ASTRA: High Throughput 3PC over Rings with Application to Secure Prediction[*]

Harsh Chaudhari
Indian Institute of Science, Bangalore India
chaudharim@iisc.ac.in

Ashish Choudhury[†]
International Institute of Information Technology
Bangalore, India
ashish.choudhury@iiitb.ac.in

Arpita Patra[‡]
Indian Institute of Science, Bangalore India
arpita@iisc.ac.in

Ajith Suresh
Indian Institute of Science, Bangalore India
ajith@iisc.ac.in

Abstract

The concrete efficiency of secure computation has been the focus of many recent works. In this work, we present concretely-efficient protocols for secure 3-party computation (3PC) over a ring of integers modulo 2^ℓ tolerating one corruption, both with semi-honest and malicious security. Owing to the fact that computation over ring emulates computation over the real-world system architectures, secure computation over ring has gained momentum of late.

Cast in the offline-online paradigm, our constructions present the most efficient online phase in concrete terms. In the semi-honest setting, our protocol requires communication of 2 ring elements per multiplication gate during the *online* phase, attaining a per-party cost of *less than one element*. This is achieved for the first time in the regime of 3PC. In the *malicious* setting, our protocol requires communication of 4 elements per multiplication gate during the online phase, beating the state-of-the-art protocol by 5 elements. Realized with both the security notions of selective abort and fairness, the malicious protocol with fairness involves slightly more communication than its counterpart with abort security for the output gates *alone*.

We apply our techniques from 3PC in the regime of secure server-aided machine-learning (ML) inference for a range of prediction functions– linear regression, linear SVM regression, logistic regression, and linear SVM classification. Our setting considers a model-owner with trained model parameters and a client with a query, with the latter willing to learn the prediction of her query based on the model parameters of the former. The inputs and computation are outsourced to a set of three non-colluding servers. Our constructions catering to both semi-honest and the malicious world, invariably perform better than the existing constructions.

1 INTRODUCTION

Secure Multi-Party Computation (MPC) [10, 36, 64], the holy grail of secure distributed computing, enables a set of n mutually distrusting parties to perform joint computation on their private inputs, in a way that no coalition of t parties can learn more information than the output (privacy) or affect the true output of the computation (correctness). While MPC, in general, has been a subject of extensive research, the area of MPC with a small number of parties in the *honest majority* setting [4, 16, 18, 33, 51] has drawn popularity of late mainly due to its efficiency and simplicity. Furthermore, most real-time applications involve a small number of parties. Applications such as statistical and financial data analysis [14], email-filtering [44], distributed credential encryption [51], Danish sugar beet auction [15] involve 3 parties. Well-known MPC frameworks such as VIFF [35], Sharemind [13] have been explored with 3 parties. Recent advances in secure machine learning (ML) based on MPC have shown applications with a small number of parties [49, 50, 52, 59, 63]. MPC with a small number of parties helps solve MPC over large population as well via server-aided computation, where a small number of servers jointly hold the input data of the large population and run an MPC protocol evaluating the desired function.

With motivations galore, the specific problem of three-party computation (3PC) tolerating one corruption has received phenomenal attention of late [2, 4, 16, 19, 33, 38, 47, 51, 55, 55, 58]. Leveraging honest majority, this setting allows to attain stronger security goals such as *fairness* (corrupt party receives the output only if all honest parties receive output) which are otherwise impossible with dishonest-majority [21]. In this work, we revisit the concrete efficiency of 3PC and to be specific, the efficiency of the input-dependent computation.

The two typical lines of constructions that the regime of MPC over small population offer are– high-throughput [2–4, 19, 33, 55], and low-latency [16, 18, 37, 38, 51, 58] protocols. Relying on secret sharing mechanism, the former category requires low communication overhead (bandwidth) and simple computations. Catering to low-latency networks, this category takes a number of communication rounds proportional to the multiplicative depth of the circuit representing the function to be computed. On the other hand, the other category, relying on garbled circuits, requires a constant number of communication rounds and serve better in high-latency networks such as the Internet. The focus of this work is high-throughput 3PC.

Almost all high-throughput protocols evaluate a circuit that represents the function f to be computed in a secret-shared fashion. Informally, the parties jointly maintain the invariant that for

[*] This article is the full and extended version of an earlier article to appear in ACM CCSW 2019

[†] This Publication is an outcome of the R&D work undertaken in the project under the Visvesvaraya PhD Scheme of Ministry of Electronics & Information Technology, Government of India, being implemented by Digital India Corporation (formerly Media Lab Asia)

[‡] Arpita Patra is supported by SERB Women Excellence Award 2017 (DSTO 1706).

each wire in the circuit, the exact value over that wire is available in a secret-shared fashion among the parties, in a way that the adversary learns no information about the exact value from the shares of the corrupt parties. Upon completion of the circuit evaluation, the parties jointly reconstruct the secret-shared function output. Intuitively, the security holds as no intermediate value is revealed during the computation. The deployed secret-sharing schemes are typically linear, ensuring non-interactive evaluation of the linear gates. The communication is required *only* for the non-linear (i.e.multiplication) gates in the circuit. The focus then turns on improving the communication overhead per multiplication gate. Recent literature has seen a range of customized linear secret-sharing schemes over a small number of parties, boosting the performance for multiplication gate spectacularly [2, 33, 37].

In an interesting direction towards improving efficiency, MPC protocols are suggested to be cast in two phases– an offline phase that performs *input-independent* computation and an online phase that performs *fast input-dependent* computation utilizing the offline computation [6]. The offline phase, run in advance, generates 'raw material' in a relatively expensive way to yield a blazing-fast online phase. This is very useful in a scenario where a set of parties agreed to perform a specific computation repetitively over a period of time. The parties can batch together the offline computations and generate a large volume of offline data to support the execution of multiple online phases. Popularly referred as offline-online paradigm [6], there are constructions abound that show effectiveness of this paradigm both in the theoretical [6–9, 11, 20] and practical [5, 22, 26–28, 41–43, 59] regime.

In yet another direction to improve practical efficiency, secure computation for arithmetic circuits over rings has gained momentum of late, while traditionally fields have been the default choice. Computation over rings models computation in the real-life computer architectures such as computation over CPU words of size 32 or 64 bits. In 3PC setting, the work of [13] supports arithmetic circuits over arbitrary rings with passive security, while [2] offers active security. The works of [27, 31] improve online communication over arbitrary rings with active security, yet fall back to computation over large prime-order fields in the offline phase. This forces the developer to depend on external libraries for fields (which are 10×-100× slower) compared to the real-world system architectures based on 32-bit and 64-bit rings.

1.1 Our Contribution

In this work, we follow the offline-online paradigm and propose 3PC constructions over a ring $\mathbb{Z}_{2^\ell}$ (that include Boolean ring $\mathbb{Z}_{2^1}$) with the most efficient online phase in concrete terms. Though the focus lies on the online phase, the cost of offline phase is respected and is kept in check. We present a range of constructions satisfying semi-honest and malicious security. We apply our techniques for secure prediction for a range of prediction functions in the outsourced setting and build a number of constructions tolerating semi-honest and malicious adversary. A common feature that all our constructions exude is that function-dependent communication is needed amongst *fewer* than three pairs in the online phase, yielding better online performance. We elaborate on our contributions.

Secure 3PC. Our 3PC protocol with semi-honest security requires communication of two elements per multiplication during the online phase. The per-party online cost of our protocol is less than one element per multiplication, a property achieved for the first time in the 3PC setting. This improvement comes from the use of a form of linear secret-sharing scheme inspired from the work of [37] that allows offloading the task of one of the parties in the offline phase and requires *only* two parties to talk to each other in the online phase. This essentially implies that the evaluation of multiplication gates in the online phase requires the presence of just two parties, unlike the previous protocols [2, 4, 19, 33, 47] that insist all the three parties be awake throughout the computation. One exception is the case of Chameleon [59], where two parties perform the online computation with the help of correlated randomness generated by a *semi-trusted* party in the offline phase. Though the model looks similar in the semi-honest setting, we achieve a stronger security guarantee by allowing the third party to be maliciously corrupted. Moreover, our multiplication protocol in the semi-honest setting requires an online communication of 2 ring elements as opposed to 4 of [59]. We achieve this 2× improvement while maintaining the same offline cost (1 element) of [59].

For the malicious case, our protocol requires a *total* communication of four elements per multiplication during the online phase. The state-of-the-art protocol over *rings* requires nine ring elements per multiplication in the online phase. Lastly, we boost the security of our malicious protocol to fairness without affecting its cost per multiplication. The inflation inflicted is purely for the output gates and to be specific for output reconstruction. The key contribution of the fair protocol lies in constructing a fair reconstruction protocol that ensures a corrupt party receives the output if and only if the honest parties receive. The fair reconstruction does not resort to a broadcast channel and instead rely on a new concept of 'proof of origin' that tackles the confusion a sender can infuse in the absence of broadcast channel by sending different messages to its fellow parties over private channels.

In Table 1, we compare our work with the most relevant works. The communication specifies the number of bits that needs to be communicated per multiplication gate in the amortized sense.

Semi-honest			Malicious			
Ref.	Offline	Online	Ref.	Offline	Online	Fair?
[4]	0	3ℓ	[2]	12ℓ	9ℓ	✗
This	ℓ	2ℓ	**This**	21ℓ	4ℓ	✓

Table 1: Concrete Comparison of our 3PC protocols

Secure ML Prediction. The growing influx of data makes ML a promising applied science, touching human life like never before. Its potential can be leveraged to advance areas such as medicine [32], facial recognition [60], banking, recommendation services, threat analysis, and authentication technologies. Many technology giants such as Amazon, Microsoft, Google, Apple are offering cloud-based ML services to their customers both in the form of training platforms that train models on customer data and pre-trained models that can be used for inference, often referred as 'ML as a Service

(MLaaS)'. However, these huge promises can only be unleashed when rightful privacy concerns, due to ethical, legal or competitive reasons, can be brought to control via privacy-preserving techniques. This is when privacy-preserving techniques such as MPC meets ML, with the former serving extensively in an effective way both for secure training and prediction [25, 45, 48, 50, 52, 59, 63]. This has a huge impact on the efficiency

In this work, we target secure prediction where a model-owner holding the model parameters enables a client to receive a prediction result to its query as per the model, respecting privacy concerns of each other. Following the works of [49, 50, 52, 59, 63], we envision a server-aided setting where the inputs and computation are outsourced to a set of servers. We consider some of the widely used ML algorithms, namely linear regression and linear support vector machines (SVM) regression for *regression* task and logistic regression and SVM classification for *classification* task [12, 30]. We propose an efficient protocol for *secure comparison* that is an important building block for classification task. We exploit the asymmetry in our secret sharing scheme and forgo expensive primitives such as garbled circuits or parallel prefix adders, which are used in [52] and [50]. As emphasized below, our technique allows attaining a constant round complexity for classification tasks.

In Table 2, we compare our results with the best-known construction of ABY3 [50] that uses 3-server setting. As the main focus of ABY3 is training, they develop an efficient technique for fixed-point multiplication in shared fashion, tackling the overflow and accuracy issues in the face of repeated multiplications. Such techniques can be avoided for functions inducing circuit of multiplicative depth one. Hence we compare with the version of ABY3 that skips this and present below a consolidated comparison in terms of communication. Following the works in the domain of server-aided prediction, we only count the cost incurred by the servers to compute the output in shared form from the inputs in shared form, ignoring the cost for sharing the inputs and reconstructing the output. 'Reg' denotes regression, 'Class' denotes classification and 'Round' denotes the number of online rounds. Here ℓ denotes the size of the underlying ring $\mathbb{Z}_{2^\ell}$ (in bits) and d denotes the number of features. The values in Table 2 indicate that our protocol clearly outperforms ABY3, in terms of online communication in all the settings. In the semi-honest setting, this is achieved since we are able to shift 33% of the overall communication to the offline phase. In the malicious setting, online communication is further improved because of our efficient dot-product protocol. Moreover, our novel construction for secure comparison allows the classification protocols to be *round constant* unlike ABY3 which requires $\log \ell + 1$ rounds.

Ref.	Param.	Semi-honest		Malicious	
		Reg	Class	Reg	Class
ABY3	Offline	0	0	$12d\ell$	$12d\ell + 24\ell$
	Online	3ℓ	9ℓ	$9d\ell$	$9d\ell + 18\ell$
	Round	1	$\log \ell + 1$	1	$\log \ell + 1$
This	Offline	ℓ	ℓ	$21d\ell$	$21d\ell + 46\ell$
	Online	$\mathbf{2\ell}$	$\mathbf{4\ell + 2}$	$\mathbf{2d\ell + 2\ell}$	$\mathbf{2d\ell + 8\ell + 1}$
	Round	1	**3**	1	**4**

Table 2: Concrete Comparison of Our ML Protocols

Implementation. For 3PC, we implement our protocols over a ring $\mathbb{Z}_{2^{32}}$ and compare with the state-of-the-art protocols, namely [4] in the semi-honest setting and [2] in the malicious setting. We use latency (runtime) and online throughput as the parameters for the comparison. The online throughput in LAN setting is computed as the number of AES circuits computed per second in the online phase. As an AES circuit requires more than a second in WAN setting, we take a different measure which is the number of AND gates per second. We observe that our protocols improve the online throughput of the existing one by a factor of 1.05× to 1.51× over various settings. For the WAN setting, this improvement translates to computing *additional* AND gates of the range 1.44 to 4.39 millions per second.

For secure prediction, we implement our work using MNIST [46] dataset where $d = 784$ and with $\ell = 64$ in both LAN and WAN setting. We observe an improvement of 1.02× to 2.56× over ABY3 [50], in terms of online throughput, over various settings for regression algorithms. For classification algorithms, the improvement ranges from 1.5× to 2.93×.

Open Problems. Our techniques are tailor-made for 3PC with 1 corruption. Extending these techniques to the case of an arbitrary $Q^{(2)}$ adversary structure [61] is left as an open problem.

2 PRELIMINARIES AND DEFINITIONS

We consider a set of three parties $\mathcal{P} = \{P_0, P_1, P_2\}$ that are connected by pair-wise private and authentic channels in a synchronous network. The function f to be evaluated is expressed as a circuit ckt over an arithmetic ring $\mathbb{Z}_{2^\ell}$, consisting of 2-input addition and multiplication gates. The topology of the circuit is assumed to be publicly known. The term D denotes the multiplicative depth of the circuit, while $\mathsf{I}, \mathsf{O}, \mathsf{A}, \mathsf{M}$ denote the number of input wires, output wires, addition gates and multiplication gates respectively in ckt. We use the notation w_x to denote a wire w with value x flowing through it. We use $\mathsf{g} = (\mathsf{w}_x, \mathsf{w}_y, \mathsf{w}_z)$ to denote a gate in the ckt with left input wire w_x, right input wire w_y and output wire w_z. In our protocols, we divide $\mathcal{P}$ into disjoint sets $\{P_0\}$ and $\{P_1, P_2\}$, where P_0 acts as a "distributor" to do the "pre-processing" during the offline phase, which is utilized by the "evaluators" P_1, P_2 to evaluate ckt during the online phase. We use the superscripts "s" and "m" to distinguish the protocols in the semi-honest and malicious setting respectively. The protocols over boolean ring $\mathbb{Z}_{2^1}$ can be obtained by replacing the arithmetic operations addition (+) and multiplication (×) with XOR (⊕) and AND (·) respectively. Below, we present the tools needed for our protocol.

2.1 Collision Resistant Hash

Consider a hash function family $\mathsf{H} = \mathcal{K} \times \mathcal{L} \rightarrow \mathcal{Y}$. The hash function H is said to be collision resistant if for all probabilistic polynomial-time adversaries $\mathcal{A}$, given the description of H_k where $k \in_R \mathcal{K}$, there exists a negligible function negl() such that

$\Pr[(x_1, x_2) \leftarrow \mathcal{A}(k) : (x_1 \neq x_2) \wedge \mathsf{H}_k(x_1) = \mathsf{H}_k(x_2)] \leq \mathsf{negl}(\kappa)$, where $m = \mathsf{poly}(\kappa)$ and $x_1, x_2 \in_R \{0,1\}^m$.

2.2 Shared Key Setup

To save communication between the parties, a one-time setup that establishes pre-shared random keys for a pseudo-random function (PRF) F is used. A similar setup has been used in the known protocols in the 3PC setting [2, 33, 50]. Here $F : 0,1^{\kappa} \times 0,1^{\kappa} \rightarrow X$ is a secure PRF, with co-domain X being $\mathbb{Z}_{2^\ell}$. The set of keys are:

- One key shared between every pair– k_{01}, k_{02}, k_{12} for the parties $(P_0, P_1), (P_0, P_2), (P_1, P_2)$ respectively.
- One shared key amongst all– $k_{\mathcal{P}}$.

If parties P_0, P_1 wish to sample a random value r non-interactively, they invoke $F_{k_{01}}(id_{01})$ to obtain r, where id_{01} is a counter that the parties update locally after every PRF invocation. The key used to sample a value will be clear from the context (from the identities of the pair that samples or from the fact that it is sampled by all) and will be omitted. We model the key setup via a functionality $\mathcal{F}_{\mathsf{setup}}$ that can be realized using any secure MPC protocol.

3 SHARING SEMANTICS

In this section, we explain two variants of secret sharing that are used in this work. Both the variants operate over arithmetic ($\mathbb{Z}_{2^\ell}$) and boolean ($\mathbb{Z}_{2^1}$) rings.

$[\cdot]$-sharing. A value v is said to be $[\cdot]$-shared among parties P_1, P_2, if the parties P_1 and P_2 respectively holds the values v_1 and v_2 such that $\mathsf{v} = \mathsf{v}_1 + \mathsf{v}_2$. We use $[\cdot]_{P_i}$ to denote the $[\cdot]$-share of party P_i for $i \in \{1, 2\}$.

$[\![\cdot]\!]$-sharing. A value v is said to be $[\![\cdot]\!]$-shared among parties P_0, P_1 and P_2, if

- there exists values $\lambda_{\mathsf{v}}, \mathsf{m}_{\mathsf{v}}$ such that $\mathsf{v} = \mathsf{m}_{\mathsf{v}} - \lambda_{\mathsf{v}}$.
- P_0 holds $\lambda_{\mathsf{v},1}$ and $\lambda_{\mathsf{v},2}$.
- P_1 and P_2 hold $(\mathsf{m}_{\mathsf{v}}, \lambda_{\mathsf{v},1})$ and $(\mathsf{m}_{\mathsf{v}}, \lambda_{\mathsf{v},2})$ respectively.

We denote $[\![\cdot]\!]$-share of the parties as $[\![\mathsf{v}]\!]_{P_0} = (\lambda_{\mathsf{v},1}, \lambda_{\mathsf{v},2})$, $[\![\mathsf{v}]\!]_{P_1} = (\mathsf{m}_{\mathsf{v}}, \lambda_{\mathsf{v},1})$ and $[\![\mathsf{v}]\!]_{P_2} = (\mathsf{m}_{\mathsf{v}}, \lambda_{\mathsf{v},2})$. We use $[\![\mathsf{v}]\!] = (\mathsf{m}_{\mathsf{v}}, [\lambda_{\mathsf{v}}])$ to denote the $[\![\cdot]\!]$-share of v.

Linearity of the secret sharing schemes. Given the $[\cdot]$-sharing of $x, y \in \mathbb{Z}_{2^\ell}$ and public constants $c_1, c_2 \in \mathbb{Z}_{2^\ell}$, parties can locally compute $[c_1x + c_2y]$. To see this,

$$[c_1x + c_2y] = (c_1x_1 + c_2y_1, c_1x_2 + c_2y_2) = c_1[x] + c_2[y]$$

It is easy to see that the linearity trivially extends to $[\![\cdot]\!]$-sharing as well. That is, given the $[\![\cdot]\!]$-sharing of x, y and public constants c_1, c_2, parties can locally compute $[\![c_1x + c_2y]\!]$.

$$\begin{aligned}[\![c_1x + c_2y]\!] &= (c_1\mathsf{m}_x + c_2\mathsf{m}_y, c_1[\lambda_x] + c_2[\lambda_y]) \\ &= c_1[\![x]\!] + c_2[\![y]\!]\end{aligned}$$

The linearity property enables parties to *locally* perform the operations such as addition and multiplication with a public constant.

