) allows servers to generate \([z^d]\), where \(z^d\) denotes the truncated value of \(z = x \cdot y\). A naive way is to compute the dot product using \(\Pi_{\text{dotpt}}(\cdot)\), followed by performing truncation using the \((r, r^d)\) pair. Instead, we follow the optimization of BLAZE where the online phase of \(\Pi_{\text{dotpt}}(\cdot)\) is modified to integrate the truncation using \((r, r^d)\) at no additional cost.

The preprocessing phase now consists of the execution of one instance of \(\Pi_{\text{trgen}}(\cdot)\) and the preprocessing corresponding to \(\Pi_{\text{dotpt}}(\cdot)\) (Fig. 11). In the online phase, servers enable \(P_1, P_2\) to obtain \((z - r)^d\) instead of \(\beta_2\), where \(z^* = \beta_2^* - \alpha\). Using \((z - r)^d\), both \(P_1, P_2\) then compute \((z - r)^d\) locally, truncate it to obtain \((z - r)^d\) and execute \(\Pi_{\text{sh}}(\cdot)\) to generate \([z^d]\). Finally, servers locally compute the result as \([z^d]\) = \([z^d - (z - r)^d]\) + \([r^d]\). The formal details for \(\Pi_{\text{dotpt}}(\cdot)\) protocol appear in Fig. 14.

**Protocol \(\Pi_{\text{dotpt}}(\cdot)\)**

- Servers execute the preprocessing of \(\Pi_{\text{dotpt}}(\cdot)\).
- In parallel, servers execute \(\Pi_{\text{trgen}}(\cdot)\) to generate the truncation pair \((r, r^d)\).

**Online:**

- \(P_0, P_j\), for \(j \in \{1, 2\}\), compute \([\Psi] = -\sum_{i=1}^{\ell} \beta_{\alpha_i, j} + (\beta_{\alpha_i, j}, \{\alpha_i\}) - \[r\] \) and set \([z^d]\) = \([\Psi]\) + \([r^d]\).
- \(P_1, P_0\) jamp-send \([z^d]\) to \(P_2\) and \(P_2, P_0\) jamp-send \([z^d]\) to \(P_1\).
- \(P_1, P_2\) locally compute \((z - r)^d = [z^d]\) + \([z - r]^d\) and set \((z - r) = (z - r)^d + \sum_{i=1}^{\ell} (\beta_{\alpha_i, j}) \) + \([\psi]\).
- \(P_1, P_2\) locally truncate \((z - r)\) to obtain \((z - r)^d\) and execute \(\Pi_{\text{sh}}(\cdot)\) to generate \([z][r^d]\).
- Servers locally compute \([z]\) = \([z - (z - r)^d] + [r^d]\).

Figure 14: 3PC: Dot Product Protocol with Truncation

**Lemma 3.12** (Communication). Protocol \(\Pi_{\text{dotpt}}(\cdot)\) requires an amortized communication of 15\ell bits in the preprocessing phase and requires 1 round and an amortized communication of 3\ell bits in the online phase.

**Proof.** During the preprocessing phase, servers execute the preprocessing phase of \(\Pi_{\text{dotpt}}(\cdot)\) resulting in an amortized communication of 3\ell bits (Lemma 3.10). In parallel, servers ex-
cute one instance of $\Pi_{\text{trgen}}$ protocol resulting in an additional communication of $12\ell$ bits (Lemma 3.11).

The online phase follows from that of $\Pi_{\text{dotp}}$ protocol except that, now, $P_1, P_2$ compute additive shares of $z - r$, where $z = \hat{x} \odot \hat{y}$, which is achieved using two executions of $\Pi_{\text{imp}}$ in parallel. This requires one round and an amortized communication cost of $2\ell$ bits. $P_1, P_2$ then jointly share the truncated value of $z - r$ with $P_0$, which requires one round and $6$ bits. However, this step can be deferred till the end for multiple dot product with truncation instances, which amortizes the cost.

Secure Comparison Secure comparison allows servers to check whether $x < y$, given their $[\cdot]-shares$. In FPA representation, checking $x < y$ is equivalent to checking the msb of $v = x - y$. Towards this, servers locally compute $[v] = [x] - [y]$ and extract the msb of $v$ using $\Pi_{\text{bitext}}$. In case an arithmetic sharing is desired, servers can apply $\Pi_{\text{bit2a}}$ (Fig. 10) protocol on the outcome of $\Pi_{\text{bitext}}$ protocol.

Activation Functions We now elaborate on two of the most prominently used activation functions: i) Rectified Linear Unit (ReLU) and (ii) Sigmoid (Sig).

(i) ReLU: The ReLU function, $\text{relu}(v) = \max(0,v)$, can be viewed as $\text{relu}(v) = B \cdot v$, where bit $b = 1$ if $v < 0$ and $0$ otherwise. Here $B$ denotes the complement of $b$. Given $[v]$, servers execute $\Pi_{\text{bitext}}$ on $[v]$ to generate $[b]B$. $[\cdot]B$-sharing of $B$ is locally computed by setting $b_{\text{trgen}} = 1 \odot b$. Servers execute $\Pi_{\text{bitlnj}}$ protocol on $[b]B$ and $[v]$ to obtain the desired result.

Lemma 3.13 (Communication). Protocol $\text{relu}$ requires an amortized communication of $21\ell - 6$ bits in the preprocessing phase and requires $\log \ell + 3$ rounds and an amortized communication of $16\ell - 6$ bits in the online phase.

Proof. One instance of relu protocol comprises of execution of one instance of $\Pi_{\text{bitext}}$, followed by $\Pi_{\text{bitlnj}}$. The cost, therefore, follows from Lemma 3.7, and Lemma 3.9.

(ii) Sig: In this work, we use the MPC-friendly variant of the Sigmoid function [15,43,45]. Note that $\text{sig}(v) = \overline{B_1} b_2(v + 1/2) + \overline{B_2}$, where $b_1 = 1$ if $v + 1/2 < 0$ and $b_2 = 1$ if $v - 1/2 < 0$. To compute $[\text{sig}(v)]$, servers proceed in a similar fashion as in ReLU, and hence, we skip the details.

The formal details of the MPC-friendly variant of the Sigmoid function [15,43,45] is given below:

$$\text{sig}(v) = \begin{cases} 
0 & v < -\frac{1}{2} \\
\frac{v + 1}{2} & -\frac{1}{2} \leq v \leq \frac{1}{2} \\
1 & v > \frac{1}{2}
\end{cases}$$

Lemma 3.14 (Communication). Protocol $\text{sig}$ requires an amortized communication of $39\ell - 9$ bits in the preprocessing phase and requires $\log \ell + 4$ rounds and an amortized communication of $29\ell - 9$ bits in the online phase.

Proof. An instance of sig protocol involves the execution of the following protocols in order– i) two parallel instances of $\Pi_{\text{bitext}}$ protocol, ii) once instance of $\Pi_{\text{mult}}$ protocol over boolean value, and iii) one instance of $\Pi_{\text{bitlnj}}$ and $\Pi_{\text{bit2a}}$ in parallel. The cost follows from Lemma 3.7, Lemma 3.8 and Lemma 3.9.

Maxpool, Matrix Operations and Convolutions The goal of maxpool is to find the maximum value in a vector $\hat{x}$ of $m$ values. Maximum between two elements $x_i, x_j$ can be computed by applying secure comparison, which returns a binary sharing of a bit $b$ such that $b = 0$ if $x_i > x_j$, or $1$, otherwise, followed by computing $(b)^B_x(x_i - x_j) + x_i$, which can be performed using bit injection (3.3). To find the maximum value in vector $\hat{x}$, the servers first group the values in $\hat{x}$ into pairs and securely compare each pair to obtain the maximum of the two. This results in a vector of size $m/2$. This process is repeated for $\Theta(\log m)$ rounds to obtain the maximum value in the entire vector.

Linear matrix operations, such as addition of two matrices $A, B$ to generate matrix $C = A + B$, can be computed by extending the scalar operations (addition, in this case) with respect to each element of the matrix. Matrix multiplication, on the other hand, can be expressed as a collection of dot products, where the element in the $i$th row and $j$th column of $C = A \times B$, where $A, B$ are matrices of dimension $p \times q, q \times r$, respectively, can be computed as a dot product of the $i$th row of $A$ and the $j$th column of $B$. Thus, computing $C$ of dimension $p \times r$ requires $pr$ dot products whose communication cost (amortized) is equal to that of computing $pr$ multiplications in our case. This improves the cost of matrix multiplication over the naive approach which requires $pqr$ multiplications.

Convolutions form an important building block in several neural network architectures and can be represented as matrix multiplications, as explained in the example below. Consider a 2-dimensional convolution $(CV)$ of a $3 \times 3$ input matrix $X$ with a kernel $K$ of size $2 \times 2$. This can be represented as a matrix multiplication as follows.

$$CV \left( \begin{bmatrix} x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 \\ x_7 & x_8 & x_9 \end{bmatrix}, \begin{bmatrix} k_1 & k_2 \\ k_3 & k_4 \end{bmatrix} \right) = \begin{bmatrix} x_1 & x_2 & x_3 & x_5 & x_6 \\ x_2 & x_3 & x_4 & x_6 & x_7 \\ x_3 & x_4 & x_5 & x_7 & x_8 \end{bmatrix} \begin{bmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \end{bmatrix}$$

Generally, convolving a $f \times f$ kernel over a $w \times h$ input with $p \times p$ padding using $s \times s$ stride having $i$ input channels and $o$ output channels, is equivalent to performing a matrix multiplication on matrices of dimension $(w' \times h') \times (i \cdot f \cdot f)$ and $(i \cdot f \cdot f) \times (o)$ where $w' = \frac{w - f + 2p}{s} + 1$ and $h' = \frac{h - f + 2p}{s} + 1$. We refer readers to [56] (cf. “Linear and Convolutional Layer”) and [54] for more details.
4 Robust 4PC and PPML

In this section, we extend our 3PC results to the 4-party case and observe substantial efficiency gain. First, the use of broadcast is eliminated. Second, the preprocessing of multiplication becomes substantially computationally light, eliminating the multiplication protocol (used in the preprocessing) altogether. Third, we achieve a dot product protocol with communication cost independent of the size of the vector, completely eliminating the complex machinery required as in the 3PC case. At the heart of our 4PC constructions lies an efficient 4-party jmp4 primitive, denoted as jmp4, that allows two servers to send a common value to a third server robustly.

This section is organized as follows. We begin with the secret-sharing semantics for 4 parties, for which we only use an extended version of \([\cdot,\cdot,\cdot]\)-sharing. We then explain the joint message passing primitive for four servers, followed by our 4PC protocols. We conclude this section with a detailed analysis about achieving private robustness.

Secret Sharing Semantics For a value \(v\), the shares for \(P_0, P_1\) and \(P_2\) remain the same as that for 3PC case. That is, \(P_0\) holds \([(\alpha_v), (\alpha_v)\] \(\cdot\beta_v + \gamma_v\) while \(P_i\) for \(i \in \{1, 2\}\) holds \([(\alpha_v)\] \(\cdot\beta_v, \gamma_v\). The shares for the fourth server \(P_3\) is defined as \([(\alpha_v), (\alpha_v)\] \(\cdot\beta_v\). Clearly, the secret is defined as \(v = \beta_v - \alpha_v\).

4.1 4PC Joint Message Passing Primitive

The jmp4 primitive enables two servers \(P_1, P_2\) to send a common value \(v \in \mathbb{Z}_{2^\ell}\) to a third server \(P_3\), or identify a TTP in case of any inconsistency. This primitive is analogous to jmp (Fig. 2) in spirit but is significantly optimized and free from broadcast calls. Similar to the 3PC counterpart, each server maintains a bit and \(P_1\) sends the value, and \(P_2\) holds the hash of it to \(P_3\). \(P_3\) sets its inconsistency bit to 1 when the (value, hash) pair is inconsistent. This is followed by relaying the bit to all the servers, who exchange it among themselves and agree on the bit that forms majority (1 indicates the presence of inconsistency, and 0 indicates consistency). The presence of an honest majority among \(P_1, P_2, P_3\) guarantees agreement on the presence/absence of an inconsistency as conveyed by \(P_3\). Observe that inconsistency can only be caused either due to a corrupt sender sending an incorrect value (or hash), or a corrupt receiver falsely announcing the presence of inconsistency. Hence, the fourth server, \(P_4\), can safely be employed as TTP. The ideal functionality appears in Fig. 15, and the protocol appears in Fig. 16.

Notation 4.1. We say that \(P_1, P_2\) jmp4-send \(v\) to \(P_3\) when they invoke \(\Pi_{\text{jmp4}}(P_1, P_2, P_3, v, P_3)\).

We note that the end goal of jmp4 primitive relates closely to the bi-convey primitive of FLASH [12]. Bi-convey allows two servers \(S_1, S_2\) to convey a value to a server \(R\), and in case of an inconsistency, a pair of honest servers mutually identify each other, followed by exchanging their internal randomness to recover the clear inputs, computing the circuit, and sending the output to all. Note, however, that jmp4 primitive is more efficient and differs significantly in techniques from the bi-convey primitive. Unlike in bi-convey, in case of an inconsistency, jmp4 enables servers to learn the TTP’s identity unanimously. Moreover, bi-convey demands that honest servers, identified during an inconsistency, exchange their internal randomness (which comprises of the shared keys established during the key-setup phase) to proceed with the computation. This enforces the need for a fresh key-setup every time inconsistency is detected. On the efficiency front, jmp4 simply halves the communication cost of bi-convey, giving a 2× improvement.

![Functionality \(\mathcal{F}_{\text{jmp4}}\)](image)

\(\mathcal{F}_{\text{jmp4}}\) interacts with the servers in \(\mathcal{P}\) and the adversary \(S\).

1. **Step 1:** \(\mathcal{F}_{\text{jmp4}}\) receives (Input, \(v_i\)) from senders \(P_s\) for \(s \in \{i, j\}\), (Input, \(\bot\)) from receiver \(P_R\) and fourth server \(P_4\) while it receives (Select, ttp) from \(S\). Here ttp is a boolean value, with a 1 indicating that TTP = \(P_4\) should be established.

2. **Step 2:** If \(v_i = v_j\) and ttp = 0, or if \(S\) has corrupted \(P_s\), set \(msg_i = msg_j = msg_4 = \bot\), \(msg_k = v_i\) and go to **Step 4**.

3. **Step 3:** Else : Set \(msg_i = msg_j = msg_4 = msg_k = P_4\).

4. **Step 4:** Send (Output, \(msg_i\)) to \(P_s\) for \(s \in \{0, 1, 2, 3\}\).

![Protocol \(\Pi_{\text{jmp4}}(P_1, P_2, P_3, v, P_4)\)](image)

\(P_s \in \mathcal{P}\) initializes an inconsistency bit \(b_s = 0\). If \(P_s\) remains silent instead of sending \(b_s\) in any of the following rounds, the recipient sets \(b_s\) to 1.

