MPClan: Protocol Suite for Privacy-Conscious Computations

Nishat Koti*, Shravani Patil*, Arpita Patra*, Ajith Suresh†,
*Indian Institute of Science, Bangalore, Email: {kotis, shravanip, arpita}@iisc.ac.in
†Technical University of Darmstadt, Germany, Email: suresh@encrypto.cs.tu-darmstadt.de

***Abstract*—The growing volumes of data being collected and its analysis to provide better services are creating worries about digital privacy. To address privacy concerns and give practical solutions, the literature has relied on secure multiparty computation. However, recent research has mostly focused on the small-party honest-majority setting of up to four parties, noting efficiency concerns. In this work, we extend the strategies to support a larger number of participants in an honest-majority setting with efficiency at the center stage.**

Cast in the preprocessing paradigm, our semi-honest protocol improves the online complexity of the decade-old state-of-the-art protocol of Damgård and Nielson (CRYPTO'07). In addition to having an improved online communication cost, we can shut down almost half of the parties in the online phase, thereby saving up to 50% in the system's operational costs. Our maliciously secure protocol also enjoys similar benefits and requires only half of the parties, except for one-time verification, towards the end.

To showcase the practicality of the designed protocols, we benchmark popular applications such as deep neural networks, graph neural networks, genome sequence matching, and biometric matching using prototype implementations. Our improved protocols aid in bringing up to 60-80% savings in monetary cost over prior work.

I. Introduction

Today's world is seeing a visible transition from offline services to a heavy dependency on online platforms for banking, socializing, healthcare, etc. This is leading to an increased user presence online, which leaves a trail of online activity and personal data over the Internet. The availability of such user-specific data opens up possibilities for its misuse. For instance, there has been a lot of concern raised regarding advertisement service providers such as Google, Facebook breaching user privacy for targeted advertisement services [34]. In the process of providing enhanced targeted advertisement services, service providers are allegedly learning more information about their users than they are entitled to (e.g., user's shopping activity, browsing history) from various data collection entities. These entities collect user data via website cookies, loyalty cards, etc. [65]. While such targeted advertisements offer a personalized online experience, they may come at the cost of revealing unauthorized user data to these service providers. Such a challenge is also encountered in the healthcare sector. Collaborative analysis among healthcare institutes over patient data is known to facilitate better diagnosis and improved treatment. However, laws such as GDPR, which prevent sharing of patient records, hinder such collaborations, thereby re-emphasizing the need for mechanisms that enable privacy-preserving computations.

Such mechanisms that ensure privacy-preserving computations can be facilitated via several privacy-enhancing technologies such as homomorphic encryption [16], [41], differential privacy [36], secure multiparty computation [89], [11], [43], to name a few. We focus on secure multiparty computation (MPC) as it has been the cornerstone of research lately, showcasing its effectiveness in various applications such as privacy-preserving machine learning [56], [68], [85], secure collaborative analytics [75], secure genome matching [79], [5], etc. Essentially, it offers a solution to the potential privacy issues which may arise in collaborative computation scenarios such as targeted advertisements described earlier. MPC allows mutually distrusting parties to perform computations on their private inputs such that they learn nothing beyond the output of the computation. The distrust among the parties is captured by the notion of a centralized adversary, which is said to corrupt up to t out of the n participating parties. Depending on its behaviour, the adversary can be categorized as either *semi-honest* or *malicious* [42]. Semi-honest adversary models the corruption scenario where the corrupt parties are restricted to follow the protocol and cannot deviate arbitrarily, as in the stronger notion of malicious corruption.

MPC with honest majority, where only a minority of the parties are corrupt, enables construction of efficient protocols for multiple parties [13], [31], [1], [48], [12], [75]. The recent concretely efficient protocols have only considered small number of parties [56], [74], [22], [85], [68], [28], [66], [84], which restricts the number of corruptions to at most one ($t = 1$). Although the small-party setting has found application in the outsourced computation paradigm too, the generic *multiparty* setting is a better fit for real-world deployments due to its resiliency to a higher number of corruptions ($t < n/2$). Thus, for larger n, the number of corruptions that can be tolerated is also higher, thereby increasing the trust in the system. Moreover, multiparty setting allows for privacy-conscious computations even in a non-outsourced deployment scenario, such as in providing targeted advertisement services (described in Fig. 1 and elaborated below), when outsourcing the computation is not feasible/preferable. Hence, to design efficient protocols, we focus on honest majority multiparty computation.

a) Use Case: Consider the scenario of targeted advertisement services depicted in Fig. 1(a). Typically, data collection entities track a user's online activities via website cookies while browsing the Internet (①). Also known as cookie profiling, such data collection allows the entities to create a "profile" for each user, which may contain information such as browsing habits, gender, marital status, and age, to name a few, as shown in ②. These profiles can facilitate targeted advertisements via specialized algorithms (④), which is leveraged by the advertisement service providers such as Google and Facebook.

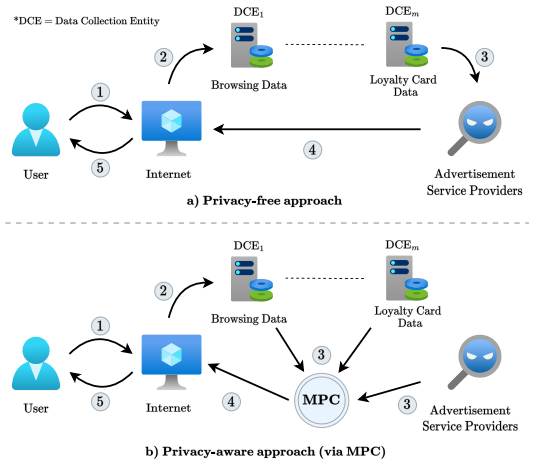


Fig. 1: Use case for privacy-conscious solutions

While such services offer a personalized experience, it comes at the expense of users' private data being revealed to the service providers, as indicated in ③. A feasible solution (Fig. 1(b)) instead is to place a solution box at the interface between these service providers and the data collection entities such that it provides mechanisms to ensure the privacy of user data while also facilitating the required computations over the same (to provide targeted advertisement services). MPC being a technology that supports privacy-preserving computations, lends itself well to such tasks. Instead of the data collection entities directly revealing the user data to the advertisement service providers, they can engage in an instance of MPC protocol (③) which securely runs the required algorithm on the user data while maintaining its privacy. Moreover, such a computation does not require the data collection entities to reveal their data to each other, thus offering a viable solution. Furthermore, as studied in [49], the effectiveness of targeted advertisements can greatly benefit from the use of machine learning algorithms. In particular, neural networks, and more recently graph neural networks [24], [72], [90], [62], [88] have shown the potential to better analyse the data available via user profiles, in turn allowing for a refined personalized experience. We thus focus on protocols for securely evaluating the standard neural networks such as VGG16 [82] (deep neural network) and graph neural network, and provide benchmarks for the same in Section V.

A. Related work

We restrict related work to MPC protocols in honest-majority setting. Despite the interest in MPC for small population [4], [3], [39], [23], [1], [21], [74], [22], [18], [56], [85], [28], MPC protocols for arbitrary number of parties have been studied largely [38], [31], [48], [6], [8], [10], [15], [13], [17], [78], [2], [19], [12], [45]. In the honest majority ($t < n/2$) semi-honest setting, [31], [40] forms the state of the art MPC protocols over fields in the information theoretic setting. This was further optimized in the computational setting in [13] using a one-time setup for correlated randomness. We will often refer to this optimized honest-majority semi-honest protocol of [31] as DN07. In the information-theoretic setting, the work of [46], improves upon the communication and round complexity of [31]. The work of [38] recently demonstrates MPC protocols in the honest majority setting in the preprocessing model with malicious security, which requires communicating $3t$ field elements in the online as well as the preprocessing phase. We observe that the semi-honest protocol derived from this requires communicating $2t$ elements in the online and $3t$ elements in the preprocessing phase. The recent work of [12], [6] provides semi-honest MPC protocols which require *each* party to communicate roughly t elements per multiplication gate, resulting in quadratic communication in the number of parties. DN07 has served as the basis for obtaining malicious security for free (i.e. amortized communication cost of $3t$ elements per multiplication gate) in the computational setting [13], [15] as well as in the information-theoretic setting [48], [46]. Both [48] and [15] follow the approach of executing a semi-honest protocol, followed by a verification phase to check the correctness of multiplication which involves heavy polynomial interpolation operations. As mentioned earlier, the recent work of [38] focuses on maliciously secure protocols for honest-majority setting in the preprocessing model. Their protocol relies on an instantiation of [48] in the preprocessing phase that requires communicating $3t$ elements while requiring another $3t$ element communication in the online phase. However, their protocol is inefficient due to a consistency check required after each level of multiplication and introduces depth-dependent overhead in communication complexity. The absence of this check results in a privacy breach as described in [47] and is elaborated in §B-B0c.

B. Towards practically efficient protocols

Before stating our contributions, we elaborate on the choices made in designing a practically efficient protocol.

1. *Preprocessing paradigm.* With the goal of attaining as fast a response time as possible, the protocols are cast in the preprocessing paradigm [33], [30], [54], [52], [7], [32], [25], [76], [53], [74], [22]. Here, expensive *data-independent* computations are carried out in a preprocessing phase, thereby making way for a fast and efficient *data-dependent* online phase. We thus focus on improving the online phase without hampering the overall protocol complexity.

2. *Algebraic structure.* To further enhance efficiency by utilizing the underlying CPU architecture, several protocols work over rings [68], [56], [22], [85], [57], [66]. We follow this approach and design MPC protocols operating over the ring $\mathbb{Z}_{2^\ell}$ and rely on replicated secret sharing (RSS). Note that usage of RSS inherently results in exponential blow-up in the number of shares for an arbitrary number of parties. Hence, it is well-suited for the practically-oriented scenarios comprising of a constant number of parties [13], [15], which we restrict to for benchmarking our protocols.

3. *Masked evaluation.* To make our protocols efficient in the preprocessing paradigm, we use the masked evaluation paradigm, a variant of the replicated secret sharing scheme. The secret data is masked using a masking value in this case, and the mask is RSS shared. The computation is done on the publicly available masked values and the shared masks. This technique was first introduced in the context of circuit garbling schemes (see [64], [87]), and was then adapted to secret sharing-based protocols in dishonest majority (see [51], [9]). It was later applied to small-population honest-majority settings such as [44], [21], [74], [28], [56] and [57] to aid in the development of practically efficient protocols.

4. *Adversarial strategy.* Based on the deployment scenario, different levels of security may be desired. While semi-honest security suffices for several applications as shown in [4], [59], [21], [70], [5], [79], [20], [83], malicious security is always desirable. Thus, to cater to different scenarios, our protocols are designed to provide semi-honest and malicious security, where each security goal has its merit.

5. *Monetary cost.* To reduce the operational costs in the online phase, several recent works [74], [56], [22], [57] reduce the number of (online) computing parties. This is useful in long computations such as those involved in privacy-preserving machine learning (PPML) applications, which span several days or even weeks. Reducing the number of online parties is especially advantageous for protocols deployed in the secure outsourced computation (SOC) setting since one has to pay for the up-time of every hired server. Shutting down even a single server significantly helps in reducing the monetary cost [67], [57] of the system. We thus focus on ensuring the participation of a minimal number of parties during the online computation in our protocols. This is achieved for the first time in generic n-party protocols [1]. Specifically, all the protocols for the semi-honest setting in our framework benefit from using only $t+1$ parties in the online phase. The protocols in the malicious setting also enjoy this benefit except that the remainder t parties are required to come online for a short verification phase at the end. The reduction in online parties aids in improving the operational cost of the framework by almost 50%. This is unlike prior works [31], [13], [15], [48], [46] which require active participation from all parties throughout the computation.

C. Our Contributions

We begin with a quick overview of the contributions of this work, followed by the details.

- We construct an n-party semi-honest protocol in the preprocessing paradigm which offers an improved online phase than the decade-old state-of-the-art protocol of [31], without inflating its total cost. Moreover, our protocol reduces the number of active parties in the online phase, thereby improving the system's operational cost when deployed in SOC setting.
- We extend our semi-honest protocol to the malicious setting, while retaining the benefits of requiring reduced number of parties in online phase for majority of the computation. Our offer over state-of-the-art protocol of [38] is a stronger security guarantee of fairness, and $\mathcal{O}(d)$ improvement in round complexity. Here, d denotes depth of the circuit to be evaluated.
- We provide support for 3 and 4 input multiplication, at the same online complexity as that of the 2 input multiplication. In addition to improving the communication cost over the approach of sequential multiplications, multi-input multiplication offers a $2\times$ improvement in the round complexity which is beneficial for high latency networks.
- We design building blocks for a range of applications such as deep neural networks, graph neural networks, genome sequence matching and biometric matching. When the applications are benchmarked, our semi-honest protocol witnesses a saving of up to 69% in monetary cost, and has $3.5\times$ to $4.6\times$ improvements in online run time and throughput over [31]. Interestingly, our maliciously secure protocols outperforms the semi-honest protocol of [31] in terms of online run time and throughput for the applications under consideration, achieving the goal of fast online phase.

We now elaborate on the contributions and highlight the technical details and novelty of our work.

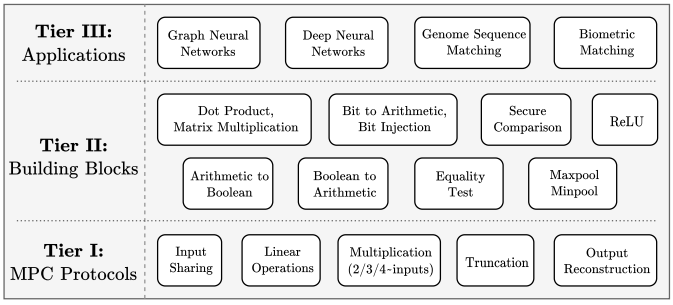


Fig. 2: Hierarchy of primitives in our 3-tier framework

Our protocol suite follows a 3-tier architecture (Fig. 2) to attain the final goal of privacy-conscious computations. The first tier comprises fundamental primitives such as input sharing, reconstruction, multiplication (with truncation), and multi-input multiplication. The second tier includes building blocks such as dot product, matrix multiplication, conversion between Boolean and arithmetic worlds, comparison, equality, non-linear activation functions, to name a few, as required in the applications considered. Finally, the third tier is applications. Our main contribution lies in Tier I and is detailed below.

1) Tier I - MPC protocols: Our goal is to design protocols with a fast online phase. Thus, working over $\mathbb{Z}_{2^\ell}$ and relying on RSS, we design a semi-honest MPC protocol in the computational setting assuming a one-time shared-key setup for correlated randomness.

Note that the straightforward extension of semi-honest multiplication protocol of [31] to the preprocessing model, which can also be derived from the recent work of [38], incurs a communication of $3t$ elements in the preprocessing phase while communicating $2t$ elements in the online. This amounts to a $1.6\times$ overhead in the total cost over [31]. Our contribution lies in ensuring a fast online phase, without inflating the total communication cost of the protocol. Specifically, our protocol requires communicating only $2t$ ring elements in the online phase and t in the preprocessing, for a multiplication gate. We are the first to achieve a communication cost of $2t$ in the online phase (unlike $3t$ in the prior works [31], [40]), without incurring any overhead in the total cost, i.e., our total cost still matches that of the best known (optimized) semi-honest honest-majority protocol [31], [40].

We extend our protocol to provide malicious security with *fairness*[2] at the cost of additionally communicating t elements in the online phase and $2t$ in the preprocessing phase. Although (*abort*[3]) protocol of [38] has the same communication as our maliciously secure protocol, we achieve a stronger security notion of fairness. Moreover, [38] requires an additional round of communication for consistency checks after each level, the

[1] A recent work [38] also claims to achieve this reduction in online parties. However, their protocol suffers from a privacy breach as explained in §B-B0c.

[2] Guarantees either all parties receive the output or none do.

[3] Honest parties may not receive the output while corrupt parties do.

absence of which results in a privacy breach (described in [47] and elaborated in §B-B0c), and necessitates participation from all parties. However, by relying on a variant of RSS, our protocol avoids the consistency check after each level of circuit evaluation and ensures privacy. Notably, we only require participation from all parties for a one-time verification at the end of evaluation, thus reducing the number of rounds by d (d denotes circuit depth).

3 and 4 input multiplications: Following [73], [71], [57], to reduce the online communication cost and round complexity, we design protocols to enable the multiplication of 3 and 4 inputs in a single shot. Compared to the naive approach of performing sequential multiplications to multiply 3/4 inputs, the *multi-input multiplication* protocol enjoys the benefit of having the same online phase complexity as that of the 2-input multiplication protocol. This brings in a $2\times$ improvement in the online round complexity and improves the online communication cost. Support for multi-input multiplication enables usage of optimized adder circuits [73] for secure comparison and Boolean addition, thereby resulting in a faster online phase. The recent work of [46] also proposes a method to improve the round complexity of circuit evaluation by evaluating all gates in two consecutive layers in a circuit in parallel. We observe that their method can be viewed as a variant of multi-input multiplication with 3 and 4 inputs. Thus, our protocols need not be limited to facilitate faster comparison and Boolean additions alone (as described above), but can be used to reduce the round and communication complexity of any general circuit evaluation. Note that [46] only improves the round complexity ($2\times$) without inflating the communication cost when compared to [31]. However, we focus on improving round complexity ($2\times$) *as well as* communication of the online phase by trading off an increase in the preprocessing.