4 OUR 3PC PROTOCOL

We start with our 3PC protocol $\Pi^{\mathsf{s}}_{\mathsf{3pc}}$ that securely evaluates any arithmetic circuit over $\mathbb{Z}_{2^\ell}$ for $\ell \geq 1$, tolerating semi-honest adversaries.

4.1 3PC with semi-honest security

Our protocol $\Pi^{\mathsf{s}}_{\mathsf{3pc}}$ has three stages– input-sharing, circuit-evaluation, and output-reconstruction. During input-sharing stage, each party generates a random $[\![\cdot]\!]$-sharing of its input. During the circuit-evaluation stage, the parties evaluate ckt in a $[\![\cdot]\!]$-shared fashion. During the output-reconstruction stage, the parties reconstruct the $[\![\cdot]\!]$-shared circuit outputs. All the stages (except output-reconstruction) can be cast in the offline and online phase, where steps independent of the actual inputs can be executed in the offline phase. At a high level, the $[\cdot]$-sharing needed behind every $[\![\cdot]\!]$-shared value in the online phase is precomputed, while the $[\![\cdot]\!]$-sharing of values themselves are computed in the online phase. We distinguish these steps as *Offline* and *Online* steps respectively. While the *Offline* steps are executed *only* by the distributor P_0, the *Online* steps are executed *only* by the evaluators P_1 and P_2. We now individually elaborate on each of the stages.

Input-sharing Stage. Protocol $\Pi^{\mathsf{s}}_{\mathsf{Sh}}(P_i, x)$ (Figure 1) allows party $P_i \in \mathcal{P}$, the designated party to give input $x \in \mathbb{Z}_{2^\ell}$ to wire w_x, to $[\![\cdot]\!]$-share its input. In the offline step, parties locally sample $\lambda_{x,1}$ and $\lambda_{x,2}$ using their shared randomness such that parties P_0 and P_i learns the entire λ_x. In the online step, P_i computes m_x using λ_x and sends it to the evaluators.

Offline:

- If $P_i = P_0$, parties P_0, P_j for $j \in \{1, 2\}$ locally sample a random $\lambda_{x,j} \in \mathbb{Z}_{2^\ell}$.
- If $P_i = P_1$, parties P_0, P_1 sample a random $\lambda_{x,1} \in \mathbb{Z}_{2^\ell}$ while all the parties in $\mathcal{P}$ sample a random $\lambda_{x,2}$.
- If $P_i = P_2$, parties P_0, P_2 sample a random $\lambda_{x,2} \in \mathbb{Z}_{2^\ell}$ while all the parties in $\mathcal{P}$ sample a random $\lambda_{x,1}$.

Online: P_i sends $\mathsf{m}_x = x + \lambda_x$ to every P_j for $j \in \{1, 2\}$ who then sets $[\![x]\!]_{P_j} = (\mathsf{m}_x, \lambda_{x,j})$.

Figure 1: Protocol $\Pi^{\mathsf{s}}_{\mathsf{Sh}}(P_i, x)$

Circuit-evaluation Stage. Here parties evaluate each gate g in the ckt in the *topological* order, where they maintain the invariant that given inputs of g in $[\![\cdot]\!]$-shared fashion, parties generate $[\![\cdot]\!]$-sharing for the output of g. If g is an addition gate $(\mathsf{w}_x, \mathsf{w}_y, \mathsf{w}_z)$, then this is done locally using the linearity of $[\![\cdot]\!]$-sharing, as per the protocol Π_{Add} (Figure 2).

Offline: P_0, P_1 set $\lambda_{z,1} = \lambda_{x,1} + \lambda_{y,1}$, while P_0, P_2 set $\lambda_{z,2} = \lambda_{x,2} + \lambda_{y,2}$.

Online: P_1 and P_2 set $\mathsf{m}_z = \mathsf{m}_x + \mathsf{m}_y$.

Figure 2: Protocol $\Pi_{\mathsf{Add}}(\mathsf{w}_x, \mathsf{w}_y, \mathsf{w}_z)$

If $\mathsf{g} = (\mathsf{w}_x, \mathsf{w}_y, \mathsf{w}_z)$ is a multiplication gate, then given $[\![x]\!] = (\mathsf{m}_x, [\lambda_x])$ and $[\![y]\!] = (\mathsf{m}_y, [\lambda_y])$, the parties compute $[\![z]\!]$ by running the protocol $\Pi^{\mathsf{s}}_{\mathsf{Mul}}$ (Figure 3). During the offline phase, parties generate λ_z for the gate output. In addition, P_0 also $[\cdot]$-shares the product of the masks of the gate inputs $(\lambda_x\lambda_y)$, both of which are known to P_0 as a part of $\llbracket x \rrbracket_{P_0}$ and $\llbracket y \rrbracket_{P_0}$. Online phase is executed by $\{P_1, P_2\}$, where they locally generate $[\mathsf{m}_z]$, followed by reconstructing m_z.

Offline:

- P_0 and P_1 locally sample random $\lambda_{z,1}, \gamma_{xy,1} \in \mathbb{Z}_{2^\ell}$, while P_0 and P_2 locally sample a random $\lambda_{z,2}$.
- P_0 computes $\gamma_{xy} = \lambda_x\lambda_y$ and sends $\gamma_{xy,2} = \gamma_{xy} - \gamma_{xy,1}$ to P_2.

Online:

- P_i for $i \in \{1, 2\}$ locally computes $[\mathsf{m}_z]_{P_i} = (i-1)\mathsf{m}_x\mathsf{m}_y - \mathsf{m}_x[\lambda_y]_{P_i} - \mathsf{m}_y[\lambda_x]_{P_i} + [\lambda_z]_{P_i} + [\gamma_{xy}]_{P_i}$.
- P_1, P_2 mutually exchange their shares and reconstruct m_z.

Figure 3: Protocol $\Pi^{\mathsf{s}}_{\mathsf{Mul}}(\mathsf{w}_x, \mathsf{w}_y, \mathsf{w}_z)$

Output-reconstruction Stage. To reconstruct the output from $\llbracket y \rrbracket$, we observe that the missing share of party P_i, for $i \in \{0, 1, 2\}$, is held by the other two parties. Thus, one among the other two parties can send the missing share to P_i, who then computes the output as $y = \mathsf{m}_y - \lambda_{y,1} - \lambda_{y,2}$. We call the resultant protocol as $\Pi^{\mathsf{s}}_{\mathsf{Rec}}$.

We combine the aforementioned stages and present $\Pi^{\mathsf{s}}_{\mathsf{3pc}}$ in Figure 4.

Pre-processing (Offline Phase):

- *Input wires*: For $j = 1, \ldots, \mathsf{I}$, corresponding to the circuit-input x_j, parties execute the offline steps of the instance $\Pi^{\mathsf{s}}_{\mathsf{Sh}}(P_i, x_j)$.
- For each gate g in ckt in the topological order, execute the offline steps of the instance $\Pi^{\mathsf{s}}_{\mathsf{Mul}}(\mathsf{w}_{x\,j}, \mathsf{w}_{y\,j}, \mathsf{w}_{z\,j})$ if g is the jth multiplication gate where $j \in \{1, \ldots, \mathsf{M}\}$ or respectively the offline steps of the instance $\Pi_{\mathsf{Add}}(\mathsf{w}_{x\,j}, \mathsf{w}_{y\,j}, \mathsf{w}_{z\,j})$ if g is the jth addition gate where $j \in \{1, \ldots, \mathsf{A}\}$.

Circuit Evaluation (Online Phase):

- *Sharing Circuit-input Values*: For $j = 1, \ldots, \mathsf{I}$, corresponding to the circuit-input x_j, party P_i executes the online steps of the instance $\Pi^{\mathsf{s}}_{\mathsf{Sh}}(P_i, x_j)$, where P_i is the party designated to provide x_j.
- *Gate Evaluation*: For each gate g in ckt in the topological order, P_1, P_2 execute the online steps of the instance $\Pi^{\mathsf{s}}_{\mathsf{Mul}}(\mathsf{w}_{x\,j}, \mathsf{w}_{y\,j}, \mathsf{w}_{z\,j})$ if g is the jth multiplication gate where $j \in \{1, \ldots, \mathsf{M}\}$ or respectively the online steps of the instance $\Pi_{\mathsf{Add}}(\mathsf{w}_{x\,j}, \mathsf{w}_{y\,j}, \mathsf{w}_{z\,j})$ if g is the jth addition gate where $j \in \{1, \ldots, \mathsf{A}\}$.
- *Output Reconstruction*: Let $\llbracket y_1 \rrbracket, \ldots, \llbracket y_{\mathsf{O}} \rrbracket$ be the shared function outputs, where for $j = 1, \ldots, \mathsf{O}$, we have $\llbracket y_j \rrbracket_{P_0} = [\lambda_{y_j}]$, $\llbracket y_j \rrbracket_{P_1} = (\mathsf{m}_{y_j}, [\lambda_{y_j}]_{P_1})$ and $\llbracket y_j \rrbracket_{P_2} = (\mathsf{m}_{y_j}, [\lambda_{y_j}]_{P_2})$. The parties in $\mathcal{P}$ reconstruct y_j by executing the instance $\Pi^{\mathsf{s}}_{\mathsf{Rec}}(\llbracket y_j \rrbracket, \mathcal{P})$.

Figure 4: The semi-honest 3PC protocol $\Pi^{\mathsf{s}}_{\mathsf{3pc}}$

Correctness and Security. We prove correctness and argue security informally below.

Theorem 4.1 (Correctness). *Protocol $\Pi^{\mathsf{s}}_{\mathsf{3pc}}$ is correct.*

Proof. We claim that for every wire in ckt, the parties hold a $\llbracket \cdot \rrbracket$-sharing of the wire value in $\Pi^{\mathsf{s}}_{\mathsf{3pc}}$. The correctness then follows from the fact that for the circuit-output wires, the corresponding $\llbracket \cdot \rrbracket$-sharing is reconstructed correctly. The claim for circuit-input wires follows from $\Pi^{\mathsf{s}}_{\mathsf{Sh}}$, while for addition gates it follows from the linearity of $\llbracket \cdot \rrbracket$-sharing. Consider a multiplication gate $(\mathsf{w}_x, \mathsf{w}_y, \mathsf{w}_z)$, evaluated as per $\Pi^{\mathsf{s}}_{\mathsf{Mul}}$, where $\mathsf{m}_x = x + \lambda_x$, $\mathsf{m}_y = y + \lambda_y$ and $\gamma_{xy} = \lambda_x\lambda_y$. We argue that m_z as computed in online step of $\Pi^{\mathsf{s}}_{\mathsf{Mul}}$ results in $xy + \lambda_z$ and hence at the end of $\Pi^{\mathsf{s}}_{\mathsf{Mul}}$, the parties hold $\llbracket z \rrbracket$. This is because $\mathsf{m}_z = \mathsf{m}_x\mathsf{m}_y - \mathsf{m}_x\lambda_y - \mathsf{m}_y\lambda_x + \lambda_z + \gamma_{xy} = (\mathsf{m}_x - \lambda_x)(\mathsf{m}_y - \lambda_y) + \lambda_z = xy + \lambda_z$. The linearity of $[\cdot]$-sharing implies that P_1 and P_2 correctly compute a $[\cdot]$-sharing of m_z. □

The security is argued as follows. If P_0 is corrupt, then the security follows since P_0 never sees the masked values over the intermediate wires. If one of the evaluators is corrupt, then the security holds since the corrupt evaluator knows only one of the shares of the mask while the other share is picked at random. The detailed security proof appear in Appendix B where we show our protocol emulates the functionality $\mathcal{F}_{\mathsf{3pc}}$ for computing a 3-party function f in the semi-honest setting as given in Figure 5.

$\mathcal{F}_{\mathsf{3pc}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$ and is parameterized by a 3-ary function f, represented by a publicly known arithmetic circuit ckt over $\mathbb{Z}_{2^\ell}$.

Upon receiving the input $x_1, \ldots, x_{\mathsf{I}}$ from the respective parties in $\mathcal{P}$, where each $x_i \in \mathbb{Z}_{2^\ell}$, the functionality computes $(y_1, \ldots, y_{\mathsf{O}}) = f(x_1, \ldots, x_{\mathsf{I}})$ and sends $y_1, \ldots, y_{\mathsf{O}}$ to the parties in $\mathcal{P}$.

Figure 5: Functionality $\mathcal{F}_{\mathsf{3pc}}$

Theorem 4.2. *$\Pi^{\mathsf{s}}_{\mathsf{3pc}}$ requires one round with communication of M ring elements during the offline phase. In the online phase, $\Pi^{\mathsf{s}}_{\mathsf{3pc}}$ requires one round with communication of at most $2\mathsf{I}$ ring elements in the Input-sharing stage, D rounds with communication of $2\mathsf{M}$ ring elements for circuit-evaluation stage and one round with communication of $3\mathsf{O}$ elements for the output-reconstruction stage.*

Proof. During the offline phase, the $[\cdot]$-shares of every λ are generated non-interactively. For the multiplication gates, generating $[\cdot]$-sharing of γ_{xy} values requires one round and communication of M elements. During the online phase, generating the $\llbracket \cdot \rrbracket$-sharing of circuit-inputs requires one round. For each input of P_0, generating the $\llbracket \cdot \rrbracket$-sharing requires a communication of 2 elements, while the same for P_1/P_2 requires one element. So, the Input-sharing phase needs one round and communication of at most $2\mathsf{I}$ elements. Evaluating the addition gates is free, while the same for each multiplication gate requires one round and communication of 2 elements to reconstruct the m_z value. Hence the circuit-evaluation phase needs D rounds and communication of $2\mathsf{M}$ elements. Reconstructing the circuit-outputs require one round and communication of $3\mathsf{O}$ elements. □

4.2 3PC with malicious security

In this section, we describe our maliciously secure 3PC protocol $\Pi^{\mathsf{m}}_{\mathsf{3pc}}$ that securely evaluates any arithmetic circuit over $\mathbb{Z}_{2^\ell}$. Similar to $\Pi^{\mathsf{s}}_{\mathsf{3pc}}$, protocol $\Pi^{\mathsf{m}}_{\mathsf{3pc}}$ has three stages– input-sharing, circuit-evaluation and output-reconstruction.

Input Sharing and Output Reconstruction Stages. We begin with the sharing and reconstruction protocols in the malicious setting, which can readily replace $\Pi^{\mathsf{s}}_{\mathsf{Sh}}$ and $\Pi^{\mathsf{s}}_{\mathsf{Rec}}$ in $\Pi^{\mathsf{m}}_{\mathsf{3pc}}$ to help obtain maliciously-secure input sharing and output reconstruction stage.

In the malicious setting, we need to ensure that the shares possessed by the honest parties are *consistent.* By consistent shares, we mean that the common share possessed by the honest parties should be the same. In protocol $\Pi^{\mathsf{s}}_{\mathsf{Sh}}$, the λ-shares will be consistent since they are generated non-interactively. But, if a corrupt P_0 owns a value x and wants to create an inconsistent $\llbracket x \rrbracket$-sharing, he can send two different versions of m_x to P_1 and P_2. To detect this inconsistency, P_1, P_2 exchange $\mathsf{H}(\mathsf{m}_x)$ and $\mathtt{abort}$ if there is a mismatch. The parties can exchange a combined hash for all the wires where P_0 is the owner and thus the cost reduces to two hash values in the amortized sense. We call the resultant protocol as $\Pi^{\mathsf{m}}_{\mathsf{Sh}}$.

For reconstruction, let $\llbracket y \rrbracket$ be a sharing to be reconstructed where $\llbracket y \rrbracket_{P_0} = (\lambda_{y,1}, \lambda_{y,2})$, $\llbracket y \rrbracket_{P_1} = (\mathsf{m}'_y, \lambda'_{y,1})$ and $\llbracket y \rrbracket_{P_2} = (\mathsf{m}''_y, \lambda'_{y,2})$ (the distinction in the notation is done to differentiate the shares held by each party). Protocol $\Pi^{\mathsf{m}}_{\mathsf{Rec}}(\llbracket y \rrbracket, \mathcal{P})$ (Figure 6) enables each *honest* party in $\mathcal{P}$ to either compute y or output $\perp$.

Online:

- P_0 and P_2 send $\lambda_{y,2}$ and $\mathsf{H}(\lambda'_{y,2})$ respectively to P_1.
- P_0 and P_1 send $\lambda_{y,1}$ and $\mathsf{H}(\lambda'_{y,1})$ respectively to P_2.
- P_1 and P_2 send m'_y and $\mathsf{H}(\mathsf{m}''_y)$ respectively to P_0.

P_i for $i \in \{0, 1, 2\}$ $\mathtt{abort}$ if the received values mismatch. Else P_i sets $y = \mathsf{m}_y - \lambda_{y,1} - \lambda_{y,2}$.

Figure 6: Protocol $\Pi^{\mathsf{m}}_{\mathsf{Rec}}(\llbracket y \rrbracket, \mathcal{P})$

Now the input sharing and output reconstruction stages in $\Pi^{\mathsf{m}}_{\mathsf{3pc}}$ are similar to those in $\Pi^{\mathsf{s}}_{\mathsf{3pc}}$ apart from protocols $\Pi^{\mathsf{s}}_{\mathsf{Sh}}$ and $\Pi^{\mathsf{s}}_{\mathsf{Rec}}$ being replaced with $\Pi^{\mathsf{m}}_{\mathsf{Sh}}$ and $\Pi^{\mathsf{m}}_{\mathsf{Rec}}$ respectively.