Send Phase: \(P_s\) sends \(v\) to \(P_k\).
Verify Phase: \(P_s\) sends \(H(v)\) to \(P_k\).
- \(P_k\) sets \(b_k = 1\) if the received values are inconsistent or if the value is not received.
- \(P_k\) sends \(b_k\) to all servers. \(P_s\) for \(s \in \{i, j, l\}\) sets \(b_s = b_k\).
- \(P_3\) for \(s \in \{i, j, l\}\) mutually exchange their bits. \(P_3\) resets \(b_s = b'\) where \(b'\) denotes the bit which appears in majority among \(b_i, b_j, b_l\).
- All servers set TTP = \(P_4\) if \(b' = 1\), terminate otherwise.

![Figure 15: 4PC: Ideal functionality for jmp4 primitive](image)

![Figure 16: 4PC: Joint Message Passing Primitive](image)

**Lemma 4.2 (Communication).** Protocol \(\Pi_{\text{jmp4}}\) (Fig. 16) requires 1 round and an amortized communication of \(\ell\) bits in the online phase.

**Proof:** Server \(P_s\) sends the value \(v\) to \(P_k\) while \(P_s\) sends hash of the same to \(P_k\). This accounts for one round of communication. Values sent by \(P_s\) for several instances can be concatenated and hashed to obtain a single value. Hence the cost of sending the hash gets amortized over multiple instances. Similarly, the
two round exchange of inconsistency bits to establish a TTP can be combined for multiple instances, thereby amortizing this cost. Thus, the amortized cost of this protocol is $\ell$ bits.

\subsection{4PC Protocols}

In this section, we revisit the protocols from 3PC (§3) and suggest optimizations leveraging the presence of an additional honest party in the system. We provide security proofs in §C.2.

\paragraph{Sharing Protocol} To enable $P_i$ to share a value $v$, protocol $\Pi_{\text{sh}d}$ (Fig. 17) proceeds similar to that of 3PC case with the addition that $P_3$ also samples the values $[\alpha_r]_1, [\alpha_r]_2, \nu_r$ using the shared randomness with the respective servers. On a high level, $P_i$ computes $\beta_v = v + [\alpha_r]_1 + [\alpha_r]_2$ and sends $\beta_v$ (or $\beta_v + \gamma_r$) to another server and they together jmp4-send this information to the intended servers. The formal protocol for sharing a value $v$ by $P_i$ is given in Fig. 17.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{protocol_sharing.png}
\caption{4PC: Generating $[v]$-shares by server $P_i$}
\end{figure}

\paragraph{Joint Sharing Protocol} Protocol $\Pi_{\text{sh}d}$ enables a pair of (unordered) servers $(P_i, P_j)$ to jointly generate a $[v]$-sharing of value $v \in \mathbb{Z}_{2^\ell}$ known to both of them. In case of an inconsistency, the server outside the computation serves as a TTP. The protocol is described in Fig. 18.

When $P_3, P_0$ want to jointly share a value $v$ which is available in the preprocessing phase, protocol $\Pi_{\text{sh}d}$ can be performed with a single element of communication (as opposed to 2 elements in Fig. 18). $P_0, P_3$ can jointly share $v$ as follows. $P_0, P_3$ sample a random $r \in \mathbb{Z}_{2^\ell}$ and set $[\alpha_r]_1 = r$. $P_0, P_3$ set $[\alpha_r]_2 = - (r + v)$ and jmp4-send $[\alpha_r]_2$ to $P_2$. This is followed by servers locally setting $\gamma_r = \beta_v = 0$.

We further observe that servers can generate a $[v]$-sharing of $v$ non-interactively when $v$ is available with $P_0, P_1, P_2$. For this, servers set $[\alpha_r]_1 = [\alpha_r]_2 = \nu_r = 0$ and $\beta_v = v$. We abuse notation and use $\Pi_{\text{sh}d}(P_0, P_1, P_2, v)$ to denote this sharing.

\paragraph{Lemma 4.4 (Communication)} In the online phase, $\Pi_{\text{sh}d}$ (Fig. 18) requires 1 round and an amortized communication of $2\ell$ bits when $(P_3, P_s)$ for $s \in \{0, 1, 2\}$ share a value, and requires an amortized communication of $\ell$ bits, otherwise.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{protocol_joint_sharing.png}
\caption{4PC: Generating $[v]$-shares by server $P_i$}
\end{figure}

\begin{proof}
When $(P_3, P_s)$ for $s \in \{0, 1, 2\}$ want to share a value $v$, there are two parallel calls to $\Pi_{\text{mp}d}$ which requires an amortized communication of $2\ell$ bits and one round. In the other cases, $\Pi_{\text{mp}d}$ is invoked only once, resulting in an amortized communication of $\ell$ bits.
\end{proof}

\paragraph{Joint Sharing Protocol} In some protocols, $P_3$ is required to generate $(\cdot)$-sharing of a value $v$ in the preprocessing phase, where $(\cdot)$-sharing of $v$ is same as that defined in 3PC (where $v = v_0 + v_1 + v_2$, and $P_0$ possesses $(v_0, v_1)$, $P_1$ possesses $(v_1, v_2)$, and $P_2$ possess $(v_2, v_0)$) with the addition that $P_3$ now


possesses \((v_0, v_1, v_2)\). We call the resultant protocol \(\Pi_{ash4}\) and it appears in Fig. 19.

**Protocol \(\Pi_{ash4}(P_3, v)\)**

**Preprocessing:**

- Servers \(P_0, P_1, P_2\) sample a random \(v_1 \in \mathbb{Z}_2^m\), while servers \(P_0, P_3, P_4\) sample a random \(v_0 \in \mathbb{Z}_2^m\).
- \(P_3\) computes \(v_2 = v - v_0 - v_1\) and sends \(v_2\) to \(P_2, P_3, P_1\). 

Figure 19: 4PC: \(\langle \cdot \rangle\)-sharing of value \(v\) by \(P_3\).

Note that servers can locally convert \(\langle v \rangle\) to \([v]\) by setting their shares as shown in Table 5.

<table>
<thead>
<tr>
<th>([v])</th>
<th>((-v_1, -v_0, 0))</th>
<th>((-v_1, v_2, -v_1))</th>
<th>((-v_0, v_2, -v_2))</th>
<th>((-v_0, -v_1, -v_2))</th>
</tr>
</thead>
<tbody>
<tr>
<td>([v])</td>
<td>((-v_1, -v_0, 0))</td>
<td>((-v_1, v_2, -v_1))</td>
<td>((-v_0, v_2, -v_2))</td>
<td>((-v_0, -v_1, -v_2))</td>
</tr>
</tbody>
</table>

Table 5: Local conversion of shares from \(\langle \cdot \rangle\)-sharing to \([ \cdot \) -sharing for a value \(v\). Here, \([a_i]_1 = -v_1, [a_i]_2 = -v_0, [a_i] = v_2, [a_i] = -v_2\).

**Lemma 4.5 (Communication).** Protocol \(\Pi_{ash4}\) (Fig. 19) requires 2 rounds and an amortized communication of \(2\ell\) bits.

**Proof.** Communicating \(v_2\) to \(P_2\) requires \(\ell\) bits and 1 round. This is followed by one invocation of \(\Pi_{mp4}\) which requires \(\ell\) bits and 1 round. Thus, the amortized communication cost is \(2\ell\) bits and two rounds. \(\Box\)

**Multiplication Protocol**

Given the \([\cdot\) -shares of \(x\) and \(y\), protocol \(\Pi_{mp4}\) (Fig. 20) allows servers to compute \([z]\) with \(z = xy\). When compared with the state-of-the-art 4PC GOD protocol of FLASH [12], our solution improves communication in both, the preprocessing and online phase, from 6 to 3 ring elements. Moreover, our communication cost matches with the state-of-the-art 4PC protocol of Trident [16] that only provides security with fairness.

Recall that the goal of preprocessing in 3PC multiplication was to enable \(P_1, P_2\) to obtain \(\psi\), and \(P_0, P_1\) for \(i \in \{1, 2\}\) obtain \([\chi_i]\), where \(\chi = \gamma_x \alpha_x + \gamma_y \alpha_y + \gamma_x \gamma_y - \psi\). Here \(\psi\) is a random value known to both \(P_1, P_2\). With the help of \(P_3\), we let the servers obtain the respective preprocessing data as follows:

- \(P_0, P_3, P_1\) together sample random \([\Gamma_{xy}]_1 \in \mathbb{Z}_2^m\), \(P_0, P_3\) locally compute \(\Gamma_x = \alpha_x \alpha_y\), set \([\Gamma_{xy}]_2 = \Gamma_{xy} - [\Gamma_{xy}]_1\), and \(mp4\)-send \([\Gamma_{xy}]_2\) to \(P_2, P_1, P_2, P_3\) locally sample \(\psi, r\) and generate \([\cdot\) -shares of \(\psi\) by setting \([\psi]_1 = r\) and \([\psi]_2 = \psi - r\). Then \(P_2, P_3\) for \(j \in \{1, 2\}\) compute \([\chi]_j = \gamma_x [\alpha_j]_1 + \gamma_y [\alpha_j]_2 + [\Gamma_{xy}]_1 - [\psi]_j\) and \(mp4\)-send \([\chi]_j\) to \(P_0\). The online phase is similar to that of 3PC, apart from \(\Pi_{mp4}\) being used instead of \(\Pi_{mp}\) for communication. Since \(P_3\) is not involved in the online computation phase, we can safely assume \(P_3\) to serve as the TTP for the \(\Pi_{mp}\) executions in the online phase.

**Protocol \(\Pi_{mp4}(P_3, [x], [y])\)**

**Preprocessing:**

- \(P_0, P_3, P_1, P_2\), for \(j \in \{1, 2\}\), sample random \([\alpha_i]_j \in \mathbb{Z}_2^m\), and \(P_0, P_3, P_1\) sample random \([\Gamma_{xy}]_1 \in \mathbb{Z}_2^m\).
- \(P_1, P_2, P_3\) sample random \(\gamma_x, \gamma_y, r \in \mathbb{Z}_2^m\) and set \([\psi]_1 = r, [\psi]_2 = \psi - r\).
- \(P_0, P_3\) set \([\Gamma_{xy}]_2 = \Gamma_{xy} - [\Gamma_{xy}]_1\), where \(\Gamma_{xy} = \alpha_x \alpha_y\), \(P_0, P_3\) \(mp4\)-send \([\Gamma_{xy}]_2\) to \(P_2, P_3\).
- \(P_3, P_2, P_0\), for \(j \in \{1, 2\}\), set \([\chi]_j = \gamma_x [\alpha_j]_1 + \gamma_y [\alpha_j]_2 + [\Gamma_{xy}]_1 - [\psi]_j\), \(P_3, P_3\) \(mp4\)-send \([\chi]_j\) to \(P_0\), while \(P_2, P_3\) \(mp4\)-send \([\chi]_j\) to \(P_2\).

**Online:**

- \(P_0, P_j, P\), for \(j \in \{1, 2\}\), compute \([\beta_j]_1 = -([\beta_x + \gamma_x] [\alpha_j]_1 - ([\beta_y + \gamma_y] [\alpha_j]_2 + [\psi])\), \(P_0, P_j\) \(mp4\)-send \([\beta_j]_1\) to \(P_2, P_2\), and \(P_2, P_0\) \(mp4\)-send \([\beta_j]_2\) to \(P_1\).
- \(P_j, P_0, P\), for \(j \in \{1, 2\}\), computes \([\beta_j]_2 = [\beta_j]_1 + [\beta_j]_2\) and sets \(\beta_x = [\beta_j]_1 + [\beta_j]_2\) and \(\beta_x + \gamma_x\) to \(P_0\).
At a high-level, the BA protocol proceeds as follows. Let us first focus on the BA protocol. This section provides details on robust realizations of the BA protocol. To construct a value $v$ among the three values received as input, we ensure that all honest servers are on the same page. Once the majority of the servers agree on the value $v$, we set the majority value for each share and reconstruct the output. Notice that, as opposed to the 3PC case, this protocol does not require commitments. The formal protocol for reconstruction is given in Fig. 21.

**Protocol $\Pi_{\text{rec4}}(\mathcal{P}, [v])$**

**Online**
- $P_0$ receives $\gamma_v$ from $P_1, P_2$ and $H(\gamma_v)$ from $P_3$.
- $P_1$ receives $[\alpha_v]_2$, $[\alpha_v]_2$, and $H([\alpha_v]_2)$ from $P_0$.
- $P_2$ receives $[\alpha_v]_1$, $[\alpha_v]_1$, and $H([\alpha_v]_1)$ from $P_1$.
- $P_3$ receives $\beta_v + \gamma_v$ from $P_0, P_1$, and $H(\beta_v + \gamma_v)$ from $P_2$.
- $P_4 \in \mathcal{P}$ selects the missing share forming the majority among the values received and reconstructs the output.

![Figure 21: 4PC: Reconstruction of $v$ among the servers](image)

**Lemma 4.7** (Communication). $\Pi_{\text{rec4}}$ (Fig. 21) requires an amortized communication of $8\ell$ bits and 1 round in the online phase.

**Proof.** Each $P_s$ for $s \in \{0, 1, 2, 3\}$ receives the missing share in clear from another server, while the hash of it from the third. As before, the missing share sent by the third server can be concatenated over multiple instances and hashed to obtain a single value. Thus, the amortized communication cost is $2\ell$ bits per server, resulting in a total cost of $8\ell$ bits.

### 4.3 Building Blocks for PPML using 4PC

This section provides details on robust realizations of the PPML building blocks in 4-server setting (for the same blocks as in §3.3). We provide the security proofs in §C.2.

#### Input Sharing and Output Reconstruction in SOC Setting

We extend input sharing and reconstruction in the SOC setting as follows. To generate $[\cdot]$-shares for its input $v$, $U$ receives each of the shares $[\alpha_v]_1$, $[\alpha_v]_2$, and $\gamma_v$ from three out of four servers as well as a random value $r \in \mathbb{Z}_{2^\ell}$ sampled together by $P_0, P_1, P_2$ and accepts the values that form the majority. $U$ locally computes $u = \nu + [\alpha_v]_1 + [\alpha_v]_2 + \gamma_v + r$ and sends $u$ to all the servers. Servers then execute a two round byzantine agreement (BA) [49] to agree on $u$ (or $\perp$). At a high-level, the BA protocol proceeds as follows. Let us denote the value received by $P_1$ from $U$ as $u_i$. To agree on $u$ received from $U$, the servers first arrive on an agreement regarding each $u_i$ received by $P_i$. This is followed by selecting the majority value among $u_1, u_2, u_3, u_4$. As before, the majority rule ensures that all honest servers are on the same page. Once each of the values are agreed on, every server takes the majority among $u_1, u_2, u_3, u_4$ as the value sent by $U$. If no value appears in majority, a default value is chosen. We refer the readers to [49] for the formal details of the agreement protocol. On successful completion of BA, $P_0$ computes $\beta_v + \gamma_v$ from $u$ while $P_1, P_2$ compute $\beta_v$ from $u$ locally. For the reconstruction of a value $v$, servers send their $[\cdot]$-shares of $v$ to $U$, who selects the majority value for each share and reconstructs the output. At any point, if a TTP is identified, the servers proceed as follows. All servers send their $[\cdot]$-share of the input to the TTP. TTP picks the majority value for each share and computes the function output. It then sends this output to $U$. $U$ also receives the identity of the TTP from all servers and accepts the output received from the TTP forming majority.