2) Tier II - Building Blocks: We design efficient protocols for several building blocks in semi-honest and malicious settings, which are stepping stones for Tier III applications. These are extensions from the small party setting [68], [74], [56], [73], and hence we defer the details to §C-A (semi-honest) and §C-B (malicious).

3) Tier III - Applications: To showcase the practicality of our framework and improvements of our protocols, we benchmark a range of applications such as neural networks (NN), which also includes the popular deep NN called VGG16 [82], graph neural network, genome sequence matching, and biometric matching, and are considered for the first time in the n-party honest-majority setting. We benchmark the applications in the WAN setting using Google Cloud instances. As mentioned, owing to the inherent restrictions of RSS and keeping the focus on practical scenarios, we showcase the performance of our protocols for $n = 5, 7$, and 9 and compare with the state-of-the-art semi-honest protocol of [31].

1. *Deep neural networks.* We benchmark inference phases of deep neural networks such as LeNet [60] and VGG16 [82]. We observe savings of up to 69% in monetary cost, and improvements of up to $4.3\times$ in online run-time and throughput, in comparison to [31].

2. *Graph neural network.* We benchmark the inference phase of graph neural network [35], [81] on MNIST [61] data set. In comparison to [31], our protocol improves up to $3.5\times$ in online run-time, and sees up to 15% savings in monetary cost.

3. *Genome sequence matching.* We demonstrate an efficient protocol for similar sequence queries (SSQ), which can be used to perform secure genome matching. Our protocol is based on the protocol of [79] which works for 2 parties and uses an edit distance approximation [5]. We extend and optimize the protocol for the multiparty setting. In comparison to [31], we witness improvements of up to $4\times$ in online run-time and throughput, and savings of 66% in monetary cost.

4. *Biometric matching.* We propose efficient protocols for computing Euclidean distance (ED), which forms the basis for performing biometric matching. Continuing the trend, we witness a $4.6\times$ improvement in online run-time and throughput compared to [31], and savings of up to 85% in monetary cost.

II. Preliminaries

We cast our protocols in the (function-dependent) preprocessing paradigm to enable a fast online phase. Parties rely on a one-time shared key setup (see §A) [74], [22], [56], [68], [4], [18] to enable generation of correlated randomness, non-interactively. Our protocols are designed for rings ($\mathbb{Z}_{2^\ell}$). We use fixed-point arithmetic (FPA) [70], [68], [21], [22], [74], [56] representation to operate over decimal values. Here, a decimal value is represented as an ℓ-bit integer in signed 2's complement representation. The most significant bit (msb) represents the sign bit, and d least significant bits are reserved for the fractional part. The ℓ-bit integer is then treated as an element of $\mathbb{Z}_{2^\ell}$, and operations are performed modulo 2^ℓ. We let $\ell = 64$, $d = 13$, with $\ell - d - 1$ bits for the integer part.

This work considers both semi-honest and malicious adversarial models with static and at most $t < n/2$ corruptions. The security of constructions is proved using the real-world/ ideal-world simulation paradigm [63], and the details are provided in §E. Let $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ denote the set of n parties which are connected by pair-wise private and authentic channels in a synchronous network. Set $\mathcal{E} = \{P_1, P_2, \ldots, P_{t+1}\}$, termed as the evaluator set, comprises parties that are active during the online phase. Set $\mathcal{D} = \{P_{t+2}, P_{t+3}, \ldots, P_n\}$, termed as the helper set, comprises parties which help in the preprocessing phase, and in the online verification in the malicious setting. Parties agree on a $P_{\mathsf{king}} \in \mathcal{E}$. Without loss of generality, let $P_{\mathsf{king}} = P_{t+1}$.

a) Sharing semantics: We use the following sharing semantics, based on RSS & additive sharing schemes, which facilitate a fast online phase.

- $\langle\cdot\rangle$*-sharing:* This denotes the replicated secret sharing (RSS) of a value with threshold t. A value $\mathsf{a} \in \mathbb{Z}_{2^\ell}$ is said to be RSS-shared with threshold t if for every subset $\mathcal{T} \subset \mathcal{P}$ of $n - t$ parties there exists $\langle \mathsf{a} \rangle_{\mathcal{T}} \in \mathbb{Z}_{2^\ell}$ possessed by all $P_i \in \mathcal{T}$ such that $\mathsf{a} = \sum_{\mathcal{T}} \langle \mathsf{a} \rangle_{\mathcal{T}}$.

Alternatively, for every set of t parties, the residual $h = n - t$ parties forming the set $\mathcal{T}$, hold the share $\langle \mathsf{a} \rangle_{\mathcal{T}}$. Let $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_{\mathsf{q}} \subset \mathcal{P}$ be the distinct subsets of size h, where $\mathsf{q} = \binom{n}{h}$ represents the total number of shares. Since P_i belongs to $\binom{n-1}{h-1}$ such sets, the tuple of shares $\{\langle \mathsf{a} \rangle_{\mathcal{T}}\}$ that it possesses are denoted as $\langle \mathsf{a} \rangle_i$.

- $[\cdot]$*-sharing:* A value $\mathsf{a} \in \mathbb{Z}_{2^\ell}$ is said to be $[\cdot]$-shared (additively shared) among parties in $\mathcal{P}$ if $P_i \in \mathcal{P}$ possesses $[\mathsf{a}]_i \in \mathbb{Z}_{2^\ell}$ such that $\mathsf{a} = [\mathsf{a}]_1 + [\mathsf{a}]_2 + \ldots + [\mathsf{a}]_n$.

Helper primitive	Input	Output
$\Pi_{[0]}$	-	$[\cdot]$-sharing of 0
Π_{rand}	-	$\langle\cdot\rangle$-sharing of a random value $\mathsf{r} \in \mathbb{Z}_{2^\ell}$
Π_{pRand}	Identity of a party P_s	$\langle\cdot\rangle$-sharing of a random value $\mathsf{r} \in \mathbb{Z}_{2^\ell}$ such that P_s learns all shares
$\Pi_{.\to\langle\cdot\rangle}$	$\mathsf{a} \in \mathbb{Z}_{2^\ell}$ held by at least $t+1$ parties	$\langle\!\langle\mathsf{a}\rangle\!\rangle$-sharing
$\Pi_{\langle\cdot\rangle\to\mathcal{T}[\cdot]}$	$\langle\mathsf{a}\rangle$-sharing, $\mathcal{T} \subset \mathcal{P}$ such that $\lvert\mathcal{T}\rvert = t+1$	${}^{\mathcal{T}}[\mathsf{a}]$-sharing
$\Pi_{\langle\cdot\rangle\to[\cdot]}$	$\langle\mathsf{a}\rangle$-sharing	$[\mathsf{a}]$-sharing
$\Pi_{\langle\!\langle\cdot\rangle\!\rangle\to\mathcal{T}[\cdot]}$	$\langle\!\langle\mathsf{a}\rangle\!\rangle$-sharing, $\mathcal{T} \subset \mathcal{P}$ such that $\lvert\mathcal{T}\rvert = t+1$	${}^{\mathcal{T}}[\mathsf{a}]$-sharing
$\Pi_{\langle\!\langle\cdot\rangle\!\rangle\to[\cdot]}$	$\langle\!\langle\mathsf{a}\rangle\!\rangle$-sharing	$[\mathsf{a}]$-sharing
$\Pi_{\langle\!\langle\cdot\rangle\!\rangle\to\langle\cdot\rangle}$	$\langle\!\langle\mathsf{a}\rangle\!\rangle$-sharing	$\langle\mathsf{a}\rangle$-sharing
$\Pi_{\langle\cdot\rangle\cdot\langle\cdot\rangle\to[\cdot]}$	$\langle\mathsf{a}\rangle$-sharing, $\langle\mathsf{b}\rangle$-sharing	$[\mathsf{ab}]$-sharing
Π_{agree}	$\mathcal{P}, \vec{\mathsf{v}_1}, \ldots, \vec{\mathsf{v}_n}$	'`continue`' if $\vec{\mathsf{v}_i} = \vec{\mathsf{v}_j}$ for all $P_i, P_j \in \mathcal{P}$, '`abort`' otherwise
$\Pi_{\langle\cdot\rangle}$	a, identity of a party P_s	$\langle\mathsf{a}\rangle$-sharing

TABLE I: Description of helper primitives – all primitives are non-interactive, except Π_{agree} (see §A-A for details)

- ${}^{\mathcal{T}}[\cdot]$*-sharing:* A value $\mathsf{a} \in \mathbb{Z}_{2^\ell}$ is said to be ${}^{\mathcal{T}}[\cdot]$-shared among $t+1$ parties in $\mathcal{T}$, if each $P_i \in \mathcal{T}$ holds ${}^{\mathcal{T}}[\mathsf{a}]_i$ such that $\mathsf{a} = \sum_{P_i \in \mathcal{T}} {}^{\mathcal{T}}[\mathsf{a}]_i$. We refer to this sharing scheme as $(t+1)$-additive sharing, and use ${}^{\mathcal{E}}[\mathsf{a}]$ to denote such a sharing among parties in $\mathcal{E}$.
- $\langle\!\langle\cdot\rangle\!\rangle$*-sharing:* A value $\mathsf{a} \in \mathbb{Z}_{2^\ell}$ is said to be $\langle\!\langle\cdot\rangle\!\rangle$-shared in the semi-honest setting if there exist values $\lambda_\mathsf{a}, \mathsf{m}_\mathsf{a} \in \mathbb{Z}_{2^\ell}$ such that $\mathsf{m}_\mathsf{a} = \mathsf{a} + \lambda_\mathsf{a}$ where λ_a is $\langle\cdot\rangle$-shared among $\mathcal{P}$ and every $P_i \in \mathcal{E}$ holds m_a. We denote the shares of $P_i \in \mathcal{D}$ by $\langle\!\langle\mathsf{a}\rangle\!\rangle_i = \langle\lambda_\mathsf{a}\rangle_i$ and that of $P_i \in \mathcal{E}$ as $\langle\!\langle\mathsf{a}\rangle\!\rangle_i = (\mathsf{m}_\mathsf{a}, \langle\lambda_\mathsf{a}\rangle_i)$. In the malicious setting, m_a is held by all parties, and $\langle\!\langle\mathsf{a}\rangle\!\rangle_i = (\mathsf{m}_\mathsf{a}, \langle\lambda_\mathsf{a}\rangle_i)$ for all $P_i \in \mathcal{P}$.

It is trivial to see that all the sharing schemes mentioned above are linear. This allows parties to compute linear operations such as addition and multiplication with constants locally. The Boolean world operates over $\mathbb{Z}_2$, and we denote the corresponding Boolean sharing with a superscript **B**. Notations are summarized in Table II.

Notation	Description
$n = 2t+1$	Total number of parties with t corrupt and $h = t+1$ honest
$\mathcal{T}_1, \ldots, \mathcal{T}_\mathsf{q}$	$\mathsf{q} = \binom{n}{h}$ distinct subsets of $\mathcal{P}$ with $t+1$ parties each
q	Number of replicated secret shares (RSS) of a value
$\mathsf{g} = \binom{n-1}{h-1}$	Number of RSS shares of a value held by a party
$\mathcal{E}$	Online parties $(P_1, \ldots, P_{t+1})$ that actively carry out the computation
$\mathcal{D}$	Helper parties $(P_{t+2}, \ldots, P_n)$
a_i	i^{th} element of vector $\vec{a}$
$\vec{a} \odot \vec{b}$	dot product of vectors $\vec{a}$ and $\vec{b}$
$\mathbf{A} \bigodot \mathbf{B}$	Multiplication of matrices $\mathbf{A}$ and $\mathbf{B}$
b^R	Arithmetic (Ring) equivalent over $\mathbb{Z}_{2^\ell}$ of bit $\mathsf{b} \in \mathbb{Z}_2$
$\mathsf{v}[i]$	i^{th} bit of ℓ-bit value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$
$\mathsf{m}_\mathsf{a} = \mathsf{a} + \lambda_\mathsf{a}$	Masked value m_a for $\mathsf{a} \in \mathbb{Z}_{2^\ell}$ with mask $\lambda_\mathsf{a} \in \mathbb{Z}_{2^\ell}$
$\mathsf{M}_{\mathsf{a}_1\mathsf{a}_2\ldots\mathsf{a}_k}$	$\prod_{i=1}^k \mathsf{m}_{\mathsf{a}_i}$; Product of masked values $\mathsf{m}_{\mathsf{a}_1}, \ldots, \mathsf{m}_{\mathsf{a}_k}$
$\Lambda_{\mathsf{a}_1\mathsf{a}_2\ldots\mathsf{a}_k}$	$\prod_{i=1}^k \lambda_{\mathsf{a}_i}$; Product of masks $\lambda_{\mathsf{a}_1}, \ldots, \lambda_{\mathsf{a}_k}$

TABLE II: Notations used in this work

b) Helper primitives: We use the primitives described in Table I from literature [13], [15], [74], [26] in our protocols, and their details are deferred to §A-A. The Boolean variants of corresponding primitives are denoted with a superscript **B**.

III. MPClan Protocol

This section details the semi-honest MPC protocol execution performed over the ring $\mathbb{Z}_{2^\ell}$ that comprises three phases– input sharing, evaluation (linear operations and multiplication), and output reconstruction.

a) Input sharing and Output Reconstruction: To enable $P_s \in \mathcal{P}$ to $\langle\!\langle\cdot\rangle\!\rangle$-share a value $\mathsf{v} \in \mathbb{Z}_{2^\ell}$, parties first non-interactively sample $\langle\cdot\rangle$-shares of λ_v, relying on the shared-key setup, such that P_s learns all these shares on clear (via Π_{pRand}). This enables P_s to compute and send $\mathsf{m}_\mathsf{v} = \mathsf{v} + \lambda_\mathsf{v}$ to parties in $\mathcal{E}$, thereby generating $\langle\!\langle\mathsf{v}\rangle\!\rangle$.

To reconstruct v towards all parties given $\langle\!\langle\mathsf{v}\rangle\!\rangle$, parties in $\mathcal{E}$ non-interactively generate its additive shares, ${}^{\mathcal{E}}[\mathsf{v}]$, among themselves (via $\Pi_{\langle\!\langle\cdot\rangle\!\rangle\to\mathcal{E}[\cdot]}$). These parties send their additive shares to P_{king}, who computes and sends v to all parties. Reconstruction towards a single party, say P_s, proceeds similarly except that the protocol terminates after parties in $\mathcal{E}$ send their additive shares of v to $P_{\mathsf{king}} = P_s$, who then computes v.

b) Evaluation: Evaluation comprises linear operations of addition and multiplication with public constant, and non-linear operations such as multiplication. Parties can non-interactively compute linear operations owing to the linearity of the $\langle\!\langle\cdot\rangle\!\rangle$-sharing. Concretely, given $\langle\!\langle\mathsf{a}\rangle\!\rangle, \langle\!\langle\mathsf{b}\rangle\!\rangle$ and public constants $\mathsf{c}_1, \mathsf{c}_2$, parties can non-interactively compute $\langle\!\langle\mathsf{c}_1\mathsf{a}+\mathsf{c}_2\mathsf{b}\rangle\!\rangle$ as $\mathsf{c}_1\langle\!\langle\mathsf{a}\rangle\!\rangle + \mathsf{c}_2\langle\!\langle\mathsf{b}\rangle\!\rangle$.

To compute $\langle\!\langle\cdot\rangle\!\rangle$-shares for non-linear operations such as multiplication, say $\mathsf{z} = \mathsf{ab}$ given $\langle\!\langle\mathsf{a}\rangle\!\rangle, \langle\!\langle\mathsf{b}\rangle\!\rangle$, parties proceed as follows. At a high-level, the approach is to enable generation of $\langle\!\langle\mathsf{z}-\mathsf{r}\rangle\!\rangle$ and $\langle\!\langle\mathsf{r}\rangle\!\rangle$ for a random $\mathsf{r} \in \mathbb{Z}_{2^\ell}$, which enables parties to non-interactively compute $\langle\!\langle\mathsf{z}\rangle\!\rangle = \langle\!\langle\mathsf{z}-\mathsf{r}\rangle\!\rangle + \langle\!\langle\mathsf{r}\rangle\!\rangle$. Observe that $\langle\!\langle\mathsf{r}\rangle\!\rangle$ can be generated non-interactively by locally sampling each of its shares. To generate $\langle\!\langle\mathsf{z}-\mathsf{r}\rangle\!\rangle$, we let parties in $\mathcal{E}$ obtain $\mathsf{z}-\mathsf{r}$, following which $\langle\!\langle\mathsf{z}-\mathsf{r}\rangle\!\rangle$ can be generated non-interactively (this is achieved via $\Pi_{.\to\langle\!\langle\cdot\rangle\!\rangle}$ where all parties set their shares of $\langle\lambda_{\mathsf{z}-\mathsf{r}}\rangle$ as 0, and parties in $\mathcal{E}$ set $\mathsf{m}_{\mathsf{z}-\mathsf{r}} = \mathsf{z}-\mathsf{r}$). Observe that z remains private while revealing $\mathsf{z}-\mathsf{r}$ to parties in $\mathcal{E}$ since r is a random mask not known to adversary.