Circuit Evaluation Stage. Protocol Π_{Add} remains secure in the malicious setting as well since it involves local operations only. The challenge lies in turning the multiplication protocol $\Pi^{\mathsf{s}}_{\mathsf{Mul}}$ to one that tolerates malicious behaviour. We start with the observation that $\Pi^{\mathsf{s}}_{\mathsf{Mul}}$ suffers in two *mutually-exclusive* ways in the face of one malicious corruption, each under different corruption scenario. When P_0 is corrupt, the only possible violation in $\Pi^{\mathsf{s}}_{\mathsf{Mul}}$ comes in the form of sharing $\gamma_{xy} \neq \lambda_x \lambda_y$ during the offline phase. When P_1 (or P_2) is corrupt, the violation occurs when a wrong share of m_z is handed over to the fellow honest evaluator during the online phase, causing reconstruction of a wrong m_z. While the attacks are quite distinct in nature following the asymmetric roles played by the two sets $\{P_0\}$ and $\{P_1, P_2\}$ in $\Pi^{\mathsf{s}}_{\mathsf{Mul}}$, our novel construction solves both issues at the same time via checking product-relation of a single $\llbracket \cdot \rrbracket$-shared triple. We start with the technique to tackle a corrupt evaluator (P_1 or P_2) during the online phase. To identify if an incorrect m_z is reconstructed by an honest evaluator, say P_1, he can seek the help of P_0 as follows: P_1 can send $\mathsf{m}_x, \mathsf{m}_y$ to P_0, who can then compute m_z, as P_0 already has knowledge of λ_x, λ_y and λ_z from the offline phase and send back to P_1. Note that sending $\mathsf{m}_x, \mathsf{m}_y$ in clear to P_0 breaks privacy of the scheme and hence P_1 sends padded version of the same to P_0, namely $\mathsf{m}^{\star}_x = \mathsf{m}_x + \delta_x$ and $\mathsf{m}^{\star}_y = \mathsf{m}_y + \delta_y$. P_0 then computes $\mathsf{m}^{\star}_z = -\mathsf{m}^{\star}_x \lambda_y - \mathsf{m}^{\star}_y \lambda_x + \lambda_z + 2\gamma_{xy}$. Note that,

$$\begin{aligned} \mathsf{m}^{*}_z &= -\mathsf{m}^{\star}_x \lambda_y - \mathsf{m}^{\star}_y \lambda_x + \lambda_z + 2\gamma_{xy} \\ &= -(\mathsf{m}_x + \delta_x)\lambda_y - (\mathsf{m}_y + \delta_y)\lambda_x + \lambda_z + 2\gamma_{xy} \\ &= (\mathsf{m}_z - \mathsf{m}_x \mathsf{m}_y) - (\delta_x \lambda_y + \delta_y \lambda_x - \gamma_{xy}) \\ &= (\mathsf{m}_z - \mathsf{m}_x \mathsf{m}_y) - \chi \end{aligned}$$

Assuming that P_0 knows $\chi = \delta_x \lambda_y + \delta_y \lambda_x - \gamma_{xy}$, he can then compute $\mathsf{m}^{*}_z + \chi$ and send it back to P_1. Given the knowledge of $\mathsf{m}_x, \mathsf{m}_y$, P_1 can verify the correctness of m_z. The case for a honest P_2 follows similarly. Now we describe how to enable P_0 obtain $\chi = \delta_x \lambda_y + \delta_y \lambda_x - \gamma_{xy}$. First of all, note that revealing χ in clear to P_0 leads to breach of privacy. Because, P_0 knows $\lambda_x, \lambda_y, \gamma_{xy}$ from the offline phase and he receives $\mathsf{m}_x + \delta_x, \mathsf{m}_y + \delta_y$ during the online phase. With this information, P_0 can deduce a relation between m_x and m_y. Hence, we tweak the value of χ to $\delta_x \lambda_y + \delta_y \lambda_x + \delta_z - \gamma_{xy}$ incorporating a random mask δ_z. To generate χ, in the offline phase, parties P_1, P_2 locally sample random elements $\delta_x, \delta_y, \delta_z \in \mathbb{Z}_{2^{\ell}}$, compute a $[\cdot]$-sharing of χ and sends the shares to P_0. Let $[\chi]_{P_i} = \chi_i$ for $i \in \{1, 2\}$. P_0 locally adds the $[\cdot]$-shares and obtains χ. In the above step, a corrupt evaluator can introduce an error while computing the $[\cdot]$-share of χ, affecting the correctness of the protocol. Thus, it is crucial to ensure the correctness of χ computed by P_0.

To summarize, we now have two issues to tackle in the offline phase– (i) as we pointed out earlier, during the offline phase, a corrupt P_0 can incorrectly share γ_{xy}; (ii) a corrupt evaluator can send a wrong $[\cdot]$-share of χ to P_0. Towards tackling these, once P_0 obtains the value χ, parties locally compute $\llbracket \cdot \rrbracket$-shares of values $\mathsf{a} = \delta_x - \lambda_x, \mathsf{b} = \delta_y - \lambda_y$ and $\mathsf{c} = (\delta_z + \delta_x \delta_y) - \chi$ as follows:

$$\begin{aligned} &\llbracket \mathsf{a} \rrbracket_{P_0} = (\lambda_{x,1}, \lambda_{x,2}), && \llbracket \mathsf{b} \rrbracket_{P_0} = (\lambda_{y,1}, \lambda_{y,2}), && \llbracket \mathsf{c} \rrbracket_{P_0} = (\chi_1, \chi_2) \\ &\llbracket \mathsf{a} \rrbracket_{P_1} = (\delta_x, \lambda_{x,1}), && \llbracket \mathsf{b} \rrbracket_{P_1} = (\delta_y, \lambda_{y,1}), && \llbracket \mathsf{c} \rrbracket_{P_1} = (\delta_z + \delta_x \delta_y, \chi_1) \\ &\llbracket \mathsf{a} \rrbracket_{P_2} = (\delta_x, \lambda_{x,2}) && \llbracket \mathsf{b} \rrbracket_{P_2} = (\delta_y, \lambda_{y,2}) && \llbracket \mathsf{c} \rrbracket_{P_2} = (\delta_z + \delta_x \delta_y, \chi_2) \end{aligned}$$

Now $(\llbracket \mathsf{a} \rrbracket, \llbracket \mathsf{b} \rrbracket, \llbracket \mathsf{c} \rrbracket)$ is a multiplication triple ($\mathsf{c} = \mathsf{ab}$) if and only if P_0 shares γ_{xy} correctly (when it is corrupt) and P_0 reconstructs χ correctly (when one of the evaluators is corrupt). This is because,

$$\begin{aligned} \mathsf{ab} &= (\delta_x - \lambda_x)(\delta_y - \lambda_y) = \delta_x \delta_y + \lambda_x \lambda_y - \delta_x \lambda_y - \delta_y \lambda_x \\ &= (\delta_x \delta_y + \delta_z) - (\delta_x \lambda_y + \delta_y \lambda_x + \delta_z - \gamma_{xy}) \\ &= (\delta_x \delta_y + \delta_z) - \chi = \mathsf{c} \end{aligned}$$

We first recall the two standard components needed to check the validity of a multiplication triple– i) a tool for generating $\llbracket \cdot \rrbracket$-shared random multiplication triple and ii) a technique to check securely the product relation of a $\llbracket \cdot \rrbracket$-shared triple, given a valid $\llbracket \cdot \rrbracket$-shared multiplication triple (often referred to as sacrificing technique). With a lot of constructions specifically available for the former one [2, 33], we choose to model it as an ideal functionality $\mathcal{F}_{\mathsf{trip}}$ and use it for our purpose without going into the details. For the latter component, we quickly recall the known protocol.

$\mathcal{F}_{\mathsf{trip}}$, by now a standard functionality [2, 33], allows to generate a set of $\llbracket \cdot \rrbracket$-sharing of multiplication triples over $\mathcal{P}$, each of which, say $(\mathsf{d}, \mathsf{e}, \mathsf{f})$ satisfies the following– i) d, e and f are random and private and ii) $\mathsf{f} = \mathsf{de}$. In Appendix A.1, we present an instantiation of this functionality, namely Π_{trip} (Figure 18), using the techniques proposed by [2, 33].

- Parties locally compute $\llbracket \rho \rrbracket = \llbracket \mathsf{a} \rrbracket - \llbracket \mathsf{d} \rrbracket$ and $\llbracket \sigma \rrbracket = \llbracket \mathsf{b} \rrbracket - \llbracket \mathsf{e} \rrbracket$.
- Parties reconstruct ρ and σ by executing $\Pi^{\mathsf{m}}_{\mathsf{Rec}}(\llbracket \rho \rrbracket, \mathcal{P})$ and $\Pi^{\mathsf{m}}_{\mathsf{Rec}}(\llbracket \sigma \rrbracket, \mathcal{P})$ respectively.
- Parties locally compute $\llbracket \tau \rrbracket = \llbracket \mathsf{c} \rrbracket - \llbracket \mathsf{f} \rrbracket - \sigma \llbracket \mathsf{d} \rrbracket - \rho \llbracket \mathsf{e} \rrbracket - \sigma\rho$.
- Parties reconstruct τ by executing $\Pi^{\mathsf{m}}_{\mathsf{Rec}}(\llbracket \tau \rrbracket, \mathcal{P})$ and output $\perp$, if $\tau \neq 0$.

Figure 7: Protocol Π_{prc} to check product-relation of a triple

Protocol Π_{prc} [20, 33] ('prc' stands for product-relation check) takes a pair of $\llbracket \cdot \rrbracket$-shared random and private triples as input, say $(\mathsf{a}, \mathsf{b}, \mathsf{c})$ and $(\mathsf{d}, \mathsf{e}, \mathsf{f})$, over $\mathbb{Z}_{2^\ell}$, verifies if the former is a multiplication triple or not and nothing beyond, given the latter is a valid triple. The protocol appears in Figure 7 and its properties in Appendix A.2.

By exploiting the definition of $\llbracket \cdot \rrbracket$-sharing, we reduce the cost of Π_{prc} to just 2, instead of 3, instances of $\Pi^{\mathsf{m}}_{\mathsf{Rec}}$, in an amortized sense. Recall that the goal of the third invocation of $\Pi^{\mathsf{m}}_{\mathsf{Rec}}$ inside Π_{prc} is to reconstruct $\llbracket \tau \rrbracket = (\mathsf{m}_\tau, [\lambda_\tau])$, followed by checking if $\tau = 0$. It follows that $\tau = 0$ if and only if $\mathsf{m}_\tau - \lambda_\tau = 0$ implying $\mathsf{m}_\tau = \lambda_\tau$. Hence checking $\tau = 0$ is equivalent to checking if $\mathsf{m}_\tau = \lambda_{\tau,1} + \lambda_{\tau,2}$, which can be translated to three pair-wise checks – (i) P_0 and P_1 can verify if $\mathsf{m}_\tau - \lambda_{\tau,1} \overset{?}{=} \lambda_{\tau,2}$; (ii) P_1 and P_2 can verify if $\mathsf{m}_\tau - \lambda_{\tau,2} \overset{?}{=} \lambda_{\tau,1}$; (iii) P_0 and P_2 can verify if $\mathsf{m}_\tau - \lambda_{\tau,2} \overset{?}{=} \lambda_{\tau,1}$. Parties in $\mathcal{P}$ can mutually perform the above checks for all the instances of Π_{prc} together at the end by exchanging hash of all the required values.

Offline :

- Parties P_0, P_1 locally sample random $\lambda_{z,1}, \gamma_{xy,1} \in \mathbb{Z}_{2^\ell}$, while P_0, P_2 locally sample a random $\lambda_{z,2}$. P_0 locally computes $\gamma_{xy} = \lambda_x \lambda_y$ and sends $\gamma_{xy,2} = \gamma_{xy} - \gamma_{xy,1}$ to P_2.
- Parties execute Π_{trip} to generate triple $(\llbracket \mathsf{d} \rrbracket, \llbracket \mathsf{e} \rrbracket, \llbracket \mathsf{f} \rrbracket)$.
- Parties P_1, P_2 locally sample random $\delta_x, \delta_y, \delta_z \in \mathbb{Z}_{2^\ell}$ and compute $[\delta_z]$ non-interactively.
- P_i for $i \in \{1, 2\}$ computes $[\chi]_{P_i} = \delta_x [\lambda_y]_{P_i} + \delta_y [\lambda_x]_{P_i} + [\delta_z]_{P_i} - [\gamma_{xy}]_{P_i}$ and sends $[\chi]_{P_i}$ to P_0, who computes χ.
- Parties locally compute the $\llbracket \cdot \rrbracket$-shares of the values $\mathsf{a} = \delta_x - \lambda_x$, $\mathsf{b} = \delta_y - \lambda_y$ and $\mathsf{c} = (\delta_z + \delta_x \delta_y) - \chi$.
- Parties execute Π_{prc} on $(\llbracket \mathsf{a} \rrbracket, \llbracket \mathsf{b} \rrbracket, \llbracket \mathsf{c} \rrbracket)$ and $(\llbracket \mathsf{d} \rrbracket, \llbracket \mathsf{e} \rrbracket, \llbracket \mathsf{f} \rrbracket)$.

Online :

- P_i for $i \in \{1, 2\}$ locally computes $[\mathsf{m}_z]_{P_i} = (i-1)\mathsf{m}_x \mathsf{m}_y - \mathsf{m}_x [\lambda_y]_{P_i} - \mathsf{m}_y [\lambda_x]_{P_i} + [\lambda_z]_{P_i} + [\gamma_{xy}]_{P_i}$. P_1, P_2 mutually exchange their shares and reconstruct m_z.
- P_1 sends $\mathsf{m}^\star_x = \mathsf{m}_x + \delta_x, \mathsf{m}^\star_y = \mathsf{m}_y + \delta_y$ to P_0, while P_2 sends $\mathsf{H}(\mathsf{m}^\star_x || \mathsf{m}^\star_y)$ to P_0. P_0 outputs $\perp$, if the received values are inconsistent.
- P_0 computes $\mathsf{m}^\star_z = -\mathsf{m}^\star_x \lambda_y - \mathsf{m}^\star_y \lambda_x + \lambda_z + 2\gamma_{xy} + \chi$ and sends $\mathsf{H}(\mathsf{m}^\star_z)$ to both P_1 and P_2.
- P_i for $i \in \{1, 2\}$ `abort` if $\mathsf{H}(\mathsf{m}^\star_z) \neq \mathsf{H}(\mathsf{m}_z - \mathsf{m}_x \mathsf{m}_y + \delta_z)$.

Figure 8: Protocol $\Pi^{\mathsf{m}}_{\mathsf{Mul}}(\mathsf{w}_x, \mathsf{w}_y, \mathsf{w}_z)$:

With the building blocks set, we present our maliciously-secure multiplication protocol $\Pi^{\mathsf{m}}_{\mathsf{Mul}}$ in Figure 8. Note that the use of hash function improves the amortized cost in the online phase of $\Pi^{\mathsf{m}}_{\mathsf{Mul}}$– (i) P_2 can send a single hash of all the $\mathsf{m}^\star_x$ and $\mathsf{m}^\star_y$ values for all the instances of $\Pi^{\mathsf{m}}_{\mathsf{Mul}}$ to P_0 in the end of the circuit-evaluation; (ii) P_0 can send a single hash of all the $\mathsf{m}^\star_z$ values for all the instances of $\Pi^{\mathsf{m}}_{\mathsf{Mul}}$ to the evaluators at the end of the circuit-evaluation. The former step can be coupled with the communication of $(\mathsf{m}^\star_x, \mathsf{m}^\star_y)$ by P_1 to P_0. Party P_1 sending to P_0 attributes to the increase of the communication cost per multiplication gate in the malicious setting, compared to the semi-honest setting. On the positive note, coupling the above communication for all the multiplication gates together results in a couple of rounds overhead compared to the semi-honest protocol. As a consequence, the latency of the malicious protocol remains as good as the semi-honest protocol.

The correctness of the protocol $\Pi^{\mathsf{m}}_{\mathsf{Mul}}$ is stated in Lemma 4.3.

Lemma 4.3 (Correctness). *In the protocol* $\Pi^{\mathsf{m}}_{\mathsf{Mul}}$, *the following holds: During the offline phase, if* P_0 *is corrupt and* $[\cdot]$*-shares* $\gamma_{xy} \neq \lambda_x \lambda_y$, *then the honest evaluators output* $\perp$. *On the other hand, if one of the evaluators is corrupt and enforces the honest* P_0 *to obtain an incorrect* χ, *then the honest parties output* $\perp$. *During the online step, if one of the evaluators is corrupt and enforces the honest evaluator to obtain an incorrect* m_z, *then the honest evaluator outputs* $\perp$

Proof. For correctness, first consider the case when P_0 is *corrupt* and $[\cdot]$-shares $\gamma_{xy} \neq \lambda_x \lambda_y$ during offline step. Let $\gamma_{xy} = \lambda_x \lambda_y + \Delta$ where Δ is the error introduced by P_0. Now,

$$\begin{aligned} \mathsf{c} &= (\delta_x \delta_y + \delta_z) - (\delta_x \lambda_y + \delta_y \lambda_x + \delta_z - (\gamma_{xy} - \Delta)) \\ &= (\delta_x - \lambda_x)(\delta_y - \lambda_y) - \Delta = \mathsf{a} - \Delta \neq \mathsf{a} \end{aligned}$$

and thus $(\mathsf{a}, \mathsf{b}, \mathsf{c})$ is not a multiplication triple. Then, from Lemma A.1, honest evaluators output $\perp$.

Second, we consider the case when one of the evaluators, say P_1, sends $\chi_1 + \Delta$ to P_0 who reconstructs $\chi' = \chi + \Delta$. Then, the value

$$\begin{aligned} \mathsf{c} &= (\delta_x \delta_y + \delta_z) - \chi' = (\delta_x \delta_y + \delta_z) - (\chi + \Delta) \\ &= (\delta_x \delta_y + \delta_z) - (\delta_x \lambda_y + \delta_y \lambda_x + \delta_z - \gamma_{xy}) - \Delta \\ &= (\delta_x - \lambda_x)(\delta_y - \lambda_y) - \Delta = \mathsf{a} - \Delta \neq \mathsf{a} \end{aligned}$$

and hence $(\mathsf{a}, \mathsf{b}, \mathsf{c})$ is not a multiplication triple. Thus, similar to the previous case, honest parties output $\perp$.

Lastly, we consider the case, when one of the evaluators, say P_1, is *corrupt* and during online step sends $[\mathsf{m}_z]_{P_1} + \Delta$ for some non-zero Δ during the reconstruction, so that P_2 reconstructs $\mathsf{m}_z + \Delta$, instead of m_z. In this case, the honest P_0 would have $\chi = \delta_x \lambda_y + \delta_y \lambda_x + \delta_z - \gamma_{xy}$ from offline step. Moreover, during online step, P_0 correctly learns $\mathsf{m}^\star_x = \mathsf{m}_x + \delta_x$ and $\mathsf{m}^\star_y = \mathsf{m}_y + \delta_y$. Furthermore, $\gamma_{xy} = \lambda_x \lambda_y$ holds. It then follows that $\mathsf{m}^\star_z$ received by P_2 from P_0 will be different from $\mathsf{m}_z + \Delta - \mathsf{m}_x \mathsf{m}_y + \delta_z$ locally computed by P_2 and hence P_2 will output $\perp$. □

The informal privacy argument of $\Pi^{\mathsf{m}}_{\mathsf{Mul}}$ is as follows. We first consider the case when P_0 is corrupt, where $\llbracket x \rrbracket$, $\llbracket y \rrbracket$ and $\llbracket z \rrbracket$ are defined by the shares of P_1, P_2. The privacy for this case follows from the fact that P_0 does not learn anything about m_x, m_y and m_z, neither during the offline step, nor during the online step. Clearly, the communication between P_0 and P_1, P_2 during offline step is independent of $\mathsf{m}_x, \mathsf{m}_y$ and m_z. Moreover, the value χ reveals nothing about δ_x and δ_x since it is padded with a random δ_z. During the online step, P_0 learns $\mathsf{m}^\star_x$ and $\mathsf{m}^\star_y$, which reveals nothing about $\mathsf{m}_x, \mathsf{m}_y$, as δ_x and δ_y remains random and private for P_0. We next consider the case when one of the evaluators, say P_1 is corrupt. The privacy for this case follows from the fact that $\lambda_x, \lambda_y, \lambda_z$ and γ_{xy} remains private from the view point of P_1. On the other hand, no additional information is revealed from $\mathsf{m}_z^\star$ during the online step, as adversary will already know that $\mathsf{m}_z^\star = \mathsf{m}_z - \mathsf{m}_x\mathsf{m}_y + \delta_z$.

We present a detailed security proof for our 3PC protocol $\Pi_{\mathsf{3pc}}^{\mathsf{m}}$ in Appendix C, showing that it emulates the functionality $\mathcal{F}_{\mathsf{3pc}}^{\mathsf{Abort}}$ as given in Figure 9.

$\mathcal{F}_{\mathsf{3pc}}^{\mathsf{Abort}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$ and is parameterized by a 3-ary function f, represented by a publicly known arithmetic circuit ckt over $\mathbb{Z}_{2^\ell}$.

Input: Upon receiving the input $x_, \ldots, x_\mathsf{I}$ from the respective parties in $\mathcal{P}$, do the following: if $(\mathsf{Input}, *)$ message was received from P_j corresponding to x_j, then ignore. Otherwise record $x'_j = x_j$ internally. If $x'_j \neq \mathbb{Z}_{2^\ell}$, consider $x'_j = \mathtt{abort}$.

Output to adversary: If there exists $j \in \{1, \ldots, \mathsf{I}\}$ such that $x'_j = \mathtt{abort}$, send (Output, $\perp$) to all the parties. Else, send (Output, $(y_1, \ldots, y_\mathsf{O})$) to the adversary $\mathcal{S}$, where $(y_1, \ldots, y_\mathsf{O}) = f(x'_1, \ldots, x'_\mathsf{I})$.