#### Bit Extraction Protocol

This protocol enables the servers to compute a boolean sharing of the most significant bit (MSB) of a value $v \in \mathbb{Z}_{2^\ell}$ given the arithmetic sharing $[v]$. To compute the MSB, we use the optimized Parallel Prefix Adder (PPA) circuit from ABY3 [43], which takes as input two boolean values and outputs the MSB of the sum of the inputs. The circuit requires $2(\ell - 1)$ AND gates and has a multiplicative depth of $\log \ell$. The protocol for bit extraction ($\Pi_{\text{bext4}}$) involves computing the boolean PPA circuit using the protocols described in §4. The two inputs to this boolean circuit are generated as follows. The value $v$ whose MSB needs to be extracted can be represented as the sum of two values as $v = \beta_v + (-\alpha_v)$ where the first input to the circuit will be $\beta_v$ and the second input will be $-\alpha_v$. Since $\beta_v$ is held by $P_1, P_2$, servers execute $\Pi_{\text{bext4}}$ to generate $[\beta_v]^B$. Similarly, $P_0, P_1$ possess $\alpha_v$, and servers execute $\Pi_{\text{bext4}}$ to generate $[-\alpha_v]^B$. Servers in $\mathcal{P}$ use the $[\cdot]^B$ shares of these two inputs ($\beta_v, -\alpha_v$) to compute the optimized PPA circuit which outputs the $[\text{msb}(v)]^B$.

**Lemma 4.8** (Communication). The protocol $\Pi_{\text{bext4}}$ requires an amortized communication of $7\ell - 6$ bits in the preprocessing phase, and $\log \ell$ rounds with amortized communication of $7\ell - 6$ bits in the online phase.

**Proof.** Generation of boolean sharing of $\alpha_v$ requires $\ell$ bits in the preprocessing phase (since $\Pi_{\text{bext4}}$ with $P_0, P_1$ can be achieved with $\ell$ bits of communication in the preprocessing phase), and generation of boolean sharing of $\beta_v$ requires $\ell$ bits and one round (which can be deferred towards the end of the protocol thereby requiring one round for several instances) in the online phase. Further, the boolean PPA circuit to be computed requires $2(\ell - 1)$ AND gates. Since each AND gate requires $\Pi_{\text{mult4}}$ to be executed, it requires an amortized communication of $6\ell - 6$ bits in both the preprocessing phase and the online phase. Thus, the overall communication is $7\ell - 6$ bits, in both, the preprocessing and online phase. The circuit has a multiplicative depth of $\log \ell$ which results in $\log \ell$ rounds in the online phase.

\[ \square \]
Bit2A Protocol
This protocol enables servers to compute the arithmetic sharing of a bit $b$ given its boolean sharing. Let $b^R$ denote the value of $b$ in the ring $\mathbb{Z}/2\mathbb{Z}$. We observe that $b^R$ can be written as follows, $b^R = (\alpha_b \oplus \beta_b)^R = \alpha_b^R + \beta_b^R - 2\alpha_b \beta_b^R$. Thus, to obtain an arithmetic sharing of $b^R$, the servers can compute an arithmetic sharing of $\alpha_b^R$, $\beta_b^R$ and $\beta_b^R$ respectively. This can be done as follows, $P_0$, $P_1$, $P_2$, $P_3$ execute $\Pi_{ash4}$ on $\alpha_b^R$ in the preprocessing phase to generate $[\alpha_b^R]$. Similarly, $P_1$, $P_2$ execute $\Pi_{ash4}$ on $\beta_b^R$ in the online phase to generate $[\beta_b^R]$. This is followed by $\Pi_{mul4}$ on $[\alpha_b^R], [\beta_b^R]$, followed by local computation to obtain $[b^R]$.

Protocol $\Pi_{bit2A4}(P, [b]_B)$

**Preprocessing:**
- Servers execute $\Pi_{ash4}(P_3, e^R)$ (Fig. 19) where $e = \alpha_b \oplus \gamma_b$. Let the shares be $e^R_0 = (e_0, e_1), e^R_1 = (e_1, e_2), e^R_2 = (e_2, e_0)$. $\Pi_{ash4}$ computes $x_1 = y_0 \oplus y_b$, and $\Pi_{mp4}$ sends $x_1$ to $P_0$.
- $P_1$, $P_2$ compute $y_1 = (e_1 + e_2)(1 - 2\beta_b^R) + \beta_b^R + r$, and $\Pi_{mp4}$ sends $y_1$ to $P_0$.
- $P_0$ computes $y_2 = e_0(1 - 2\beta_b^R) - r$, and $\Pi_{mp4}$ sends $y_2$ to $P_0$.
- $P_0$ computes $x = e \oplus r_b = [\alpha_b]_1 \oplus [\alpha_b]_2 \oplus x_1$ and checks if $H(x^R - y_1) = H(y_2)$.
- If verification fails, $P_0$ sets flag = 1, else it sets flag = 0. $P_0$ sends flag to $P_1$. Next, $P_1$, $P_0$ $\Pi_{mp4}$-send flag to $P_2$ and $P_3$.
- Servers set $\Pi_{TP}$ if flag = 1.
- If verification succeeds, servers locally convert $e^R$ to $[e^R]$ by setting their shares according to Table 5.

**Online:**
- Servers execute $\Pi_{ash4}(P_0, P_1, P_2, e^R)$ where $c = \beta_b \oplus \gamma_b$.
- Servers execute $\Pi_{mul4}(P, [e]_R, [e^R])$ to generate $[e^R][e^R]$.
- Servers compute $[b^R] = [e^R] + [e^R] - 2[e^R][e^R]$.

![Figure 22: 4PC: Bit2A Protocol](image)

While the above approach serves the purpose, we now provide an improved version, which further helps in reducing the online cost. We observe that $b^R$ can be written as follows, $b^R = (\alpha_b \oplus \beta_b)^R = (\alpha_b \oplus \beta_b)(\beta_b \oplus \gamma_b) = (\alpha_b + \gamma_b - 2\alpha_b \beta_b^R)$ where $\alpha = \alpha_b \oplus \gamma_b$ and $c = \beta_b \oplus \gamma_b$. Thus, to obtain an arithmetic sharing of $b^R$, $P_3$ generates $\langle \cdot \rangle$-sharing of $\alpha_b^R$. To ensure the correctness of the shares, the servers $P_0$, $P_1$, $P_2$ check whether the following equation holds: $(e \oplus r_b)^R = e^R + r_b^R - 2e^R\beta_b^R$. If the verification fails, a $\Pi_{TP}$ is identified. Else, this is followed by servers locally converting $e^R$-shares to $[e^R]$ according to Table 5, followed by multiplying $[e^R][e^R]$ and locally computing $[b^R] = [e^R] + [e^R] - 2[e^R][e^R]$. Note that during $\Pi_{ash4}(P_0, P_1, P_2, e^R)$ since $\alpha_b$ and $\gamma_b$ are set to 0, the preprocessing of multiplication can be performed locally. The formal protocol appears in Fig. 22.

**Lemma 4.9 (Communication).** $\Pi_{bit2A4}$ (Fig. 22) requires an amortized communication of $3\ell + 4$ bits in the preprocessing phase, and 1 round with amortized communication of $3\ell$ bits in the online phase.

**Proof.** During preprocessing, one instance of $\Pi_{ash4}$ requires $2\ell$ bits of communication. Further, sending $x_1$ requires 1 bit, while sending $y_1$ requires $\ell$ bits. Sending of $H(y_2)$ can be amortized over several instances. Finally, communicating flag requires 3 bits. Thus, the overall amortized communication cost in the preprocessing phase is $3\ell + 4$ bits. In the online phase, joint sharing of $c^R$ can be performed non-interactively. The only cost is due to the online phase of multiplication which requires $3\ell$ bits and one round. Thus, the amortized communication cost in the online phase is $3\ell$ bits with one round of communication.

**Bit Injection Protocol**
Given the boolean sharing of a bit $b$, denoted as $[b]_B$, and the arithmetic sharing of $v \in \mathbb{Z}/2\mathbb{Z}$, protocol $\Pi_{binj4}$ (Fig. 23) computes $[v]_B \cdot [b]_B$ to generate $[v]_B$, followed by servers computing $[v]_B$ by executing $\Pi_{mul4}$ protocol on $[b]_B$ and $[v]_B$. Instead, we provide an optimized variant which helps in reducing the communication cost of the naive approach in both, the preprocessing and online phase. We give the details next.

Let $z = b^R \cdot v$, where $b^R$ denotes the value of $b$ in $\mathbb{Z}/2\mathbb{Z}$. Then, during the communication of $[z]$, we observe the following:

$z = b^R \cdot v = (\alpha_b \oplus \beta_b)(\beta_b - \alpha_b)$

$= ((\alpha_b \oplus \gamma_b) \oplus (\beta_b \oplus \gamma_b))(\beta_b - \alpha_b)$

$= (c_b + \gamma_b) - (c_b - \gamma_b)$

$= (c_b - \gamma_b) + (c_b \cdot \gamma_b) - (c_b - \gamma_b) + (c_b \cdot \gamma_b)$

$= c_b \cdot \gamma_b - c_b \cdot \gamma_b + (c_b \cdot \gamma_b) - (c_b \cdot \gamma_b)$

$= e^R \cdot \gamma_b - e^R \cdot \gamma_b + (c_b \cdot \gamma_b)$

where $c_b = \beta_b \oplus \gamma_b$, $e_b = \alpha_b \oplus \gamma_b$, $c_v = \beta_v \oplus \gamma_v$ and $e_v = \alpha_v \oplus \gamma_v$. The protocol proceeds with $P_0$ generating $\langle \cdot \rangle$-shares of $e_b^R$ and $e_x = e^R \cdot e_v$, followed by verification of the same by $P_0$, $P_1$, $P_2$. If verification succeeds, then to enable $P_2$ to compute $b_v = z + \alpha_v$, $P_1$, $P_2$ $\Pi_{mp4}$-send the missing share of $b_v$ to $P_2$. Similarly for $P_1$. Next, $P_1$, $P_2$ reconstruct $b_v$, $\Pi_{mp4}$-send $b_v + \gamma_v$ to $P_0$ completing the protocol.

**Lemma 4.10 (Communication).** $\Pi_{binj4}$ requires an amortized communication cost of $6\ell + 4$ bits in the preprocessing phase, and requires 1 round with an amortized communication of $3\ell$ bits in the online phase.

**Proof.** The preprocessing phase requires two instances of $\Pi_{ash4}$ which requires $4\ell$ bits of communication. Verifying correctness of $\langle e^R_b \rangle$ requires $\ell + 1$ bits, whereas for $\langle e_x \rangle$ we require $\ell$ bits. Finally, communicating flag requires 3 bits. This results in the amortized communication of $6\ell + 4$ bits in the preprocessing phase. The online phase consists of three calls to $\Pi_{mp4}$ which requires $3\ell$ bits of amortized communication. Note that the last call can be deferred towards the
end of the computation, thereby requiring a single round for multiple instances. Thus, the number of rounds required in the online phase is one.

Protocol $\Pi_{\text{bitin4}}(P, \lfloor b \rfloor^B, \lfloor v \rfloor)$

Let $c_0 = b_0 \oplus y_b$, $e_b = \alpha_b \oplus y_b$, $c_0 = b_0 \oplus y_0$, $e_v = \alpha_v \oplus y_v$ and $e_z = z^R e_v$.

**Preprocessing:**

- $P_0, P_3, P_j$ for $j \in \{1, 2\}$ sample $\lfloor a \rfloor_1 \in \mathbb{Z}_{2^e}$ while $P_1, P_2, P_3$ sample $\lfloor y \rfloor_2 \in \mathbb{Z}_{2^e}$.
- Servers execute $\Pi_{\text{ash4}}(P_1, e_z^B)$ and $\Pi_{\text{ash4}}(P_3, e_z^B)$. Shares of $(e_v)$ are set locally as $e_{v_1} = [\alpha_{v_1}], e_{v_2} = [\alpha_{v_2}], e_{v_3} = [\gamma_{v_3}]$.
- Servers verify correctness of $(e_z)$ using steps similar to $\Pi_{\text{bit2AA}}$. Correctness of $(e_z)$ is verified as follows.

- $P_0, P_3, P_j$ for $j \in \{1, 2\}$ sample a random $r_j \in \mathbb{Z}_{2^e}$. While $P_1, P_2, P_3$ sample a random $r_0 \in \mathbb{Z}_{2^e}$, $P_0, P_3$ set $a_0 = r_1 - r_2$.
- $P_1, P_2, P_3$ compute $a_1 = r_0 - r_1$, $r_2 - r_0$.
- $P_1, P_2, P_3$ compute $x_1 = e_{v_1} + e_{v_2} + e_{v_3} + e_{v_4} + e_{v_5}$.
- $P_2, P_3$ compute $x_2 = e_{v_1} + e_{v_2} + e_{v_3} + e_{v_4} + e_{v_5} + a_1$.
- $P_1$ computes $y_1 = e_{v_1} + e_{v_2} + e_{v_3} + e_{v_4} + e_{v_5}$.
- $P_1, P_2, P_3$ jmp4-send $H(-y_2)$ to $P_0$, where $y_2 = y_2 - e_{v_2}$.
- $P_0$ computes $y_0 = x_0 - e_{v_0}$, and checks if $H(y_0 + y_1) = H(-y_2)$.
- If verification fails, $P_0$ sets flag = 1, else it sets flag = 0. $P_0$ sends flag to $P_1$. Next, $P_1, P_0$ jmp4-send flag to $P_2$ and $P_3$.
- Servers set TTP if flag = 1.

**Online:**

- $P_0, P_1$ compute $u_1 = -c_z^R e_v + (c_0 - 2c_z^R e_v) e_z^B + (2c_z^R - 1) e_z + [a_1]$, and jmp4-send $u_1$ to $P_2$.
- $P_0, P_2$ compute $u_2 = -c_z^R e_v + (c_0 - 2c_z^R e_v) e_z^B + (2c_z^R - 1) e_z + [a_2]$, and jmp4-send $u_2$ to $P_1$.
- $P_1, P_2$ compute $\beta_z = u_1 + u_2 - c_z^R e_v + (c_0 - 2c_z^R e_v) e_z^B + (2c_z^R - 1) e_z + c_z^B e_v$.
- $P_1, P_2$ jmp4-send $\beta_z + y_2$ to $P_0$.

Figure 23: 4PC: Bit Injection Protocol

**Dot Product** Given $[\cdot]$-shares of two $n$-sized vectors $\mathbf{x}, \mathbf{y}$, protocol $\Pi_{\text{dot4p}}$ (Fig. 24) enables servers to compute $[z]$ with $z = \mathbf{x} \cdot \mathbf{y}$. The protocol is essentially similar to $n$ instances of multiplications of the form $z_i = x_i y_i$ for $i \in [n]$. But instead of communicating values corresponding to each of the $n$ instances, servers locally sum up the shares and communicate a single value. This helps to obtain a communication cost independent of the size of the vectors.