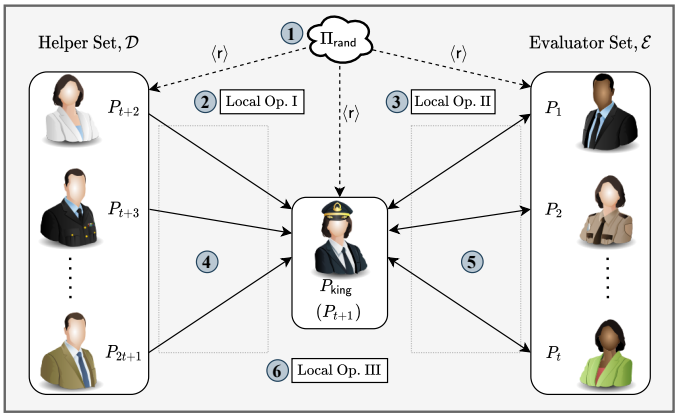
To enable parties in $\mathcal{E}$ to obtain $\mathsf{z}-\mathsf{r}$, we let $\mathsf{z}-\mathsf{r} = \mathsf{D}+\mathsf{E}$, where D is additively shared among parties in $\mathcal{D}$ while E is additively shared among parties in $\mathcal{E}$ (D, E are defined in the following paragraphs). Thus, to reconstruct $\mathsf{z}-\mathsf{r}$ towards parties in $\mathcal{E}$, parties send their respective additive shares of D

or E towards $P_{\mathsf{king}} \in \mathcal{P}$. P_{king} reconstructs D, E, and sends $\mathsf{z}-\mathsf{r} = \mathsf{D}+\mathsf{E}$ to parties in $\mathcal{E}$. Elaborately, as seen in [21], [57], $\mathsf{z}-\mathsf{r}$ can be computed as

$$\begin{aligned}\mathsf{z}-\mathsf{r} &= \mathsf{ab}-\mathsf{r} = (\mathsf{m_a}-\lambda_\mathsf{a})(\mathsf{m_b}-\lambda_\mathsf{b})-\mathsf{r} \\ &= \mathsf{M_{ab}}-\mathsf{m_a}\lambda_\mathsf{b}-\mathsf{m_b}\lambda_\mathsf{a}+\Lambda_\mathsf{ab}-\mathsf{r} \\ &= \underbrace{\mathsf{M_{ab}}-\mathsf{m_a}\lambda_\mathsf{b}-\mathsf{m_b}\lambda_\mathsf{a}+(\Lambda_\mathsf{ab}-\mathsf{r})_\mathcal{E}}_{\mathsf{E}} + \underbrace{(\Lambda_\mathsf{ab}-\mathsf{r})_\mathcal{D}}_{\mathsf{D}}\end{aligned} \tag{1}$$

where $\Lambda_\mathsf{ab}-\mathsf{r} = (\Lambda_\mathsf{ab}-\mathsf{r})_\mathcal{D} + (\Lambda_\mathsf{ab}-\mathsf{r})_\mathcal{E}$.

We next detail the steps in the multiplication protocol, and its schematic representation is provided in Fig. 3.



① Generation of random $\mathsf{r} \in \mathbb{Z}_{2^\ell}$ ② Computing $[\mathsf{r}]$ &$\langle\!\langle\mathsf{r}\rangle\!\rangle$
③ Computing ${}^{\mathcal{E}}[\lambda_\mathsf{a}], {}^{\mathcal{E}}[\lambda_\mathsf{b}]$ ④ $\mathcal{D}$ sending $\{[\mathsf{z}-\mathsf{r}]\}_\mathcal{D}$ to P_{king}
⑤ $\mathcal{E}$ sending $\{[\mathsf{z}-\mathsf{r}]\}_\mathcal{E}$ to P_{king} and receiving result from P_{king}
⑥ Computing $\langle\!\langle\mathsf{z}\rangle\!\rangle$

Fig. 3: Steps of multiplication protocol

► *Step* ①: Parties non-interactively generate $\langle\mathsf{r}\rangle$ by locally sampling each of its shares (via Π_{rand}). Parties locally compute $[\mathsf{r}]$ and $\langle\!\langle\mathsf{r}\rangle\!\rangle$ from $\langle\mathsf{r}\rangle$ using $\Pi_{\langle\cdot\rangle\to[\cdot]}$ and $\Pi_{\langle\cdot\rangle\to\langle\!\langle\cdot\rangle\!\rangle}$, respectively. Looking ahead, $[\mathsf{r}]$ aids in generating additive shares of D, E, while $\langle\!\langle\mathsf{r}\rangle\!\rangle$ aids in computing $\langle\!\langle\mathsf{z}\rangle\!\rangle$ from $\langle\!\langle\mathsf{z}-\mathsf{r}\rangle\!\rangle$.

► *Step* ②: This step involves computing additive shares of $\Lambda_\mathsf{ab}-\mathsf{r}$ among all parties. For this, parties non-interactively generate $[\Lambda_\mathsf{ab}]$ from $\langle\lambda_\mathsf{a}\rangle, \langle\lambda_\mathsf{b}\rangle$ (via $\Pi_{\langle\cdot\rangle\cdot\langle\cdot\rangle\to[\cdot]}$). $P_i \in \mathcal{P}$ sets its additive share of $\Lambda_\mathsf{ab}-\mathsf{r}$ as $[\Lambda_\mathsf{ab}-\mathsf{r}]_i = [\Lambda_\mathsf{ab}]_i - [\mathsf{r}]_i$. Observe that the shares $[\Lambda_\mathsf{ab}-\mathsf{r}]_i$ of $P_i \in \mathcal{D}$ define the additive shares of $\mathsf{D} = (\Lambda_\mathsf{ab}-\mathsf{r})_\mathcal{D}$ among parties in $\mathcal{D}$. Similarly, the shares $[\Lambda_\mathsf{ab}-\mathsf{r}]_i$ of $P_i \in \mathcal{E}$ define the additive shares of $(\Lambda_\mathsf{ab}-\mathsf{r})_\mathcal{E}$ among parties in $\mathcal{E}$ (i.e. ${}^{\mathcal{E}}[(\Lambda_\mathsf{ab}-r)_\mathcal{E}]$).

► *Step* ③: Parties in $\mathcal{E}$ generate additive shares of $\lambda_\mathsf{a}, \lambda_\mathsf{b}$ among themselves (${}^{\mathcal{E}}[\cdot]$-shares, via $\Pi_{\langle\cdot\rangle\to\mathcal{E}[\cdot]}$). Looking ahead, ${}^{\mathcal{E}}[\lambda_\mathsf{a}], {}^{\mathcal{E}}[\lambda_\mathsf{b}]$ aid in generating additive shares of E among $\mathcal{E}$.

► *Step* ④: Parties in $\mathcal{D}$ send their additive shares of D (as defined in step ②) to P_{king}, who reconstructs D.

► *Step* ⑤: $P_i \in \mathcal{E} \setminus \{P_{\mathsf{king}}\}$ non-interactively generates additive share, ${}^{\mathcal{E}}[\mathsf{E}]_i$, of E among parties in $\mathcal{E}$ as ${}^{\mathcal{E}}[\mathsf{E}]_i = -\mathsf{m_a}\,{}^{\mathcal{E}}[\lambda_\mathsf{b}]_i - \mathsf{m_b}\,{}^{\mathcal{E}}[\lambda_\mathsf{a}]_i + {}^{\mathcal{E}}[(\Lambda_\mathsf{ab}-\mathsf{r})_\mathcal{E}]_i$. Note that it suffices for only one designated party in $\mathcal{E}$ to add $\mathsf{M_{ab}}$ in its share of ${}^{\mathcal{E}}[\mathsf{E}]$, and without loss of generality we let this designated party be P_{king}. For $P_{\mathsf{king}} = P_{t+1}$ in our case, ${}^{\mathcal{E}}[\mathsf{E}]_{t+1} = \mathsf{M_{ab}} - \mathsf{m_a}\,{}^{\mathcal{E}}[\lambda_\mathsf{b}]_{t+1} - \mathsf{m_b}\,{}^{\mathcal{E}}[\lambda_\mathsf{a}]_{t+1} + {}^{\mathcal{E}}[(\Lambda_\mathsf{ab}-\mathsf{r})_\mathcal{E}]_{t+1}$. Parties send their additive shares of E to P_{king}, who reconstructs E, and sends $\mathsf{z}-\mathsf{r} = \mathsf{D}+\mathsf{E}$ to parties in $\mathcal{E}$.

► *Step* ⑥: Parties non-interactively generate $\langle\!\langle\mathsf{z}-\mathsf{r}\rangle\!\rangle$ (via $\Pi_{\cdot\to\langle\!\langle\cdot\rangle\!\rangle}$) as explained earlier. Using $\langle\!\langle\mathsf{r}\rangle\!\rangle$ generated in step ①, parties compute $\langle\!\langle\mathsf{z}\rangle\!\rangle = \langle\!\langle\mathsf{z}-\mathsf{r}\rangle\!\rangle + \langle\!\langle\mathsf{r}\rangle\!\rangle$, as required.

Protocol $\Pi_{\mathsf{mult}}(\mathcal{P}, \langle\!\langle\mathsf{a}\rangle\!\rangle, \langle\!\langle\mathsf{b}\rangle\!\rangle, \mathsf{isTr})$

$\mathsf{isTr} = 1$ denotes perform truncation, $\mathsf{isTr} = 0$ denotes otherwise.

Preprocessing:

① If $\mathsf{isTr} = 0$: invoke Π_{rand} to generate $\langle\mathsf{r}\rangle$ where $\mathsf{r} \in \mathbb{Z}_{2^\ell}$. Invoke $\Pi_{\langle\cdot\rangle\to[\cdot]}$ and $\Pi_{\langle\cdot\rangle\to\langle\!\langle\cdot\rangle\!\rangle}$ on $\langle\mathsf{r}\rangle$ to generate $[\mathsf{r}]$ and $\langle\!\langle\mathsf{r}\rangle\!\rangle$, respectively.

- Else, invoke $\Pi_{\mathsf{dsBits}}(\mathcal{P}, 1)$ (Fig. 22) to generate $\langle\!\langle\mathsf{r}\rangle\!\rangle, \langle\!\langle\mathsf{r}^d\rangle\!\rangle$, and $\Pi_{\langle\!\langle\cdot\rangle\!\rangle\to[\cdot]}$ on $\langle\!\langle\mathsf{r}\rangle\!\rangle$ to generate $[\mathsf{r}]$.

② Invoke $\Pi_{\langle\cdot\rangle\cdot\langle\cdot\rangle\to[\cdot]}$ on $\langle\lambda_\mathsf{a}\rangle, \langle\lambda_\mathsf{b}\rangle$ to generate $[\Lambda_\mathsf{ab}]$, and compute $[\Lambda_\mathsf{ab}-\mathsf{r}] = [\Lambda_\mathsf{ab}] - [\mathsf{r}]$.

- $P_i \in \mathcal{E}$ sets ${}^{\mathcal{E}}[(\Lambda_\mathsf{ab}-r)_\mathcal{E}]_i = [\Lambda_\mathsf{ab}-r]_i$.

③ $P_i \in \mathcal{E}$ invokes $\Pi_{\langle\cdot\rangle\to\mathcal{E}[\cdot]}$ on $\langle\lambda_\mathsf{a}\rangle, \langle\lambda_\mathsf{b}\rangle$ to generate ${}^{\mathcal{E}}[\lambda_\mathsf{a}]_i, {}^{\mathcal{E}}[\lambda_\mathsf{b}]_i$, respectively.

④ $P_i \in \mathcal{D}$ sends $[\Lambda_\mathsf{ab}-\mathsf{r}]_i$ to P_{king}, who sets $\mathsf{D} = \sum_{i:P_i\in\mathcal{D}}[\Lambda_\mathsf{ab}-\mathsf{r}]_i$.

Online:

⑤ $P_i \in \mathcal{E}$ computes ${}^{\mathcal{E}}[\zeta]_i = -\mathsf{m_a}\,{}^{\mathcal{E}}[\lambda_\mathsf{b}]_i - \mathsf{m_b}\,{}^{\mathcal{E}}[\lambda_\mathsf{a}]_i + {}^{\mathcal{E}}[(\Lambda_\mathsf{ab}-\mathsf{r})_\mathcal{E}]_i$, and sends ${}^{\mathcal{E}}[\zeta]_i$ to P_{king}.

- P_{king} computes $\mathsf{E} = \mathsf{M_{ab}} + \sum_{i:P_i\in\mathcal{E}} {}^{\mathcal{E}}[\zeta]_i$ and sends $\mathsf{z}-\mathsf{r} = \mathsf{D}+\mathsf{E}$ to all parties in $\mathcal{E}$.

⑥ If $\mathsf{isTr} = 0$: invoke $\Pi_{\cdot\to\langle\!\langle\cdot\rangle\!\rangle}$ on $\mathsf{z}-\mathsf{r}$ to generate $\langle\!\langle\mathsf{z}-\mathsf{r}\rangle\!\rangle$, and compute $\langle\!\langle\mathsf{z}\rangle\!\rangle = \langle\!\langle\mathsf{z}-\mathsf{r}\rangle\!\rangle + \langle\!\langle\mathsf{r}\rangle\!\rangle$.

- Else, invoke $\Pi_{\cdot\to\langle\!\langle\cdot\rangle\!\rangle}$ on $(\mathsf{z}-\mathsf{r})^d$ to generate $\langle\!\langle(\mathsf{z}-\mathsf{r})^d\rangle\!\rangle$, and compute $\langle\!\langle\mathsf{z}^d\rangle\!\rangle = \langle\!\langle(\mathsf{z}-\mathsf{r})^d\rangle\!\rangle + \langle\!\langle\mathsf{r}^d\rangle\!\rangle$.

Fig. 4: Semi-honest: Multiplication protocol

Lemma III.1. *Protocol* Π_{mult} *(Fig. 4) incurs a communication of* t *elements in the preprocessing phase and* $2t$ *elements in* 2 *rounds in the online phase for multiplication when* $\mathsf{isTr} = 0$.

Analysis: Observe that the communication towards P_{king} in steps ④ and ⑤, can be performed in parallel, resulting in the overall round complexity of the protocol being two. Further, a communication of t elements is required in step ④ and $2t$ elements is required in ⑤ (since $P_{\mathsf{king}} \in \mathcal{E}$), thereby having a total communication complexity of $3t$ ring elements. This complexity resembles that of DN07. However, our sharing semantics enables us to push some of the steps mentioned above to a preprocessing phase, resulting in a fast online phase, which is non-trivial to achieve in the case of DN07. Elaborately, observe that since $\mathsf{r}, \lambda_\mathsf{a}, \lambda_\mathsf{b}$ are independent of the input (owing to our sharing semantics), computation involving these terms ranging from steps ① to ④ can thus be moved to a preprocessing phase. This improves the online communication complexity by slashing the inward communication towards P_{king} by half. Thus, the online phase requires only $2t$ ring elements of communication while offloading t elements of communication to the preprocessing phase.

Note that a straightforward extension of semi-honest multiplication of [31] to the preprocessing model, which can be derived from [38], does not provide an efficient solution. Although such a protocol has the same online complexity ($2t$

elements) as our online phase, it has the drawback of inflating the overall communication cost by a factor of $1.6\times$ over [31]. Elaborately, the online communication cost of $2t$ elements can be attained by appropriately defining the sharing semantics and using the P_{king} approach, similar to our protocol. However, this requires parties to generate the sharing of $\Lambda_{\mathsf{ab}} = \lambda_{\mathsf{a}} \cdot \lambda_{\mathsf{b}}$ from the shares of λ_{a} and λ_{b} during the preprocessing phase, and requires a full-fledged multiplication, incurring a cost of $3t$ elements. This yields a protocol with a total cost of $5t$ elements in comparison to the $3t$ cost of the all-online DN07 protocol. Thus, departing from this approach, the novelty of our protocol lies in leveraging the interplay between the sharing semantics and redesigning the communication pattern among the parties to ensure that the total cost of $3t$ does not change.

Furthermore, our protocol design allows parties in $\mathcal{D}$ to remain shut in the online phase, thereby reducing the system's operational load. This is because parties in $\mathcal{D}$ only contribute towards the computation of D, which can be completed in the preprocessing phase. However, the preprocessing phase becomes function-dependent due to the linear gates, for which the λ value for the output wires cannot be chosen randomly. Concretely, if c is the output of a linear gate, say addition, with inputs a, b, then λ_{c} cannot be chosen randomly and should be defined as $\lambda_{\mathsf{c}} = \lambda_{\mathsf{a}} + \lambda_{\mathsf{b}}$.

Ideal functionality $\mathcal{F}_{\mathsf{n-PC}}$ for evaluating function f in the n-party setting with semi-honest security appears in Fig. 5.

Functionality $\mathcal{F}_{\mathsf{n-PC}}$

$\mathcal{F}_{\mathsf{n-PC}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}^{\mathsf{sh}}$. Let f denote the function to be computed. Let x_s be the input corresponding to the party P_s, and y_s be the corresponding output, i.e $(\{\mathsf{y}_s\}_{s=1}^n) = f(\{\mathsf{x}_s\}_{s=1}^n)$.

Step 1: $\mathcal{F}_{\mathsf{n-PC}}$ receives $(\mathsf{Input}, \mathsf{x}_s)$ from $P_s \in \mathcal{P}$, & computes $(\{\mathsf{y}_s\}_{s=1}^n) = f(\{\mathsf{x}_s\}_{s=1}^n)$.

Step 2: $\mathcal{F}_{\mathsf{n-PC}}$ sends $(\mathsf{Output}, \mathsf{y}_s)$ to $P_s \in \mathcal{P}$.