Output to selected honest parties: Receive (select, $\{I\}$) from adversary $\mathcal{S}$, where $\{I\}$ denotes a subset of the honest parties. If an honest party belongs to I, send (Output, $\perp$), else send (Output, $(y_1, \ldots, y_\mathsf{O})$).

Figure 9: Functionality $\mathcal{F}_{\mathsf{3pc}}^{\mathsf{Abort}}$

We now prove the communication complexity of protocol $\Pi_{\mathsf{3pc}}^{\mathsf{m}}$ below.

Theorem 4.4. *Protocol $\Pi_{\mathsf{3pc}}^{\mathsf{m}}$ has the following complexities.*

Input-sharing Stage: *It is non-interactive during the offline phase and requires one round and an amortized communication of at most* $2\mathsf{I}$ *ring elements during the online phase.*

Circuit-evaluation Stage: *Assuming $\mathsf{M} = 2^{20}$ and a statistical security parameter $s = 40$, in the amortized sense, evaluating each multiplication gate requires* 4 *rounds and communication of* 21 *ring elements in the offline phase, while the online phase needs* 1 *round with a communication of* 4 *ring elements.*

Output-reconstruction Stage: *It requires one round and an amortized communication of* $3\mathsf{O}$ *ring elements.*

Proof. The complexity for the Input-sharing Stage follows from Theorem 4.2 and the fact that the cost of $\Pi_{\mathsf{Sh}}^{\mathsf{m}}$ reduces to that of $\Pi_{\mathsf{Sh}}^{\mathsf{s}}$ in an amortized sense due to the use of the hash function. During the circuit-evaluation stage, the addition gates need no interaction, as usual. For a multiplication gate, the offline communication include– (i) sending a share of $[\gamma_{xy}]$ to P_2; (ii) the amortized cost of generating one shared triple via $\mathcal{F}_{\mathsf{trip}}$; (iii) the cost of reconstructing χ towards P_0 and lastly (iv) the cost of one Π_{prc}. The first one requires one round and communication of one element. The second one requires 3 rounds and an amortized communication of $9B - 6$ ring elements, where $B = \frac{s}{\log_2 \mathsf{M}}$, using the techniques of [2] (see Appendix A.1), where s is the statistical parameter dictating the performance of underlying cut-and-choose technique. Assuming $\mathsf{M} = 2^{20}$, $s = 40$, this ensures that generating a single multiplication triple require 3 rounds and an amortized communication of 12 ring elements. The third one requires one round and communication of two elements. The fourth and last one requires one round and an amortized communication of 6 elements as part of the two underlying instances of $\Pi_{\mathsf{Rec}}^{\mathsf{m}}$. This sums up to communication of 21 elements per multiplication gate.

The total number of rounds for evaluating the multiplication gates during the offline phase turns to be 4 as follows: P_0 can send the share of $[\gamma_{xy}]$ to P_2 and in parallel, the parties can start generating a shared triple via $\mathcal{F}_{\mathsf{trip}}$; while the former requires one round, the latter requires three rounds. Once the share of $[\gamma_{xy}]$ is available with P_2, party P_1 and P_2 can reconstruct χ towards P_0, requiring one round, which overlaps with the second round of the instantiation of $\mathcal{F}_{\mathsf{trip}}$. Once the third round of the instantiation of $\mathcal{F}_{\mathsf{trip}}$ is over, the parties execute the instance of Π_{prc}, which requires one additional round.

During the online phase, evaluating a multiplication gate requires one round and communication of two elements for the reconstruction of m_z. Also, P_1 needs to send $\mathsf{m}_x^\star$ and $\mathsf{m}_y^\star$ values to P_0 per instance, which requires just one round for all the multiplication gates and communication of 2 ring elements per gate. Summing up, evaluating a multiplication gate in the online phase requires an amortized round complexity of 1 and communication of 4 elements.

The output-reconstruction phase requires one round and an amortized communication of $3\mathsf{O}$ elements, as the cost of $\Pi_{\mathsf{Rec}}^{\mathsf{m}}$ reduces to $\Pi_{\mathsf{Rec}}^{\mathsf{s}}$ in an amortized sense due to the use of the hash function. □

4.3 Achieving Fairness

We boost the security of $\Pi_{\mathsf{3pc}}^{\mathsf{m}}$ from abort to fairness via a fair reconstruction protocol Π_{fRec} that substitutes $\Pi_{\mathsf{Rec}}^{\mathsf{m}}$ for the reconstruction of the circuit outputs. To fairly reconstruct $[\![y]\!]$, the pair $\{P_0, P_1\}$ commit their common share $\lambda_{y,1}$ to P_2 and likewise the pair P_0, P_2 commit their common share $\lambda_{y,2}$ to P_1 in the offline phase. In the online phase, the evaluator pair $\{P_1, P_2\}$ commit their common information m_y to P_0. In all the three cases, shared random (PRF) key is used to derive the randomness for preparing the commitments. As a result, each pair should prepare an identical commitment ideally. The recipient in each case can abort when the received commitments do not match. If no abort happens, P_0 signals P_1 and P_2 to start opening the commitments which will help the parties to get their missing share and reconstruct the output. As there is at least one honest party in each pair of (P_0, P_1), (P_0, P_2) and (P_1, P_2), the opened value of the honest party from each pair is used for reconstructing y. Lastly, if the protocol aborts before, then none receive the output maintaining fairness.

A very subtle issue arises in the above protocol in the absence of broadcast channel. A corrupt P_0 can send distinct signals to P_1 and P_2 (abort to one and continue to the other), breaching unanimity in the end. To settle this, we make the pair $\{P_0, P_1\}$ to commit a value r_1 chosen from their common random source to P_2 and likewise the pair P_0, P_2 to commit a common value r_2 to P_1 in the offline phase. In the online phase, when P_0 signals abort to P_1, it sends the opening of r_2 along. Similarly, when P_0 signals abort to P_2, it sends the opening of r_1 along. Now an evaluator, say P_1 on receiving the abort can convince P_2 that it has indeed received abort from P_0, using r_2 as the *proof of origin* for the abort message. Because the only way P_1 can secure r_2 is via P_0. Put differently, a corrupt P_1 cannot simply claim that it received abort from P_0, while P_0 is really instructed to continue. A single pair of (r_1, r_2) can be used as a proof of origin for multiple instances of reconstruction running in parallel. Protocol $\Pi_{\mathsf{fRec}}(\llbracket y \rrbracket, \mathcal{P})$ is formally presented in Figure 10.

Offline:

- Parties P_0, P_1 locally sample a random $r_1 \in \mathbb{Z}_{2^\ell}$, prepare and send commitments of $\lambda_{y,1}$ and r_1 to P_2. Similarly, parties P_0, P_2 sample r_2 and send commitments of $\lambda_{y,2}$ and r_2 to P_1 The randomness needed for both commitments are sampled from the PRF key-setup.
- P_1 (resp. P_2) aborts if the received commitments mismatch.

Online:

- P_1, P_2 compute a commitment of m_y using randomness sampled from their PRF key-setup and send it to P_0.
- If the commitments do not match, P_0 sends $(\mathtt{abort}, o_1)$ to P_2, while he sends $(\mathtt{abort}, o_2)$ to P_1 and aborts, where o_i denotes opening information for the commitment of r_i. Else P_0 sends $\mathtt{continue}$ to both P_1 and P_2.
- P_1, P_2 exchange the messages received from P_0.
- P_1 aborts if he receives either (i) $(\mathtt{abort}, o_2)$ from P_0 and o_2 opens the commitment of r_2 or (ii) $(\mathtt{abort}, o_1)$ from P_2 and o_1 is the correct opening information of r_1. The case for P_2 is similar to that of P_1
- If no abort happens, parties obtain their missing share of a as follows:
  - P_0, P_1 open $\lambda_{y,1}$ towards P_2.
  - P_0, P_2 open $\lambda_{y,2}$ towards P_1.
  - P_1, P_2 open m_y towards P_0.
- Parties reconstruct the value y using missing share that matches with the agreed upon commitment.

Figure 10: Protocol $\Pi_{\mathsf{fRec}}(\llbracket y \rrbracket, \mathcal{P})$

The complexity of Π_{fRec} is stated below. The commitment can be implemented via a hash function $\mathcal{H}()$ e.g. $(c, o) = (\mathcal{H}(x||r), x||r) = Com(x; r)$, whose security can be proved in the random-oracle model (ROM) [40]. We do not include the cost of commitment and opening of r_1 and r_2, as they will get amortized away over many instances of Π_{fRec}.

Lemma 4.5. *Protocol* Π_{fRec} *requires one round and an amortized communication of* 4 *commitments in the offline phase.* Π_{fRec} *requires four rounds and an amortized communication of at most* 2 *commitments and* 6 *opening of commitments in the online phase.*

5 PRIVACY PRESERVING MACHINE LEARNING

We apply our techniques for 3PC developed so far to the regime of ML prediction for a range of prediction functions– linear regression, logistic regression, linear SVM classification, and linear SVM regression.

5.1 The Model

A model-owner M, holding a vector of *trained model parameters*, would like to offer ML prediction service to a client C holding a *query vector* as per certain prediction function. In the server-aided setting, M and C outsource their respective inputs in shared fashion to three untrusted but non-colluding servers $\{P_0, P_1, P_2\}$ who perform the computation in shared fashion via techniques developed for our 3PC protocols and reconstruct the output to the client alone. The client learns the output and nothing beyond. We assume a *computationally bounded* adversary $\mathcal{A}$, who can corrupt at most one of the servers $\{P_0, P_1, P_2\}$ and one of $\{\mathsf{M}, \mathsf{C}\}$ in either semi-honest or malicious fashion. The security against an $\mathcal{A}$ corrupting parties in both sets $\{P_0, P_1, P_2\}$ and $\{\mathsf{M}, \mathsf{C}\}$ semi-honestly and likewise maliciously reduces to the semi-honest and respectively malicious security of our 3PC protocols. Adversarial machine learning [56, 57, 62] that includes attacks launched by a client to learn the model using its outputs, lies outside the scope of this work. Following the existing literature on server-aided secure ML [34, 39, 53, 54], we do not count the cost of M and C making their inputs available in secret-shared form amongst the servers and the cost of reconstructing the output to the client. We assume that the inputs are available to the servers in a secret-shared form and focus on efficient computation of a prediction function on the shared inputs to obtain shared outputs.

5.2 Notations

For a vector $\vec{\mathsf{a}}$, a_i denotes the i^{th} element in the vector. For two vectors $\vec{\mathsf{a}}$ and $\vec{\mathsf{b}}$ of length d, their scalar dot product is $\vec{\mathsf{a}} \odot \vec{\mathsf{b}} = \sum_{i=1}^{d} \mathsf{a}_i \mathsf{b}_i$. The definitions of $[\cdot]$-sharing and $\llbracket \cdot \rrbracket$-sharing are extended in a natural way for the vectors. A vector $\vec{\mathsf{a}} = (\mathsf{a}_1, \ldots, \mathsf{a}_d)$ is said to be $[\cdot]$-shared, denoted as $[\vec{\mathsf{a}}]$, if each a_i is $[\cdot]$-shared. We use the notations $[\vec{\mathsf{a}}]_{P_1}$ and $[\vec{\mathsf{a}}]_{P_2}$ to denote the vector of $[\cdot]$-shares of P_1 and P_2 respectively, corresponding to $[\vec{\mathsf{a}}]$. Similarly, a vector $\vec{\mathsf{a}} = (\mathsf{a}_1, \ldots, \mathsf{a}_d)$ is said to be $\llbracket \cdot \rrbracket$-shared, denoted as $\llbracket \vec{\mathsf{a}} \rrbracket$, if each a_i is $\llbracket \cdot \rrbracket$-shared. We use the notation $\overrightarrow{\lambda_{\mathsf{a}}}$ and $\overrightarrow{\mathsf{m}_{\mathsf{a}}}$ to denote the vector of masks and vector of masked values corresponding to $\llbracket \vec{\mathsf{a}} \rrbracket$. Finally, we note that the linearity of $[\cdot]$ and $\llbracket \cdot \rrbracket$-sharings hold even over vectors.

5.3 Fixed Point Arithmetic

We represent decimal values as ℓ-bit integers in signed 2's complement representation with the most significant bit representing the sign bit and x least significant bits representing the fractional part. For our purpose, we choose $\ell = 64$ and $x = 13$, keeping $i = \ell - x - 1 = 50$ bits for the integral part of the value. We then treat these ℓ-bit strings as elements of $\mathbb{Z}_{2^\ell}$. A product of two numbers from this domain would lead to expanding x to 26 and yet leaving 37 bits for the integer part which keeps the accuracy unaffected. As the prediction functions of our concern require multiplication of depth one, the prediction function output values have the above format. Noticeably, since SecureML [52] and ABY3 [50] need to do multiplication in sequence multiple times for the task of training, they propose a new method of truncation to maintain a representation invariant across the sequential products. This is necessary to keep accuracy in check in their works.

5.4 Protocols for ML

We begin with some of the building blocks required.

Secure Dot Product. Given the $\llbracket \cdot \rrbracket$-shares of d element vectors $\vec{\mathsf{p}}$ and $\vec{\mathsf{q}}$, the goal of a secure dot-product is to compute $\llbracket \cdot \rrbracket$-sharing of $\vec{\mathsf{p}} \odot \vec{\mathsf{q}}$. Using Π_{Mul} naively to compute the product of each component would require a communication complexity that is linearly dependent on d in both the offline and online phase. In the *semi-honest* setting, following the literature [17, 23, 29, 50, 59], we make the communication of Π_{dp} independent of d as follows: during the offline phase, P_0 $[\cdot]$-shares *only* $\gamma_{\mathbf{pq}} = \overrightarrow{\lambda_p} \odot \overrightarrow{\lambda_q}$, instead of each individual $\lambda_{\mathsf{p}_i}\lambda_{\mathsf{q}_i}$. During the online phase, instead of reconstructing each $\mathsf{m}_{\mathsf{p}_i\mathsf{q}_i}$ separately to compute m_{u} where $\mathsf{u} = \vec{\mathbf{p}} \odot \vec{\mathbf{q}}$, the evaluators P_1, P_2 locally compute $[\mathsf{m}_{\mathsf{u}}]$ and then reconstruct m_{u}. We call the resultant protocol as $\Pi^{\mathsf{s}}_{\mathsf{dp}}$ (Figure 11).

Offline : P_0, P_1 sample random $\lambda_{\mathsf{u},1}, \gamma_{\mathbf{pq},1} \in \mathbb{Z}_{2^\ell}$, while P_0, P_2 sample random $\lambda_{\mathsf{u},2} \in \mathbb{Z}_{2^\ell}$. P_0 locally computes $\gamma_{\mathbf{pq}} = \overrightarrow{\lambda_p} \odot \overrightarrow{\lambda_q}$, sets $\gamma_{\mathbf{pq},2} = \gamma_{\mathbf{pq}} - \gamma_{\mathbf{pq},1}$ and sends $\gamma_{\mathbf{pq},2}$ to P_2.

Online :

- P_i for $i \in \{1, 2\}$ locally computes $[\mathsf{m}_{\mathsf{u}}]_{P_i} = \sum_{j=1}^{d} \left((i-1)\mathsf{m}_{\mathsf{p}_j}\mathsf{m}_{\mathsf{q}_j} - \mathsf{m}_{\mathsf{p}_j}[\lambda_{\mathsf{p}_j}]_{P_i} - \mathsf{m}_{\mathsf{q}_j}[\lambda_{\mathsf{p}_j}]_{P_i}\right) + [\gamma_{\mathbf{pq}}]_{P_i} + [\lambda_{\mathsf{u}}]_{P_i}$.
- P_1 and P_2 mutually exchange $[\mathsf{m}_{\mathsf{u}}]$ to reconstruct m_{u}.

Figure 11: Protocol $\Pi^{\mathsf{s}}_{\mathsf{dp}}$

Due to the extra checks we introduce for tolerating a maliciously adversary in our multiplication protocol, the optimization done above for semi-honest protocol in the offline phase does not work. As a result, we resort to d invocations of our multiplication protocol. Invoking Theorem 4.4, our protocol for dot product then needs to communicate $21d$ ring elements in the offline phase. However, we improve the online cost from $4d$ (as per Theorem 4.4) to $2d + 2$ as follows. The parties execute the online stage of protocol $\Pi^{\mathsf{s}}_{\mathsf{dp}}$. In parallel, P_1 sends $\mathsf{m}^\star_{\mathsf{p}_i}, \mathsf{m}^\star_{\mathsf{q}_i}$ for $i \in \{1, \ldots, d\}$ to P_0, while P_2 sends the corresponding hash to P_0. Instead of sending $\mathsf{m}^\star_{\mathsf{p}_i\mathsf{q}_i}$ for each $\mathsf{p}_i\mathsf{q}_i$, P_0 can "combine" all the $\mathsf{m}^\star_{\mathsf{p}_i\mathsf{q}_i}$ values and send a single $\mathsf{m}^\star_{\mathsf{u}}$ to P_1, P_2 for verification. In detail, P_0 can compute $\mathsf{m}^\star_{\mathsf{u}} = \sum_{j=1}^{d} \mathsf{m}^\star_{\mathsf{u}_j}$ and send a hash of the same to both P_1 and P_2, who can then cross check with a hash of $\mathsf{m}_{\mathsf{u}} - \sum_{j=1}^{d}(\mathsf{m}_{\mathsf{p}_j}\mathsf{m}_{\mathsf{q}_j} - \delta_{\mathsf{u}_j})$. We call the resultant protocol as $\Pi^{\mathsf{m}}_{\mathsf{dp}}$ and the communication complexity is given below.

Lemma 5.1. $\Pi^{\mathsf{s}}_{\mathsf{dp}}$ *requires communication of one ring element during the offline step and communication of two ring elements in online step.* $\Pi^{\mathsf{m}}_{\mathsf{dp}}$ *requires communication of* $21d$ *ring elements during the offline step and communication of* $2d + 2$ *ring elements in online step.*

Secure Comparison. Comparing two arithmetic values is one of the major hurdles in realizing efficient secure ML algorithms. Given arithmetic shares $\llbracket \mathsf{u} \rrbracket, \llbracket \mathsf{v} \rrbracket$, parties wish to check whether $\mathsf{u} < \mathsf{v}$, which is equivalent to checking if $\mathsf{a} < 0$, where $\mathsf{a} = \mathsf{u} - \mathsf{v}$. In the fixed-point arithmetic representation, this task can be accomplished by checking the $\mathsf{msb}(\mathsf{a})$. Thus the goal reduces to generating boolean-shares of $\mathsf{msb}(\mathsf{a})$ given the arithmetic-sharing $\llbracket \mathsf{a} \rrbracket$. Here, we exploit the asymmetry in our secret sharing scheme and forgo expensive primitives such as garbled circuits or parallel prefix adders, which are used in SecureML [52] and ABY3 [50].

Offline: P_1, P_2 together sample random $\mathsf{r}, \mathsf{r}' \in \mathbb{Z}_{2^\ell}$ and set $\mathsf{p} = \mathsf{msb}(\mathsf{r})$. Parties non-interactively generate boolean shares of p as $\llbracket \mathsf{p} \rrbracket^{\mathbf{B}}_{P_0} = (0, 0)$, $\llbracket \mathsf{p} \rrbracket_{P_1} = (\mathsf{p}, 0)$ and $\llbracket \mathsf{p} \rrbracket_{P_2} = (\mathsf{p}, 0)$.

Online: P_1 sets $[\mathsf{a}]_{P_1} = \mathsf{m}_{\mathsf{a}} - \lambda_{\mathsf{a},1}$, P_2 sets $[\mathsf{a}]_{P_2} = -\lambda_{\mathsf{a},2}$.