In more detail, the dot product protocol proceeds as follows. During the preprocessing phase, similar to the multiplication protocol $P_0, P_1, P_3$ sample a random $[\Gamma_{\mathbf{x} \cdot \mathbf{y}}]_1$. $P_0, P_3$ compute $\Gamma_{\mathbf{x} \cdot \mathbf{y}} = \sum_{i=1}^{n} a_i \alpha_j$ and jmp4-send $[\Gamma_{\mathbf{x} \cdot \mathbf{y}}]_2 = \Gamma_{\mathbf{x} \cdot \mathbf{y}} - [\Gamma_{\mathbf{x} \cdot \mathbf{y}}]_1$ to $P_2$. $P_1, P_2, P_3$ sample a random $\psi$, and generate its $[\cdot]$-shares locally. Servers $P_3, P_j$ for $j \in \{1, 2\}$ then compute $[\chi_j] = \sum_{i=1}^{n} (\gamma_i, [\alpha_j] + \gamma_i [\alpha_j]) + \Gamma_{\mathbf{x} \cdot \mathbf{y}} - [\psi]$, and jmp4-send $[\chi_j]$ to $P_0$. The formal protocol is given in Fig. 24.

Protocol $\Pi_{\text{dot4p}}(P, \lfloor [\mathbf{x}] \rfloor, \lfloor [\mathbf{y}] \rfloor)$

**Preprocessing:**

- $P_0, P_3, P_j$, for $j \in \{1, 2\}$, sample random $[\alpha_j] \in \mathbb{Z}_{2^e}$, while $P_0, P_1, P_3$ sample random $[\Gamma_{\mathbf{x} \cdot \mathbf{y}}]_1 \in \mathbb{Z}_{2^e}$.
- $P_1, P_2, P_3$ together sample random $\gamma, \psi, \tau \in \mathbb{Z}_{2^e}$ and set $[\psi]_1 = \tau, [\psi]_2 = \psi - \tau$.
- $P_0, P_3$ compute $[\Gamma_{\mathbf{x} \cdot \mathbf{y}}]_2 = [\Gamma_{\mathbf{x} \cdot \mathbf{y}}]_1 - [\Gamma_{\mathbf{x} \cdot \mathbf{y}}]_2$, where $[\Gamma_{\mathbf{x} \cdot \mathbf{y}}]_2 = \sum_{i=1}^{n} \alpha_i \alpha_j, P_0, P_3$ jmp4-send $[\Gamma_{\mathbf{x} \cdot \mathbf{y}}]_2$ to $P_2$.
- $P_3, P_j$, for $j \in \{1, 2\}$, set $[\chi_j] = \sum_{i=1}^{n} (\gamma_i [\alpha_j] + \gamma_i [\alpha_j]) + \Gamma_{\mathbf{x} \cdot \mathbf{y}} - [\psi]_j$.
- $P_1, P_3$ jmp4-send $[\chi_1]$ to $P_0$, and $P_2, P_3$ jmp4-send $[\chi_2]$ to $P_0$.

**Online:**

- $P_0, P_j$, for $j \in \{1, 2\}$, compute $[\beta_j] = -\sum_{i=1}^{n} ([\beta_i + \gamma_i [\alpha_j] + \gamma_i [\alpha_j]) + [\chi_j]$.
- $P_1, P_0$ jmp4-send $[\beta_1]$ to $P_2$, while $P_2, P_0$ jmp4-send $[\beta_2]$ to $P_1$.
- $P_j$, for $j \in \{1, 2\}$ computes $\beta_1 = \beta_1 + [\beta_2]$ and sets $\beta_2 = \beta_2 + [\sum_{i=1}^{n} \alpha_i \gamma_j] + \psi$.
- $P_1, P_2$ jmp4-send $\beta_x + \gamma_x$ to $P_0$.

Figure 24: 4PC: Dot Product Protocol ($z = \mathbf{x} \cdot \mathbf{y}$)

**Lemma 4.11 (Communication).** $\Pi_{\text{dot4p}}$ (Fig. 24) requires an amortized communication of $3 \ell$ bits in the preprocessing phase, and $1$ round and an amortized communication of $3 \ell$ bits in the online phase.

**Proof.** The preprocessing phase requires three calls to $\Pi_{\text{jmp4p}}$, one to jmp4-send $[\Gamma_{\mathbf{x} \cdot \mathbf{y}}]_2$ to $P_2$, and two to jmp4-send $[\chi_j]_2$ to $P_0$. Each invocation of $\Pi_{\text{jmp4p}}$ requires $\ell$ bits resulting in the amortized communication cost of preprocessing phase to be $3 \ell$ bits. In the online phase, there are $2$ parallel invocations of $\Pi_{\text{jmp4p}}$ to jmp4-send $[\beta_1]_2, [\beta_2]_2$ to $P_0, P_1$, respectively, which require amortized communication of $2 \ell$ bits and one round. This is followed by another call to $\Pi_{\text{jmp4p}}$ to jmp4-send $\beta_x + \gamma_x$ to $P_0$ which requires one more round and amortized communication of $\ell$ bits. As in the multiplication protocol, this step can be delayed till the end of the protocol and clubbed for multiple instances. Thus, the online phase requires one round and an amortized communication of $3 \ell$ bits.

**Truncation** Given the $[\cdot]$-sharing of a value $v$ and a random truncation pair $([r], [r^d])$, the $[\cdot]$-sharing of the truncated value $v^d$ (right shifted value by, say, $d$ positions) can be computed as follows. Servers open the value $(v - r)$, truncate it and add it to $[r^d]$ to obtain $[v^d]$. The protocol for generating the truncation pair $([r], [r^d])$ is described in Fig. 25.
Lemma 4.12 (Communication). $\Pi_{\text{trgen}4}(P)$ requires $1$ round and an amortized communication of $\ell$ bits in the online phase.

Proof. The cost follows directly from that of $\Pi_{\text{jmp}4}$ (Lemma 4.2 and 4.4).

Dot Product with Truncation Protocol $\Pi_{\text{dotpt}4}$ (Fig. 26) enables servers to generate $\lceil\cdot\rceil$-sharing of the truncated value of $z = \vec{x} \odot \vec{y}$, denoted as $z^d$, given the $\lceil\cdot\rceil$-sharing of $n$-sized vectors $\vec{x}$ and $\vec{y}$. This protocol is similar to the 3PC protocol.

Protocol $\Pi_{\text{dotpt}4}(P, \{[[x]]_i, [[y]]_i\}_{i \in [n]})$

Preprocessing:
- $P_0, P_j$, for $j \in \{1, 2\}$, sample random $R_j \in \mathbb{Z}_{2^\ell}$. $P_0, P_3$ sets $r = R_1 + R_2$ while $P_j$ sets $|r|_j = R_j$.
- $P_0, P_3$ locally truncate $r$ to obtain $r^d$ and execute $\Pi_{\text{bitinj}4}(P_0, P_3, r^d)$ to generate $[r^d]$.

Online:
- $P_0, P_j$, for $j \in \{1, 2\}$, compute $[\Psi]_j = -\sum_{i=1}^{n}((\beta_x + y_i)\alpha_{y_i})_j + (\beta_y + y_i)\alpha_{y_i} - |r|_j$ and sets $[[z - r]^*]_j = [[\Psi]]_j + [[x]]_j$.
- $P_1, P_0$ jmp4-send $[[z - r]^*]_1$ to $P_2$ and $P_2, P_0$ jmp4-send $[[z - r]^*]_2$ to $P_1$.
- $P_1, P_2$ locally compute $(z - r)^* = [[z - r]^*]_1 + [[z - r]^*]_2$ and set $(z - r) = (z - r) + \sum_{i=1}^{n}(\beta_x, p_{r_i}) + \psi$.
- $P_1, P_2$ locally truncate $(z - r)$ to obtain $(z - r)^d$ and execute $\Pi_{\text{bitpt}4}(P_1, P_2, (z - r)^d)$ to generate $[[z - r]^d]$.
- Servers locally compute $[z^d] = [[z - r]^d] + [r^d]$.

Figure 26: 4PC: Dot Product Protocol with Truncation

Lemma 4.13 (Communication). $\Pi_{\text{dotpt}4}(P)$ requires an amortized communication of $4\ell$ bits in the preprocessing phase, and $1$ round with amortized communication of $3\ell$ bits in the online phase.

Proof. The preprocessing phase comprises of the preprocessing phase of $\Pi_{\text{dotpt}4}$ and $\Pi_{\text{trgen}4}$ which results in an amortized communication of $3\ell + \ell = 4\ell$ bits. The online phase follows from that of $\Pi_{\text{dotpt}4}$ protocol except that, now, $P_1, P_2$ compute $\lceil\cdot\rceil$-shares of $z - r$. This requires one round and an amortized communication cost of $2\ell$ bits. $P_1, P_2$ then jointly share the truncated value of $z - r$ with $P_0$, which requires $1$ round and $\ell$ bits. However, this step can be deferred till the end for multiple instances, which results in amortizing this round. The total amortized communication is thus $3\ell$ bits in online phase.

Activation Functions Here, as in the 3PC case, we consider two activation functions – ReLU and Sig.

Lemma 4.14 (Communication). Protocol for relu requires an amortized communication of $13\ell - 2$ bits in the preprocessing phase and requires $\log\ell + 1$ rounds and an amortized communication of $10\ell - 6$ bits in the online phase.

Proof. One instance of relu protocol comprises of execution of one instance of $\Pi_{\text{bitext}4}$, followed by $\Pi_{\text{bitinj}4}$. The cost, therefore, follows from Lemma 4.8, and Lemma 4.10.

Lemma 4.15 (Communication). Protocol for sig requires an amortized communication of $23\ell - 1$ bits in the preprocessing phase and requires $\log\ell + 2$ rounds and an amortized communication of $20\ell - 9$ bits in the online phase.

Proof. An instance of sig protocol involves the execution of the following protocols in order– i) two parallel instances of $\Pi_{\text{bitext}4}$ protocol, ii) one instance of $\Pi_{\text{mult}4}$ protocol over boolean value, and iii) one instance of $\Pi_{\text{bitinj}4}$ and $\Pi_{\text{bit2A}4}$ in parallel. The cost follows from Lemma 4.8, Lemma 4.9 and Lemma 4.10.

5 Applications and Benchmarking

In this section, we empirically show the practicality of our protocols for PPML. We consider training and inference for Logistic Regression, and inference for 3 different Neural Networks (NN). NN training requires additional tools to allow mixed word computations, which we leave as future work. We refer readers to SecureML [45], ABY3 [43], BLAZE [48], FALCON [57] for a detailed description of the training and inference steps for the aforementioned ML algorithms. All our benchmarking is done over the publicly available MNIST [39] and CIFAR-10 [37] dataset. For training, we use a batch size of $B = 128$ and define 1 KB = 8192 bits.

In 3PC, we compare our results against the best-known framework BLAZE that provides fairness in the same setting. We observe that the technique of making the dot product cost independent of feature size can also be applied to BLAZE to obtain better costs. Hence, for a fair comparison, we additionally report these improved values for BLAZE. Further, we only consider the PPA circuit based variant of bit extraction for BLAZE since we aim for high throughput; the GC based variant results in huge communication and is not efficient for deep NNs. Our results imply that we get GOD at no additional cost compared to BLAZE. For 4PC, we compare our results with two best-known works FLASH [12] (which is robust) and Trident [16] (which is fair). Our results halve the cost of FLASH and are on par with Trident.
Benchmarks

Benchmarks Environment We use a 64-bit ring \((\mathbb{Z}_{256})\). The benchmarking is performed over a WAN that was instantiated using 8 instances of Google Cloud\(^6\), with machines located in East Australia \((P_0)\), South Asia \((P_1)\), South East Asia \((P_2)\), and West Europe \((P_3)\). The machines are equipped with 2.3 GHz Intel Xeon E5 v3 (Haswell) processors supporting hyper-threading, with 8 vCPUs, and 30 GB of RAM Memory and with a bandwidth of 40 Mbps. The average round-trip time \((\text{rtt})\) was taken as the time for communicating 1 KB of data between a pair of parties, and the rtt values were as follows.

<table>
<thead>
<tr>
<th>(P_0-P_1)</th>
<th>(P_0-P_2)</th>
<th>(P_0-P_3)</th>
<th>(P_1-P_2)</th>
<th>(P_1-P_3)</th>
<th>(P_2-P_3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>151.40ms</td>
<td>59.95ms</td>
<td>275.02ms</td>
<td>92.94ms</td>
<td>173.93ms</td>
<td>219.37ms</td>
</tr>
</tbody>
</table>

Software Details We implement our protocols using the publicly available ENCRYPTO library [22] in C++17. We obtained the code of BLAZE and FLASH from the respective authors and executed them in our environment. The collision-resistant hash function was instantiated using SHA-256. We have used multi-threading, and our machines were capable of handling a total of 32 threads. Each experiment is run for 20 times, and the average values are reported.

Datasets We use the following datasets:

- MNIST [39] is a collection of 28 × 28 pixel, handwritten digit images along with a label between 0 and 9 for each image. It has 60,000 and respectively, 10,000 images in the training and test set. We evaluate logistic regression, and NN-1, NN-2 (cf. §5.2) on this dataset.

- CIFAR-10 [37] consists of 32 × 32 pixel images of 10 different classes such as dogs, horses, etc. There are 50,000 images for training and 10,000 for testing, with 6000 images in each class. We evaluate NN-3 (cf. §5.2) on this dataset.

Benchmarks Parameters We use throughput \((\text{TP})\) as the benchmarking parameter following BLAZE and ABY3 [43] as it would help to analyse the effect of improved communication and round complexity in a single shot. Here, TP denotes the number of operations (“iterations” for the case of training and “queries” for the case of inference) that can be performed in unit time. We consider minute as the unit time since most of our protocols over WAN requires more than a second to complete. An iteration in ML training consists of a forward propagation phase followed by a backward propagation phase. In the former phase, servers compute the output from the inputs. At the same time, in the latter, the model parameters are adjusted according to the difference in the computed output and the actual output. The inference can be viewed as one forward propagation of the algorithm alone. In addition to TP, we provide the online and overall communication and latency for all the benchmarked ML algorithms.

We observe that due to our protocols’ asymmetric nature, the communication load is unevenly distributed among all the servers, which leaves several communication channels underutilized. Thus, to improve the performance, we perform load balancing, where we run several parallel execution threads, each with roles of the servers changed. This helps in utilizing all channels and improving the performance. We report the communication and runtime of the protocols for online phase and total (= preprocessing + online).