Fig. 5: Semi-honest: Ideal functionality for function f

c) Incorporating truncation: To retain FPA semantics, it is required to truncate the result of multiplication, $\mathsf{z} = \mathsf{ab}$, which ends up having $2d$ bits in the fractional part, by d bits, i.e. compute $\mathsf{z}^d = \mathsf{z}/2^d$. For this, we extend the *probabilistic* truncation technique of [68], [56], [57] proposed in the small party domain to the n-party setting. Given $(\mathsf{r}, \mathsf{r}^d)$-pair, with $\mathsf{r}^d = \mathsf{r}/2^d$, the truncated value of z can be obtained as $\mathsf{z}^d = (\mathsf{z} - \mathsf{r})^d + \mathsf{r}^d$. Accuracy and correctness of this method follows from [68], [66].

Functionality $\mathcal{F}_{\mathsf{TrGen}}$

– Samples random $\mathsf{r} \in \mathbb{Z}_{2^\ell}$, and computes $\mathsf{r}^d = \mathsf{r}/2^d$.

– Generates $\langle\!\langle \cdot \rangle\!\rangle$-shares of r, r^d and set output share for $P_s \in \mathcal{P}$ as $\mathsf{y}_s = \{\langle\!\langle \mathsf{r} \rangle\!\rangle_s, \langle\!\langle \mathsf{r}^d \rangle\!\rangle_s\}$.

Output: Send $(\mathsf{Output}, \mathsf{y}_s)$ to $P_s \in \mathcal{P}$.

Fig. 6: Ideal functionality $\mathcal{F}_{\mathsf{TrGen}}$

Our multiplication protocol can be modified to additionally perform truncation by incorporating the following two changes– (i) generate $\langle\!\langle \mathsf{r}^d \rangle\!\rangle$ in step ①, and (ii) compute $\langle\!\langle \mathsf{z}^d \rangle\!\rangle = \langle\!\langle (\mathsf{z} - \mathsf{r})^d \rangle\!\rangle + \langle\!\langle \mathsf{r}^d \rangle\!\rangle$, instead, in step ⑥. For (i), we rely on the ideal functionality, $\mathcal{F}_{\mathsf{TrGen}}$ (Fig. 6), for computing $\langle\!\langle \mathsf{r} \rangle\!\rangle, \langle\!\langle \mathsf{r}^d \rangle\!\rangle$. $\mathcal{F}_{\mathsf{TrGen}}$ can be instantiated using the appropriate MPC protocol which will be used as a black-box in our multiplication. Thus, improvements in the MPC protocol that realizes $\mathcal{F}_{\mathsf{TrGen}}$ can be inherited in our multiplication protocol. In our work, we instantiate $\mathcal{F}_{\mathsf{TrGen}}$ using Π_{dsBits} (Fig. 22), which is a slightly modified version of the doubly-shared random bit generation protocol of [29], adapted to our n-party setting. Concretely, Π_{dsBits} generates ℓ doubly-shared random bits instead of a single bit, as done in the protocol of [29]. Here, a doubly-shared random bit is a bit which is arithmetic as well as Boolean shared. We defer the details of Π_{dsBits} to §B-A since it follows easily from the protocol of [29]. With respect to (ii), observe that it is a local operation, and hence performing truncation does not incur any additional overhead in the online phase.

d) Dot product: Given $\langle\!\langle \cdot \rangle\!\rangle$-shares of vectors $\vec{x}$ and $\vec{y}$ of size n, dot product outputs $\langle\!\langle \mathsf{z} \rangle\!\rangle$ where $\mathsf{z} = \vec{x} \odot \vec{y} = \sum_{k=1}^{\mathsf{n}} \mathsf{x}_k \mathsf{y}_k$ and $\odot$ denotes the dot product operation. The design of our multiplication protocol enables easy extension to support dot product computation without incurring any overhead. Concretely, similar to multiplication,

$$\begin{aligned} \mathsf{z} - \mathsf{r} &= (\vec{x} \odot \vec{y}) - \mathsf{r} \\ &= \sum_{k=1}^{\mathsf{n}} \mathsf{M}_{\mathsf{x}_k \mathsf{y}_k} - \sum_{k=1}^{\mathsf{n}} \mathsf{m}_{\mathsf{x}_k} \lambda_{\mathsf{y}_k} - \sum_{k=1}^{\mathsf{n}} \mathsf{m}_{\mathsf{y}_k} \lambda_{\mathsf{x}_k} + \sum_{k=1}^{\mathsf{n}} \Lambda_{\mathsf{x}_k \mathsf{y}_k} - \mathsf{r} \end{aligned} \tag{2}$$

In each of the summands of $\mathsf{z} - \mathsf{r}$, each of the n product terms can be generated similar to that in the multiplication protocol, which can then be locally summed up before sending it towards P_{king}. Due to this simple extension, we defer the formal dot product protocol (Fig. 23) to §B-A. Looking ahead, for matrix multiplication, each element of the resultant matrix can be computed via a dot product.

e) Multi input multiplication: 3-input and 4-input multiplication protocols have showcased their wide applicability in improving the online phase complexity [57], [73], [71]. Concretely, computing $\mathsf{z} = \mathsf{abc}$ (3-input) or $\mathsf{z} = \mathsf{abcd}$ (4-input) naively requires at least two sequential invocations of 2-input multiplication protocol in the online phase. Instead, 3-input and 4-input multiplication protocol, respectively, enables performing this computation with the same online complexity as that of a *single* 2-input multiplication. Thus, we design 3-input and 4-input multiplication protocols by extending the techniques of [73], [57] to the n-party setting. Designing these protocols require modifications in the preprocessing steps. Consider 3-input multiplication where the goal is to generate $\langle\!\langle \cdot \rangle\!\rangle$-sharing of $\mathsf{z} = \mathsf{abc}$ given $\langle\!\langle \mathsf{a} \rangle\!\rangle, \langle\!\langle \mathsf{b} \rangle\!\rangle, \langle\!\langle \mathsf{c} \rangle\!\rangle$. Note that

$$\begin{aligned} \mathsf{z} - \mathsf{r} &= \mathsf{abc} - \mathsf{r} = (\mathsf{m}_{\mathsf{a}} - \lambda_{\mathsf{a}})(\mathsf{m}_{\mathsf{b}} - \lambda_{\mathsf{b}})(\mathsf{m}_{\mathsf{c}} - \lambda_{\mathsf{c}}) - \mathsf{r} \\ &= \mathsf{M}_{\mathsf{abc}} - \mathsf{M}_{\mathsf{ac}} \lambda_{\mathsf{b}} - \mathsf{M}_{\mathsf{bc}} \lambda_{\mathsf{a}} - \mathsf{M}_{\mathsf{ab}} \lambda_{\mathsf{c}} \\ &\quad + \mathsf{m}_{\mathsf{a}} \Lambda_{\mathsf{bc}} + \mathsf{m}_{\mathsf{b}} \Lambda_{\mathsf{ac}} + \mathsf{m}_{\mathsf{c}} \Lambda_{\mathsf{ab}} - \Lambda_{\mathsf{abc}} - \mathsf{r} \end{aligned}$$

We follow an approach closely related to 2-input multiplication, with the difference being that parties additionally require to generate the additive sharing of $\Lambda_{\mathsf{bc}}, \Lambda_{\mathsf{ac}}$ and Λ_{abc} during preprocessing. Given these sharings, parties proceed with a similar online phase as in Π_{mult} to compute the 3-input multiplication without inflating the online cost. Similarly, for

4-input multiplication, parties need to generate the additive sharing of $\Lambda_{ad}, \Lambda_{bd}, \Lambda_{cd}, \Lambda_{abd}, \Lambda_{acd}, \Lambda_{bcd}, \Lambda_{abcd}$ in addition to those required in the case of 3-input multiplication. The generation of these sharings follows a similar approach as the 2-input multiplication, and the details are deferred to §B-A. Table III compares the cost of computing $z = abc$ via a 2-input multiplication sequentially vs a 3-input multiplication, and computing $z = abcd$ via a 2-input and 4-input multiplication.

Multiplication type	Building Block	Communication		Online Rounds
		Prep.	Online	
$z = abc$	2-input mult.	$2t\ell$	$4t\ell$	4
	3-input mult.	$6t\ell$	$2t\ell$	2
$z = abcd$	2-input mult.	$3t\ell$	$6t\ell$	4
	4-input mult.	$15t\ell$	$2t\ell$	2

TABLE III: Semi-honest: Communication and round complexity for computing multi-input multiplications

The recent work of [46] provides a method to reduce the round complexity of circuit evaluation. They group the (distinct) consecutive layers in the circuit into pairs and perform a parallel evaluation of all gates in the two layers in a group. Consider a multiplication gate with inputs x, y (obtained as output from a previous layer) and output z. Their approach considers three cases: (i) if x and y are not the outputs of a multiplication gate, (ii) exactly one among x, y is the output of a multiplication gate, and (iii) both x, y are outputs of a multiplication gate. We observe that case (ii) and (iii) in their approach resembles multi-input multiplication, which allows evaluating the second layer of multiplication ($z = x \cdot y$) non-interactively, thereby saving on rounds.

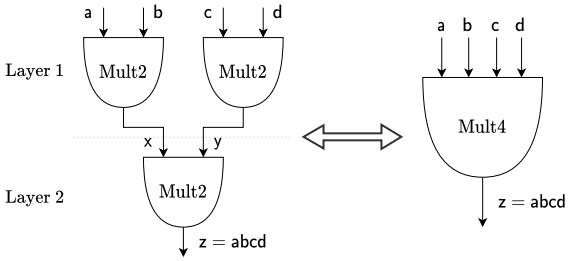


Fig. 7: 4-input multiplication

For instance, consider a 2-layer sub-circuit as in Fig. 7 where $x = a \cdot b, y = c \cdot d$ are outputs of a multiplication gate which are fed as input to a multiplication gate in the next level. The approach of [46] allows computation of $z = (a \cdot b) \cdot (c \cdot d)$ in a single shot, which is equivalent to computing z via a 4-input multiplication in our case. Similarly, when only one of the inputs (either x or y) is the output of multiplication, computation of $z = x \cdot y$ resembles a 3-input multiplication. Thus, cases (i), (ii), (iii) correspond to 2-input, 3-input, and 4-input multiplication, respectively, in our work and are sufficient to reduce round complexity of any circuit evaluation by half. Hence, we restrict our focus to 3 and 4-input multiplication.

IV. Extending to Malicious Security

Using standard approaches [74], [56], [38], it is straightforward to adapt the semi-honest protocols such as input sharing and output reconstruction to the malicious setting. The details are provided in §B-B for completeness. Hence, in this section, we focus on the challenges encountered and their resolutions for obtaining a maliciously secure *multiplication* protocol.

Note that although a maliciously secure multiplication protocol can be achieved by compiling our semi-honest protocol using compiler techniques such as [1], [15], the resultant protocol has an expensive online phase. For instance, using the compiler of [1] yields a protocol that requires computation over *extended* rings and communicating $4t$ extended ring elements in the online phase. This is not favourable compared to working over plain rings, especially in the online phase. Further, compilers such as those in [15] require heavy computational machinery like reliance on zero-knowledge proofs in the online phase, which is also not desirable. Thus, to attain a computation and communication efficient online phase, departing from the aforementioned compiler-based approaches, we design a maliciously secure multiplication protocol that requires communicating $3t$ *ring* elements in each phase. It is worth noting that we can do this while retaining the benefits of requiring only $t+1$ parties in the online phase (for most of the computation). The remaining t parties are required to come online only for a short one-time verification phase, that is deferred to the end of the computation. Deferring verification may result in a privacy breach [47]. However, we describe later why the privacy breach does not arise in our protocol.

To enable generation of $\langle\!\langle z \rangle\!\rangle = \langle\!\langle ab \rangle\!\rangle$ from $\langle\!\langle a \rangle\!\rangle$ and $\langle\!\langle b \rangle\!\rangle$, we retain the high-level ideas from the semi-honest protocol. Our task reduces to (i) generating additive shares of Λ_{ab} among parties in $\mathcal{E}$ (i.e. ${}^{\mathcal{E}}[\Lambda_{ab}]$) given $\langle \lambda_a \rangle$ and $\langle \lambda_b \rangle$, in the preprocessing phase, and (ii) reconstructing $z - r$ in the online phase. Given (i), computing ${}^{\mathcal{E}}[z - r]$ in the online phase is a local operation. Given (ii), parties can invoke $\Pi_{\rightarrow \langle\!\langle \cdot \rangle\!\rangle}$ to generate $\langle\!\langle z - r \rangle\!\rangle$, and compute $\langle\!\langle z \rangle\!\rangle = \langle\!\langle z - r \rangle\!\rangle + \langle\!\langle r \rangle\!\rangle$, where $\langle\!\langle r \rangle\!\rangle$ is generated in the preprocessing phase, as discussed in the semi-honest case.

Functionality $\mathcal{F}_{\mathsf{MulPre}}$

$\mathcal{F}_{\mathsf{MulPre}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$. Let $\mathcal{T}_i$ be the set of the honest parties.

Input: $\mathcal{F}_{\mathsf{MulPre}}$ receives the $\langle \cdot \rangle$-shares of a, b from the parties. It also receives $\langle \cdot \rangle$-shares of $z = ab$ of corrupt parties from $\mathcal{S}$. $\mathcal{S}$ is also allowed to send a special command, $(\mathtt{abort}, P)$, which indicates that parties in $\mathcal{P}$ with indices in P should `abort`.

$\mathcal{F}_{\mathsf{MulPre}}$ proceeds as follows.

– Reconstruct a, b using the shares received from honest parties, and compute $z = ab$.

– Compute the $\langle \cdot \rangle$-share of z to be held by the set of honest parties as the difference between z and the sum of $\langle \cdot \rangle$-shares of z received from corrupt parties.

– Let y_s denote the $\langle \cdot \rangle$-shares of z for party $P_s \in \mathcal{P}$. If received $(\mathtt{abort}, P)$ from $\mathcal{S}$, set $y_s = \mathtt{abort}$ for P_s, where $s \in P$.

Output: Send (Output, y_s) to every $P_s \in \mathcal{P}$.

Fig. 8: Ideal functionality $\mathcal{F}_{\mathsf{MulPre}}$

For task (i), our idea for the semi-honest case, of making parties in $\mathcal{D}$ to send their shares to P_{king}, does not work in the presence of a malicious adversary. To address this, we make black-box use of a maliciously secure multiplication protocol, abstracted as a functionality $\mathcal{F}_{\mathsf{MulPre}}$ in Fig. 8, that

computes $\langle \Lambda_{\mathsf{ab}} \rangle$ from $\langle \lambda_{\mathsf{a}} \rangle, \langle \lambda_{\mathsf{b}} \rangle$. In this work, we instantiate $\mathcal{F}_{\mathsf{MulPre}}$ with the state-of-the-art multiplication protocol of [15] that provides `abort` security. Note that although the protocol of [15] relies on zero-knowledge proofs, this computation is carried out in the preprocessing phase of our multiplication protocol. Moreover, since preprocessing is done for many instances in one shot, the zero-knowledge proof can benefit from amortization. The parties then invoke $\Pi_{\langle\cdot\rangle \to \mathcal{E}[\cdot]}$ to obtain ${}^{\mathcal{E}}[\Lambda_{\mathsf{ab}}]$ from $\langle \Lambda_{\mathsf{ab}} \rangle$. Looking ahead, $\langle \Lambda_{\mathsf{ab}} \rangle$ also aids in performing the online verification check.

For task (ii), in the online phase, we retain the idea of parties in $\mathcal{E}$ optimistically reconstructing $\mathsf{z}-\mathsf{r}$ from their additive shares (${}^{\mathcal{E}}[\cdot]$-share) to ensure that only the parties in $\mathcal{E}$ remain active for most of the computation. Moreover, this optimistic reconstruction requires only $\mathcal{O}(t)$-element communication rather than the $\mathcal{O}(t^2)$ required for reconstruction from $\langle\cdot\rangle$-shares (which is what will be used later for performing verification, albeit to perform only one such reconstruction). Thus, similar to the semi-honest protocol, parties in $\mathcal{E}$ optimistically reconstruct $\mathsf{z}-\mathsf{r}$ towards P_{king}, who further sends the reconstructed value to the parties in $\mathcal{E}$. In the malicious setting, this approach requires additional care since a malicious party may send a wrong ${}^{\mathcal{E}}[\cdot]$-share of $\mathsf{z}-\mathsf{r}$ to P_{king} or a malicious P_{king} may send an incorrectly reconstructed (inconsistent) $\mathsf{z}-\mathsf{r}$ to the parties. To account for these behaviours, the protocol is augmented with a short one-off verification phase to verify the consistency and correctness of $\mathsf{z}-\mathsf{r}$. This phase is executed in the end of the protocol and requires the presence of *all* parties, and hence the possession of $\mathsf{z}-\mathsf{r}$ by all. This is in contrast to the semi-honest protocol where $\mathsf{z}-\mathsf{r}$ is given to only parties in $\mathcal{E}$. To keep $\mathcal{D}$ disengaged for most of the online phase, sending $\mathsf{z}-\mathsf{r}$ to them is deferred till the end of the protocol. This send is a one-off and can be combined for all multiplication gates. Details of verification protocol Π_{Vrfy} (Fig. 9) are given next.