- P_1 sends $[\mathsf{ra}]_{P_1} = \mathsf{r}[\mathsf{a}]_{P_1} + \mathsf{r}'$ to P_0, while P_2 sends $[\mathsf{ra}]_{P_2} = \mathsf{r}[\mathsf{a}]_{P_2} - \mathsf{r}'$ to P_0, who adds them to obtain ra.
- P_0 executes $\Pi^{\mathsf{s}}_{\mathsf{Sh}}(P_0, \mathsf{q})$ to generate $\llbracket \mathsf{q} \rrbracket$ where $\mathsf{q} = \mathsf{msb}(\mathsf{ra})$.
- Parties locally compute $\llbracket \mathsf{msb}(\mathsf{a}) \rrbracket^{\mathbf{B}} = \llbracket \mathsf{p} \rrbracket^{\mathbf{B}} \bigoplus \llbracket \mathsf{q} \rrbracket^{\mathbf{B}}$.

Figure 12: Protocol $\Pi^{\mathsf{s}}_{\mathsf{BitExt}}(\llbracket \mathsf{a} \rrbracket, \mathcal{P})$

We observe that in the signed 2's complement representation, if we multiply two values, then the sign of the result is the sign of the underlying product. Consequently, if a value a is multiplied with r, then $\mathsf{sign}(\mathsf{a} \cdot \mathsf{r}) = \mathsf{sign}(\mathsf{a}) \oplus \mathsf{sign}(\mathsf{r})$. On a high level, the semi-honest protocol (Figure 12) proceeds as follows: P_1, P_2 reconstruct ra towards P_0 where a is the value we need the sign of, and r is a random value sampled by P_1, P_2 together. P_0 in turn boolean-shares the sign of ra. Parties retrieve the sign of a by XORing the sign of ra with the sign of r. For the sake of clarity, we use the superscript **B** to denote the boolean shares.

Offline: P_1, P_2 sample random $\mathsf{r}_1 \in \mathbb{Z}_{2^\ell}$ and set $\mathsf{p}_1 = \mathsf{msb}(\mathsf{r}_1)$ while P_0, P_2 sample random r_2 and set $\mathsf{p}_2 = \mathsf{msb}(\mathsf{r}_2)$.

- Parties non-interactively generate $\llbracket \cdot \rrbracket$-shares of r_1 as $\llbracket \mathsf{r}_1 \rrbracket_{P_0} = (0, 0)$, $\llbracket \mathsf{r}_1 \rrbracket_{P_1} = (\mathsf{r}_1, 0)$ and $\llbracket \mathsf{r}_1 \rrbracket_{P_2} = (\mathsf{r}_1, 0)$.
- Parties non-interactively generate $\llbracket \cdot \rrbracket$-shares of r_2 as $\llbracket \mathsf{r}_2 \rrbracket_{P_0} = (0, -\mathsf{r}_2)$, $\llbracket \mathsf{r}_2 \rrbracket_{P_1} = (0, 0)$ and $\llbracket \mathsf{r}_2 \rrbracket_{P_2} = (0, -\mathsf{r}_2)$.
- Parties execute $\Pi^{\mathsf{m}}_{\mathsf{Mul}}$ on r_1 and r_2 to generate $\llbracket \mathsf{r} \rrbracket = \llbracket \mathsf{r}_1\mathsf{r}_2 \rrbracket$.
- Parties non-interactively generate boolean shares of p_1 as $\llbracket \mathsf{p}_1 \rrbracket^{\mathbf{B}}_{P_0} = (0, 0)$, $\llbracket \mathsf{p}_1 \rrbracket^{\mathbf{B}}_{P_1} = (\mathsf{p}_1, 0)$ and $\llbracket \mathsf{p}_1 \rrbracket^{\mathbf{B}}_{P_2} = (\mathsf{p}_1, 0)$.
- Parties non-interactively generate boolean shares of p_2 as $\llbracket \mathsf{p}_2 \rrbracket^{\mathbf{B}}_{P_0} = (0, \mathsf{p}_2)$, $\llbracket \mathsf{p}_2 \rrbracket^{\mathbf{B}}_{P_1} = (0, 0)$ and $\llbracket \mathsf{p}_2 \rrbracket^{\mathbf{B}}_{P_2} = (0, \mathsf{p}_2)$.
- Parties locally compute $\llbracket \mathsf{p} \rrbracket^{\mathbf{B}} = \llbracket \mathsf{p}_1 \rrbracket^{\mathbf{B}} \oplus \llbracket \mathsf{p}_2 \rrbracket^{\mathbf{B}}$

Online:

- Parties execute $\Pi^{\mathsf{m}}_{\mathsf{Mul}}$ on r and a to generate ra followed by executing $\Pi^{\mathsf{m}}_{\mathsf{Rec}}(P_0, \mathsf{ra})$ and $\Pi^{\mathsf{m}}_{\mathsf{Rec}}(P_1, \mathsf{ra})$ to enable P_0, P_1 obtain ra.
- P_1 execute $\Pi^{\mathsf{m}}_{\mathsf{Sh}}(P_1, \mathsf{q})$ to generate $\llbracket \mathsf{q} \rrbracket^{\mathbf{B}}$ where $\mathsf{q} = \mathsf{msb}(\mathsf{ra})$. In parallel, P_0 locally computes m_{q} and sends $\mathsf{H}(\mathsf{m}_{\mathsf{q}})$ to P_2, who `abort` if the value mismatch with one received from P_1.
- Parties locally compute $\llbracket \mathsf{msb}(\mathsf{a}) \rrbracket^{\mathbf{B}} = \llbracket \mathsf{p} \rrbracket^{\mathbf{B}} \bigoplus \llbracket \mathsf{q} \rrbracket^{\mathbf{B}}$.

Figure 13: Protocol $\Pi^{\mathsf{m}}_{\mathsf{BitExt}}(\llbracket \mathsf{a} \rrbracket, \mathcal{P})$

For the malicious case, we cannot solely rely on P_0 to generate $\llbracket \mathsf{msb}(\mathsf{ra}) \rrbracket^{\mathbf{B}}$. The modified protocol for the malicious setting appears in Figure 13. The correctness for the malicious version appears in Appendix D. The communication and round complexity are given below.

Lemma 5.2. $\Pi^{\mathsf{s}}_{\mathsf{BitExt}}$ *requires no communication during the offline step, while it requires two rounds and communication of* $2\ell + 2$ *bits during the online step.* $\Pi^{\mathsf{m}}_{\mathsf{BitExt}}$ *requires four rounds and an amortized communication of* 46ℓ *bits during the offline step, while it requires three rounds and an amortized communication of* $6\ell + 1$ *bits during the online step.*

5.5 ML Prediction Functions and Abstractions

We consider four prediction functions – two from regression category with real or continuous value as the output and two from classification type with a bit as the output. The inputs to the functions are vectors of decimal values. We provide a high-level overview of the functions below and more details can be found in [49, 50, 52].

- **Linear Regression**: Model $\mathtt{M}$ owns a d-dimensional model parameter $\vec{\mathrm{w}}$ and a bias b, while client $\mathtt{C}$ has a d-dimensional query vector $\vec{\mathrm{z}}$. $\mathtt{C}$ obtains $f_{\mathsf{linr}}\big((\vec{\mathrm{w}}, \mathrm{b}), \vec{\mathrm{z}}\big) = \vec{\mathrm{w}} \odot \vec{\mathrm{z}} + \mathrm{b}$, where $\vec{\mathrm{w}} \odot \vec{\mathrm{z}}$ denotes the dot-product of $\vec{\mathrm{w}}$ and $\vec{\mathrm{z}}$.
- **SVM Regression**: $\mathtt{M}$ holds $\{\alpha_j, y_j\}_{j=1}^{k}$, d-dimensional support vectors $\{\vec{\mathrm{x}}_j\}_{j=1}^{k}$ and bias b, while P_c holds a d-dimensional query $\vec{\mathrm{z}}$. $\mathtt{C}$ obtains $f_{\mathsf{svmr}}\big((\{\alpha_j, y_j, \vec{\mathrm{x}}_j\}_{j=1}^{k}), \vec{\mathrm{z}}\big) = \sum_{j=1}^{k} \alpha_j y_j (\vec{\mathrm{x}}_j \odot \vec{\mathrm{z}}) + \mathrm{b}$.
- **Logistic Regression**: The inputs of $\mathtt{M}$ and $\mathtt{C}$ are similar to linear regression. $\mathtt{M}$ needs to provide an additional input t in the range $[0, 1]$. $\mathtt{C}$ obtains $f_{\mathsf{logr}}\big((\vec{\mathrm{w}}, \mathrm{b}, t), \vec{\mathrm{z}}\big) = \mathsf{sign}((\vec{\mathrm{w}} \odot \vec{\mathrm{z}} + \mathrm{b}) - \ln(\frac{t}{1-t}))$, where $\mathsf{sign}(\cdot)$ returns the sign bit of its argument. Since the values are represented in 2's complement representation, $\mathsf{sign}()$ returns the most significant bit (MSB) of its argument.
- **SVM Classification**: The inputs of $\mathtt{M}$ and $\mathtt{C}$ remain the same as in SVM regression. But the output to $\mathtt{C}$ changes to $f_{\mathsf{svmc}}\big((\{\alpha_j, y_j, \vec{\mathrm{x}}_j\}_{j=1}^{k}), \vec{\mathrm{z}}\big) = \mathsf{sign}(\sum_{j=1}^{k} \alpha_j y_j (\vec{\mathrm{x}}_j \odot \vec{\mathrm{z}}) + \mathrm{b})$.

6 IMPLEMENTATION AND BENCHMARKING

In this section, we provide empirical results for our 3PC and secure prediction protocols. We start with the description of the setup environment– software, hardware, and network.

Network & Hardware Details. We have experimented both in a LAN (local) and a WAN (cloud) setting. In the LAN setting, our machines (P_0, P_1, P_2) are equipped with Intel Core i7-7790 CPU with 3.6 GHz processor speed and 32 GB RAM. In the WAN setting, we use Microsoft Azure Cloud Services with machines located in South East Asia (P_0), North Europe (P_1) and North Central US (P_2). We used Standard E4s v3 instances, where machines are equipped with 32 GB RAM and 4 vcpus. Every pair of parties are connected by bi-directional communication channels in both the LAN and WAN setting, facilitating simultaneous data exchange between them. We consider a LAN with 1Gbps and a WAN with 25Mbps channel bandwidth. We measured the average round-trip time (rtt) for communicating 1 KB of data between P_0-P_1, P_1-P_2 and P_0-P_2 in both the setting. In the LAN setting, the average rtt turned out to be $0.47ms$. In the WAN setting, the rtt between P_0-P_1, P_1-P_2 and P_0-P_2 are $201.928ms$, $81.736ms$ and $229.792ms$ respectively. We use a TCP-IP connection between each set of parties.

Software Details. Our code follows the standards of C++11. We implemented our protocols in both semi-honest and malicious setting, using ENCRYPTO library [24]. We used SHA-256 to instantiate the hash function. We use multi-threading to facilitate efficient computation and communication among the parties. For benchmarking, we use the AES-128 [1] circuit. For ML prediction, since the code for ABY3 [50] was not available, we implemented their protocols in our framework for benchmarking. We run each experiment 20 times and report the average for our measurements.

Parameters for Comparison. All our constructions are compared against their closest competitors which are implemented in our environment for a fair comparison. We consider five parameters for comparison– latency (calculated as the maximum of the runtime of the parties or servers in case of secure prediction) in both LAN and WAN, total communication complexity and throughput of the *online* phase over LAN and WAN. For 3PC over LAN, the throughput is calculated as the number of AES circuits that can be computed per second. As an AES evaluation takes more than a second in WAN, we change the notion of throughput in WAN to the number of AND gates that can be computed per second. For the case of secure prediction, throughput is taken as a number of queries that can be processed per second in LAN and per minute in WAN. For simplicity, we use *online throughput* to denote the throughput of the online phase. The discrepancy across the benchmarking parameters for LAN and WAN comes from the difference in rtt (order of microseconds for LAN and milliseconds for WAN).

6.1 Experimental Results

6.1.1 Results for 3PC. In Table 3, we compare our 3PCs over the boolean ring ($\mathbb{Z}_2$) both in semi-honest and malicious setting with their closest competitors [4] and [2] respectively in terms of latency and communication.

Protocol	Work	LAN Latency (ms)		WAN Latency (s)		Communication (KB)	
		Offline	Online	Offline	Online	Offline	Online
Semi-honest	[4]	0	254.8	0	8.96	0	1.99
	This	0.48	254.8	0.23	3.19	0.66	1.33
Malicious	[2]	1.44	260.72	0.71	9.42	8.06	6.06
	This	2.37	248.38	0.88	3.57	10.72	2.69

Table 3: Comparison of Our 3PC with [4] and [2]

Note that Table 3 does not include the runtime and communication for input-sharing and output-reconstruction phases. We provide the runtime and communication of our protocol for the aforementioned phases in Table 4. For benchmarking, we let P_0 own 48 out of the 128 input wires of AES while P_1 and P_2 own 40 wires each. The table provides benchmarking for the fair reconstruction phase as well, which sees an increase in the latency for the online phase due to increased round complexity.

Phase	Protocol	LAN Latency (ms)		WAN Latency (s)		Comm. (KB)	
		Offline	Online	Offline	Online	Offline	Online
Input Sharing	Semi-honest	0	0.47	0	0.23	0.01	0.02
	Malicious	0.47		0.23		0.02	0.03
Output Reconstruction	Semi-honest	0	0.47	0	0.23	0	0.05
	Malicious						0.09
Fair Output Reconstruction	Malicious	0.47	1.91	0.23	0.77	0.25	0.19

Table 4: Benchmarking for Input Sharing and Output Reconstruction Phases of Our 3PC Protocol

In the semi-honest setting, we observe that the online latency for [4] and our protocol remain same over LAN. This is because both protocols require the same number of rounds of interaction during the online phase and the rtt among every pair of parties remain the same. Over WAN, our protocol outperforms [4] in terms of online latency. We observe that this improvement comes from the asymmetry in the rtt among the parties. In detail, our protocol has *only* one pair amongst the three pairs of parties to communicate for most of the rounds in the online phase. Thus, when compared with existing protocols, we have an additional privilege where we can assign the roles of the parties effectively across the machines so that the pair of parties having the most communication in the online phase is assigned the lowest rtt. As a result, the time taken by a single round of communication comes down to the *minimum* of the rtts among all the pairs, as opposed to the *maximum*. Thus we achieve a gain of (maximum rtt)/(minimum rtt) in time *per* round of communication, compared to the existing protocols.

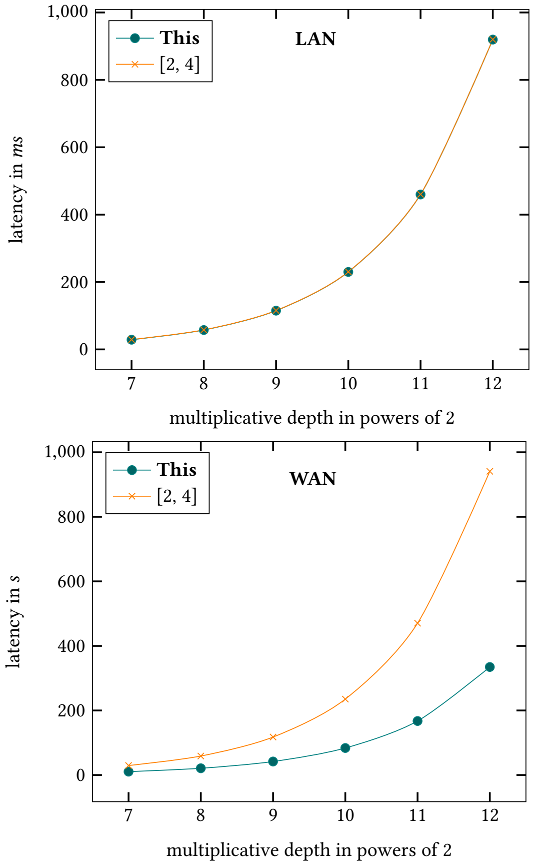

Figure 14: Plot of Online Latency against Multiplicative Depth for 3PC Protocols

In Figure 14, we compare the online latency of our protocols with their competitors, for a varying multiplicative depth (that dictates the round complexity). The same plot applies to both the semi-honest setting and malicious setting, as they differ by a single round and its impact vanishes with the growing number of rounds. It is clear from the plot that the impact of rtt becomes more visible with the increase in the number of online rounds, leading to improved efficiency.

Now, we compare the online throughput for 3PC over both LAN (#AES/sec) and WAN (#AND/sec) setting and the results appear in Table 5 ('M' denotes million and 'Improv.' denotes improvement). Table 5 shows that our protocol's online throughput is clearly better than that of its competitors. This is mainly because of the improvement in online communication, though the asymmetry in our protocol has a contribution to it. In the semi-honest setting, our protocol is able to effectively push around 33% of the total communication to the offline phase, resulting in an improved online phase. In the malicious setting, our protocol reduces online communication by a factor of 2.25× with an increase in the offline phase by a factor of 1.75×, when compared with the state-of-the-art protocols.

Setting	Semi-honest			Malicious		
	[4]	**This**	Improv.	[2]	**This**	Improv.
LAN	3296.7	3296.7	1×	3221.85	3381.91	**1.05×**
WAN	8.71 M	13.1 M	**1.51×**	2.9 M	4.34 M	**1.50×**

Table 5: Comparison of 3PC Online Throughput

6.1.2 Results for Secure Prediction. We benchmark our ML protocols that cover regression functions (linear and SVM) and classification functions (logistic and SVM) over a ring $\mathbb{Z}_{2^{64}}$. We report our performance for MNIST database [46] that has $d = 784$ features and compare our results with ABY3 [50] (with the removal of extra tools as mentioned in the introduction). The comparison of latency and communication appears below.

Regression. For regression, the servers compute $[\![\cdot]\!]$-shares of the function $\vec{\mathbf{w}} \odot \vec{\mathbf{z}} + \mathrm{b}$, given the $[\![\cdot]\!]$-shares of $[\![\vec{\mathbf{w}}]\!]$, $[\![\vec{\mathbf{z}}]\!]$ and $[\![\mathrm{b}]\!]$. This is computed by parties executing secure dot-product on $[\![\vec{\mathbf{w}}]\!]$ and $[\![\vec{\mathbf{z}}]\!]$, followed by locally adding the result with $[\![\cdot]\!]$-shares of b. Here we provide benchmarking for two regression algorithms, namely Linear Regression and Linear SVM Regression. Though the aforementioned algorithms serve a different purpose, we observe that their underlying computation is same from the viewpoint of the servers, apart from the values $\vec{\mathbf{w}}, \vec{\mathbf{z}}$ and b being different as mentioned in Section 5.5. Thus we provide a single benchmark, capturing both the algorithms and the results appear in Table 6.

Setting	Work	Semi-honest		Malicious	
		Offline	Online	Offline	Online
LAN (*ms*)	ABY3	0	0.62	1.61	1.56
	This	0.52	0.61	2.56	1.07
WAN (*s*)	ABY3	0	0.23	0.72	0.70
	This	0.23	0.09	1.1	0.44
Comm. (KB)	ABY3	0	0.02	73.5	55.13
	This	0.01	0.01	128.63	12.27

Table 6: Comparison of Latency and Communication for Regression Protocols

In the semi-honest setting, similar online latency for both protocols over LAN can be justified by the similar rtt among parties. Over WAN, the asymmetry in the rtt among the parties (as mentioned for the case of 3PC) adds benefit to our protocol. In the malicious setting, the result is further improved, since we require one less round when compared with ABY3 in the online phase. We now provide an *online* throughput comparison of our regression protocols over LAN (queries/sec) and WAN (queries/min) setting and the result appear in Table 7. We observe that the throughput was further boosted in the malicious setting because of our efficient dot-product protocol (Section 5.4) with which we could improve the online communication by a factor of 4.5× when compared to ABY3.