5.1 Logistic Regression

In Logistic Regression, one iteration comprises updating the weight vector \(\mathbf{w}\) using the gradient descent algorithm (GD). It is updated according to the function given below:

\[
\mathbf{w} = \mathbf{w} - \frac{\alpha}{B} \left[ \mathbf{X}_i \circ (\mathbf{y} \circ \mathbf{w}) - \mathbf{y}_i \right],
\]

where \(\alpha\) and \(\mathbf{X}_i\) denote the learning rate, and a subset, of batch size \(B\), randomly selected from the entire dataset in the \(i\)th iteration, respectively. The forward propagation comprises of computing the value \(\mathbf{X}_i \circ \mathbf{w}\) followed by an application of a sigmoid function on it. The weight vector is updated in the backward propagation, which internally requires the computation of a series of matrix multiplications, and can be achieved using a dot product. The update function can be computed using \(\mathbf{w} = \mathbf{w} - \frac{\alpha}{B} \left[ \mathbf{X}_i \circ (\mathbf{y} \circ \mathbf{w}) - \mathbf{y}_i \right]\). We summarize our results in Table 6.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Ref.</th>
<th>Online (TP in (10^3))</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Latency (s)</td>
<td>Com [KB]</td>
</tr>
<tr>
<td>3PC Training</td>
<td>BLAZE</td>
<td>0.74</td>
<td>50.26</td>
</tr>
<tr>
<td>3PC Training</td>
<td>SWIFT</td>
<td>1.05</td>
<td>50.32</td>
</tr>
<tr>
<td>3PC Inference</td>
<td>BLAZE</td>
<td>0.66</td>
<td>0.28</td>
</tr>
<tr>
<td>3PC Inference</td>
<td>SWIFT</td>
<td>0.97</td>
<td>0.34</td>
</tr>
<tr>
<td>4PC Training</td>
<td>BLAZE</td>
<td>0.83</td>
<td>88.93</td>
</tr>
<tr>
<td>4PC Training</td>
<td>SWIFT</td>
<td>41.32</td>
<td>11969.48</td>
</tr>
<tr>
<td>4PC Inference</td>
<td>BLAZE</td>
<td>0.75</td>
<td>0.50</td>
</tr>
<tr>
<td>4PC Inference</td>
<td>SWIFT</td>
<td>0.75</td>
<td>0.27</td>
</tr>
</tbody>
</table>

Table 6: Logistic Regression training and inference. TP is given in (hit/min) for training and (#queries/min) for inference.

We observe that the online TP for the case of 3PC inference is slightly lower compared to that of BLAZE. This is because the total number of rounds for the inference phase is slightly higher in our case due to the additional rounds introduced by the verification mechanism (aka verify phase which also needs broadcast). This gap becomes less evident for protocols with more number of rounds, as is demonstrated in the case of NN (presented next), where verification for several iterations is clubbed together, making the overhead for verification insignificant.

For the case of 4PC, our solution outperforms FLASH in terms of communication as well as throughput. Concretely, we observe a \(2\times\) improvement in TP for inference and a \(2.3\times\)

---

\(^6\)https://cloud.google.com/
improvement for training. For Trident [16], we observe a drop of 15.86% in TP for inference due to the extra rounds required for verification to achieve GOD. This loss is, however, traded-off with the stronger security guarantee. For training, we are on par with Trident as the effect of extra rounds becomes less significant for more number of rounds, as will also be evident from the comparisons for NN inference.

As a final remark, note that our 4PC sees roughly 2.5× improvement compared to our 3PC for logistic regression.

5.2 NN Inference

We consider the following popular neural networks for benchmarking. These are chosen based on the different range of model parameters and types of layers used in the network. We refer readers to [57] for a detailed architecture of the neural networks.

NN-1: This is a 3-layered fully connected network with ReLU activation after each layer. This network has around 118K parameters and is chosen from [43, 48].

NN-2: This network, called LeNet [38], contains 2 convolutional layers and 2 fully connected layers with ReLU activation after each layer, additionally followed by maxpool for convolutional layers. This network has approximately 431K parameters.

NN-3: This network, called VGG16 [52], was the runner-up of ILSVRC-2014 competition. This network has approximately 15 million parameters.

Table 7 summarises our benchmarking results for 3PC NN inference. TP is given in (#queries/min).

<table>
<thead>
<tr>
<th>Network</th>
<th>Ref.</th>
<th>Online</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Online</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Latency [s]</td>
<td>Com [MB]</td>
</tr>
<tr>
<td>NN-1</td>
<td>BLAZE</td>
<td>1.92</td>
<td>0.04</td>
</tr>
<tr>
<td></td>
<td>SWIFT</td>
<td>2.22</td>
<td>0.04</td>
</tr>
<tr>
<td>NN-2</td>
<td>BLAZE</td>
<td>4.77</td>
<td>3.54</td>
</tr>
<tr>
<td></td>
<td>SWIFT</td>
<td>5.08</td>
<td>3.54</td>
</tr>
<tr>
<td>NN-3</td>
<td>BLAZE</td>
<td>15.58</td>
<td>52.58</td>
</tr>
<tr>
<td></td>
<td>SWIFT</td>
<td>15.89</td>
<td>52.58</td>
</tr>
</tbody>
</table>

Table 7: 3PC NN Inference. TP is given in (#queries/min).

Table 7 summarises our benchmarking results for 3PC NN inference. As illustrated, the performance of our 3PC framework is on par with BLAZE while providing better security guarantee.

Table 8 summarises NN inference for 4PC setting. Here, we outperform FLASH in every aspect, with the improvement in TP being at least 2× for each NN architecture. Further, we are on par with Trident [16] because the extra rounds required for verification get amortized with an increase in the number of rounds required for computing NN inference. This establishes the practical relevance of our work.

As a final remark, note that our 4PC sees roughly 3× improvement compared to our 3PC for NN inference. This reflects the improvements brought in by the additional honest server in the system.

6 Conclusion

In this work, we presented an efficient framework for PPML that achieves the strongest security of GOD or robustness. Our 3PC protocol builds upon the recent work of BLAZE [48] and achieves almost similar (in some cases, better) performance albeit improving the security guarantee. For the case of 4PC, we outperform the best-known—(a) robust protocol of FLASH [12] by 2× performance-wise and (b) fair protocol of Trident [16] by uplifting its security.

We leave the problem of extending our framework to support mixed-world conversions as well as to design protocols to support algorithms like Decision Trees, k-means Clustering etc. as open problem.

Acknowledgements

We thank our shepherd Guevara Noubir, and anonymous reviewers for their valuable feedback.

Nishat Koti would like to acknowledge financial support from Cisco PhD Fellowship 2020. Mahak Pancholi would like to acknowledge financial support from Cisco MTech Fellowship 2020. Arpita Patra would like to acknowledge financial support from SERB MATRICS (Theoretical Sciences) Grant 2020 and Google India AI/ML Research Award 2020. Ajith Suresh would like to acknowledge financial support from Google PhD Fellowship 2019. The authors would also like to acknowledge the financial support from Google Cloud to perform the benchmarking.

References


R. Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In ACM STOC, pages 364–369, 1986.


A Preliminaries

A.1 Shared Key Setup

Let $F : \{0,1\}^k \times \{0,1\}^k \rightarrow X$ be a secure pseudo-random function (PRF), with co-domain $X$ being $\mathbb{Z}_{2^r}$. The set of keys established between the servers for 3PC is as follows:

- One key shared between every pair— $k_{0i}, k_{02}, k_{12}$ for the servers $(P_0, P_1), (P_0, P_2)$ and $(P_1, P_2)$, respectively.

- One shared key known to all the servers— $k_p$.

Suppose $P_0, P_1$ wish to sample a random value $r \in \mathbb{Z}_{2^r}$ non-interactively. To do so they invoke $F_{\text{setup}}(id_{01})$ and obtain $r$. Here, $id_{01}$ denotes a counter maintained by the servers, and is updated after every PRF invocation. The appropriate keys used to sample is implicit from the context, from the identities of the pair that sample or from the fact that it is sampled by all, and, hence, is omitted.

![Functionality $F_{\text{setup}}$](image)

Figure 27: 3PC: Ideal functionality for shared-key setup

The key setup is modelled via a functionality $F_{\text{setup}}$ (Fig. 27) that can be realised using any secure MPC protocol. Analogously, key setup functionality for 4PC is given in Fig. 28.

![Functionality $F_{\text{setup}}$](image)

Figure 28: 4PC: Ideal functionality for shared-key setup

To generate a 3-out-of-3 additive sharing of 0 i.e. $\zeta_s$ for $s \in \{0,1,2\}$ such that $P_i$ holds $\zeta_s$, and $\zeta_0 + \zeta_1 + \zeta_2 = 0$, servers proceed as follows. Every pair of servers, $P_i, P_{(i+1)\%3}$, non-interactively generate $r_s$, as described earlier, and each $P_i$ sets $\zeta_s = r_s - r_{(i-1)\%3}$.

A.2 Collision Resistant Hash Function

Consider a hash function family $H = \mathcal{K} \times \mathcal{L} \rightarrow \mathcal{Y}$. The hash function $H$ is said to be collision resistant if, for all probabilistic polynomial-time adversaries $A$, given the description of $H_0$ where $k \in \mathcal{K}$, there exists a negligible function $\text{negl}(\cdot)$ such that $\Pr[(x_1, x_2) \leftarrow A(k) : (x_1 \neq x_2) \land H_0(x_1) = H_0(x_2)] \leq \text{negl}(\cdot)$, where $m = \text{poly}(\cdot)$ and $x_1, x_2 \in \{0,1\}^m$.

A.3 Commitment Scheme

Let $\text{Com}(x)$ denote the commitment of a value $x$. The commitment scheme $\text{Com}(x)$ possesses two properties: hiding and binding. The former ensures privacy of the value $x$ given just its commitment $\text{Com}(x)$, while the latter prevents a corrupt server from opening the commitment to a different value $x' \neq x$. The practical realization of a commitment scheme is via a hash function $H()$ given below, whose security can be proved in the random-oracle model (ROM) for $(c, o) = (H(x||r), x||r) = \text{Com}(x;r)$.

B Instantiating $F_{\text{DotPPre}}$

As mentioned earlier, a trivial way to instantiate $\Pi_{\text{dotPPre}}$ is to treat a dot product operation as $n$ multiplications. However, this results in a communication cost that is linearly dependent on the feature size. Instead, we instantiate $\Pi_{\text{dotPPre}}$ by a semi-honest dot product protocol followed by a verification phase to check the correctness. For the verification phase, we extend the techniques of [9, 10] to provide support for verification of dot product tuples. Setting the verification phase parameters appropriately gives a $\Pi_{\text{dotPPre}}$ whose (amortized) communication cost is independent of the feature size. We provide the details next.

To realize $F_{\text{DotPPre}}$, the approach is to run a semi-honest dot product protocol followed by a verification phase to check the correctness of the output. For verification, the work of [9] provides techniques to verify the correctness of $m$ multiplication triples (and degree-two relations) at a cost of $O(\sqrt{m})$ extended ring elements, albeit with abort security. While [10] improves their techniques to provide robust verification for multiplication, we show how to extend the techniques in [10] to robustly verify the correctness of $m$ dot product tuples (dot product being a degree two relation), with vectors of dimension $n$, at a cost of $O(\sqrt{nm})$ extended ring elements. Thus, the cost to realize one instance of $F_{\text{DotPPre}}$ can be brought down to only the cost of a semi-honest dot product computation (which is 3 ring elements and independent of the vector dimension), where the cost due to verification can be amortized away by setting $n, m$ appropriately.

Given vectors $\vec{d} = (d_1, \ldots, d_i), \vec{e} = (e_1, \ldots, e_n)$, let server $P_i$, for $i \in \{0,1,2\}$, hold $d_{(j)i} = (d_{j,i}, d_{j,(i+1)\%3})$ and $e_{(j)i} = (e_{j,i}, e_{j,(i+1)\%3})$ where $j \in [n]$ (henceforth, we omit the use of $\%3$ in the subscript as it is understood from the context). The semi-honest dot product protocol proceeds as follows. The servers, using the shared key setup, non-interactively generate 3-out-of-3 additive shares of zero (as described in A.1), i.e.
$P_i$ has $\zeta$, such that $\zeta_0 + \zeta_1 + \zeta_2 = 0$. Then, each $P_i$ locally computes 3-out-of-3 additive share of $f = d \odot e$ as:

$$f_i = \zeta_i + \sum_{j=1}^{n} (d_{j,i} \cdot e_{j,i} + d_{j,i+1} \cdot e_{j,i} + d_{j,i+1} \cdot e_{j,i+1})$$  \hspace{1cm} (7)

Now, to complete the $\langle \cdot \rangle$-sharing of $f$, $P_i$ sends $f_i$ to $P_{i+1}$. To check the correctness of the computation ($f = \langle d \odot e \rangle$), each $P_i \in \mathcal{P}$ needs to prove that the $f_i$ it sent in the semi-honest protocol satisfies 7, i.e.

$$\zeta_i + \sum_{j=1}^{n} (d_{j,i} \cdot e_{j,i} + d_{j,i+1} \cdot e_{j,i+1} + d_{j,i+1} \cdot e_{j,i+1}) - f_i = 0$$  \hspace{1cm} (8)

This difference in the expected message that should be sent (computed using $P_i$'s correct input shares) and actual message that is sent by $P_i$ is captured by a circuit $c$, defined below.

$$c(\{d_{j,i}, d_{j,i+1}, e_{j,i}, e_{j,i+1}\}_{j=1}^{n}, \zeta_i, f_i) = \zeta_i + \sum_{j=1}^{n} (d_{j,i} \cdot e_{j,i} + d_{j,i+1} \cdot e_{j,i+1} + d_{j,i+1} \cdot e_{j,i+1}) - f_i$$  \hspace{1cm} (9)

Here, $c$ takes as input $u = 4n + 2$ values: $\langle \cdot \rangle$-shares of $d, e$ held by $P_i$, i.e. $\{d_{j,i}, d_{j,i+1}, e_{j,i}, e_{j,i+1}\}_{j=1}^{n}$, the additive share of zero, $\zeta$, that $P_i$ holds, and the additive share $f_i$ sent by $P_i$. For correct computation with respect to $P_i$, we require the difference in the expected message and the actual message to be 0, i.e.,

$$c(\{d_{j,i}, d_{j,i+1}, e_{j,i}, e_{j,i+1}\}_{j=1}^{n}, \zeta_i, f_i) = 0$$  \hspace{1cm} (10)

We now explain how to verify the correctness for $m$ dot product tuples assuming that the operations are carried out over a prime-order field. The verification can be extended to support operations over rings following the techniques of [9, 10]. To verify the correctness for $m$ dot product tuples, $\{d_k, e_k, f_k\}_{k=1}^{m}$ where $f_k = d_k \odot e_k$, the output of $c$ (which is the difference in the expected and actual message sent) for each of the corresponding dot product tuple must be 0. To check correctness of all dot products at once, it suffices to check if a random linear combination of the output of each $c$ (for each dot product) is 0. This is because the random linear combination of the differences will be 0 with high probability if $f_k = d_k \odot e_k$ for each $k \in \{1, \ldots, m\}$. We remark that the definition of $c(\cdot)$ in [10] enables the verification of only multiplication triples. With the re-definition of $c$ as in 9, we can now verify the correctness of dot products while the rest of the verification steps remain similar to that in [10]. We elaborate on the details, next.