Verification comprises two checks– a *consistency* check to first verify that P_{king} has indeed sent the same $\mathsf{z}-\mathsf{r}$ to all the parties, followed by a *correctness* check to verify the correctness of the $\mathsf{z}-\mathsf{r}$. For the former, parties perform a hash-based consistency check of $\mathsf{z}-\mathsf{r}$, and abort in case of any inconsistency. If $\mathsf{z}-\mathsf{r}$ is consistent, parties verify its correctness. The high-level idea for verifying correctness is to *robustly* reconstruct $\mathsf{z}-\mathsf{r}$, but now from its $\langle\cdot\rangle$-shares (can be computed given $\langle \lambda_{\mathsf{a}} \rangle, \langle \lambda_{\mathsf{a}} \rangle, \langle \Lambda_{\mathsf{ab}} \rangle$ that are generated in the preprocessing phase). Parties can then verify if this reconstructed value equals the value received from P_{king}. Concretely, this is equivalent to robustly reconstructing $\langle \Omega \rangle = \langle \mathsf{z} - \mathsf{r} - (\mathsf{M}_{\mathsf{ab}} - \mathsf{m}_{\mathsf{a}}\lambda_{\mathsf{b}} - \mathsf{m}_{\mathsf{b}}\lambda_{\mathsf{a}} + \Lambda_{\mathsf{ab}} - \mathsf{r}) \rangle$, where $\mathsf{z}-\mathsf{r}$ is the value received from P_{king}, and verifying if $\Omega = 0$. For robust reconstruction of $\langle \Omega \rangle$, every party sends its $\langle\cdot\rangle$-share to every other party who misses this share, and `aborts` in case of inconsistencies in the received values. Elaborately, reconstruction of Ω towards $P_s \in \mathcal{P}$ proceeds as follows. For each missing $\langle\cdot\rangle$-share of Ω at P_s, each of the $t+1$ parties holding this share send it to P_s. P_s uses this share for reconstruction if all the $t+1$ received values are consistent, else it `aborts`. Presence of at least one honest party among the $t+1$ guarantees that inconsistency, if any, can be detected. Since each share in $\langle \Omega \rangle$ is held by $t+1$ parties, comprising at least one honest party, any cheating by up to t corrupt parties is guaranteed to be detected. Note that this reconstruction requires communicating $\mathcal{O}(t^2)$ ring elements to verify the correct computation of a single multiplication gate, the cost of which can be optimized using standard optimization techniques [1], [23]. Concretely, the correctness of $\mathsf{z}-\mathsf{r}$ for several multiplication gates can be verified with a single reconstruction by reconstructing a linear combination of Ω for several gates and verifying equality with 0. Thus, only one robust reconstruction from $\langle\cdot\rangle$-shares is required for several multiplication gates, whose cost gets amortized due to verification across multiple gates.

Protocol $\Pi_{\mathsf{Vrfy}}\left(\mathcal{P}, \{\langle\!\langle \mathsf{a}_i \rangle\!\rangle, \langle\!\langle \mathsf{b}_i \rangle\!\rangle, \mathsf{z}_i - \mathsf{r}_i, \langle \Lambda_{\mathsf{a}_i\mathsf{b}_i} \rangle, \langle \mathsf{r}_i \rangle\}_{i=1}^{m}\right)$

Let $(\mathsf{a}_1, \mathsf{b}_1, \mathsf{z}_1), \ldots, (\mathsf{a}_m, \mathsf{b}_m, \mathsf{z}_m)$ denote the inputs and outputs of the m multiplication gates to be verified.

– *Consistency Check.* Invoke Π_{agree} on $\{\mathsf{z}_1 - \mathsf{r}_1, \ldots, \mathsf{z}_m - \mathsf{r}_m\}$.

– *Correctness Check.* Repeat the following κ times.

- Generate random $\theta_1, \ldots, \theta_m \in \mathbb{Z}_{2^\ell}$ and compute

$$\langle \Omega \rangle = \sum_{i=1}^{m} \theta_i \left(\mathsf{z}_i - \mathsf{r}_i - \left(\mathsf{M}_{\mathsf{a}_i\mathsf{b}_i} - \mathsf{m}_{\mathsf{a}_i} \langle \lambda_{\mathsf{b}_i} \rangle - \mathsf{m}_{\mathsf{b}_i} \langle \lambda_{\mathsf{a}_i} \rangle + \langle \Lambda_{\mathsf{a}_i\mathsf{b}_i} \rangle - \langle \mathsf{r}_i \rangle \right) \right)$$

- For each $\langle\cdot\rangle$-share of Ω, the $t+1$ parties possessing this share send it to every party that misses this share. If the recipient party receives inconsistent values for any missing share, it `aborts`.
- Reconstruct Ω and `abort` if $\Omega \neq 0$.

Fig. 9: Malicious: Verification protocol for all multiplication gates

It is worth noting that this random linear combination technique does not trivially work over rings. This is due to the existence of zero divisors which results in the linear combination being 0 with a probability $1/2$ (which denotes the cheating probability of the adversary) [1]. Hence, to obtain the desired security, the verification check is repeated κ times where κ is the security parameter. This bounds the cheating probability of adversary to $1/2^\kappa$. Another approach is to perform the verification over extended rings [13], [14]. Specifically, verification operations are carried out over a ring $\mathbb{Z}_{2^\ell}/f(x)$ which is a ring of all polynomials with coefficients in $\mathbb{Z}_{2^\ell}$ modulo a degree d polynomial $f(x)$ that is irreducible over $\mathbb{Z}_2$. Each element of $\mathbb{Z}_{2^\ell}$ is lifted to a degree d polynomial in $\mathbb{Z}_{2^\ell}[x]/f(x)$, which increases the communication required to perform verification by a factor of d.

To summarize, the maliciously secure multiplication protocol (see Fig. 27) can be broken down into the following.

– Preprocessing phase which involves generation of $\langle \Lambda_{\mathsf{ab}} \rangle$ by invoking $\mathcal{F}_{\mathsf{MulPre}}$. Malicious behaviour, if any, will be caught by $\mathcal{F}_{\mathsf{MulPre}}$. $\langle \Lambda_{\mathsf{ab}} \rangle$ is non-interactively converted into ${}^{\mathcal{E}}[\cdot]$-shares of λ_{ab}. ${}^{\mathcal{E}}[\lambda_{\mathsf{a}}], {}^{\mathcal{E}}[\lambda_{\mathsf{b}}]$ is also generated non-interactively.

– Generation of ${}^{\mathcal{E}}[\cdot]$-shares of $\lambda_{\mathsf{a}}, \lambda_{\mathsf{b}}, \Lambda_{\mathsf{ab}}$ during preprocessing enables computation of ${}^{\mathcal{E}}[\mathsf{z}-\mathsf{r}]$ in the online phase, and thereby reconstruction of $\mathsf{z}-\mathsf{r}$ via P_{king}. The crucial point to note here is that this requires the presence of only parties in $\mathcal{E}$ in the online phase. This is followed by non-interactive generation of $\langle\!\langle \mathsf{z}-\mathsf{r} \rangle\!\rangle$ from which $\langle\!\langle \mathsf{z} \rangle\!\rangle$ is computed as $\langle\!\langle \mathsf{z} \rangle\!\rangle = \langle\!\langle \mathsf{z}-\mathsf{r} \rangle\!\rangle + \langle\!\langle \mathsf{r} \rangle\!\rangle$, where $\langle\!\langle \mathsf{r} \rangle\!\rangle$ is generated during preprocessing.

– Finally, to catch malicious behaviour in the online phase, if any, in the verification phase the correctness of the generated $\langle\!\langle \mathsf{z} \rangle\!\rangle$ is checked simultaneously, for each z that is the output of a multiplication gate. This is done by invoking Π_{Vrfy}. Note that before this verification begins, P_{king} sends $\mathsf{z}-\mathsf{r}$ corresponding to all multiplication gates to parties in $\mathcal{D}$ in a single shot.

As pointed out in [47], deferring the correctness check to later may result in a privacy breach when using a sharing scheme that allows for redundancy (such as RSS or Shamir sharing). The details are elaborated in §B-B0c. However, the crucial point to note here is that although we rely on a variant of RSS which introduces redundancy, recall that while performing a reconstruction towards P_{king}, we rely on $^{\mathcal{E}}[\cdot]$-sharing of $\mathsf{z}-\mathsf{r}$, which is a $(t+1)$-additive sharing. The use of additive sharing while performing reconstruction towards P_{king} eliminates any redundancy in the sharing scheme and thus, helps in overcoming this subtle privacy breach, as also shown in [47]. This privacy breach persists in [38], and is discussed in §B-B0c.

Lemma IV.1. *Protocol* $\Pi^{\mathsf{M}}_{\mathsf{mult}}$ *(Fig. 27) incurs a communication of* $3t$ *elements in the preprocessing phase and* $3t$ *elements in* 2 *rounds in the online phase for multiplication when* $\mathsf{isTr}=0$.

The ideal functionality $\mathcal{F}_{\mathsf{n-PC}}$ for evaluating a function f in the n-party setting while providing malicious security (with `abort`) appears in Fig. 10.

Functionality $\mathcal{F}^{\mathsf{mal}}_{\mathsf{n-PC}}$

$\mathcal{F}_{\mathsf{n-PC}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}^{\mathsf{mal}}$. Let f denote the function to be computed. Let x_s be the input of party P_s, and y_s be the corresponding output, i.e $(\{\mathsf{y}_s\}^n_{s=1}) = f(\{\mathsf{x}_s\}^n_{s=1})$. $\mathcal{S}^{\mathsf{mal}}$ is also allowed to send a special command, $(\mathtt{abort}, \mathsf{P})$, which indicates that honest parties in $\mathcal{P}$ with indices in P should `abort`.

Step 1: $\mathcal{F}_{\mathsf{n-PC}}$ receives $(\mathsf{Input}, \mathsf{x}_s)$ from $P_s \in \mathcal{P}$. If $(\mathsf{Input}, *)$ already received from P_s, then ignore the current message. Otherwise, record $x'_s = x_s$ internally.

Step 2: Compute $(\{\mathsf{y}_s\}^n_{s=1}) = f(\{\mathsf{x}_s\}^n_{s=1})$ and send the output y_s for a corrupt P_s to $\mathcal{S}^{\mathsf{mal}}$.

Step 3: If received $(\mathsf{Signal}, \mathtt{abort}, \mathsf{P})$ from $\mathcal{S}^{\mathsf{mal}}$, set $\mathsf{y}_s = \mathtt{abort}$ for P_s, where $s \in \mathsf{P}$. Send $(\mathsf{Output}, \mathsf{y}_s)$ to honest $P_s \in \mathcal{P}$.

Fig. 10: Malicious: Ideal functionality for evaluating function f

a) Multiplication with truncation: Similar to the semi-honest protocol, truncation can be incorporated in the malicious multiplication as well without inflating the online communication. For this, we rely on maliciously secure ideal functionality, $\mathcal{F}^{\mathsf{M}}_{\mathsf{TrGen}}$ (Fig. 28), to generate the $\langle\langle\cdot\rangle\rangle$-shares of $(\mathsf{r}, \mathsf{r}^d)$ and is instantiated using $\Pi^{\mathsf{M}}_{\mathsf{dsBits}}$ [29] protocol in our work. On a high level, the semi-honest versions of interactive operations such as multiplication and reconstruction in Π_{dsBits} are replaced with their maliciously secure counterparts in $\Pi^{\mathsf{M}}_{\mathsf{dsBits}}$, and more details are provided in §B-B.

b) Dot product: Similar to the maliciously secure multiplication protocol that relied on $\mathcal{F}_{\mathsf{MulPre}}$ to generate $\langle\cdot\rangle$-shares of the multiplicative term, Λ_{ab} in the preprocessing phase, the maliciously secure dot product protocol invokes $\mathcal{F}_{\mathsf{DotPPre}}$ (Fig. 29) to generate $\langle\cdot\rangle$-shares of the multiplicative term, $\sum^{\mathsf{n}}_{k=1} \Lambda_{\mathsf{x}_k\mathsf{y}_k}$, required to compute the dot product as per equation (2). Given $\langle\cdot\rangle$-shares of $\sum^{\mathsf{n}}_{k=1} \Lambda_{\mathsf{x}_k\mathsf{y}_k}$, online phase proceeds similar to that of multiplication.

Observe that a trivial realization of $\mathcal{F}_{\mathsf{DotPPre}}$ can be reduced to n instances of multiplication. However, we extend the ideas from [56] and rely on a distributed zero-knowledge proof [15] to eliminate the vector-size dependency in the preprocessing phase. Concretely, we instantiate $\mathcal{F}_{\mathsf{DotPPre}}$ using a semi-honest dot product protocol [48] whose cost matches that of semi-honest multiplication [31] (and thus is independent of the vector-size), followed by a verification phase to verify the correctness of the dot product computation. For the verification, we extend the verification technique for multiplication in [15], to now verify the correctness of dot product, such that the cost due to verification can be amortized away for multiple dot products, thereby resulting in vector-size independent preprocessing. Details of this extension are deferred to §B-B.

c) Multi input multiplication: This protocol is similar to its semi-honest counterpart with the difference that the preprocessing phase relies on invoking $\mathcal{F}_{\mathsf{MulPre}}$ for generating the required multiplicative terms. The details are deferred to §B-B0d. Table IV compares the cost of computing multi-input multiplication via a 2-input multiplication sequentially vs. the multi-input multiplication protocol.

Multiplication type	Building Block	Communication		Online Rounds
		Prep.	Online	
$\mathsf{z}=\mathsf{abc}$	2-input mult.	$6t\ell$	$6t\ell$	4
	3-input mult.	$12t\ell$	$3t\ell$	2
$\mathsf{z}=\mathsf{abcd}$	2-input mult.	$9t\ell$	$9t\ell$	4
	4-input mult.	$33t\ell$	$3t\ell$	2

TABLE IV: Malicious: Communication and round complexity for computing multi-input multiplications

V. Applications & Benchmarks

To evaluate the performance of our protocols, we benchmark some of the popular applications such as deep neural networks (NN), graph neural networks (GNN), similar sequence queries (SSQ), and biometric matching where MPC is used to achieve privacy. While these applications have been looked at in the small party setting [70], [56], [81], [5], [79], [85], [73], [68], we believe the n-party setting is a better fit for reasons described in the introduction. To the best of our knowledge, we are the first to benchmark these in the multiparty honest-majority setting for more than four parties.

a) Benchmark environment: The performance of our protocols is analyzed using a prototype implementation building over the ENCRYPTO library [27] in C++17.

We chose 64 bit ring ($\mathbb{Z}_{2^{64}}$) for our arithmetic world, and the operations over extended ring were carried out using the NTL library[4]. Since the correctness and accuracy of the applications considered in the secure computation setting are already established, our benchmark aims to demonstrate our protocols' performance and is not fully functional. Moreover, we believe that incorporating state-of-the-art code optimizations like GPU-assisted computing can enhance the efficiency of our protocols, which is left as future work. Since there is no defined way to capture an adversary's misbehaviour, following standard practice [68], [56], [28], we benchmark honest executions of the protocols, which also include the steps performed for verification in the malicious case.

[4] https://libntl.org

We use multi-threading, wherever possible, to facilitate efficient computation and communication among the parties. The parties in the computation are emulated using Google Cloud (n1-standard-64 instances, 2.0 GHz Intel Xeon Skylake, 64 vCPUs, 240 GB RAM) with machines located in East Australia, South Asia, South East Asia, and West Europe. All our experiments are run for 5, 7, and 9 parties, each.

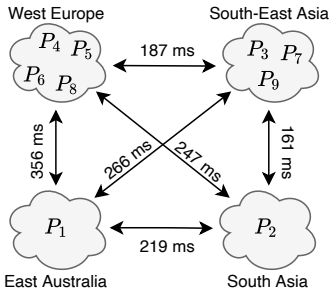


Fig. 11: Round trip time (rtt)

b) Benchmark parameters: We report the run-time and communication of the online phase and total (= preprocessing + online). To capture the effect of online round complexity and communication in one go, we also report the throughput (TP [4], [68], [56]) of the online phase. TP denotes the number of operations that can be performed in one minute. Finally, when deployed in the outsourced setting, one pays the price for the communication and up-time of the hired servers. To demonstrate how our protocols fare in this scenario, we additionally report the monetary cost (Cost) [67], [57] for the applications considered. This cost is estimated using Google Cloud Platform [80] pricing, where 1 GB and 1 hour of usage costs USD 0.08 and USD 3.04, respectively.

A. Comparison with DN07

In this section, we benchmark our semi-honest and malicious protocols over synthetic circuits comprising one million multiplications with varying depths of 1, 100, and 1000, and compare against the optimized ring variant of DN07 [13]. The gates are distributed equally across each level in the circuit.

a) Communication: The communication cost for 1 million multiplications is tabulated in Table V for the 5, 7, and 9 party settings. As can be observed, the online phase of our semi-honest protocol enjoys the benefits of pushing 33% communication to a preprocessing phase compared to DN07. The observed values corroborate the claimed improvement in the online complexity of our protocol. Our malicious protocol retains the online communication cost of DN07 while incurring a similar overhead in the preprocessing.