Setting	Semi-honest			Malicious		
	ABY3	**This**	Improv.	ABY3	**This**	Improv.
LAN	0.645 M	0.656 M	**1.02×**	0.007 M	0.010 M	**1.5×**
WAN	0.104 M	0.267 M	**2.56×**	0.010 M	0.016 M	**1.5×**

Table 7: Online Throughput of Regression Protocols

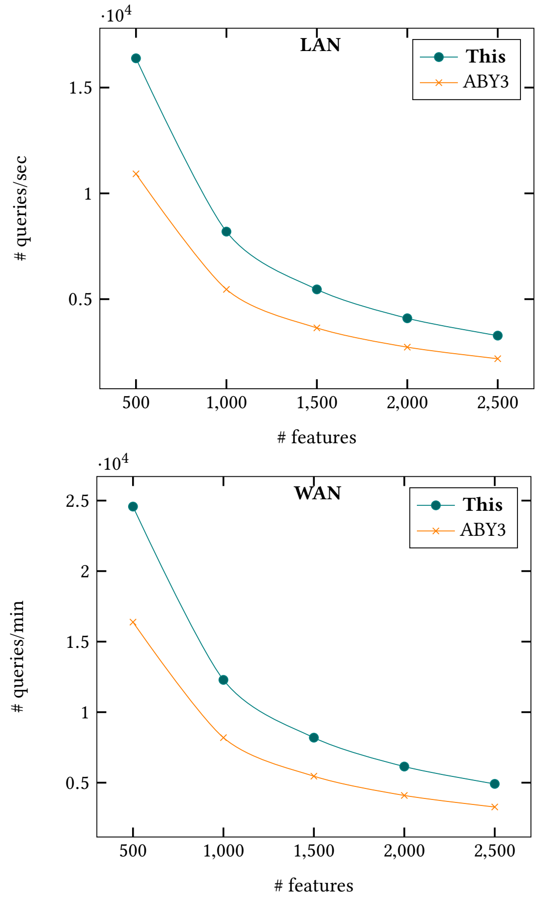

Figure 15: Plot of Online Throughput against Multiplicative Depth for Regression Protocols

In Figure 15, we present a comparison of online throughput (#queries/sec for LAN and #queries/min for WAN) against the number of features in the malicious setting, for a number of features varying from 500 to 2500. Since the online communication cost is independent of the feature size in the semi-honest setting, we omit to plot the same. The plot clearly shows that our protocol for regression outperforms ABY3 in terms of online throughput. The reduction in throughput with the increase in feature size for both ours as well as ABY3's can be explained with the increase in communication for higher feature sizes.

Classification. For classification, the servers compute $[\![\cdot]\!]^{\mathbf{B}}$-shares of the function sign($\vec{\mathbf{w}} \odot \vec{\mathbf{z}} + \mathrm{b}$), given the $[\![\cdot]\!]$-shares of $[\![\vec{\mathbf{w}}]\!]$, $[\![\vec{\mathbf{z}}]\!]$ and $[\![\mathrm{b}]\!]$. Towards this, parties first execute secure dot-product on $[\![\vec{\mathbf{w}}]\!]$ and $[\![\vec{\mathbf{z}}]\!]$, followed by locally adding the result with $[\![\mathrm{b}]\!]$. Then parties execute secure comparison protocol on the result obtained from the previous step to generate the boolean share of sign($\vec{\mathbf{w}} \odot \vec{\mathbf{z}} + \mathrm{b}$). Here we consider two classification algorithms, namely Logistic Regression and Linear SVM Classification. Similar to the case with Regression, both algorithms share the same computation from the server's perspective and thus we provide a single benchmark. The results appear in Table 8 and the online throughput comparison appears in Table 9.

Setting	Work	Semi-honest		Malicious	
		Offline	Online	Offline	Online
LAN (*ms*)	ABY3	0	3.48	1.63	4.42
	This	0.54	1.58	2.57	2.53
WAN (*s*)	ABY3	0	1.61	0.72	2.08
	This	0.23	0.55	1.1	0.98
Comm. (KB)	ABY3	0	0.07	73.7	55.3
	This	0.01	0.04	129	12.4

Table 8: Comparison of Latency and Communication for Classification Protocols

Setting	Semi-honest			Malicious		
	ABY3	**This**	Improv.	ABY3	**This**	Improv.
LAN	0.115 M	0.253 M	**2.2×**	0.007 M	0.010 M	**1.5×**
WAN	0.015 M	0.044 M	**2.93×**	0.010 M	0.016 M	**1.5×**

Table 9: Online Throughput of Classification Protocols

In this case, we observe that our protocol outperforms ABY3 in all the settings. This is mainly due to our Secure Comparison protocol (Section 5.4) where we improve upon both communication and rounds in the online phase. The effect of this improvement becomes more visible for applications where the secure comparison is used extensively. Similar to Regression, in Figure 16, we provide below a comparison of online throughput (#queries/sec for LAN and #queries/min for WAN) against the number of features in the malicious setting.

6.2 Restricted Bandwidth Setting

We observe that the asymmetry of our constructions further comes to our advantage for throughput. That is, while a drop in bandwidth between any pair of parties significantly affects the throughput of the existing protocols, the throughput of ours does not get affected much as long as the drop occurs between the pair(s) of parties handling a low volume of data. The purpose of this setting is to show that for setups with varying bandwidths among the servers, our protocol has an advantage in choosing the roles of the servers whereas existing works cannot. To demonstrate this positive impact, we test the throughput of our ML constructions in a modified network setting where the bandwidth between one of the pairs, namely P_0 and P_2 is restricted to 100Mbps (instead of 1Gbps) in LAN

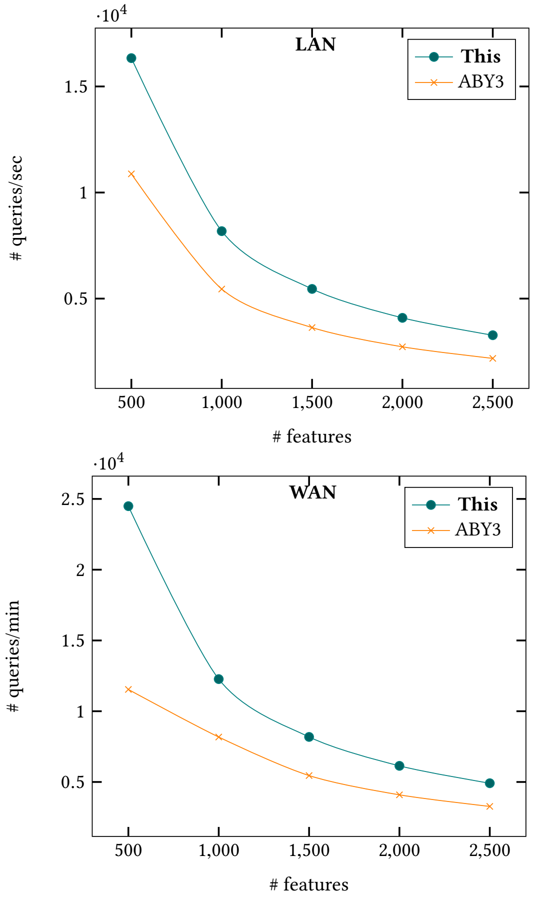

Figure 16: Plot of Online Throughput against Multiplicative Depth for Classification Protocols

and 10Mbps (instead of 25Mbps) in WAN setting. This restriction significantly drops the throughput of the existing constructions as they need all the pairs to communicate equally, while ours remain unaffected. The cut-down on bandwidth does not make any difference in latency (that is measured for *one* execution) and communication complexity. We provide a comparison of throughput in the malicious setting in Table 10.

Setting	Regression			Classification		
	ABY3	**This**	Improv.	ABY3	**This**	Improv.
LAN	0.001 M	0.010 M	**15×**	0.001 M	0.010 M	**15.01×**
WAN	0.004 M	0.016 M	**3.75×**	0.004 M	0.016 M	**3.75×**

Table 10: Online Throughput of ML Protocols in the Malicious Setting under Restricted Bandwidth

The comparison of online throughput (#queries/sec for LAN while #queries/min for WAN) against the number of features in the malicious setting for classification protocols appear in Figure 17.

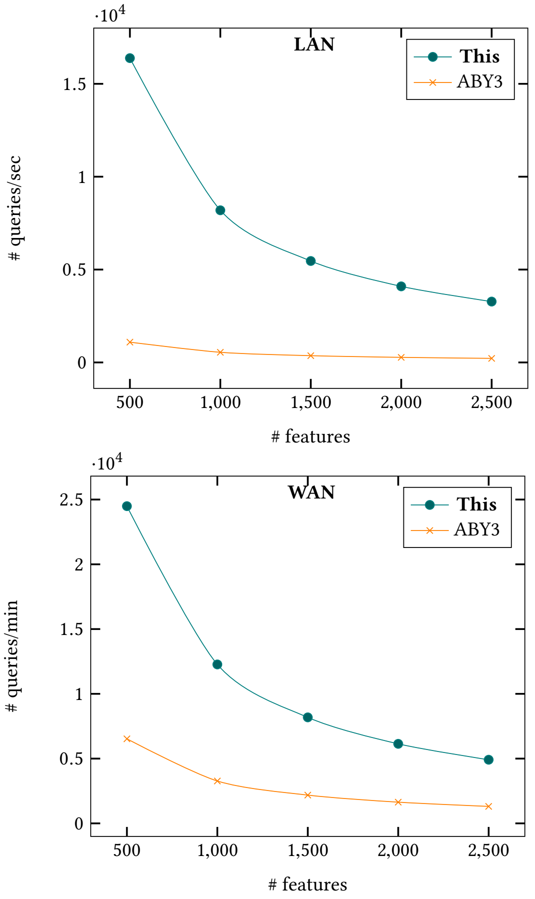

Figure 17: Plot of Online Throughput against Multiplicative Depth for Classification Protocols in the Malicious Setting under Restricted Bandwidth

REFERENCES

[1] V. A. Abril, P. Maene, N. Mertens, and N. P. Smart. [n.d.]. Bristol Fashion MPC Circuits. https://homes.esat.kuleuven.be/~nsmart/MPC/.

[2] T. Araki, A. Barak, J. Furukawa, T. Lichter, Y. Lindell, A. Nof, K. Ohara, A. Watzman, and O. Weinstein. 2017. Optimized Honest-Majority MPC for Malicious Adversaries - Breaking the 1 Billion-Gate Per Second Barrier. In *IEEE S&P*.

[3] T. Araki, A. Barak, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara. 2016. DEMO: High-Throughput Secure Three-Party Computation of Kerberos Ticket Generation. In *ACM CCS*.

[4] T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara. 2016. High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority. In *ACM CCS*.

[5] C. Baum, I. Damgård, T. Toft, and R. W. Zakarias. 2016. Better Preprocessing for Secure Multiparty Computation. In *ACNS*.

[6] D. Beaver. 1991. Efficient Multiparty Protocols Using Circuit Randomization. In *CRYPTO*.

[7] D. Beaver. 1995. Precomputing Oblivious Transfer. In *CRYPTO*.

[8] Z. Beerliová-Trubíniová and M. Hirt. 2006. Efficient Multi-party Computation with Dispute Control. In *TCC*.

[9] Z. Beerliová-Trubíniová and M. Hirt. 2008. Perfectly-Secure MPC with Linear Communication Complexity. In *TCC*.

[10] M. Ben-Or, S. Goldwasser, and A. Wigderson. 1988. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract). In *ACM STOC*.

[11] E. Ben-Sasson, S. Fehr, and R. Ostrovsky. 2012. Near-Linear Unconditionally-Secure Multiparty Computation with a Dishonest Minority. In *CRYPTO*.

[12] Christopher Bishop. 2006. *Pattern Recognition and Machine Learning*.

[13] D. Bogdanov, S. Laur, and J. Willemson. 2008. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *ESORICS*.

[14] D. Bogdanov, R. Talviste, and J. Willemson. 2012. Deploying Secure Multi-Party Computation for Financial Data Analysis. In *FC*.

[15] P. Bogetoft, D. L. Christensen, I. Damgård, M. Geisler, T. P. Jakobsen, M. Krøigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, M. I. Schwartzbach, and T. Toft. 2009. Secure Multiparty Computation Goes Live. In *FC*.

[16] M. Byali, A. Joseph, A. Patra, and D. Ravi. 2018. Fast Secure Computation for Small Population over the Internet. *ACM CCS* (2018).

[17] O. Catrina and S. de Hoogh. 2010. Secure Multiparty Linear Programming Using Fixed-Point Arithmetic. In *ESORICS*.

[18] N. Chandran, J. A. Garay, P. Mohassel, and S. Vusirikala. 2017. Efficient, Constant-Round and Actively Secure MPC: Beyond the Three-Party Case. In *ACM CCS*.

[19] K. Chida, D. Genkin, K. Hamada, D. Ikarashi, R. Kikuchi, Y. Lindell, and A. Nof. 2018. Fast Large-Scale Honest-Majority MPC for Malicious Adversaries. In *CRYPTO*.

[20] A. Choudhury and A. Patra. 2017. An Efficient Framework for Unconditionally Secure Multiparty Computation. *IEEE Trans. Information Theory* (2017).

[21] R. Cleve. 1986. Limits on the Security of Coin Flips when Half the Processors Are Faulty (Extended Abstract). In *ACM STOC*.

[22] R. Cramer, I. Damgård, D. Escudero, P. Scholl, and C. Xing. 2018. SPDZ2k: Efficient MPC mod 2ˆk for Dishonest Majority. *CRYPTO* (2018).

[23] R. Cramer, I. Damgård, and Y. Ishai. 2005. Share Conversion, Pseudorandom Secret-Sharing and Applications to Secure Computation. In *TCC*.

[24] Cryptography and Privacy Engineering Group at TU Darmstadt. 2017. ENCRYPTO Utils. https://github.com/encryptogroup/ENCRYPTO_utils.

[25] M. Dahl. 2018. Private Image Analysis with MPC: Training CNNs on Sensitive Data using SPDZ. (2018).

[26] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. 2013. Practical Covertly Secure MPC for Dishonest Majority - Or: Breaking the SPDZ Limits. In *ESORICS*.

[27] I. Damgård, C. Orlandi, and M. Simkin. 2018. Yet Another Compiler for Active Security or: Efficient MPC Over Arbitrary Rings. *CRYPTO* (2018).

[28] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. 2012. Multiparty Computation from Somewhat Homomorphic Encryption. In *CRYPTO*.

[29] S. de Hoogh, B. Schoenmakers, P.Chen, and H. Akker. 2014. Practical Secure Decision Tree Learning in a Teletreatment Application. In *FC*.

[30] Richard O. Duda, Peter E. Hart, and David G. Stork. 2000. *Pattern Classification (2nd Edition)*.

[31] H. Eerikson, C. Orlandi, P. Pullonen, J. Puura, and M. Simkin. 2019. Use your Brain! Arithmetic 3PC For Any Modulus with Active Security. *IACR Cryptology ePrint Archive* (2019).

[32] A. Esteva, B. Kuprel, R. A. Novoa, J. Ko, S. M. Swetter, H. M. Blau, and S. Thrun. 2017. Dermatologist-level classification of skin cancer with deep neural networks. *Nature* (2017).

[33] J. Furukawa, Y. Lindell, A. Nof, and O. Weinstein. 2017. High-Throughput Secure Three-Party Computation for Malicious Adversaries and an Honest Majority. In *EUROCRYPT*.

[34] A. Gascón, P. Schoppmann, B. Balle, M. Raykova, J. Doerner, S. Zahur, and D. Evans. 2016. Secure Linear Regression on Vertically Partitioned Datasets. *IACR Cryptology ePrint Archive* (2016).

[35] M. Geisler. 2007. Viff: Virtual ideal functionality framework.

[36] O. Goldreich, S. Micali, and A. Wigderson. 1987. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *STOC*.

[37] S. D. Gordon, S. Ranellucci, and X. Wang. 2018. Secure Computation with Low Communication from Cross-Checking. In *ASIACRYPT*.

[38] Y. Ishai, R. Kumaresan, E. Kushilevitz, and A. Paskin-Cherniavsky. 2015. Secure Computation with Minimal Interaction, Revisited. In *CRYPTO*.

[39] S. Kamara, P. Mohassel, and M. Raykova. 2011. Outsourcing Multi-Party Computation. *IACR Cryptology ePrint Archive* (2011).

[40] J. Katz and Y. Lindell. 2014. *Introduction to Modern Cryptography, Second Edition*. CRC Press.

[41] M. Keller, E. Orsini, and P. Scholl. 2016. MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer. In *ACM CCS*.

[42] M. Keller, V. Pastro, and D. Rotaru. 2018. Overdrive: Making SPDZ Great Again. In *EUROCRYPT*.

[43] M. Keller, P. Scholl, and N. P. Smart. 2013. An architecture for practical actively secure MPC with dishonest majority. In *ACM CCS*.

[44] J. Launchbury, D. Archer, T. DuBuisson, and E. Mertens. 2014. Application-Scale Secure Multiparty Computation. In *ESOP*.

[45] S. Laur, H. Lipmaa, and T. Mielikäinen. 2006. Cryptographically private support vector machines. In *ACM SIGKDD*.

[46] Yann LeCun and Corinna Cortes. 2010. MNIST handwritten digit database. (2010). http://yann.lecun.com/exdb/mnist/

[47] Y. Lindell and A. Nof. 2017. A Framework for Constructing Fast MPC over Arithmetic Circuits with Malicious Adversaries and an Honest-Majority. In *ACM CCS*.

[48] J. Liu, M. Juuti, Y. L., and N. Asokan. 2017. Oblivious Neural Network Predictions via MiniONN Transformations. In *ACM CCS*.

[49] E. Makri, D. Rotaru, N. P. Smart, and F. Vercauteren. 2018. EPIC: Efficient Private Image Classification (or: Learning from the Masters). *CT-RSA* (2018).

[50] P. Mohassel and P. Rindal. 2018. ABY^3: A Mixed Protocol Framework for Machine Learning. In *ACM CCS*.

[51] P. Mohassel, M. Rosulek, and Y. Zhang. 2015. Fast and Secure Three-party Computation: Garbled Circuit Approach. In *CCS*.

[52] P. Mohassel and Y. Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *IEEE S&P*.

[53] V. Nikolaenko, S. Ioannidis, U. Weinsberg, M. Joye, N. Taft, and D. Boneh. 2013. Privacy-preserving matrix factorization. In *ACM CCS*.

[54] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft. 2013. Privacy-Preserving Ridge Regression on Hundreds of Millions of Records. In *IEEE S&P*.

[55] P. S. Nordholt and M. Veeningen. 2018. Minimising Communication in Honest-Majority MPC by Batchwise Multiplication Verification. In *ACNS*.

[56] T. Orekondy, B. Schiele, and M. Fritz. 2018. Knockoff Nets: Stealing Functionality of Black-Box Models. *CoRR* (2018).

[57] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami. 2017. Practical Black-Box Attacks Against Machine Learning. In *ASIA CCS*.

[58] A. Patra and D. Ravi. 2018. On the Exact Round Complexity of Secure Three-Party Computation. *CRYPTO* (2018).

[59] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar. 2018. Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications. In *AsiaCCS*.

[60] F. Schroff, D. Kalenichenko, and J. Philbin. 2015. FaceNet: A unified embedding for face recognition and clustering. In *IEEE CVPR*.

[61] N. P. Smart and T. Wood. 2019. Error Detection in Monotone Span Programs with Application to Communication-Efficient Multi-party Computation. In *CT-RSA*.

[62] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. 2016. Stealing Machine Learning Models via Prediction APIs. In *USENIX*.

[63] S. Wagh, D. Gupta, and N. Chandran. [n.d.]. SecureNN: Efficient and Private Neural Network Training. *IACR Cryptology ePrint Archive* 2018 ([n. d.]).

[64] A. C. Yao. 1982. Protocols for Secure Computations. In *FOCS*.