A verification circuit, constructed as follows, enables $P_i$ to prove the correctness of the additive share of $f$ that it sent, for $m$ instances of dot product at once. Note that the proof system is designed for the distributed-verifier setting where the proof generated by $P_i$ will be shared among $P_{i-1}, P_{i+1}$, who can together verify its correctness. First, a sub-circuit $g$ is defined as: group $L$ small $c$ circuits and take a random linear combination of the values on their output wires. Since each $c$ circuit takes $u = 4n + 2$ inputs as described earlier, $g$ takes in $uL$ inputs. Precisely, $g$ is defined as follows:

$$g(x_1, \ldots, x_{uL}) = \sum_{k=1}^{L} \theta_k \cdot c(x_{(k-1)u+1}, \ldots, x_{(k-1)u+u})$$

Since there are total $m$ dot products to be verified, there will be $M = m/L$ sub-circuits $g$. Looking ahead, this grouping technique enables obtaining a sub-linear communication cost for verification because the communication cost turns out to be $O(uL + M)$ and setting $uL = M$ gives the desired result. The sub-circuits $g$ make up the circuit $G$ which outputs a random linear combination of the values on the output wires of each $g$, i.e.,

$$G(x_1, \ldots, x_{um}) = \sum_{k=1}^{M} \eta_k \cdot g(x_{(k-1)uL+1}, \ldots, x_{(k-1)uL+uL})$$

Here, $\theta_k$ and $\eta_k$ are randomly sampled (non-interactively) by all parties. To prove correctness, $P_i$ needs to prove that $G$ outputs 0. For this, $P_i$ defines $f_1, \ldots, f_{uL}$ random polynomials of degree $M$, one for each input wire of $g$. For $\ell \in \{1, \ldots, M\}$ and $j \in \{1, \ldots, uL\}$, $f_{j}(0)$ is chosen randomly and $f_{j}(\ell) = x_{(j-1)u+\ell}$ (i.e. the $j$th input of the $\ell$th gate). $P_i$ further defines a $2M$ degree polynomial $p(\cdot)$ on the output wires of $g$, i.e $p(\cdot) = g(f_1, \ldots, f_{uL})$ where $p(\ell)$ for $\ell \in \{1, \ldots, M\}$ is the output of the $\ell$th gate. The additional $M + 1$ points required to interpolate the $2M$ degree polynomial $p$, are obtained by evaluating $f_1, \ldots, f_{uL}$ on $M + 1$ additional points, followed by an application of $g$ circuit. The proof generated by $P_i$ consists of $f_1(0), \ldots, f_{uL}(0)$ and the coefficients of $p$. Recall that since we are in the distributed-verifier setting, the prover $P_i$ additionally shares the proof with $P_{i-1}, P_{i+1}$. Note here, that shares of $f_1(0), \ldots, f_{uL}(0)$ can be generated non-interactively.

To verify the proof, verifiers $P_{i-1}, P_{i+1}$ need to check if the output of $G$ is 0. This can be verified by computing the output of $G$ as $b = \sum_{k=1}^{M} \eta_k \cdot p(\ell)$ and checking if $b = 0$, where $\eta_k$’s are non-interactively sampled by all after the proof is sent. If $p$ is defined correctly, then this is indeed a random linear combination of the outputs of all the $g$-circuits. This necessitates the second check to verify the correctness of $p$ as per its definition i.e $p(\cdot) = g(f_1(\cdot), \ldots, f_{uL}(\cdot))$. This is performed by checking if $p(r) = p(f_1(r), \ldots, f_{uL}(r))$ for a random $r \notin \{1, \ldots, M\}$ (for privacy to hold) sampled non-interactively by all after the proof is sent. For the first check, verifiers can locally compute additive shares of $b$ (using the additive shares of coefficients of $p$ obtained as part of the proof) and reconstruct $b$ to check for equality with 0. For the second, verifiers locally compute additive shares of $p(r)$ using the shares of coefficients of $p$, and shares of $f_1(r), \ldots, f_{uL}(r)$ by interpolating $f_1, \ldots, f_{uL}$ using $(P_i)$’s inputs to the $c$-circuits which are implicitly additively shared between them (owing to the replicated sharing property). Verifiers exchange these.
values among themselves, reconstruct it and check if \( p(r) = g(f_1(r), \ldots, f_M(r)) \). Note that, the messages computed and exchanged by the verifiers, depend only on the proof sent by \( P_i \) and the random values \((r, \eta)\) sampled by all. These messages can also be independently computed by \( P_i \). Thus, in order to prevent a verifier from falsely rejecting a correct proof, we use jmp to exchange these messages. To optimize the communication cost further, it suffices if a single verifier computes the output of verification.

**Setting the parameters:** The proof sent by \( P_i \) consists of the constant terms \( f_j(0) \), for \( j \in \{1, \ldots, uL\} \) and \( 2M + 1 \) coefficients of \( p \). The former can be generated non-iteratively. Hence, \( P_i \) needs to communicate \( 2M + 1 \) elements to the verifiers (one of which can be performed non-iteratively). The message sent by the verifier consists of the additive share of \( \sum_{\ell=1}^{M+1} \eta_{\ell} \cdot p(\ell) \) (for the first check) and \( f_1(r), \ldots, f_u(r), p(r) \) (for the second check). Thus, the verifier communicates \( uL + 2 \) elements. As the proof is executed three times, each time with one party acting as the prover and the other two acting as the verifier, overall, each party communicates \( uL + 2M + 3 \) elements. Setting \( uL = 2M \) and \( M = n/d \) results in the total communication required for verifying \( m \) dot products to be \( O(\sqrt{n}\log n) \). Thus, verifying a single dot product has an amortized cost of \( O \left( \frac{\sqrt{2}}{\sqrt{d}} \right) \) which can be made very small by appropriately setting the values of \( n, m \). Thus, the (amortized) cost of a maliciously secure dot product protocol can be made equal to that of a semi-honest dot product protocol, which is 3 ring elements.

To support verification over rings [10], verification operations are carried out on the extended ring \( \mathbb{Z}_{2^d}/f(x) \), which is the ring of all polynomials with coefficients in \( \mathbb{Z}_{2^d} \) modulo a polynomial \( f \), of degree \( d \), irreducible over \( \mathbb{Z}_{2^d} \). Each element in \( \mathbb{Z}_{2^d} \) is lifted to a \( d \)-degree polynomial in \( \mathbb{Z}_{2^d}[x]/f(x) \) (which results in blowing up the communication by a factor \( d \)). Thus, the per party communication amounts to \((uL + 2M + 3)d\) elements of \( \mathbb{Z}_{2^d} \) for verifying \( m \) dot products of vector size \( n \) where \( u = 4n + 2 \). Further, the probability of a cheating prover is bounded by \( \frac{n^{(d-1)d}}{2^d - M} \) (cf. Theorem 4.7 of [10]). Thus, if \( \gamma \) is such that \( 2\gamma \geq \sqrt{2} \), then the cheating probability is

\[
2^{(\ell-1)d} \cdot 2M + 1 \leq 2^{(\ell-1)d} \cdot 2\gamma + 1 \approx 2^{-(d-\gamma)}
\]

We note that both, [10] and our technique require a communication cost of \( O(\sqrt{mn}) \) ring elements for verifying \( m \) dot products of vector size \( n \). This is because multiplication is a special case of dot product with \( n = 1 \). However, since our verification is for dot products, we can get away with performing only \( m \) semi-honest dot products whose cost is equivalent to computing \( m \) semi-honest multiplications, whereas [10] requires to execute \( mn \) multiplications (as their technique can only verify correctness of multiplications), resulting in a dot product cost dependent on the vector size. Concretely, to get 40 bits of statistical security and for a vector size of \( 2^{10} \) (CIFAR-10 [37] dataset), the aforementioned parameters can be set as given in Table 9.

<table>
<thead>
<tr>
<th>( m )</th>
<th>( M )</th>
<th>( \gamma )</th>
<th>( d )</th>
<th>Cost (per dot product)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 2^{20} )</td>
<td>( 2^{16} )</td>
<td>17</td>
<td>57</td>
<td>7.125</td>
</tr>
<tr>
<td>( 2^{30} )</td>
<td>( 2^{21} )</td>
<td>22</td>
<td>62</td>
<td>0.242</td>
</tr>
<tr>
<td>( 2^{40} )</td>
<td>( 2^{26} )</td>
<td>27</td>
<td>67</td>
<td>0.008</td>
</tr>
<tr>
<td>( 2^{50} )</td>
<td>( 2^{31} )</td>
<td>32</td>
<td>72</td>
<td>0.0002</td>
</tr>
</tbody>
</table>

Table 9: Cost of verification in terms of the number of ring elements communicated per dot product, and parameters for vector size \( n = 2^{10} \) and 40 bits of statistical security. Here, \( m \) - #dot products to be verified, \( M \) - #g sub-circuits, \( d \)-degree of extension.

It is possible to further bring down the communication cost required for verifying \( m \) dot product tuples to \( O(\log(nm)) \) at the expense of requiring more rounds by further extending the technique of [9], which we leave as an exercise. We refer readers to [10] for formal details.

**C Security Analysis of Our Protocols**

In this section, we provide detailed security proofs for our constructions in both the 3PC and 4PC domains. We prove security using the real-world/ ideal-word simulation based technique. We provide proofs in the \( \mathcal{F}_{\text{setup}} \)-hybrid model for the case of 3PC, where \( \mathcal{F}_{\text{setup}} \) (Fig. 27) denotes the ideal functionality for the three server shared-key setup. Similarly, 4PC proofs are provided in \( \mathcal{F}_{\text{setup4}} \)-hybrid model (Fig. 28).

Let \( \mathcal{A} \) denote the real-world adversary corrupting at most one server in \( \mathcal{P} \), and \( \mathcal{S} \) denote the corresponding ideal world adversary. The strategy for simulating the computation of function \( f \) (represented by a circuit \( \text{ckt} \)) is as follows: The simulation begins with the simulator emulating the shared-key setup (\( \mathcal{F}_{\text{setup}}/\mathcal{F}_{\text{setup4}} \)) functionality and giving the respective keys to the adversary. This is followed by the input sharing phase in which \( \mathcal{S} \) extracts the input of \( \mathcal{A} \), using the known keys, and sets the inputs of the honest parties to be 0. \( \mathcal{S} \) now knows all the inputs and can compute all the intermediate values for each of the building blocks in clear. Also, \( \mathcal{S} \) can obtain the output of the \( \text{ckt} \) in clear. \( \mathcal{S} \) now proceeds simulating each of the building block in topological order using the aforementioned values (inputs of \( \mathcal{A} \), intermediate values and circuit output).

In some of our sub protocols, adversary is able to decide on which among the honest parties should be chosen as the Trusted Third Party (TTP) in that execution of the protocol. To capture this, we consider corruption-aware functionalities [5] for the sub-protocols, where the functionality is provided the identity of the corrupt server as an auxiliary information.

For modularity, we provide the simulation steps for each of the sub-protocols separately. These steps, when carried
out in the respective order, result in the simulation steps for the entire 3/4PC protocol. If a TTP is identified during the simulation of any of the sub-protocols, simulator will stop the simulation at that step. In the next round, the simulator receives the input of the corrupt party in clear on behalf of the TTP for the 3PC case, whereas it receives the input shares from adversary for 4PC.

C.1 Security Proofs for 3PC protocols

The ideal functionality \(F_{3PC}\) for 3PC appears in Fig. 29.

**Functionality \(F_{3PC}\)**

\(F_{3PC}\) interacts with the servers in \(P\) and the adversary \(S\). Let \(f\) denote the functionality to be computed. Let \(x_i\) be the input corresponding to the server \(P_i\), and \(y_i\) be the corresponding output, i.e. \((y_0, y_1, y_2) = f(x_0, x_1, x_2)\).

**Step 1:** \(F_{3PC}\) receives \((Input, x_i)\) from \(P_i \in P\), and computes \((y_0, y_1, y_2) = f(x_0, x_1, x_2)\).

**Step 2:** \(F_{3PC}\) sends \((Output, y_i)\) to \(P_i \in P\).

![Figure 29: 3PC: Ideal functionality for evaluating a function \(f\)](image)

C.1.1 Joint Message Passing (jmp) Protocol

This section provides the security proof for the jmp primitive, which forms the crux for achieving GOD in our constructions. Let \(F_{jmp}\) (Fig. 1) denote the ideal functionality and let \(S_{jmp}\) denote the corresponding simulator for the case of corrupt \(P_i \in P\). We begin with the case for a corrupt sender, \(P_i\). The case for a corrupt \(P_j\) is similar and hence we omit details for the same.

**Simulator \(S_{jmp}^{P_i}\)**

- \(S_{jmp}^{P_i}\) initializes \(ttp = 1\) and receives \(v_i\) from \(A\) on behalf of \(P_k\).
- In case, \(A\) fails to send a value \(S_{jmp}^{P_i}\) broadcasts \(\text{"(accuse, } P_j\text{)"}, \) sets \(ttp = P_j, v_i = 1\), and skip to the last step.
- Else, it checks if \(v_i = v\), where \(v\) is the value computed by \(S_{jmp}^{P_i}\) based on the interaction with \(P\), and using the knowledge of the shared keys. If the values are equal, \(S_{jmp}^{P_i}\) sets \(b_k = 0\), else, sets \(b_k = 1\), and sends the same to \(A\) on behalf of \(P_k\).
- If \(A\) broadcasts \(\text{"(accuse, } P_k\text{)"}, \) \(S_{jmp}^{P_i}\) sets \(v_i = 1, ttp = P_j, \) and skips to the last step.
- \(S_{jmp}^{P_i}\) computes and sends \(b_j\) to \(A\) on behalf of \(P_j\) and receives \(b_{\beta}\) from \(A\) on behalf of honest \(P_j\).
- If \(S_{jmp}^{P_i}\) does not receive a \(b_{\beta}\) on behalf of \(P_j\), it broadcasts \(\text{"(accuse, } P_i\text{)"}, \) sets \(v_i = 1, ttp = P_k\). If \(A\) broadcasts \(\text{"(accuse, } P_i\text{)"}, \) \(S_{jmp}^{P_i}\) sets \(v_i = 1, ttp = P_k\). If ttp is set, skip to the last step.
- If \((v_i = v)\) and \(b_{\beta} = 1, S_{jmp}^{P_i}\) broadcasts \(H_j = H(v)\) on behalf of \(P_j\).
- Else if \(v_i \neq v\): \(S_{jmp}^{P_i}\) broadcasts \(H_j = H(v)\) and \(H_k = H(v_i)\) on behalf of \(P_j\) and \(P_k\), respectively. If \(A\) does not broadcast, \(S_{jmp}^{P_i}\) sets \(ttp = P_k\). Else if, \(A\) broadcasts a value \(H_{A}\):
  - If \(H_{A} \neq H_j: S_{jmp}^{P_i}\) sets \(ttp = P_k\).
  - Else if \(H_{A} \neq H_k: S_{jmp}^{P_i}\) sets \(ttp = P_j\).
  - \(S_{jmp}^{P_i}\) invokes \(F_{jmp}\) on \((Input, v_i)\) and \((Select, ttp)\) on behalf of \(A\).