Ref.	$n = 5$	$n = 7$	$n = 9$
DN07 (semi)	(0, 45.78)	(0, 68.66)	(0, 91.55)
This (semi)	(15.26, 30.52)	(22.88, 45.78)	(30.51, 61.04)
This (mal)	(45.79, 45.78)	(68.67, 68.67)	(91.57, 91.57)

TABLE V: Communication (Preprocessing, Online) in MB for 1 million multiplications

Note that pushing the communication to the preprocessing phase has several benefits. First, communication with respect to several instances can happen in a single shot and leverage the benefit of serialization. Second, with respect to resource-constrained devices such as mobile phones, the preprocessing communication can occur whenever they have access to a high-bandwidth Wi-Fi network (for instance, when the device is at home overnight). These benefits facilitate a fast online phase, as observed, that may happen over a low-bandwidth network.

b) Run-time: The time taken to evaluate circuits of different depths appears in Table VI. Since the time for the 5, 7, and 9 party settings vary within the range [0, 0.5], we report values only for the 7-party setting in Table VI. With respect to the online run-time, our semi-honest protocol's time is expected to be similar to that of DN07. However, DN07 demonstrates around 1.5× higher run-time. This difference can be attributed to the asymmetry in the rtt among parties, which vanished when benchmarked over a symmetric rtt setting. Compared to the semi-honest protocol, the malicious variant incurs a minimal overhead of less than one second in the online run-time due to the one-time verification phase. However, the overhead is higher for the case of the overall run-time. Concretely, it is around 10 seconds and is due to the distributed zero-knowledge proof computation in the preprocessing phase. Note that this overhead is independent of the circuit depth and gets amortized for deeper circuits as evident from Table VI (depth 1 vs. 1000).

Ref.	$d = 1$	$d = 100$	$d = 1000$
DN07 (semi)	(0, 0.65)	(0, 54.97)	(0, 549.69)
This (semi)	(0.47, 0.45)	(0.47, 30.75)	(0.47, 307.48)
This (mal)	(10.52, 1.36)	(10.53, 68.67)	(10.54, 308.39)

TABLE VI: Latency in seconds (Preprocessing, Online) for varying depth (d) circuits with 1 million multiplications for $n = 7$

c) Monetary Cost: Another key highlight of our protocols is their improved monetary cost, as evident from Fig. 12. Concretely, for 9 parties (semi-honest), we observe a saving of 17% over DN07 for a depth-1 circuit, and it increases up to 72% for circuits with depth 1000. This is primarily due to the reduction in the number of online parties over DN07. Comparing our semi-honest and malicious variants, the latter has an overhead of 8× for depth-1 circuit, and it reduces to 1.14× for depth-1000 circuit. This is justified because the verification cost is amortized for deeper circuits, as mentioned earlier. Interestingly, our malicious variant outperforms even the semi-honest DN07 upon reaching circuit depths of 100 and above. A similar analysis holds in the symmetric rtt setting as well, where the saving is up to 56% (for $d = 1000$).

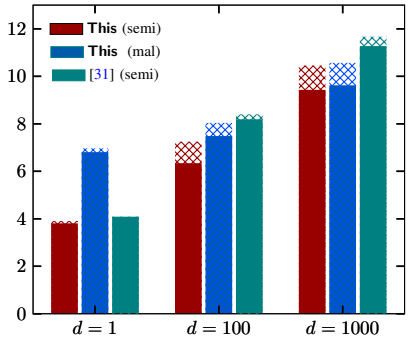
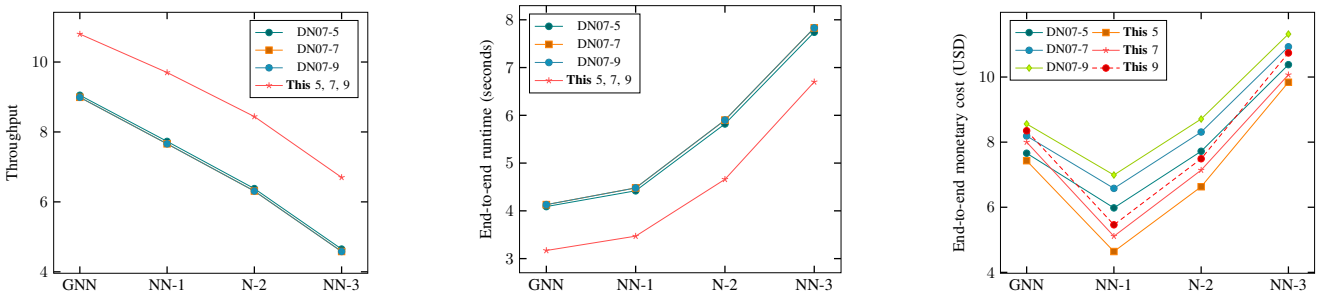


Fig. 12: Monetary cost (in USD) for evaluating circuits (1000 instances) of various depths (d) for $n = 9$ parties. The values are reported in $\log_2$ scale. Bars in solid colors denote computation over network given in Fig. 11, while the area represented via crosshatch pattern denotes the additional cost incurred in the symmetric rtt setting (356 ms).

d) Online Throughput (TP):: Owing to the asymmetric rtt as described earlier, our semi-honest variant witnesses up to 1.78× improvements in TP (for a single execution) over



(a) Online throughput (b) End-to-end runtime (c) End-to-end monetary cost (1000 queries)

Fig. 13: Comparison for GNN and deep NN between our semi-honest protocol and DN07 (values plotted are logarithmic in base 2)

DN07, which vanishes in the symmetric rtt setting. However, recall that our protocol requires only $t+1$ active parties in the online phase, which leaves several channels among the parties underutilized. Hence, we can leverage the load balancing technique where parties' roles are interchanged across various parallel executions. For instance, one approach is to make every party act as P_{king}, i.e., in 5PC, in one execution, $P_{\mathsf{king}} = P_1, \mathcal{E} = \{P_1, P_2, P_3\}, \mathcal{D} = \{P_4, P_5\}$, while in another execution $P_{\mathsf{king}} = P_2, \mathcal{E} = \{P_2, P_3, P_4\}, \mathcal{D} = \{P_5, P_1\}$, and so on. To analyse the effect of load balancing, we performed experiments with similar rtt among the parties and observed a 1.5× improvement in our semi-honest variant over DN07. This is justified as we communicate over four channels among the parties as opposed to six in DN07. We note that while enhancing the security from semi-honest to malicious, we observe a significant drop in TP, which is about 3× for the depth-1 circuit. This is primarily due to increased run time owing to the verification in online phase for malicious setting. However, this drop tends to zero for deeper circuits (as verification cost gets amortized), making online phase of our maliciously secure protocol on par with semi-honest one.

B. Deep Neural Networks (DNN) and Graph Neural Networks (GNN)

We benchmark three different neural networks (NN) [68], [74], [85] with increasing number of parameters–(i) NN-1: a 3-layer fully connected one from [70], (ii) NN-2: the LeNet [60] architecture, and (iii) NN-3: VGG16 [82] architecture (further details are deferred to §D-A0a). We benchmark the inference phase of the above NNs, which comprises computing activation matrices, followed by applying an activation function or pooling operation, depending on the network architecture. NN-1 and NN-2 are benchmarked over MNIST dataset [61] while NN-3 is benchmarked using CIFAR-10 dataset [58]. We also benchmark GNN inference, for which we use the simplified architecture of [35] given in [81]. This architecture (§D-A0b) is shown to achieve an accuracy of more than 99% on MNIST classification [81]. To analyse the improvement of our protocols, we also benchmark (semi-honest) DN07 for the applications by adapting our building blocks to their setting.

The semi-honest benchmarks for the different NNs and GNN appear in Table X (§D-A0a) while the malicious ones appear in Table XI (§D-A0a). Fig. 13 gives a pictorial view of the trends observed while comparing the semi-honest variants and are described next. We incur a very minimal overhead in the run-time of our protocols when moving from five to nine parties over all the networks considered. Hence, we use $\pm\delta$ to denote this variation in the table. The trends witnessed in synthetic circuit benchmarks (§V-A) carry forward to neural networks as well due to reasons discussed previously. For instance, the improvement in the online run-time for our semi-honest variant is up to 4.3× over DN07. The effect of reduced run-time and improved communication results in a significant improvement in online throughput of our protocol over DN07. Concretely, the gain ranges up to 4.3×. Further, the improved run-time coupled with the reduced number of online parties for our case brings in a saving of up to 69% in monetary cost for NN-1. However, the improvement drops to 33% for deep network NN-3. The reduction in savings is due to improved run-time getting nullified by increased communication from NN-1 to NN-3, making communication the dominant factor in determining monetary cost.

Observe that, unlike the case in synthetic circuits (Table V), the total communication here is an order of magnitude higher. This is primarily due to the higher communication cost incurred for performing the truncation operation–specifically, generation of the doubly-shared bits (Π_{dsBits}, Fig. 22) in the preprocessing phase. It is worth noting that Π_{dsBits} is used as a black-box, and an improved instantiation for it will lower the communication. Similar trends are observed for GNN as well, where the online run-time of DN07 is up to 3.5× higher than our semi-honest protocol. This is reflected in the throughput where we gain up to 3.4×. Further, we observe savings of up to 15% in monetary cost due to the reduced number of active parties and lesser run-time.

Moving to the malicious setting, we incur an overhead of up to 3% in online run-time, 6% in communication, and 13% in monetary cost over the semi-honest counterpart. Details are deferred to §D-A0a.

C. Genome Sequence Matching

Given a genome sequence as a query, genome matching aims to identify the most similar sequence from a database of sequences. This task is also known as similar sequence query (SSQ). It requires the computation of Edit Distance (ED), which quantifies how different two sequences are by identifying the minimum number of additions, deletions, and substitutions required to transform one sequence to the other. To compute the ED, we extend the (2-party) protocol from [79] which builds on top of the approximation from [5], to the n-party setting. The details of the approximation algorithm for ED computation appear in §D-C. The accuracy and correctness of this algorithm follow from [5]. Among the two phases of the ED algorithm, where the first phase happens non-interactively,

we only focus on the second phase of ED, which requires interaction and benchmark the same.

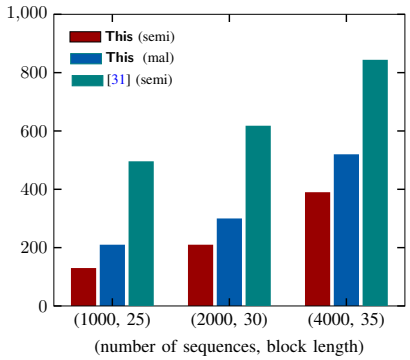


Fig. 14: Monetary cost for SSQ evaluation for varying number of sequences and block lengths ((1000,25), (2000, 30), (4000,35)) for $n = 9$ parties. Costs for 1000 instances are reported in USD.

The benchmarks for genome sequence matching appear in Table VII, Table XIII (§D-C). Following [79], we consider three cases with different number of sequences in the database (m) and different block lengths (ω). The benchmarks for $m = 2000, \omega = 30$ are reported in Table VII, while the ones for $m = 1000, \omega = 25$ and $m = 4000, \omega = 35$ appear in Table XIII. We witness similar trends here, where our semi-honest protocol has improvements of up to 4× in both online run-time and throughput over DN07. Our malicious variant incurs a minimal overhead in the range of 5-6% in online run-time and total communication over the semi-honest counterpart. For the monetary cost (Fig. 14), our semi-honest protocol has up to 66% saving over DN07, and malicious variant has around 42%-54% overhead over semi-honest counterpart.

Ref.	n	Online			End-to-end		
		Comm[a]	Time	TP[b]	Comm[c]	Time[d]	Cost[e]
DN07	5	25.82	66.33	57.89	0.40	82.24	0.31
	7	38.75	69.63	55.15	0.60	86.99	0.47
	9	51.67	69.66	55.09	0.80	87.39	0.62
This (semi)	5	15.39	17.61 ±.02	217.93 ±0.2	0.40	21.69 ±.02	0.11
	7	23.08			0.60		0.16
	9	30.78			0.80		0.21
This (mal)	5	22.79	18.3 ±.2	209.84	0.42	34.52 ±.2	0.17
	7	33.88		209.49	0.64		0.25
	9	44.06		207.23	0.85		0.30

[a]communication in MB [b]TP denotes throughput [c]communication in GB [d]Time in seconds [e]monetary cost in USD

TABLE VII: Genome sequence matching for $m = 2000, \omega = 30$.

D. Biometric Matching

We extend support for biometric matching, which finds application in many real-world tasks such as face recognition [37] and fingerprint matching [50]. The goal of such computation is to identify a sample from a database of m samples that is "closest" to a sample $\vec{u}$ held by a user. We follow the general trend and reduce the biometric matching problem to that of finding the sample from the database which has the least Euclidean Distance (**EuD**) with the user's sample $\vec{u}$. Details of the protocol are deferred to §D-B.

The benchmarks for biometric matching appear in Table VIII, Table XII (§D-B). The former table considers the case with 1024 and 65536 sequences in the database, while the latter considers 4096 and 16384 sequences. As is evident from Table VIII, our semi-honest protocol witnesses a 4.6× improvement over DN07 in both online run-time and throughput. Further, in terms of monetary cost, we observe a saving of around 85%. With respect to our maliciously secure protocol, we incur a minimal overhead of around 9.5% in terms of total communication and around 4% in online throughput over our semi-honest variant. We note that our malicious variant outperforms semi-honest DN07 in both online run-time and throughput, thereby achieving our goal of a fast online phase.

#seq	Ref.	n	Online			End-to-end		
			Comm	Time	TP[a]	Comm	Time	Cost[b]
1024	DN07	5	0.63	55.52	69.17	6.92	66.55	0.20
		7	0.94	58.27	65.90	10.38	69.32	0.30
		9	1.25	58.30	65.88	13.84	69.35	0.40
	This (semi)	5	0.09	12.61 ±.02	304.62 ±.03	6.93	14.79 ±.02	0.03
		7	0.13			10.40		0.05
		9	0.18			13.86		0.06
	This (mal)	5	0.14	13.43 ±.02	285.93 ±.2	7.61	26.67 ±.02	0.08
		7	0.21			11.42		0.11
		9	0.28			15.22		0.14
65536	DN07	5	40.14	88.64	43.32	443.34	108.53	0.40
		7	60.23	93.04	41.27	665.00	114.45	0.59
		9	80.31	93.10	41.16	886.67	114.55	0.79
	This (semi)	5	5.62	19.99 ±.02	192.09 ±.04	443.99	24.62 ±.1	0.13
		7	8.44			665.99		0.18
		9	11.25			887.99		0.23
	This (mal)	5	8.44	20.86 ±.02	183.88 ±.07	486.85	37.33 ±.05	0.18
		7	12.67			730.28		0.26
		9	16.89			972.72		0.33

Communication in MB and time in seconds.
[a]TP denotes throughput [b]monetary cost in USD

TABLE VIII: Benchmarks for biometric matching.

Conclusion

This work improves the practical efficiency of n-party honest-majority protocols using *function-dependent* preprocessing. While our first construction achieves a fast online phase compared to the semi-honest protocol of DN07, the second enhances security by tolerating malicious adversaries with minimal overhead in the online phase. The active participation of half of the participants in both of our constructions is a major highlight. This reduction in online parties results in monetary benefits in real-world deployments.

Acknowledgements

The authors would like to acknowledge support from Centre for Networked Intelligence (a Cisco CSR initiative) at the Indian Institute of Science, Bengaluru, SERB MATRICS (Theoretical Sciences) Grant 2020, Google India AI/ML Research Award 2020, DST National Mission on Interdisciplinary Cyber-Physical Systems (NM-CPS) 2020, National Security Council, India, and the support from Google Cloud to perform the benchmarking.

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850990 (PSOTI). This work was co-funded by the Deutsche Forschungsgemeinschaft (DFG) – SFB 1119 CROSSING/236615297.

REFERENCES

[1] M. Abspoel, A. P. K. Dalskov, D. Escudero, and A. Nof, "An efficient passive-to-active compiler for honest-majority MPC over rings," in *ACNS*, 2021.

[2] A. Aly, E. Orsini, D. Rotaru, N. P. Smart, and T. Wood, "Zaphod: Efficiently Combining LSSS and Garbled Circuits in SCALE," in *ACM WAHC@CCS*, 2019.

[3] T. Araki, A. Barak, J. Furukawa, T. Lichter, Y. Lindell, A. Nof, K. Ohara, A. Watzman, and O. Weinstein, "Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier," in *IEEE S&P*, 2017.

[4] T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara, "High-throughput semi-honest secure three-party computation with an honest majority," in *ACM CCS*, 2016.

[5] G. Asharov, S. Halevi, Y. Lindell, and T. Rabin, "Privacy-preserving search of similar patients in genomic data," *PETS*, 2018.

[6] A. Baccarini, M. Blanton, and C. Yuan, "Multi-party replicated secret sharing over a ring with applications to privacy-preserving machine learning," *ePrint Archive*, 2020, https://eprint.iacr.org/2020/1577.

[7] C. Baum, I. Damgård, T. Toft, and R. W. Zakarias, "Better preprocessing for secure multiparty computation," in *ACNS*, 2016.

[8] A. Ben-Efraim, Y. Lindell, and E. Omri, "Optimizing semi-honest secure multiparty computation for the internet," in *CCS*, 2016.

[9] A. Ben-Efraim, M. Nielsen, and E. Omri, "Turbospeedz: Double your online spdz! improving SPDZ using function dependent preprocessing," in *ACNS*, 2019.

[10] A. Ben-Efraim and E. Omri, "Concrete efficiency improvements for multiparty garbling with an honest majority," in *LATINCRYPT*, 2017.

[11] M. Ben-Or, S. Goldwasser, and A. Wigderson, "Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract)," in *STOC*, 1988.

[12] M. Blanton, A. Kang, and C. Yuan, "Improved building blocks for secure multi-party computation based on secret sharing with honest majority," in *ACNS*, 2020.