A BUILDING BLOCKS FOR MALICIOUS SECURITY

A.1 Instantiating $\mathcal{F}_{\mathsf{trip}}$

Here, we present a protocol Π_{trip} (Figure 18) that instantiate functionality $\mathcal{F}_{\mathsf{trip}}$ over $\mathbb{Z}_{2^\ell}$, inspired by the works of [2, 33]. The techniques of [2, 33] work for any underlying linear secret-sharing scheme. We avoid the detailed security proof for Π_{trip}, which can be easily derived from [2, 33]. We begin with a sub-protocol Π_{rand}, used in Π_{trip}. Protocol Π_{rand} allows the parties to generate a random and private $\llbracket\cdot\rrbracket$-shared value v. Towards this, parties P_0, P_1 locally sample $\lambda_{\mathsf{v},1}$, P_0, P_2 sample $\lambda_{\mathsf{v},2}$ while parties P_1, P_2 sample m_{v}. The value v is defined as $\mathsf{v} = \mathsf{m}_{\mathsf{v}} - \lambda_{\mathsf{v},1} - \lambda_{\mathsf{v},2}$.

If a party obtains $\perp$ during any stage of the protocol or did not receive an expected message, then it outputs $\perp$ and `abort`.

- **Generating Multiplication Triples Optimistically**: Let $M = BN + C$. The parties execute $2M$ instances of Π_{rand} to generate $\{(\llbracket \mathsf{d}_k \rrbracket, \llbracket \mathsf{e}_k \rrbracket)\}_{k=1,\ldots,M}$. For $k = 1, \ldots, BN+C$, the parties execute $\Pi^{\mathsf{s}}_{\mathsf{Mul}}$ on $\llbracket \mathsf{d}_k \rrbracket$ and $\llbracket \mathsf{e}_k \rrbracket$ to obtain $\llbracket \mathsf{f}_k \rrbracket$. Let $\vec{D} = [(\llbracket \mathsf{d}_k \rrbracket, \llbracket \mathsf{e}_k \rrbracket, \llbracket \mathsf{f}_k \rrbracket)]_{k=1,\ldots,M}$.
- **Cut and Bucket**: Here the parties perform the first verification by opening C triples, and then randomly divide the remainder into buckets as follows.
  - The parties generate a random permutation π over $\{1, \ldots, BN + C\}$ and permute the elements of $\vec{D}$ according to π.
  - The parties publicly reconstruct each of the first C triples in $\vec{D}$ (by executing $\Pi^{\mathsf{s}}_{\mathsf{Rec}}(\llbracket\cdot\rrbracket, \mathcal{P})$ and output $\perp$, if any of these C triples is not a multiplication triple.
  - The remaining BN triples in $\vec{D}$ are arranged into buckets $B_1, \ldots, B_N$, each containing B triples.
- **Check Buckets**: The parties initialize a vector $\vec{d}$ of length N. Then, for $k = 1, \ldots, N$, the parties do the following:
  - Let $\{(\llbracket \mathsf{d}_{k,j} \rrbracket, \llbracket \mathsf{e}_{k,j} \rrbracket, \llbracket \mathsf{f}_{k,j} \rrbracket)\}_{j=1,\ldots,B}$ denote the B shared triples in the bucket B_k.
  - For $j = 2, \ldots, B$, the parties execute Π_{prc} on $(\llbracket \mathsf{d}_{k,1} \rrbracket, \llbracket \mathsf{e}_{k,1} \rrbracket, \llbracket \mathsf{f}_{k,1} \rrbracket)$ and $(\llbracket \mathsf{d}_{k,j} \rrbracket, \llbracket \mathsf{e}_{k,j} \rrbracket, \llbracket \mathsf{f}_{k,j} \rrbracket)$.
  - The parties set $(\llbracket \mathsf{d}_{k,1} \rrbracket, \llbracket \mathsf{e}_{k,1} \rrbracket, \llbracket \mathsf{f}_{k,1} \rrbracket)$ as the kth entry of $\vec{d}$.

The parties output $\vec{d}$.

Figure 18: Protocol to generate N random and private $\llbracket\cdot\rrbracket$-shared multiplication triples

Following the technique of [33], protocol Π_{trip} generates N independent $\llbracket\cdot\rrbracket$-shared random and private multiplication triplets over $\mathbb{Z}_{2^\ell}$ at one go. Informally, the parties first optimistically generate $BN + C$ shared random triples, followed by deploying the cut-and-choose technique. Namely C triples from the set of $BN + C$ triples are randomly selected and opened to check if they are multiplication triples. The remaining BN triples are randomly grouped into N buckets, each containing B triples. In each bucket, parties check if the first triple is a multiplication triple without opening it using the protocol Π_{prc} (Figure 7), by deploying the remaining $B - 1$ triples in the bucket, one by one. If any of these verifications fail, then the parties abort, else they consider the first triple in each of the N buckets as the final output. Following [33], it follows that except with an error probability of at most $\frac{1}{N^{B-1}}$, if any of the N output triplets is not a multiplication triplet, then the honest parties abort the protocol.

In their follow-up work [2], the authors have shown how to reduce the error probability of cut-and-choose technique from $\frac{1}{N^{B-1}}$ to $\frac{1}{N^B}$, thus reducing the bucket size B to $\frac{s}{\log_2 N}$ to attain a statistical-security of 2^{-s}. The idea behind their improvement is as follows: if the array of multiplication triples from the offline phase is randomly shuffled *after* all multiplication gates are evaluated (optimistically), then adversary can successfully cheat only if the random shuffle happens to match correct triples with correctly evaluated multiplication gates and incorrect triples with incorrectly evaluated multiplication gates.

We observe that the above modification is applicable in our context as well. Following [2], the parties can postpone verification of offline step of all the instances of $\Pi^{\mathsf{m}}_{\mathsf{Mul}}$. Once the offline step of all the instances of $\Pi^{\mathsf{m}}_{\mathsf{Mul}}$ corresponding to all the multiplication gates in the circuit is executed, the parties can randomly shuffle the set of triples. The parties can then use the ith triple from the reshuffled set to perform the pending verification corresponding to the offline step of the ith instance of $\Pi^{\mathsf{m}}_{\mathsf{Mul}}$. Notice that unlike [2], in our context, the reshuffling of the set of triples happens in the offline phase itself. Excluding the cost of generating the random permutation π in the protocol of Figure 18, the amortized cost of generating a single multiplication triple will be as follows: there will be $2B$ instances of Π_{rand} followed by B instances of $\Pi^{\mathsf{s}}_{\mathsf{Mul}}$, followed by $B - 1$ instances of Π_{prc}.

A.2 Properties of Π_{prc}

Lemma A.1 (Correctness [20, 33]). *Let* $(\llbracket \mathsf{d} \rrbracket, \llbracket \mathsf{e} \rrbracket, \llbracket \mathsf{f} \rrbracket)$ *be* $\llbracket\cdot\rrbracket$ *sharing of random and private values* d, e *and* f*, such that* $\mathsf{f} = \mathsf{de}$*. Moreover, let* $(\llbracket \mathsf{a} \rrbracket, \llbracket \mathsf{b} \rrbracket, \llbracket \mathsf{c} \rrbracket)$ *be* $\llbracket\cdot\rrbracket$ *sharing of* a, b *and* c*, such that* $\mathsf{c} = \mathsf{ab} + \Delta$*, where* $\Delta \in \mathbb{Z}_{2^\ell}$*. Then the following hold in* Π_{prc}*: If* $\Delta \neq 0$*, then every honest* P_i *outputs* $\perp$*.*

Proof. In Π_{prc}, during the reconstruction of ρ, σ and τ, protocol $\Pi^{\mathsf{m}}_{\mathsf{Rec}}$ ensures that no two honest parties output two different non-$\perp$ values. Now, in order to show the correctness, it suffices to show that $\tau = 0$ iff $\Delta = 0$. Note that,

$$\begin{aligned} \tau &= \mathsf{c} - \mathsf{f} - \sigma\mathsf{d} - \rho\mathsf{e} - \sigma\rho \\ &= \mathsf{c} - \mathsf{de} - (\mathsf{b} - \mathsf{e})\mathsf{d} - (\mathsf{a} - \mathsf{d})\mathsf{e} - (\mathsf{b} - \mathsf{e})(\mathsf{a} - \mathsf{d}) \\ &= \mathsf{c} - \mathsf{ab} = \Delta \end{aligned}$$

It is straightforward from the protocol step that every honest party outputs $\perp$ if $\Delta \neq 0$. □

The privacy of Π_{prc} requires it to maintain the privacy of a, b and c. Note that the values ρ and σ reveal nothing about a and b, as d, e are random and private. The privacy now follows since $\tau = \Delta$ and independent of a, b and c.

B 3PC WITH SEMI-HONEST SECURITY

Here we prove that $\Pi^{\mathsf{s}}_{\mathsf{3pc}}$ securely realizes the standard ideal-world functionality $\mathcal{F}_{\mathsf{3pc}}$ (Figure 5) for securely evaluating any arithmetic circuit over $\mathbb{Z}_{2^\ell}$. Our proof works in the $\mathcal{F}_{\mathsf{setup}}$-hybrid model.

$\mathcal{F}_{\mathsf{setup}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$ who may corrupt one of the parties.

$\mathcal{F}_{\mathsf{setup}}$ picks random keys $k_{01}, k_{02}, k_{12}, k_{\mathcal{P}} \in \{0,1\}^\kappa$ and sends (k_{01}, k_{02}), (k_{01}, k_{12}) and (k_{02}, k_{12}) to P_0, P_1 and P_2 respectively. In addition, $\mathcal{F}_{\mathsf{setup}}$ sends $k_{\mathcal{P}}$ to all the parties.

Figure 19: Functionality $\mathcal{F}_{\mathsf{setup}}$ (semi-honest)

We first consider the simple case, when P_0 is corrupted. Intuitively, the security follows from the fact, that P_0 does not see the messages exchanged between P_1, P_2 during the online phase, who actually perform the circuit-evaluation. So in essence, this is equivalent to P_1, P_2 using the preprocessing done by a trusted third party to do the circuit-evaluation (in the semi-honest setting, even a corrupt P_0 will do the pre-processing honestly).

Theorem B.1. *Protocol $\Pi^{\mathsf{s}}_{\mathsf{3pc}}$ securely realizes the functionality $\mathcal{F}_{\mathsf{3pc}}$ against a static, semi-honest adversary $\mathcal{A}$ in the $\mathcal{F}_{\mathsf{setup}}$-hybrid model, who corrupts P_0.*

Proof. Let $\mathcal{A}$ be a real-world semi-honest adversary corrupting the distributor P_0 during the protocol $\Pi^{\mathsf{s}}_{\mathsf{3pc}}$. We present an ideal-world adversary (simulator) $\mathcal{S}^{\mathsf{s}}_{\mathsf{3pc}}$ for $\mathcal{A}$ in Figure 20 that simulates messages for corrupt P_0. The only communication to P_0 is during the output-reconstruction stage in the online phase. $\mathcal{S}^{\mathsf{s}}_{\mathsf{3pc}}$ can easily simulate these messages, with the knowledge of function output and the masks corresponding to the circuit-output wires.

The simulator plays the role of the honest parties P_1, P_2 and simulates each step of $\Pi^{\mathsf{s}}_{\mathsf{3pc}}$ to corrupt P_0 as follows and finally outputs $\mathcal{A}$'s output.

Offline Phase: $\mathcal{S}^{\mathsf{s}}_{\mathsf{3pc}}$ emulates $\mathcal{F}_{\mathsf{setup}}$ and gives k_{01}, k_{02} and $k_{\mathcal{P}}$ to P_0. In addition, $\mathcal{S}^{\mathsf{s}}_{\mathsf{3pc}}$ on behalf of P_2 receives $\gamma_{xy,2}$ from $\mathcal{A}$ for every multiplication gate $\mathsf{g} = (\mathsf{w}_x, \mathsf{w}_y, \mathsf{w}_z)$. From these, it learns the λ-masks for all the wires in ckt.

Online Phase: On input $\{x_i\}$'s, the inputs of corrupt P_0 and the function output $(y_1, \ldots, y_{\mathrm{O}})$, $\mathcal{S}^{\mathsf{s}}_{\mathsf{3pc}}$ simulates the output-reconstruction stage to $\mathcal{A}$ as follows. For every y_j, it computes $\mathsf{m}_{y_j} = y_j + \lambda_{y_j}$ and sends it to $\mathcal{A}$, on the behalf of P_1. Here λ_{y_j} is the mask corresponding to the output y_j which $\mathcal{S}^{\mathsf{s}}_{\mathsf{3pc}}$ can compute since he learns the entire λ-masks during the offline phase.

Figure 20: Simulator $\mathcal{S}^{\mathsf{s}}_{\mathsf{3pc}}$ for the case of corrupt P_0

The proof now simply follows from the fact that simulated view and real-world view of the adversary are computationally indistinguishable. □

We next consider the case, when the adversary corrupts one of the evaluators. Without loss of generality, we consider the case of a corrupt P_1 and the case of a corrupt P_2 is handled symmetrically. Intuitively, the security, in this case, follows from the fact that each λ-mask is random (from the properties of the underlying PRF) and the one share that is learned by corrupt P_1 for each mask leaks nothing about them and hence the masked values reveal no additional information about the actual values over the wires.

Theorem B.2. *Protocol $\Pi^{\mathsf{s}}_{\mathsf{3pc}}$ securely realizes the functionality $\mathcal{F}_{\mathsf{3pc}}$ against a static, semi-honest adversary $\mathcal{A}$ in the $\mathcal{F}_{\mathsf{setup}}$-hybrid model, who corrupts P_1 (and similarly P_2).*

Proof. Let $\mathcal{A}$ be a real-world semi-honest adversary corrupting the evaluator P_1 during the protocol $\Pi^{\mathsf{s}}_{\mathsf{3pc}}$. We now present the steps of the ideal-world adversary (simulator) $\mathcal{S}^{\mathsf{s}}_{\mathsf{3pc}}$ for $\mathcal{A}$ for this case in Figure 21. At a high level, $\mathcal{S}^{\mathsf{s}}_{\mathsf{3pc}}$ itself does the honest pre-processing on the behalf of P_0 and will simulate the entire circuit-evaluation, assuming the circuit-inputs of P_0 and P_2 to be 0. In the output-reconstruction stage, it "adjusts" the shares of circuit-output values on the behalf of P_2 so that $\mathcal{A}$ sees the same function output as in the real-world protocol.

The simulator plays the role of the honest parties P_0, P_2 and simulates each step of the protocol $\Pi^{\mathsf{s}}_{\mathsf{3pc}}$ to corrupt P_1 as follows and finally outputs $\mathcal{A}$'s output.

Offline Phase: $\mathcal{S}^{\mathsf{s}}_{\mathsf{3pc}}$ emulates $\mathcal{F}_{\mathsf{setup}}$ and gives k_{01}, k_{12} and $k_{\mathcal{P}}$ to P_1. $\mathcal{S}^{\mathsf{s}}_{\mathsf{3pc}}$ chooses a random key k_{02}. With these, $\mathcal{S}^{\mathsf{s}}_{\mathsf{3pc}}$, on the behalf of P_0, executes the offline steps of the instances of $\Pi^{\mathsf{s}}_{\mathsf{Sh}}$, $\Pi^{\mathsf{s}}_{\mathsf{Mul}}$ and Π_{Add} for circuit-inputs, multiplication and addition gates respectively. In the process, it learns the masks for each wire in the ckt and γ-values for each multiplication gate.

Online Phase:

- *Sharing Circuit-input Values*: For every circuit-input x_j that P_0 inputs, $\mathcal{S}^{\mathsf{s}}_{\mathsf{3pc}}$ sets $x_j = 0$ and simulates the messages of P_0 as part of the online steps of $\Pi^{\mathsf{s}}_{\mathsf{Sh}}(P_0, x_j)$.The inputs owned by P_2 are simulated similarly.
  - *Gate Evaluation*: The simulator simulates the evaluation of each gate g according to the topological order. No simulation is needed for an addition gate. If g is a multiplication gate, then the simulator simulates the messages of P_2 as part of the online steps of the corresponding instance of $\Pi^{\mathsf{s}}_{\mathsf{Mul}}$.
  - *Output Reconstruction*: For $j = 1, \ldots, \mathrm{O}$ let $[\lambda_{y_j}] = (\lambda_{y_j,1}, \lambda_{y_j,2})$ be the sharing, available with the simulator and let m_{y_j} be the simulated masked output, corresponding to y_j, available with $\mathcal{A}$. On input $\{x_i\}$'s, the inputs of corrupt P_1 and the function output $(y_1, \ldots, y_{\mathrm{O}})$, as part of online steps of the instance $\Pi^{\mathsf{s}}_{\mathsf{Rec}}([\lambda_{y_j}])$, the simulator sends $\mathsf{m}_{y_j} - \lambda_{y_j,1} - y_j$ as the share of λ_{y_j}, on the behalf of P_2 to $\mathcal{A}$.

Figure 21: Simulator $\mathcal{S}^{\mathsf{s}}_{\mathsf{3pc}}$ for the case of corrupt P_1

It is easy to see that the simulated view and the real-world view of the adversary are computationally indistinguishable. □

C 3PC WITH MALICIOUS SECURITY

Here we prove that $\Pi^{\mathsf{m}}_{\mathsf{3pc}}$ securely realizes the standard ideal-world functionality $\mathcal{F}^{\mathsf{Abort}}_{\mathsf{3pc}}$ (Figure 9) for securely evaluating any arithmetic circuit over $\mathbb{Z}_{2^\ell}$ with selective abort. Our proof works in $\{\mathcal{F}_{\mathsf{setup}}, \mathcal{F}_{\mathsf{trip}}\}$-hybrid model.

$\mathcal{F}_{\mathsf{setup}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{setup}}$ picks random keys $k_{01}, k_{02}, k_{12}, k_{\mathcal{P}} \in \{0, 1\}^{\kappa}$.

Output to adversary: If $\mathcal{S}$ sends abort, then send (Output, $\perp$) to all the parties. Otherwise, send (Output, y_i) to the adversary $\mathcal{S}$, where $y_i = (k_{i1}, k_{i2}, k_{\mathcal{P}})$ when P_0 is corrupt and $y_i = (k_{0i}, k_{12}, k_{\mathcal{P}})$ when $P_i \in \{P_1, P_2\}$ is corrupt.

Output to selected honest parties: Receive (select, $\{I\}$) from adversary $\mathcal{S}$, where $\{I\}$ denotes a subset of the honest parties. If an honest party P_i belongs to I, send (Output, $\perp$), else send (Output, y_i). Here $y_i = (k_{i1}, k_{i2}, k_{\mathcal{P}})$ when $P_i = P_0$ and $y_i = (k_{0i}, k_{12}, k_{\mathcal{P}})$ when $P_i \in \{P_1, P_2\}$

Figure 22: Functionality $\mathcal{F}_{\mathsf{setup}}$ (malicious)

Since the protocol $\Pi^{\mathsf{m}}_{\mathsf{3pc}}$ differs from $\Pi^{\mathsf{s}}_{\mathsf{3pc}}$ mainly in three protocols – sharing ($\Pi^{\mathsf{m}}_{\mathsf{Sh}}$), reconstruction ($\Pi^{\mathsf{m}}_{\mathsf{Rec}}$) and multiplication ($\Pi^{\mathsf{m}}_{\mathsf{Mul}}$) protocols, we provide the details of simulation for the same. We begin with the case, when P_0 is corrupted.