![Figure 30: Simulator \(S_{jmp}^{P_i}\) for corrupt sender \(P_i\)](image)

The case for a corrupt receiver, \(P_k\) is provided in Fig. 31.

**Simulator \(S_{jmp}^{P_k}\)**

- \(S_{jmp}^{P_k}\) initializes \(ttp = 1\), computes \(v\) honestly and sends \(v\) and \(H(v)\) to \(A\) on behalf of \(P_i\) and \(P_j\), respectively.
- If \(A\) broadcasts \(\text{"(accuse, } P_j\text{)"}, \) \(ttp = P_j\), else if \(A\) broadcasts \(\text{"(accuse, } P_i\text{)"}, \) \(ttp = P_i\). If both messages are broadcast, \(ttp = P_i\). If ttp is set skip to the last step.
- On behalf of \(P_i, P_j, S_{jmp}^{P_k}\) receives \(b_j\) from \(A\). Let \(b_j\) (resp. \(b_j\)) denote the bit received by \(P_i\) (resp. \(P_j\)) from \(A\).
- If \(A\) failed to send bit \(b_{\beta}\) to \(P, S_{jmp}^{P_k}\) broadcasts \(\text{"(accuse, } P_j\text{)"}, \) \(ttp = P_j\). Similarly, for \(P_j\). If both \(P_i, P_j\) broadcast \(\text{"(accuse, } P_i\text{)"}, \) \(ttp = P_i\). If ttp is set, skip to the last step.
- If \(b_j \vee b_j = 1: S_{jmp}^{P_k}\) broadcasts \(H_i, H_j\) where \(H_i = H_j = H(v)\) on behalf of \(P_i, P_j\), respectively.
- If \(A\) does not broadcast \(S_{jmp}^{P_k}\) sets \(ttp = 1\). If \(A\) broadcasts a value \(H_{A}\):
  - If \(H_{A} \neq H_i: S_{jmp}^{P_k}\) sets \(ttp = P_j\).
  - Else if \(H_{A} = H_i = H_j: S_{jmp}^{P_k}\) sets \(ttp = P_i\).
  - \(S_{jmp}^{P_k}\) invokes \(F_{jmp}\) on \((Input, ⊥)\) and \((Select, ttp)\) on behalf of \(A\).

![Figure 31: Simulator \(S_{jmp}^{P_k}\) for corrupt receiver \(P_k\)](image)

C.1.2 Sharing Protocol

The case for a corrupt \(P_0\) is provided in Fig. 37.

**Simulator \(S_{sh}^{P_0}\)**

Preprocessing: \(S_{sh}^{P_0}\) emulates \(F_{setup}\) and gives the keys \((k_{01}, k_{02}, k_{03})\) to \(A\). The values that are commonly held along with \(A\) are sampled using appropriate shared key. Otherwise, values are sampled randomly.

Online:

- If the dealer \(P_1 = P_0\):
  - \(S_{sh}^{P_0}\) receives \(β\) from \(P_1\) and sets \(msg = ν\) accordingly.
  - Steps for \(Π_{jmp}\) protocol are simulated according to \(S_{jmp}^{P_i}\) (Fig. 30), where \(P_0\) plays the role of one of the senders.
- If the dealer \(P_1 = P_0\):
  - \(S_{sh}^{P_0}\) sets \(ν = 0\) by assigning \(β = α\).
  - Steps for \(Π_{jmp}\) protocol are simulated similar to \(S_{jmp}^{P_i}\) (Fig. 31), with \(P_0\) acting as the receiver.
The case for a corrupt P₁ is provided in Fig. 33. The case for a corrupt P₂ is similar.

Figure 33: Simulator \( S_{sh}^{P_i} \) for corrupt P₁

### C.1.3 Multiplication Protocol

The case for a corrupt P₀ is provided in Fig. 34.

Figure 34: Simulator \( S_{mult}^{P_0} \) for corrupt P₀

### C.1.4 Reconstruction Protocol

The case for a corrupt P₀ is provided in Fig. 35. The case for a corrupt P₁, P₂ is similar.

Figure 35: Simulator \( S_{rec}^{P_i} \) for corrupt Pᵢ

### C.1.5 Joint Sharing Protocol

The case for a corrupt P₀ is provided in Fig. 37. The case for a corrupt P₁, P₂ is similar.

Figure 37: Simulator \( S_{sh}^{P_i} \) for corrupt Pᵢ

### C.1.6 Dot Product Protocol

The case for a corrupt P₀ is provided in Fig. 38.
The case for a corrupt receiver, \( P_i \) is provided in Fig. 39. The case for a corrupt \( P_j \) is similar.

Figure 39: Simulator \( S_{\text{dotp}}^{P_j} \) for corrupt \( P_j \)

C.1.7 Truncation Protocol

The case for a corrupt \( P_0 \) is provided in Fig. 40.

Figure 40: Simulator \( S_{\text{trgen}}^{P_0} \) for corrupt \( P_0 \)

The case for a corrupt \( P_1 \) is provided in Fig. 41. The case for a corrupt \( P_2 \) is similar.

Figure 41: Simulator \( S_{\text{trgen}}^{P_1} \) for corrupt \( P_1 \)

C.2 Security Proofs for 4PC protocols

The ideal functionality \( \mathcal{F}_{\text{4PC}} \) for 4PC protocols is shown in Fig. 42.

Figure 42: 4PC: Ideal functionality for computing \( f \) in 4PC setting

C.2.1 Joint Message Passing

Let \( \mathcal{F}_{\text{jmp4}} \) Fig. 15 denote the ideal functionality and let \( S_{\text{jmp4}}^{P_i} \) denote the corresponding simulator for the case of corrupt \( P_i \) in \( \mathcal{P} \).

We begin with the case for a corrupt sender, \( P_i \), which is provided in Fig. 43. The case for a corrupt \( P_j \) is similar and hence we omit details for the same.

Figure 43: Simulator \( S_{\text{jmp4}}^{P_j} \) for corrupt sender \( P_j \)

The case for a corrupt receiver, \( P_k \) is provided in Fig. 44.

Figure 44: Simulator \( S_{\text{jmp4}}^{P_k} \) for corrupt receiver \( P_k \)
\[ S^\beta_{\text{jmp4}} \] receives \( b_{i,j}, b_{j,i} \) from \( \mathcal{A} \) on behalf of \( P_i, P_j, P_k \), respectively. If \( \mathcal{A} \) fails to send a value, it is assumed to be 1.

\[- S^\beta_{\text{jmp4}} \] sets \( b_j \) to be majority value in \( b_{i,j}, b_{j,i} \). If \( b_j = 1 \), \( S^\beta_{\text{jmp4}} \) sets \( t \) to 1, else it sets \( t \) to 0. \( S^\beta_{\text{jmp4}} \) invokes \( F_{\text{ jmp4}} \) with \((\text{Input}, \bot)\) and \((\text{Select}, b_k)\) on behalf of \( \mathcal{A} \).

**Figure 44:** Simulator \( S^\beta_{\text{ jmp4}} \) for corrupt receiver \( P_k \)

The case for a corrupt receiver, \( P_k \), which is the server outside the computation involved in \( \Pi_{\text{ jmp4}} \), is provided in Fig. 45.

**Figure 45:** Simulator \( S^\beta_{\text{ jmp4}} \) for corrupt fourth server \( P_j \)

### C.2.2 Sharing Protocol

The case for corrupt \( P_0 \) is given in Fig. 46.

**Figure 46:** Simulator \( S^\beta_{\text{ jmp4}} \) for corrupt \( P_0 \)

The case for corrupt \( P_1 \) is given in Fig. 47. The case for a corrupt \( P_2 \) is similar.

**Figure 47:** Simulator \( S^\beta_{\text{ jmp4}} \) for corrupt \( P_1 \)

### C.2.3 Multiplication Protocol

The case for corrupt \( P_0 \) is given in Fig. 48.

**Figure 48:** Simulator \( S^\beta_{\text{ jmp4}} \) for corrupt \( P_3 \)

### Preprocessing:

- \( S^\beta_{\text{ jmp4}} \) emulates \( F_{\text{setup4}} \) and gives the keys \((k_{01}, k_{02}, k_{12}, k_{13}, k_{012}, k_{013}, k_{123}, k_p)\) to \( \mathcal{A} \). The values that are commonly held with \( \mathcal{A} \) are sampled using the respective keys, while others are sampled randomly.

**Online:**

- If dealer is \( P_0 \), \( S^\beta_{\Pi_{\text{mult}}} \) receives \( \beta \) from \( \mathcal{A} \) on behalf of \( P_0 \). Steps corresponding to \( \Pi_{\text{ jmp4}} \) are simulated according to \( S^\beta_{\Pi_{\text{ jmp4}}} \) where \( P_0 \) acts as one of the sender for sending \( \beta \).

- If dealer is \( P_1 \) or \( P_2 \), \( S^\beta_{\Pi_{\text{mult}}} \) sets \( v = 0 \) by assigning \( \beta = \alpha \). Steps corresponding to \( \Pi_{\text{ jmp4}} \) for sending \( \beta + \gamma \) to \( \mathcal{A} \) are simulated according to \( S^\beta_{\Pi_{\text{ jmp4}}} \) where \( P_0 \) acts as the receiver.

- If dealer is \( P_3 \), \( S^\beta_{\Pi_{\text{mult}}} \) sets \( v = 0 \) by assigning \( \beta = \alpha \). \( S^\beta_{\Pi_{\text{mult}}} \) sends \( \beta + \gamma \) to \( \mathcal{A} \) on behalf of \( P_3 \). Steps corresponding to \( \Pi_{\text{ jmp4}} \) are simulated according to \( S^\beta_{\Pi_{\text{ jmp4}}} \) where \( P_0 \) acts as one of the sender with \( P_1, P_2 \) as the receivers, separately.

- If dealer is \( P_4 \) or \( P_5 \), steps corresponding to \( \Pi_{\text{ jmp4}} \) are simulated according to \( S^\beta_{\Pi_{\text{ jmp4}}} \) where \( P_0 \) acts as the server outside the computation.

**Figure 49:** Simulator \( S^\beta_{\Pi_{\text{mult}}} \) for corrupt \( P_0 \)

### Preprocessing:

- \( S^\beta_{\Pi_{\text{mult}}} \) samples \( [\alpha_1, \alpha_2] \) using the respective keys with \( \mathcal{A} \). \( S^\beta_{\Pi_{\text{mult}}} \) samples \( [\beta_1, \beta_2, \beta_3] \) randomly on behalf of the respective honest parties, and computes \( [\gamma_1, \gamma_2] \) honestly.

- Steps corresponding to \( \Pi_{\text{ jmp4}} \) are simulated according to \( S^\beta_{\Pi_{\text{ jmp4}}} \) where \( P_0 \) acts as one of the sender for sending \( [\gamma_1, \gamma_2] \).

- \( S^\beta_{\Pi_{\text{mult}}} \) computes \( [\chi_1, \chi_2] \) honestly. Steps corresponding to \( \Pi_{\text{ jmp4}} \) are simulated according to \( S^\beta_{\Pi_{\text{ jmp4}}} \) where \( P_0 \) acts as the receiver for \( [\chi_1, \chi_2] \).
– $\mathcal{S}_{\Pi_{\text{mult}}}$ computes $[\beta_1^r]$, $[\beta_2^r]$ honestly. Steps corresponding to $\Pi_{\text{jmp4}}$ are simulated according to $S_{\Pi_{\text{jsh4}}}$ where $P_0$ acts as one of the sender for sending $[\beta_1^r]$, $[\beta_2^r]$.

– $\mathcal{S}_{\Pi_{\text{mult}}}$ computes $\beta_2 + \gamma_2$. Steps corresponding to $\Pi_{\text{jmp4}}$ are simulated according to $S_{\Pi_{\text{jsh4}}}$ where $P_0$ acts as the receiver for receiving $\beta_2 + \gamma_2$.

Figure 49: Simulator $S_{\Pi_{\text{mult}}}$ for corrupt $P_0$

The case for corrupt $P_1$ is given in Fig. 50. The case for a corrupt $P_2$ is similar.

**Simulator $S_{\Pi_{\text{mult}}}$**

**Preprocessing:**

– $S_{\Pi_{\text{mult}}}$ samples $[\alpha_2^r]$, $[\gamma_2^r]$, $[\beta_2^r]$, $[\gamma_1^r]$, $[\beta_1^r]$ using the respective keys with $\mathcal{A}$. $S_{\Pi_{\text{mult}}}$ samples $[\alpha_1^r]$ randomly on behalf of the respective honest parties.

– Steps corresponding to $\Pi_{\text{jmp4}}$ are simulated according to $S_{\Pi_{\text{jsh4}}}$ where $P_1$ acts as the server outside the computation while communicating $[\Gamma_{xy}]$.

– $\mathcal{S}_{\Pi_{\text{mult}}}$ computes $[\chi_1]$. Steps corresponding to $\Pi_{\text{jmp4}}$ are simulated according to $S_{\Pi_{\text{jsh4}}}$ where $P_1$ acts as one of the sender for $[\chi_1]$.

– Steps corresponding to $\Pi_{\text{jmp4}}$ are simulated according to $S_{\Pi_{\text{jsh4}}}$ where $P_1$ acts as the server outside the computation while communicating $[\chi_2]$.

**Online:**

– $\mathcal{S}_{\Pi_{\text{mult}}}$ computes $[\beta_1^r]$, $[\beta_2^r]$. Steps corresponding to $\Pi_{\text{jmp4}}$ are simulated according to $S_{\Pi_{\text{jsh4}}}$ where $P_1$ acts as one of the sender for sending $[\beta_1^r]$, and $P_1$ acts as the receiver for receiving $[\beta_2^r]$, respectively.

– $\mathcal{S}_{\Pi_{\text{mult}}}$ computes $\beta_2 + \gamma_2$. Steps corresponding to $\Pi_{\text{jmp4}}$ are simulated according to $S_{\Pi_{\text{jsh4}}}$ where $P_1$ acts as one of the sender for sending $\beta_2 + \gamma_2$.

Figure 50: Simulator $S_{\Pi_{\text{mult}}}$ for corrupt $P_1$

The case for corrupt $P_3$ is given in Fig. 51.

**Simulator $S_{\Pi_{\text{mult}}}$**

**Preprocessing:**

– $S_{\Pi_{\text{mult}}}$ samples $[\alpha_2^r]$, $[\alpha_1^r]$, $[\gamma_2^r]$, $[\beta_2^r]$, $[\gamma_1^r]$, $[\beta_1^r]$ using the respective keys with $\mathcal{A}$. $S_{\Pi_{\text{mult}}}$ computes $[\Gamma_{xy}]$ honestly.

– Steps corresponding to $\Pi_{\text{jmp4}}$ are simulated according to $S_{\Pi_{\text{jsh4}}}$ where $P_3$ acts as one of the sender for sending $[\Gamma_{xy}]$.