[13] D. Boneh, E. Boyle, H. Corrigan-Gibbs, N. Gilboa, and Y. Ishai, "Zero-Knowledge Proofs on Secret-Shared Data via Fully Linear PCPs," in *CRYPTO*, 2019.

[14] E. Boyle, N. Gilboa, Y. Ishai, and A. Nof, "Practical fully secure three-party computation via sublinear distributed zero-knowledge proofs," in *ACM CCS*, 2019.

[15] ——, "Efficient fully secure computation via distributed zero-knowledge proofs," in *ASIACRYPT*, 2020.

[16] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) Fully Homomorphic Encryption without Bootstrapping," *ACM Trans. Comput. Theory*, 2014.

[17] L. Braun, D. Demmler, T. Schneider, and O. Tkachenko, "MOTION - A Framework for Mixed-Protocol Multi-Party Computation," *ACM Trans. Priv. Secur.*, 2022.

[18] M. Byali, H. Chaudhari, A. Patra, and A. Suresh, "FLASH: Fast and Robust Framework for Privacy-preserving Machine Learning," *PETS*, 2020.

[19] S. Carpov, K. Deforth, N. Gama, M. Georgieva, D. Jetchev, J. Katz, I. Leontiadis, M. Mohammadi, A. Sae-Tang, and M. Vuille, "Manticore: Efficient Framework for Scalable Secure Multiparty Computation Protocols," *ePrint Archive*, 2021, https://eprint.iacr.org/2021/200.

[20] N. Chandran, N. Dasgupta, D. Gupta, S. L. B. Obbattu, S. Sekar, and A. Shah, "Efficient Linear Multiparty PSI and Extensions to Circuit/Quorum PSI," in *ACM CCS*, 2021.

[21] H. Chaudhari, A. Choudhury, A. Patra, and A. Suresh, "ASTRA: High Throughput 3PC over Rings with Application to Secure Prediction," in *ACM CCSW@CCS*, 2019.

[22] H. Chaudhari, R. Rachuri, and A. Suresh, "Trident: Efficient 4PC Framework for Privacy Preserving Machine Learning," in *NDSS*, 2020.

[23] K. Chida, D. Genkin, K. Hamada, D. Ikarashi, R. Kikuchi, Y. Lindell, and A. Nof, "Fast large-scale honest-majority MPC for malicious adversaries," in *CRYPTO*, 2018.

[24] P. Covington, J. Adams, and E. Sargin, "Deep neural networks for youtube recommendations," in *RecSys*, 2016.

[25] R. Cramer, I. Damgård, D. Escudero, P. Scholl, and C. Xing, "SPD$\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for Dishonest Majority," in *CRYPTO*, 2018.

[26] R. Cramer, I. Damgård, and Y. Ishai, "Share conversion, pseudorandom secret-sharing and applications to secure computation," in *TCC*, 2005.

[27] Cryptography and P. E. G. at TU Darmstadt, "ENCRYPTO Utils," 2017, https://github.com/encryptogroup/ENCRYPTO_utils.

[28] A. Dalskov, D. Escudero, and M. Keller, "Fantastic Four: Honest-Majority Four-Party Secure Computation With Malicious Security," in *USENIX Security*, 2021.

[29] I. Damgård, D. Escudero, T. K. Frederiksen, M. Keller, P. Scholl, and N. Volgushev, "New primitives for actively-secure MPC over rings with applications to private machine learning," in *IEEE S&P*, 2019.

[30] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart, "Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits," in *ESORICS*, 2013.

[31] I. Damgård and J. B. Nielsen, "Scalable and unconditionally secure multiparty computation," in *CRYPTO*, 2007.

[32] I. Damgård, C. Orlandi, and M. Simkin, "Yet another compiler for active security or: Efficient MPC over arbitrary rings," in *CRYPTO*, 2018.

[33] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias, "Multiparty computation from somewhat homomorphic encryption," in *CRYPTO*, 2012.

[34] E. Dans, "How signal cleverly exposed Facebook's disregard for privacy," *Forbes*, 2021, https://www.forbes.com/sites/enriquedans/2021/05/07/how-signal-cleverly-exposed-facebooks-disregard-forprivacy.

[35] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering," in *NeurIPS*, 2016.

[36] C. Dwork, "Differential Privacy: A Survey of Results," in *TAMC*, 2008.

[37] Z. Erkin, M. Franz, J. Guajardo, S. Katzenbeisser, I. Lagendijk, and T. Toft, "Privacy-preserving face recognition," *PETS*, 2009.

[38] D. Escudero and A. Dalskov, "Honest Majority MPC with Abort with Minimal Online Communication," in *LATINCRYPT*, 2021.

[39] J. Furukawa, Y. Lindell, A. Nof, and O. Weinstein, "High-throughput secure three-party computation for malicious adversaries and an honest majority," in *EUROCRYPT*, 2017.

[40] D. Genkin, Y. Ishai, M. M. Prabhakaran, A. Sahai, and E. Tromer, "Circuits resilient to additive attacks with applications to secure computation," in *STOC*, 2014.

[41] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *CRYPTO*, 2013.

[42] O. Goldreich, *Foundations of cryptography: volume 2, basic applications*. Cambridge university press, 2009.

[43] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game or A completeness theorem for protocols with honest majority," in *STOC*, 1987.

[44] S. D. Gordon, S. Ranellucci, and X. Wang, "Secure computation with low communication from cross-checking," in *ASIACRYPT*, 2018.

[45] S. D. Gordon, D. Starin, and A. Yerukhimovich, "The More The Merrier: Reducing the Cost of Large Scale MPC," in *EUROCRYPT*, 2021.

[46] V. Goyal, H. Li, R. Ostrovsky, A. Polychroniadou, and Y. Song, "ATLAS: Efficient and Scalable MPC in the Honest Majority Setting," in *CRYPTO*, 2021.

[47] V. Goyal, Y. Liu, and Y. Song, "Communication-efficient unconditional MPC with guaranteed output delivery," in *CRYPTO*, 2019.

[48] V. Goyal and Y. Song, "Malicious Security Comes Free in Honest-Majority MPC," in *CRYPTO*, 2020.

[49] G. Guido, M. I. Prete, S. Miraglia, and I. De Mare, "Targeting direct marketing campaigns by neural networks," *Journal of Marketing Management*, 2011.

[50] W. Henecka and T. Schneider, "Faster secure two-party computation with less memory," in *AsiaCCS*, 2013.

[51] J. Katz, V. Kolesnikov, and X. Wang, "Improved non-interactive zero knowledge with applications to post-quantum signatures," in *CCS*, 2018.

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

[53] M. Keller, V. Pastro, and D. Rotaru, "Overdrive: Making SPDZ great again," in *EUROCRYPT*, 2018.

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

[55] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *ICLR*, 2017.

[56] N. Koti, M. Pancholi, A. Patra, and A. Suresh, "SWIFT: Super-fast and Robust Privacy-Preserving Machine Learning," in *USENIX Security*, 2021.

[57] N. Koti, A. Patra, R. Rachuri, and A. Suresh, "Tetrad: Actively Secure 4PC for Secure Training and Inference," in *NDSS*, 2022.

[58] A. Krizhevsky, V. Nair, and G. Hinton, "The CIFAR-10 dataset," 2014, https://www.cs.toronto.edu/~kriz/cifar.html.

[59] A. Lapets, N. Volgushev, A. Bestavros, F. Jansen, and M. Varia, "Secure MPC for analytics as a web application," in *IEEE SecDev*, 2016.

[60] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, 1998.

[61] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010, http://yann.lecun.com/exdb/mnist/.

[62] C. Li, B. Pang, Y. Liu, H. Sun, Z. Liu, X. Xie, T. Yang, Y. Cui, L. Zhang, and Q. Zhang, "Adsgnn: Behavior-graph augmented relevance modeling in sponsored search," in *SIGIR*, 2021.

[63] Y. Lindell, "How to simulate it - A tutorial on the simulation proof technique," in *Tutorials on the Foundations of Cryptography*, 2017.

[64] Y. Lindell, B. Pinkas, N. P. Smart, and A. Yanai, "Efficient constant round multi-party computation combining BMR and SPDZ," in *CRYPTO*, 2015.

[65] M. Malone, "How does Facebook know what ads to show you? (example)," *Vici Media*, 2021, https://www.vicimediainc.com/how-does-facebook-know-what-ads-to-show-you/.

[66] S. Mazloom, P. H. Le, S. Ranellucci, and S. D. Gordon, "Secure parallel computation on national scale volumes of data," in *USENIX Security*, 2020.

[67] P. Miao, S. Patel, M. Raykova, K. Seth, and M. Yung, "Two-sided malicious security for private intersection-sum with cardinality," in *CRYPTO*, 2020.

[68] P. Mohassel and P. Rindal, "ABY^3: A mixed protocol framework for machine learning," in *CCS*, 2018.

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

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

[71] S. Ohata and K. Nuida, "Communication-Efficient (Client-Aided) Secure Two-Party Protocols and Its Application," in *FC*, 2020.

[72] K. Park, J. Lee, and J. Choi, "Deep neural networks for news recommendations," in *CIKM*, 2017.

[73] A. Patra, T. Schneider, A. Suresh, and H. Yalame, "ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation," in *USENIX Security*, 2021.

[74] A. Patra and A. Suresh, "BLAZE: Blazing Fast Privacy-Preserving Machine Learning," in *NDSS*, 2020.

[75] R. Poddar, S. Kalra, A. Yanai, R. Deng, R. A. Popa, and J. M. Hellerstein, "Senate: A Maliciously-Secure MPC Platform for Collaborative Analytics," in *USENIX Security*, 2021.

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

[77] P. Rogaway and T. Shrimpton, "Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance," in *FSE*, 2004.

[78] D. Rotaru and T. Wood, "MArBled Circuits: Mixing Arithmetic and Boolean Circuits with Active Security," in *INDOCRYPT*, 2019.

[79] T. Schneider and O. Tkachenko, "EPISODE: efficient privacy-preserving similar sequence queries on outsourced genomic databases," in *AsiaCCS*, 2019.

[80] G. C. C. Services, "Google Cloud Platform," 2008, network costs - https://cloud.google.com/vpc/network-pricing, Computation costs - https://cloud.google.com/compute/vm-instance-pricing.

[81] L. Shen, X. Chen, J. Shi, Y. Dong, and B. Fang, "An Efficient 3-Party Framework for Privacy-Preserving Neural Network Inference," in *ESORICS*, 2020.

[82] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in *ICLR*, 2015.

[83] J. So, B. Güler, and A. S. Avestimehr, "CodedPrivateML: A Fast and Privacy-Preserving Framework for Distributed Machine Learning," *IEEE J. Sel. Areas Inf. Theory*, 2021.

[84] A. Suresh, "MPCLeague: Robust MPC Platform for Privacy-Preserving Machine Learning," PhD Thesis, 2021, https://arxiv.org/pdf/2112.13338.

[85] S. Wagh, S. Tople, F. Benhamouda, E. Kushilevitz, P. Mittal, and T. Rabin, "FALCON: Honest-Majority Maliciously Secure Framework for Private Deep Learning," *PETS*, 2020.

[86] R. A. Wagner and M. J. Fischer, "The String-to-String Correction Problem," *J. ACM*, 1974.

[87] X. Wang, S. Ranellucci, and J. Katz, "Authenticated garbling and efficient maliciously secure two-party computation," in *CCS*, 2017.

[88] J. Yang, Z. Liu, S. Xiao, C. Li, D. Lian, S. Agrawal, A. Singh, G. Sun, and X. Xie, "Graphformers: Gnn-nested transformers for representation learning on textual graph," in *NeurIPS*, 2021.

[89] A. C. Yao, "Protocols for secure computations (extended abstract)," in *FOCS*, 1982.

[90] J. Zhu, Y. Cui, Y. Liu, H. Sun, X. Li, M. Pelger, T. Yang, L. Zhang, R. Zhang, and H. Zhao, "Textgnn: Improving text encoder via graph neural network in sponsored search," in *WWW*, 2021.

APPENDIX A
PRELIMINARIES

a) Shared key setup: $\mathcal{F}_{\mathsf{setup}}$ [4], [68], [74] enables establishment of common random keys for a pseudo-random function (PRF) F, among parties. This aids in non-interactively generating correlated randomness. Here $F : \{0,1\}^\kappa \times \{0,1\}^\kappa \to X$ is a secure PRF, with co-domain X being $\mathbb{Z}_{2^\ell}$. The semi-honest functionality, $\mathcal{F}_{\mathsf{setup}}$ appears in Fig. 15. The functionality for the malicious case is similar, except that the adversary now has the capability to `abort`.

To sample a random value $\mathsf{r} \in \mathbb{Z}_{2^\ell}$ among a set of $t+1$ parties $\mathcal{T} = \{P_1, \ldots, P_{t+1}\}$ non-interactively, each $P_i \in \mathcal{T}$ invokes $F_{k_\mathcal{T}}(id_\mathcal{T})$ and obtains r. Here, $id_\mathcal{T}$ denotes a counter maintained by the parties in $\mathcal{T}$, and is updated after every PRF invocation. The appropriate keys used to sample is implicit from the context, from the identities of the parties that sample.

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

$\mathcal{F}_{\mathsf{setup}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$. $\mathcal{F}_{\mathsf{setup}}$ picks random keys $k_\mathcal{T}, k_{\mathcal{T}'}$ for every set $\mathcal{T}, \mathcal{T}' \subseteq \mathcal{P}$ of $t+1, t+2$ parties, respectively. $\mathcal{F}_{\mathsf{setup}}$ picks random keys k_{ij} for every pair of parties $P_i, P_j \in \mathcal{P}$ and $i < j$.

- Set $\mathsf{x}_s = \{k_{si}, k_{js}\}_{\forall i:s<i\leq n, \forall j:1\leq j<s}$.
- Set $\mathsf{y}_s = \{k_\mathcal{T}\}_{\forall \mathcal{T}\subset\mathcal{P}:|\mathcal{T}|=t+1}$ when $P_s \in \mathcal{T}$.
- Set $\mathsf{z}_s = \{k_{\mathcal{T}'}\}_{\forall \mathcal{T}'\subseteq\mathcal{P}:|\mathcal{T}'|=t+2}$ when $P_s \in \mathcal{T}'$.

Output: Send $(\mathsf{Output}, \mathsf{x}_s, \mathsf{y}_s, \mathsf{z}_s)$ to every $P_s \in \mathcal{P}$.

Fig. 15: Ideal functionality for shared-key setup

b) Collision-Resistant Hash Function: A family of hash functions [77] $\{\mathsf{H} : \mathcal{K}\times\mathsf{M} \to \mathcal{Y}\}$ is said to be collision resistant if for all PPT adversaries $\mathcal{A}$, given the hash function H_k for $k \in_R \mathcal{K}$, the following holds: $\mathsf{Pr}[(x, x') \leftarrow \mathcal{A}(k) : (x \neq$

$x') \wedge \mathsf{H}_k(x) = \mathsf{H}_k(x')] = \mathsf{negl}(\kappa)$, where $x, x' \in \{0,1\}^m$, $m = \mathsf{poly}(\kappa)$, and κ is security parameter.

c) Commitment Scheme: Let $\mathsf{Com}(x)$ denote the commitment of a value x [69]. The commitment scheme $\mathsf{Com}(x)$ possesses two properties; *hiding* and *binding*. The former ensures privacy of value x given its commitment $\mathsf{Com}(x)$, while the latter prevents a corrupt party from opening the commitment to a different value $x' \neq x$.

A. Helper primitives

(1) $\Pi_{[0]} \rightarrow [0]$ (Fig. 18): To generate $[\cdot]$-shares of 0, each party non-interactively samples two values, each with one of its neighboring parties. A party's shares of 0 are defined as the difference between these values.

Protocol $\Pi_{[0]}$

1. P_i, P_{i+1}, for $i \in \{1, \ldots, n-1\}$, sample a random value $\mathsf{r}_i \in_R \mathbb{Z}_{2^\ell}$, while P_1, P_n sample a random value $\mathsf{r}_n \in_R \mathbb{Z}_{2^\ell}$, using their respective common PRF keys.

2. P_i for $i \in \{2, \ldots, n\}$ sets $[0]_i = \mathsf{r}_i - \mathsf{r}_{i-1}$, while P_1 sets $[0]_1 = \mathsf{r}_1 - \mathsf{r}_n$.

Fig. 16: Generating $[\cdot]$-shares of 0

(2) $\Pi_{\mathsf{rand}} \rightarrow \langle \mathsf{r} \rangle$ (Fig. 17): To generate $\langle \cdot \rangle$-shares of a random $\mathsf{r} \in \mathbb{Z}_{2^\ell}$, every set of $t+1$ parties non-interactively sample a random value using keys established during the setup phase and define r to be the sum of these values.

Protocol Π_{rand}

1. Every $P_i \in \mathcal{T}_j$ for $j \in \{1, \ldots, \mathsf{q}\}$, samples $\langle \mathsf{r} \rangle_{\mathcal{T}_j} \in_R \mathbb{Z}_{2^\ell}$ using the common PRF key.

2. Define $\mathsf{r} = \sum_{j=1}^{\mathsf{q}} \langle \mathsf{r} \rangle_{\mathcal{T}_j}$.

Fig. 17: Generating $\langle \cdot \rangle$-shares of a random value

(3) $\Pi_{\mathsf{pRand}}(P_s) \rightarrow \langle \mathsf{r} \rangle$ (Fig. 18): This protocol generates $\langle \cdot \rangle$-shares of a random value r such that P_s learns all the shares. Every set of $t+1$ parties non-interactively samples a random value together with P_s, using the keys established (for every set of $t+2$ parties) during the setup phase.