Theorem C.1. *In $\{\mathcal{F}_{\mathsf{setup}}, \mathcal{F}_{\mathsf{trip}}\}$-hybrid model, $\Pi^{\mathsf{m}}_{\mathsf{3pc}}$ securely realizes the functionality $\mathcal{F}^{\mathsf{Abort}}_{\mathsf{3pc}}$ against a static, malicious adversary $\mathcal{A}$, who corrupts P_0.*

Proof. Let $\mathcal{A}$ be a real-world malicious adversary corrupting P_0 during $\Pi^{\mathsf{m}}_{\mathsf{3pc}}$. We present an ideal-world adversary (simulator) $\mathcal{S}^{\mathsf{m}}_{\mathsf{3pc}}$ for $\mathcal{A}$, who plays the roles of honest P_1, P_2 and simulates the messages received by P_0 during the protocol. The simulation is similar as in the semi-honest setting, where the simulator simulates P_1, P_2 with random inputs and keeps track of all the values that the parties (both honest and corrupt) are supposed to hold. Based on this, the simulator can find out whether the corrupt P_0 is sending an incorrect message(s) in any of the sub-protocols and accordingly simulates honest parties aborting the protocol. The simulator initializes a Boolean variable $\mathsf{flag} = 0$, which indicates whether the honest parties abort during the simulation. Similar to the semi-honest setting, $\mathcal{S}^{\mathsf{m}}_{\mathsf{3pc}}$ invokes the simulator $\mathcal{S}^{\mathsf{m}}_{\mathsf{setup}}$ and learns the shared keys among P_0-P_1 and P_0-P_2, namely k_{01} and k_{02} and the key $k_{\mathcal{P}}$. From the shared keys, it learns the λ-masks for all the wires in ckt. The details of $\mathcal{S}^{\mathsf{m}}_{\mathsf{3pc}}$ for the offline phase is as follows:

- *Offline Step of the instances* $\Pi^{\mathsf{m}}_{\mathsf{Sh}}$ *and* $\Pi^{\mathsf{m}}_{\mathsf{Rec}}$: Here the simulator has to simulate nothing, as the offline phase involves no communication.
- *Offline Step of the instances* $\Pi^{\mathsf{m}}_{\mathsf{Mul}}(\mathsf{w}_{xj}, \mathsf{w}_{yj}, \mathsf{w}_{zj})$: The simulator receives $\gamma_{x_jy_j,2}$ from $\mathcal{A}$ on behalf of P_2. Simulator then picks random $\delta_{x_j}, \delta_{y_j}$ and δ_{z_j} and their $[\cdot]$-shares on behalf of P_1, P_2 and honestly simulates the messages of P_1, P_2 as per the protocol $\Pi^{\mathsf{m}}_{\mathsf{Mul}}$. Namely, the simulator learns from $\mathcal{A}$ the inputs with which P_0 wants to call $\mathcal{F}_{\mathsf{trip}}$. If the input of P_0 to $\mathcal{F}_{\mathsf{trip}}$ is $\perp$, then the simulator sets $\mathsf{flag} = 1$, else the simulator plays the role of $\mathcal{F}_{\mathsf{trip}}$ honestly with the inputs received on behalf of P_0 and generates a $[\![\cdot]\!]$-sharing of a randomly chosen multiplication triplet $(\mathsf{d}, \mathsf{e}, \mathsf{f})$. On behalf of P_1, P_2, the simulator sends to $\mathcal{A}$ the $[\cdot]$-shares of χ. For the instance of Π_{prc}, the simulator honestly simulates the messages of P_1, P_2 towards P_0. Moreover, the simulator sets $\mathsf{flag} = 1$, if it finds that $\gamma_{x_jy_j} \neq \lambda_{x_j}\lambda_{y_j}$.

The details of $\mathcal{S}^{\mathsf{m}}_{\mathsf{3pc}}$ for simulating the messages of the online phase are as follows. Informally, the simulator extracts the circuit-inputs of P_0 from the masked circuit-inputs which P_0 sends to the evaluators since the simulator will know the corresponding mask. The simulator then sets the circuit-inputs of P_1, P_2 to some arbitrary values and simulates the steps of the online phase. During the evaluation of multiplication gates, P_0 receives versions of $\mathsf{m}^{\star}_{x}$ and $\mathsf{m}^{\star}_{y}$, which can be easily simulated as the simulator has selected them. Finally, while simulating the public reconstruction of $[\![\cdot]\!]$-shared circuit-outputs, the simulator adjusts the shares of P_1, P_2, so that P_0 receives the same output as it would have received in the execution of the real-world protocol. As done in the simulation of the offline phase, the simulator keeps track of all the values that the corrupt P_0 possess and sets $\mathsf{flag} = 1$ if it finds that P_0 is sending an inconsistent value during the simulated execution.

- *Online Step of the instances* $\Pi^{\mathsf{m}}_{\mathsf{Sh}}(P_i, x_j)$: If $P_i = P_0$, then the simulator receives m_{x_j} and m'_{x_j} from $\mathcal{A}$ on behalf of P_1 and P_2 respectively. The simulator sets $\mathsf{flag} = 1$ if it finds that $\mathsf{m}_{x_j} \neq \mathsf{m}'_{x_j}$, else it extracts the inputs x_j of P_0 as $x_j = \mathsf{m}_{x_j} - \lambda_{x_j}$, where λ_{x_j} is the mask which the simulator learnt during the offline step. If $P_i \in \{P_1, P_2\}$, then nothing needs to be simulated as P_0 does not receive any message as a part of online step of such instances of $\Pi^{\mathsf{m}}_{\mathsf{Sh}}(P_i, x_j)$. For such instances, the simulator sets $x_j = 0$ and accordingly computes the simulated $[\![x_j]\!]$.
- *Online Step of the instances* $\Pi^{\mathsf{m}}_{\mathsf{Mul}}(\mathsf{w}_{xj}, \mathsf{w}_{yj}, \mathsf{w}_{zj})$: The simulator honestly performs the steps of P_1, P_2 for this instance and computes the simulated $[\![z_j]\!]$. On behalf of P_1, the simulator sends $\mathsf{m}^{\star}_{x_j} = \mathsf{m}_{x_j} + \delta_{x_j}$ and $\mathsf{m}^{\star}_{y_j} = \mathsf{m}_{y_j} + \delta_{y_j}$ to $\mathcal{A}$, while he sends hash of the same to $\mathcal{A}$ on behalf of P_2. The simulator receives $\mathsf{H}(\mathsf{m}^{\star}_{z_j})$ and $\mathsf{H}(\mathsf{m}^{\star}_{z'_j})$ from $\mathcal{A}$ on behalf of P_1 and P_2 respectively. The simulator sets $\mathsf{flag} = 1$ if $\mathsf{H}(\mathsf{m}^{\star}_{z_j}) \neq \mathsf{H}(\mathsf{m}^{\star}_{z'_j})$ or if $\mathsf{H}(\mathsf{m}^{\star}_{z_j}) \neq \mathsf{H}(\mathsf{m}_{z_j} - \mathsf{m}^{\star}_{x_j}\mathsf{m}^{\star}_{y_j} + \delta_{z_j})$.
- *Obtaining function outputs*: If flag is set to 1 during any step of the simulation till now, then the simulator sends $\perp$ to $\mathcal{F}^{\mathsf{Abort}}_{\mathsf{3pc}}$, which corresponds to the case that in the real-world protocol, the honest parties abort before reaching to the output-reconstruction stage, implying that no party receives the output. Else the simulator sends inputs $\{x_j\}$ extracted on behalf of P_0 to $\mathcal{F}^{\mathsf{Abort}}_{\mathsf{3pc}}$ and receives the function outputs $y_1, \ldots, y_{\mathsf{O}}$.
- *Simulating the instances of* $\Pi^{\mathsf{m}}_{\mathsf{Rec}}(\star, \mathcal{P})$ *during the output- reconstruction*: For $j = 1, \ldots, \mathsf{O}$, let $[\lambda_{y_j}] = (\lambda_{y_j,1}, \lambda_{y_j,2})$ be the $[\cdot]$-shared mask, corresponding to the jth circuit-output, available with the simulator. Then as a part of the jth instance of $\Pi^{\mathsf{m}}_{\mathsf{Rec}}$, the simulator sends $y_j + \lambda_{y_j}$ and $\mathsf{H}(y_j + \lambda_{y_j})$ to $\mathcal{A}$ on behalf of P_1 and P_2 respectively. Moreover, the simulator receives $\mathsf{H}(\lambda_{y'_j,i})$ from $\mathcal{A}$ on behalf of P_i for $i \in \{1, 2\}$. The simulator initializes the set I to $\emptyset$. If $\mathsf{H}(\lambda_{y'_j,i,1}) \neq \mathsf{H}(\lambda_{y_j,i,1})$ then the simulator includes P_i to the set I. The simulator then sends the set I to $\mathcal{F}^{\mathsf{Abort}}_{\mathsf{3pc}}$ and terminates.

The proof now follows from the fact that simulated view and real-world view of a corrupt P_0 are computationally indistinguishable. □

We next consider the case, when the adversary corrupts one of the evaluators, say P_1.

Theorem C.2. *In the* $\{\mathcal{F}_{\mathsf{setup}}, \mathcal{F}_{\mathsf{trip}}\}$*-hybrid model,* $\Pi^{\mathsf{m}}_{\mathsf{3pc}}$ *securely realizes the functionality* $\mathcal{F}^{\mathsf{Abort}}_{\mathsf{3pc}}$ *against a static, malicious adversary* $\mathcal{A}$*, who corrupts* P_1.

Proof. The correctness follows similar to Theorem C.1. We now focus on privacy. Let $\mathcal{A}$ be a real-world malicious adversary corrupting the evaluator P_1 during the protocol $\Pi^{\mathsf{m}}_{\mathsf{3pc}}$. We present an ideal-world adversary (simulator) $\mathcal{S}^{\mathsf{m}}_{\mathsf{3pc}}$ for $\mathcal{A}$, who plays the roles of honest P_0, P_2 and simulates the messages received by P_1 during the protocol. $\mathcal{S}^{\mathsf{m}}_{\mathsf{3pc}}$ invokes the simulator $\mathcal{S}^{\mathsf{m}}_{\mathsf{setup}}$ and learns the shared keys among P_1-P_0 and P_1-P_2, namely k_{01} and k_{12} and the key $k_{\mathcal{P}}$. In addition, $\mathcal{S}^{\mathsf{m}}_{\mathsf{3pc}}$ chooses a random key k_{02}. The details of $\mathcal{S}^{\mathsf{m}}_{\mathsf{3pc}}$ for the offline phase is as follows:

- *Offline Step of the instances* $\Pi^{\mathsf{m}}_{\mathsf{Sh}}$ *and* $\Pi^{\mathsf{m}}_{\mathsf{Rec}}$: Here the simulator has to simulate nothing, as the offline phase involves no communication.
- *Offline Step of the instances* $\Pi^{\mathsf{m}}_{\mathsf{Mul}}(\mathsf{w}_{xj}, \mathsf{w}_{yj}, \mathsf{w}_{zj})$: On behalf of P_0, the simulator computes $\gamma_{x_jy_j} = \lambda_{x_j}\lambda_{y_j}$. In addition, simulator learns $\gamma_{x_jy_j,1}$ that $\mathcal{A}$ computes, for the shared key k_{01}. With these, simulator computes $\gamma_{x_jy_j,2} = \gamma_{x_jy_j} - \gamma_{x_jy_j,1}$. On behalf of

P_2, simulator computes $\delta_{x_j}, \delta_{y_j}, \delta_{z_j,1}$ and $\delta_{z_j,2}$ using the key k_{12}. The simulator receives from $\mathcal{A}$, the input with which P_1 wants to call $\mathcal{F}_{\mathsf{trip}}$. If this input is $\perp$, then the simulator sets $\mathsf{flag} = 1$. Else the simulator itself honestly performs the steps of $\mathcal{F}_{\mathsf{trip}}$ and generates $\llbracket \cdot \rrbracket$-sharing of a random multiplication triplet $(\mathsf{d}, \mathsf{e}, \mathsf{f})$. The simulator then receives χ_1 from $\mathcal{A}$ on behalf of P_0. The simulator then computes $\llbracket \mathsf{a} \rrbracket, \llbracket \mathsf{b} \rrbracket, \llbracket \mathsf{c} \rrbracket$ and honestly executes the steps of Π_{prc} on behalf of P_0, P_2. Moreover, the simulator sets $\mathsf{flag} = 1$, if $\chi_1 \neq \delta_{x_j}\lambda_{y_j,1} + \delta_{y_j}\lambda_{x_j,1} + \delta_{z_j,1} - \gamma_{x_j y_j,1}$, else the simulator computes $\chi = \chi_1 + \chi_2$.

The details of $\mathcal{S}^{\mathsf{m}}_{\mathsf{3pc}}$ for simulating the messages of the online phase are as follows.

- *Online Step of the instances* $\Pi^{\mathsf{m}}_{\mathsf{Sh}}(P_i, x_j)$: If $P_i = P_0$, then on behalf of P_0, the simulator sets $x_j = 0$ and sends $\mathsf{m}_{x_j} = 0 + \lambda_{x_j}$ to $\mathcal{A}$. Then on behalf of P_2, the simulator receives $\mathsf{H}(\mathsf{m}_{x'_j})$ from $\mathcal{A}$, which P_1 wants to send to P_2; the simulator sets $\mathsf{flag} = 1$ if it finds that $\mathsf{H}(\mathsf{m}_{x'_j}) \neq \mathsf{H}(\mathsf{m}_{x_j})$. If $P_i = P_1$, then on behalf of P_2, the simulator receives m_{x_j} from $\mathcal{A}$, which P_1 wants to send to P_2 and extract the input $x_j = \mathsf{m}_{x_j} - \lambda_{x_j}$ of P_1. If $P_i = P_2$, then the simulator sets $x_j = 0$ and sends $\mathsf{m}_{x_j} = 0 + \lambda_{x_j}$ to $\mathcal{A}$ on behalf of P_2.
- *Online Step of the instances* $\Pi^{\mathsf{m}}_{\mathsf{Mul}}(\mathsf{w}_{x\,j}, \mathsf{w}_{y\,j}, \mathsf{w}_{z\,j})$: On behalf of P_2, the simulator honestly sends the $[\cdot]$-share of m_{z_j} to $\mathcal{A}$. Then on behalf of P_2, the simulator receives from $\mathcal{A}$ the $[\cdot]$-share of m_{z_j}, which P_1 wants to send to P_2. The simulator checks if this share is correct and accordingly sets $\mathsf{flag} = 1$. The simulator then receives $\mathsf{m}^{\star}_{x_j}$ and $\mathsf{m}^{\star}_{y_j}$ from $\mathcal{A}$ on behalf of P_0, which P_1 wants to send to P_0. The simulator sets $\mathsf{flag} = 1$, if it finds that $\mathsf{m}^{\star}_{x_j} \neq \mathsf{m}_{x_j} + \delta_{x_j}$ or $\mathsf{m}^{\star}_{y_j} \neq \mathsf{m}_{y_j} + \delta_{y_j}$. On behalf of P_0, the simulator sends $\mathsf{m}^{\star}_{z_j} = -\lambda_{y_j} \cdot \mathsf{m}^{\star}_{x_j} - \lambda_{x_j} \cdot \mathsf{m}^{\star}_{y_j} + \delta_{z_j} + 2\gamma_{x_j y_j} + \chi$ to $\mathcal{A}$.
- *Obtaining function outputs*: If flag is set to 1 during any step of the simulation till now, then the simulator sends $\perp$ to $\mathcal{F}^{\mathsf{Abort}}_{\mathsf{3pc}}$. Else the simulator sends inputs x_j extracted on behalf of P_1 to $\mathcal{F}^{\mathsf{Abort}}_{\mathsf{3pc}}$ and receives the function outputs $y_1, \ldots, y_{\mathsf{O}}$.
- *Simulating the instances of*$\Pi^{\mathsf{m}}_{\mathsf{Rec}}(\star, \mathcal{P})$ *during the output-reconstruction*: For $j = 1, \ldots, \mathsf{O}$, let $(\lambda_{y_j,1}, \mathsf{m}_{y_j})$ be the share of P_1 available with the simulator, as a part of the simulated output sharing $\llbracket y_j \rrbracket$. Then as a part of $\Pi^{\mathsf{m}}_{\mathsf{Rec}}(\llbracket y_j \rrbracket, \mathcal{P})$, on behalf of P_2 and P_0, the simulator sends $\mathsf{m}_{y_j} - \lambda_{y_j,1} - y_j$ and $\mathsf{H}(\mathsf{m}_{y_j} - \lambda_{y_j,1} - y_j)$ respectively to $\mathcal{A}$, which ensures that $\mathcal{A}$ reconstructs $\mathsf{m}_{y_j} - \lambda_{y_j,1} - (\mathsf{m}_{y_j} - \lambda_{y_j,1} - y_j) = y_j$. On behalf of P_0 and P_2 respectively, the simulator receives $\mathsf{m}_{y'_j}$ and $\mathsf{H}(\lambda'_{y_j,1})$ from $\mathcal{A}$, which P_1 wants to send to P_0 and P_2 respectively as a part of $\Pi^{\mathsf{m}}_{\mathsf{Rec}}(\llbracket y_j \rrbracket, \mathcal{P})$. The simulator initializes the set I to $\emptyset$. The simulator includes P_0 to I if it finds that $\mathsf{m}_{y'_j} \neq \mathsf{m}_{y_j}$. Similarly, the simulator includes P_2 to I, if it finds that $\mathsf{H}(\lambda'_{y_j,1}) \neq \mathsf{H}(\lambda_{y_j,1})$. The simulator then sends the set I to $\mathcal{F}^{\mathsf{Abort}}_{\mathsf{3pc}}$ and terminates.

It is easy to see that the simulated and real-world views of the adversary are computationally indistinguishable. □

D SECURE PREDICTION

Lemma D.1 (Correctness). *In the protocol* $\Pi^{\mathsf{m}}_{\mathsf{BitExt}}$, *the following holds: During the offline phase, honest parties compute either* $\mathsf{r} = \mathsf{r}_1\mathsf{r}_2$ *or output* $\perp$. *During the online phase, honest parties either obtain* $\mathsf{sign}(\mathsf{ra})$ *or output* $\perp$.

Proof. During the offline phase, parties locally set $\llbracket \mathsf{r}_1 \rrbracket_{P_0} = (0, 0), \llbracket \mathsf{r}_1 \rrbracket_{P_1} = (\mathsf{r}_1, 0)$ and $\llbracket \mathsf{r}_1 \rrbracket_{P_2} = (\mathsf{r}_1, 0)$, which effectively assign $\mathsf{m}_{\mathsf{r}_1} = \mathsf{r}_1$ and $\lambda_{\mathsf{r}_1} = 0$. Hence, the aforementioned way of computing shares non-interactively indeed generates a valid $\llbracket \cdot \rrbracket$-sharing of r_1 according to our sharing semantics. Similarly, the $\llbracket \cdot \rrbracket$-sharing of r_2 is valid since the parties effectively assign $\mathsf{m}_{\mathsf{r}_2} = 0$ and $\lambda_{\mathsf{r}_2} = -\mathsf{r}_2$. Given the $\llbracket \cdot \rrbracket$-sharing of r_1 and r_2, it follows from the correctness property of protocol $\Pi^{\mathsf{m}}_{\mathsf{Mul}}$ (Lemma 4.3) that honest parties compute either $\mathsf{r} = \mathsf{r}_1\mathsf{r}_2$ or output $\perp$ during the offline phase.

Similar to the offline phase, following the correctness of $\Pi^{\mathsf{m}}_{\mathsf{Mul}}$, honest parties either compute $\llbracket \cdot \rrbracket$-sharing of ra correctly or output $\perp$ during the online phase. During the reconstruction of ra towards P_0, P_1, since each missing share is held by two other parties and we have at most one corruption, it holds that each of P_0, P_1 either obtain ra or output $\perp$. Now that the value ra is available with both P_0 and P_1, when P_1 performs $\llbracket \cdot \rrbracket$-sharing of $\mathsf{q} = \mathsf{msb}(\mathsf{ra})$, party P_2 can cross check hash of m_{q} received from P_1 with the one received from P_0. Thus a corrupt P_0 or P_1 cannot force an honest P_2 to accept a wrong q value. Moreover, the last step where parties compute $\llbracket \cdot \rrbracket^{\mathsf{B}}$-shares of $\mathsf{p} \oplus \mathsf{q}$ is non-interactive. Hence, the correctness of online phase is ensured. □