– $\mathcal{S}_{\Pi_{\text{mult}}}$ computes $[\chi_1]$, $[\chi_2]$. Steps corresponding to $\Pi_{\text{jmp4}}$ are simulated according to $S_{\Pi_{\text{jsh4}}}$ where $P_3$ acts as one of the sender for sending $[\chi_1]$ and $[\chi_2]$.

**Online:**

– Steps corresponding to $\Pi_{\text{jmp4}}$ are simulated according to $S_{\Pi_{\text{jsh4}}}$ where $P_3$ acts as the server outside the computation involving $[\beta_1^r]$, $[\beta_2^r]$ and $\beta_2 + \gamma_2$.

Figure 51: Simulator $S_{\Pi_{\text{mult}}}$ for corrupt $P_3$

### C.2.4 Reconstruction Protocol

The case for corrupt $P_0$ is given in Fig. 52. The cases for corrupt $P_1$, $P_2$, $P_3$ are similar.

**Simulator $S_{\Pi_{\text{mult}}}$**

– $\mathcal{S}_{\Pi_{\text{mult}}}$ sends $\gamma_v$ to $\mathcal{A}$ on behalf of $P_1$, $P_2$, and $H(\gamma_v)$ on behalf of $P_3$, respectively.

– $\mathcal{S}_{\Pi_{\text{mult}}}$ receives $H(\alpha_{\mathcal{V}})$. $H(\alpha_{\mathcal{V}})$, $\beta_v + \gamma_v$ from $\mathcal{A}$ on behalf of $P_2$, $P_3$, $P_3$, respectively.

Figure 52: Simulator $S_{\Pi_{\text{mult}}}$ for corrupt $P_0$

### C.2.5 Joint Sharing Protocol

The case for corrupt $P_0$ is given in Fig. 53.

**Simulator $S_{\Pi_{\text{jsh4}}}$**

**Preprocessing:**

– $\mathcal{S}_{\Pi_{\text{jsh4}}}$ has knowledge of $\alpha_v$ and $\gamma_v$, which it obtains while emulating $\mathcal{F}_{\text{setup4}}$. The common values shared with the $\mathcal{A}$ are sampled using the appropriate shared keys, while other values are sampled at random.

**Online:**

– If dealers are $(P_0, P_1)$: $\mathcal{S}_{\Pi_{\text{jsh4}}}$ computes $\beta_v$ using $\gamma$. Steps corresponding to $\Pi_{\text{jmp4}}$ are simulated according to $S_{\Pi_{\text{mult}}}$ where $P_0$ acts as one of the sender for $\beta_v$.

– If dealers are $(P_0, P_2)$ or $(P_0, P_3)$: Analogous to the above case.

– If dealers are $(P_1, P_2)$: $\mathcal{S}_{\Pi_{\text{jsh4}}}$ sets $v = 0$ and $\beta_v = [\alpha_{\mathcal{V}}] + |\alpha_{\mathcal{V}}|_2$. Steps corresponding to $\Pi_{\text{jmp4}}$ are simulated according to $S_{\Pi_{\text{mult}}}$ where $P_0$ acts as the receiver for $\beta_v$.

– If dealers are $(P_3, P_1)$: $\mathcal{S}_{\Pi_{\text{jsh4}}}$ sets $v = 0$ and $\beta_v = [|\alpha_{\mathcal{V}}|] + ||\alpha_{\mathcal{V}}|_2$. Steps corresponding to $\Pi_{\text{jmp4}}$ are simulated according to $S_{\Pi_{\text{mult}}}$ where $P_0$ acts as the server outside the computation for $\beta_v$, and according to $S_{\Pi_{\text{jsh4}}}$ where $P_0$ acts as the receiver for $\beta_v$.

– If dealers are $(P_3, P_2)$: Analogous to the above case.

Figure 53: Simulator $S_{\Pi_{\text{jsh4}}}$ for corrupt $P_0$

The case for corrupt $P_1$ is given in Fig. 54. The case for corrupt $P_2$ is similar.

**Simulator $S_{\Pi_{\text{jsh4}}}$**

**Preprocessing:**

– $\mathcal{S}_{\Pi_{\text{jsh4}}}$ has knowledge of $\alpha$-values and $\gamma$ corresponding to $v$
which it obtains while emulating $\mathcal{F}_{\text{setup4}}$. The common values shared with the $\mathcal{A}$ are sampled using the appropriate shared keys, while other values are sampled at random.

**Online:**
- If dealers are $(P_0, P_1)$: $S_{\Pi_{\text{mp4}}}^P$ computes $\beta_\nu$ using $v$. Steps corresponding to $\Pi_{\text{mp4}}$ are simulated according to $S_{\Pi_{\text{mp4}}}^P$ where $P_1$ acts as one of the sender for $\beta_\nu$.
- If dealers are $(P_1, P_2)$: Analogous to the previous case, except that now $\beta_\nu + \gamma_\nu$ is sent instead of $\beta_\nu$.
- If dealers are $(P_3, P_1)$: $S_{\Pi_{\text{mp4}}}^P$ computes $\beta_\nu$ and $\beta_\nu + \gamma_\nu$. Steps corresponding to $\Pi_{\text{mp4}}$ are simulated according to $S_{\Pi_{\text{mp4}}}^P$ where $P_1$ acts as one of the sender for $\beta_\nu$, $\beta_\nu + \gamma_\nu$.
- If dealers are $(P_0, P_2)$ or $(P_0, P_3)$ or $(P_3, P_2)$: $S_{\Pi_{\text{shat}}}^P$ sets $v = 0$ and $\beta_\nu = [\alpha_v]_1 + [\alpha_v]_2$. Steps corresponding to $\Pi_{\text{mp4}}$ are simulated according to $S_{\Pi_{\text{mp4}}}^P$ where $P_1$ acts as the receiver for $\beta_\nu$.

Figure 54: Simulator $S_{\Pi_{\text{shat}}}^P$ for corrupt $P_1$

The case for corrupt $P_3$ is given in Fig. 55.

**Simulator $S_{\Pi_{\text{shat}}}^P$**

**Preprocessing:**
- $S_{\Pi_{\text{shat}}}^P$ has knowledge of $\alpha$-values and $\gamma$ corresponding to $v$ which it obtains while emulating $\mathcal{F}_{\text{setup4}}$. The common values shared with the $\mathcal{A}$ are sampled using the appropriate shared keys, while other values are sampled at random.

**Online:**
- If dealers are $(P_1, P_2)$: $S_{\Pi_{\text{shat}}}^P$ sets $v = 0$. Steps corresponding to $\Pi_{\text{mp4}}$ are simulated according to $S_{\Pi_{\text{mp4}}}^P$ where $P_3$ acts as the server outside the computation for $\beta_\nu + \gamma_\nu$.
- If dealers are $(P_0, P_1)$ or $(P_0, P_3)$: Analogous to the above case.
- If dealers are $(P_3, P_1)$: $S_{\Pi_{\text{shat}}}^P$ computes $\beta_\nu$ using $v$. Steps corresponding to $\Pi_{\text{mp4}}$ are simulated according to $S_{\Pi_{\text{mp4}}}^P$ where $P_3$ acts as one of the sender for sending $\beta_\nu$.
- If dealers are $(P_3, P_1)$: $S_{\Pi_{\text{shat}}}^P$ computes $\beta_\nu$ and $\beta_\nu + \gamma_\nu$. Steps corresponding to $\Pi_{\text{mp4}}$ are simulated according to $S_{\Pi_{\text{mp4}}}^P$ where $P_3$ acts as one of the sender for sending $\beta_\nu$, $\beta_\nu + \gamma_\nu$.
- If dealers are $(P_3, P_2)$: Analogous to the above case.

Figure 55: Simulator $S_{\Pi_{\text{shat}}}^P$ for corrupt $P_3$

**C.2.6 Dot Product Protocol**

The case for corrupt $P_0$ is given in Fig. 56.

**Simulator $S_{\Pi_{\text{mp4}}}^P$**

**Preprocessing:**
- $S_{\Pi_{\text{mp4}}}^P$ samples $[\alpha_z]_1$, $[\alpha_z]_2$, $[\Gamma_{\tilde{\chi} \tilde{\psi}}]$ using the respective keys with $\mathcal{A}$, $S_{\Pi_{\text{mp4}}}^P$ samples $\gamma, \psi, r$ randomly on behalf of the respective honest parties, and computes $[\Gamma_{\tilde{\chi} \tilde{\psi}}]$ honestly.
- Steps corresponding to $\Pi_{\text{mp4}}$ are simulated according to $S_{\Pi_{\text{mp4}}}^P$ where $P_0$ acts as one of the sender for $[\Gamma_{\tilde{\chi} \tilde{\psi}}]$.
- $S_{\Pi_{\text{mp4}}}^P$ computes $\chi_1, \chi_2$ honestly. Steps corresponding to $\Pi_{\text{mp4}}$ are simulated according to $S_{\Pi_{\text{mp4}}}^P$ where $P_0$ acts as the receiver for $\chi_1$ and $\chi_2$.

**Online:**
- $S_{\Pi_{\text{mp4}}}^P$ computes $[\beta_1^2]_1$, $[\beta_2^2]_2$ honestly. Steps corresponding to $\Pi_{\text{mp4}}$ are simulated according to $S_{\Pi_{\text{mp4}}}^P$ where $P_0$ acts as one of the sender for $[\beta_1^2]_1$, $[\beta_2^2]_2$.
- $S_{\Pi_{\text{mp4}}}^P$ computes $\beta_2 + \gamma_2$. Steps corresponding to $\Pi_{\text{mp4}}$ are simulated according to $S_{\Pi_{\text{mp4}}}^P$ where $P_0$ acts as the receiver for $\beta_2 + \gamma_2$.

Figure 56: Simulator $S_{\Pi_{\text{mp4}}}^P$ for corrupt $P_0$

The case for corrupt $P_1$ is given in Fig. 57. The case for corrupt $P_2$ is similar.

**Simulator $S_{\Pi_{\text{mp4}}}^P$**

**Preprocessing:**
- $S_{\Pi_{\text{mp4}}}^P$ samples $[\alpha_z]_1$, $[\alpha_z]_2$, $[\Gamma_{\tilde{\chi} \tilde{\psi}}]$ using the respective keys with $\mathcal{A}$, $S_{\Pi_{\text{mp4}}}^P$ samples $[\alpha_z]_2$ randomly on behalf of the respective honest parties.
- Steps corresponding to $\Pi_{\text{mp4}}$ are simulated according to $S_{\Pi_{\text{mp4}}}^P$ where $P_1$ acts as the server outside the computation for $[\Gamma_{\tilde{\chi} \tilde{\psi}}]$.
- $S_{\Pi_{\text{mp4}}}^P$ computes $\chi_1$. Steps corresponding to $\Pi_{\text{mp4}}$ are simulated according to $S_{\Pi_{\text{mp4}}}^P$ where $P_1$ acts as one of the sender for $\chi_1$.
- Steps corresponding to $\Pi_{\text{mp4}}$ are simulated according to $S_{\Pi_{\text{mp4}}}^P$ where $P_1$ acts as the server outside the computation for $\chi_2$.

**Online:**
- $S_{\Pi_{\text{mp4}}}^P$ computes $[\beta_1^2]_1$, $[\beta_2^2]_2$. Steps corresponding to $\Pi_{\text{mp4}}$ are simulated according to $S_{\Pi_{\text{mp4}}}^P$ and $S_{\Pi_{\text{mp4}}}^P$ where $P_1$ acts as one of the sender for $[\beta_1^2]_1$, and $P_1$ acts as the receiver for $[\beta_2^2]_2$.
- $S_{\Pi_{\text{mp4}}}^P$ computes $\beta_2 + \gamma_2$. Steps corresponding to $\Pi_{\text{mp4}}$ are simulated according to $S_{\Pi_{\text{mp4}}}^P$ where $P_1$ acts as one of the sender for $\beta_2 + \gamma_2$.

Figure 57: Simulator $S_{\Pi_{\text{mp4}}}^P$ for corrupt $P_1$

The case for corrupt $P_3$ is given in Fig. 58.

**Simulator $S_{\Pi_{\text{mp4}}}^P$**

**Preprocessing:**
- $S_{\Pi_{\text{mp4}}}^P$ samples $[\alpha_z]_1$, $[\alpha_z]_2$, $[\Gamma_{\tilde{\chi} \tilde{\psi}}]$ using the respective keys with $\mathcal{A}$, $S_{\Pi_{\text{mp4}}}^P$ computes $[\Gamma_{\tilde{\chi} \tilde{\psi}}]$ honestly.
- Steps corresponding to $\Pi_{\text{mp4}}$ are simulated according to $S_{\Pi_{\text{mp4}}}^P$ where $P_3$ acts as one of the sender for $[\Gamma_{\tilde{\chi} \tilde{\psi}}]$.
\[ S'_{\text{jmp4}} \text{ where } P_3 \text{ acts as one of the sender for } [\Gamma_{\vec{x} \cdot \vec{y}}]_2. \]

- \( S'_{\text{dotp4}} \) computes \( \chi_1, \chi_2 \). Steps corresponding to \( \Pi_{\text{jmp4}} \) are simulated according to \( S'_{\text{dotp4}} \) where \( P_3 \) acts as one of the sender for \( \chi_1, \chi_2 \).

**Online:**

- Steps corresponding to \( \Pi_{\text{jmp4}} \) are simulated according to \( S'_{\text{dotp4}} \) where \( P_3 \) acts as the server outside the computation for \( [\beta_z^*]_1, [\beta_z^*]_2, \beta_z + \gamma z \).

**C.2.7 Truncation Pair Generation**

Here we give the simulation steps for \( \Pi_{\text{trgen4}} \). The case for corrupt \( P_0 \) is given in Fig. 59. The case for corrupt \( P_3 \) is similar.

**Simulator \( S'_{\Pi_{\text{dotp4}}} \) for corrupt \( P_3 \)**

- \( S'_{\Pi_{\text{dotp4}}} \) samples \( R_1, R_2 \) using the respective keys with \( A \).
- Steps corresponding to \( \Pi_{\text{jsh4}} \) are simulated according to \( S'_{\Pi_{\text{dotp4}}} \) (Fig. 53).

Figure 58: Simulator \( S'_{\Pi_{\text{dotp4}}} \) for corrupt \( P_3 \)

**Simulator \( S'_{\Pi_{\text{trgen4}}} \) for corrupt \( P_0 \)**

The case for corrupt \( P_1 \) is given in Fig. 60. The case for corrupt \( P_2 \) is similar.

**Simulator \( S'_{\Pi_{\text{trgen4}}} \) for corrupt \( P_1 \)**

- \( S'_{\Pi_{\text{trgen4}}} \) samples \( R_1 \) using the respective keys with \( A \), and samples \( R_2 \) randomly.
- Steps corresponding to \( \Pi_{\text{jsh4}} \) are simulated according to \( S'_{\Pi_{\text{trgen4}}} \) (Fig. 54).

Figure 59: Simulator \( S'_{\Pi_{\text{trgen4}}} \) for corrupt \( P_0 \)

Figure 60: Simulator \( S'_{\Pi_{\text{trgen4}}} \) for corrupt \( P_1 \)