Protocol $\Pi_{\mathsf{pRand}}(P_s)$

1. Every $P_i \in \mathcal{T}_j$ for $j \in \{1, \ldots, \mathsf{q}\}$, samples $\langle \mathsf{r} \rangle_{\mathcal{T}_j} \in_R \mathbb{Z}_{2^\ell}$, together with P_s, using the common PRF key.

2. Define $\mathsf{r} = \sum_{j=1}^{\mathsf{q}} \langle \mathsf{r} \rangle_{\mathcal{T}_j}$.

Fig. 18: Generating $\langle \cdot \rangle$-shares of a random value along with P_s

(4) $\Pi_{\cdot \rightarrow \langle\langle \cdot \rangle\rangle}(\mathsf{a}) \rightarrow \langle\langle \mathsf{a} \rangle\rangle$: This protocol generates $\langle\langle \mathsf{a} \rangle\rangle$ when $\mathsf{a} \in \mathbb{Z}_{2^\ell}$ is held by at least $t+1$ parties, say parties in $\mathcal{E}$. For this, $P_i \in \mathcal{E}$ sets $\mathsf{m}_\mathsf{a} = \mathsf{a}$ and $\langle \cdot \rangle$-shares of λ_a as 0. To generate $\langle\langle \mathsf{a} \rangle\rangle$ in the malicious case where all parties hold a, we let parties set $\mathsf{m}_\mathsf{a} = \mathsf{a}$ and shares of λ_a as 0.

(5) $\Pi_{\langle \cdot \rangle \rightarrow \mathcal{T}[\cdot]}(\langle \mathsf{a} \rangle) \rightarrow {}^{\mathcal{T}}[\mathsf{a}]$ (Fig. 19): This protocol enables parties in $\mathcal{T} = \{E_1, E_2, \ldots, E_{t+1}\}$ to generate ${}^{\mathcal{T}}[\mathsf{a}]$ from $[\mathsf{a}]$. To generate ${}^{\mathcal{T}}[\mathsf{a}]_i$, the idea is to sum up the shares in $\langle \mathsf{a} \rangle_{\mathcal{T}_1}, \ldots, \langle \mathsf{a} \rangle_{\mathcal{T}_\mathsf{q}}$, while ensuring that every share is accounted for and no share is incorporated more than once. Concretely, for share $\langle \mathsf{a} \rangle_{\mathcal{T}_j}$ held by parties in $\mathcal{T}_j$ for $j \in \{1, \ldots, \mathsf{q}\}$, $E_i \in \mathcal{T}_j$ incorporates $\langle \mathsf{a} \rangle_{\mathcal{T}_j}$ in its share of ${}^{\mathcal{E}}[\mathsf{a}]_i$ if E_i has the least index in $\mathcal{T}_j$.

Protocol $\Pi_{\langle \cdot \rangle \rightarrow \mathcal{T}[\cdot]}(\langle \mathsf{a} \rangle)$

1. Let $\mathcal{T} = \{E_1, \ldots, E_{t+1}\}$.

2. $E_i \in \mathcal{T}$ computes ${}^{\mathcal{E}}[\mathsf{a}]_i = \sum_{j=1}^{\mathsf{q}} \langle \mathsf{a} \rangle_{\mathcal{T}_j} \cdot \mathsf{e}_j^i$, where $\mathsf{e}_j^i = 1$ if E_i has the least index in $\mathcal{T}_j$, and 0, otherwise.

Fig. 19: Conversion from $\langle \cdot \rangle$-share to ${}^{\mathcal{T}}[\cdot]$-share

(6) $\Pi_{\langle \cdot \rangle \rightarrow [\cdot]}(\langle \mathsf{a} \rangle) \rightarrow [\mathsf{a}]$: $\langle \cdot \rangle$-share can be converted to $[\cdot]$-share following similar procedure as $\Pi_{\langle \cdot \rangle \rightarrow \mathcal{T}[\cdot]}$, and is denoted as $\Pi_{\langle \cdot \rangle \rightarrow [\cdot]}(\langle \mathsf{a} \rangle)$. We omit the details due to similarity.

(7) $\Pi_{\langle\langle \cdot \rangle\rangle \rightarrow \mathcal{T}[\cdot]}(\langle\langle \mathsf{a} \rangle\rangle) \rightarrow {}^{\mathcal{T}}[\mathsf{a}]$: Parties in $\mathcal{T}$ invoke $\Pi_{\langle \cdot \rangle \rightarrow \mathcal{T}[\cdot]}$ on $-\lambda_\mathsf{a}$ to generate ${}^{\mathcal{T}}[-\lambda_\mathsf{a}]$, followed by a designated $P_i \in \mathcal{T}$ that holds m_a setting ${}^{\mathcal{T}}[\mathsf{a}]_i = \mathsf{m}_\mathsf{a} + {}^{\mathcal{E}}[-\lambda_\mathsf{a}]_i$.

(8) $\Pi_{\langle\langle \cdot \rangle\rangle \rightarrow [\cdot]}(\langle\langle \mathsf{a} \rangle\rangle) \rightarrow [\mathsf{a}]$: $[\mathsf{a}]$ can be generated from $\langle\langle \mathsf{a} \rangle\rangle$ similar to $\Pi_{\langle\langle \cdot \rangle\rangle \rightarrow \mathcal{T}[\cdot]}$, and is denoted as $\Pi_{\langle\langle \cdot \rangle\rangle \rightarrow [\cdot]}(\langle\langle \mathsf{a} \rangle\rangle)$.

(9) $\Pi_{\langle \cdot \rangle \rightarrow \langle\langle \cdot \rangle\rangle}(\langle \mathsf{a} \rangle) \rightarrow \langle\langle \mathsf{a} \rangle\rangle$: To convert $\langle \mathsf{a} \rangle$, to $\langle\langle \mathsf{a} \rangle\rangle$, set $\mathsf{m}_\mathsf{a} = 0$ and set $\langle \lambda_\mathsf{a} \rangle = -\langle \mathsf{a} \rangle$.

(10) $\Pi_{\langle\langle \cdot \rangle\rangle \rightarrow \langle \cdot \rangle}(\langle\langle \mathsf{a} \rangle\rangle) \rightarrow \langle \mathsf{a} \rangle$: To convert $\langle\langle \mathsf{a} \rangle\rangle$ to $\langle \mathsf{a} \rangle$, set $\langle \mathsf{a} \rangle_{\mathcal{T}_j} = -\langle \lambda_\mathsf{a} \rangle_{\mathcal{T}_j}$ for $j \in \{1, \ldots, \mathsf{q}-1\}$ and $\langle \mathsf{a} \rangle_{\mathcal{T}_\mathsf{q}} = \mathsf{m}_\mathsf{a} - \langle \lambda_\mathsf{a} \rangle_{\mathcal{T}_\mathsf{q}}$, where $\mathcal{T}_\mathsf{q} = \mathcal{E}$.

(11) $\Pi_{\langle \cdot \rangle \cdot \langle \cdot \rangle \rightarrow [\cdot]}(\langle \mathsf{a} \rangle, \langle \mathsf{b} \rangle) \rightarrow [\mathsf{ab}]$ (Fig. 20): Given $\langle \mathsf{a} \rangle, \langle \mathsf{b} \rangle$, parties non-interactively compute $[\mathsf{ab}]$ as follows. Observe that $[\mathsf{ab}] = \sum_{j=1}^{\mathsf{q}} \left[\langle \mathsf{a} \rangle_{\mathcal{T}_j} \mathsf{b}\right]$. To generate $\left[\langle \mathsf{a} \rangle_{\mathcal{T}_j} \mathsf{b}\right]$, the idea is to generate ${}^{\mathcal{T}_j}\left[\langle \mathsf{a} \rangle_{\mathcal{T}_j} \mathsf{b}\right]$ and perform a conversion. Parties in $\mathcal{T}_j$ generate ${}^{\mathcal{T}_j}\left[\langle \mathsf{a} \rangle_{\mathcal{T}_j} \mathsf{b}\right]$ as ${}^{\mathcal{T}_j}\left[\langle \mathsf{a} \rangle_{\mathcal{T}_j} \mathsf{b}\right] = \left(\langle \mathsf{a} \rangle_{\mathcal{T}_j}\right) \cdot \left({}^{\mathcal{T}_j}[\mathsf{b}]\right)$. To obtain $\left[\langle \mathsf{a} \rangle_{\mathcal{T}_j} \mathsf{b}\right]$ from ${}^{\mathcal{T}_j}\left[\langle \mathsf{a} \rangle_{\mathcal{T}_j} \mathsf{b}\right]$, $P_i \in \mathcal{P}$ sets $\left[\langle \mathsf{a} \rangle_{\mathcal{T}_j} \mathsf{b}\right]_i = {}^{\mathcal{T}_j}\left[\langle \mathsf{a} \rangle_{\mathcal{T}_j} \mathsf{b}\right]_i$ if $P_i \in \mathcal{T}_j$ and $\left[\langle \mathsf{a} \rangle_{\mathcal{T}_j} \mathsf{b}\right]_i = 0$, otherwise.

Protocol $\Pi_{\langle \cdot \rangle \cdot \langle \cdot \rangle \rightarrow [\cdot]}(\mathcal{P}, \langle \mathsf{a} \rangle, \langle \mathsf{b} \rangle)$

1. For $j \in \{1, \ldots, \mathsf{q}\}$:

- $P_i \in \mathcal{T}_j$ invokes $\Pi_{\langle \cdot \rangle \rightarrow \mathcal{T}[\cdot]}$ on $\langle \mathsf{b} \rangle$ to generate ${}^{\mathcal{T}_j}[\mathsf{b}]_i$.
- Set $\left[\langle \mathsf{a} \rangle_{\mathcal{T}_j} \mathsf{b}\right]_i = \left(\langle \mathsf{a} \rangle_{\mathcal{T}_j}\right) \cdot \left({}^{\mathcal{T}_j}[\mathsf{b}]_i\right)$ if $P_i \in \mathcal{T}_j$, and $\left[\langle \mathsf{a} \rangle_{\mathcal{T}_j} \mathsf{b}\right]_i = 0$, otherwise.

2. $P_i \in \mathcal{P}$ computes $[\mathsf{ab}]_i = \sum_{j=1}^{\mathsf{q}} \left[\langle \mathsf{a} \rangle_{\mathcal{T}_j} \mathsf{b}\right]_i$.

Fig. 20: $\langle \mathsf{a} \rangle, \langle \mathsf{b} \rangle$ to $[\mathsf{ab}]$

(12) $\Pi_{\mathsf{agree}}(\mathcal{P}, \{\vec{\mathsf{v}_1}, \ldots, \vec{\mathsf{v}_n}\}) \rightarrow$ `continue`/`abort`: Allows parties to check if they hold the same set of values $\vec{\mathsf{v}} = (\mathsf{v}_1, \ldots, \mathsf{v}_m)$, where parties `continue` if the values are same, and `abort` otherwise. We denote the version of $\vec{\mathsf{v}}$ held by $P_i \in \mathcal{P}$ as $\vec{\mathsf{v}_i}$. To check for consistency of $\vec{\mathsf{v}}$, parties compute hash, $\mathsf{H} = \mathsf{H}(\mathsf{v}_1 || \ldots || \mathsf{v}_m)$, of the concatenation of all values $\mathsf{v}_1, \ldots, \mathsf{v}_m$, and exchange H among themselves. If any party receives inconsistent hashes, it `aborts`; else it `continues`.

(13) $\Pi_{\langle \cdot \rangle}(P_s, \mathsf{a}) \rightarrow \langle \mathsf{a} \rangle$: To enable P_s to generate $\langle \mathsf{a} \rangle$, parties generate $\langle \mathsf{a} \rangle_{\mathcal{T}_j}$ for $j \in \{1, \ldots, \mathsf{q}-1\}$ using Π_{pRand}, with P_s learning $\langle \mathsf{a} \rangle_{\mathcal{T}_j}$ (i.e., $\langle \mathsf{a} \rangle_{\mathcal{T}_j}$ are sampled using common key amongst $t+2$ parties). P_s sets $\langle \mathsf{a} \rangle_{\mathcal{T}_\mathsf{q}} = \mathsf{a} - \sum_{j=1}^{\mathsf{q}-1} \langle \mathsf{a} \rangle_{\mathcal{T}_j}$ and sends $\langle \mathsf{a} \rangle_{\mathcal{T}_\mathsf{q}}$ to parties in $\mathcal{T}_\mathsf{q}$. For malicious case, this is followed by invoking $\Pi_{\mathsf{agree}}(\mathcal{P}, \{\langle \mathsf{a} \rangle_{\mathcal{T}_\mathsf{q}}\})$ to check consistency of value sent by P_s.

APPENDIX B
MPCLAN PROTOCOLS

A. Semi-honest protocols

a) Input sharing: The protocol for input sharing appears in Fig. 21.

Protocol $\Pi_{\mathsf{Sh}}(P_s, \mathsf{a})$

Preprocessing: Invoke $\Pi_{\mathsf{pRand}}(P_s)$ to generate $\langle \lambda_{\mathsf{a}} \rangle$, with P_s learning λ_{a} where $\lambda_{\mathsf{a}} \in \mathbb{Z}_{2^\ell}$.

Online: P_s computes and sends $\mathsf{m}_{\mathsf{a}} = \mathsf{a} + \lambda_{\mathsf{a}}$ to all $P_i \in \mathcal{E}$.

Fig. 21: Semi-honest: Input sharing protocol

b) Truncation - Instantiating $\mathcal{F}_{\mathsf{TrGen}}$: We rely on a modified version of the doubly shared random bit (a bit that is arithmetic as well as Boolean shared) generation protocol of [29], extended to our n-party setting, to generate $\langle\langle \mathsf{r} \rangle\rangle, \langle\langle \mathsf{r}^d \rangle\rangle$ as required to perform truncation. Here, r^d represents the truncated (by d bits) version of $\mathsf{r} \in \mathbb{Z}_{2^\ell}$. The resulting protocol is referred to as Π_{dsBits} (Fig. 22).

Protocol $\Pi_{\mathsf{dsBits}}(\mathcal{P}, \mathsf{isTr})$

If $\mathsf{isTr} = 1$, set $\mathsf{k} = \ell$ else set $\mathsf{k} = 1$. For $i \in \{0, \ldots, \mathsf{k}-1\}$:

1. Invoke Π_{rand} to generate $\langle \mathsf{u}_i \rangle^{\ell+2}$ for $\mathsf{u}_i \in \mathbb{Z}_{2^{\ell+2}}$, and $\Pi_{[0]}$ to generate $[0]^{\ell+2}$.

2. Compute $\langle \mathsf{a}_i \rangle^{\ell+2} = 2\langle \mathsf{u}_i \rangle^{\ell+2} + 1$.

3. Invoke $\Pi_{\langle\cdot\rangle\cdot\langle\cdot\rangle \to [\cdot]}$ on $\langle \mathsf{a}_i \rangle^{\ell+2}$ to generate $[\mathsf{e}_i]^{\ell+2}$ where $\mathsf{e}_i = \mathsf{a}_i^2$.

4. Send $[\mathsf{e}_i]^{\ell+2} + [0]^{\ell+2}$ to P_{king}, who reconstructs $\mathsf{e}_i + 0 = \mathsf{e}_i$ and sends to all.

5. Let c_i be the smallest root of e_i modulo $2^{\ell+2}$, and c_i^{-1} its inverse. Compute $\langle \mathsf{d}_i \rangle^{\ell+2} = \mathsf{c}_i^{-1} \langle \mathsf{a}_i \rangle^{\ell+2} + 1$.

6. P_j sets $\langle \mathsf{b}_i \rangle_j^{\ell+2} = \langle \mathsf{d}_i \rangle_j^{\ell+2}/2$, and $\langle \mathsf{b}_i^{\mathsf{R}} \rangle_j, \langle \mathsf{b}_i \rangle_j^{\mathbf{B}}$ as the least significant ℓ bits and the least significant bit of $\langle \mathsf{b}_i \rangle_j^{\ell+2}$, respectively.

7. Invoke $\Pi_{\langle\cdot\rangle \to \langle\langle\cdot\rangle\rangle}$ on $\langle \mathsf{b}_i^{\mathsf{R}} \rangle, \langle \mathsf{b}_i \rangle^{\mathbf{B}}$ to generate $\langle\langle \mathsf{b}_i^{\mathsf{R}} \rangle\rangle, \langle\langle \mathsf{b}_i \rangle\rangle^{\mathbf{B}}$.

If $\mathsf{isTr} = 1$, set:

$$(\langle\langle \mathsf{r} \rangle\rangle, \langle\langle \mathsf{r}^d \rangle\rangle) = \left(\sum_{i=0}^{\mathsf{k}-1} 2^i \langle\langle \mathsf{b}_i^{\mathsf{R}} \rangle\rangle, \sum_{i=d}^{\mathsf{k}-1} 2^{i-d} \langle\langle \mathsf{b}_i^{\mathsf{R}} \rangle\rangle \right)$$

Fig. 22: Semi-honest: Doubly shared bits

At a high-level, generation of doubly shared bits relies on the property that every non-zero quadratic residue has exactly one root when working over fields. The work of [29], operating over rings, shows that something similar holds over rings as well. Concretely, according to lemma 4.1 of [29]: *if* a *is such that* $\mathsf{a}^2 \equiv_\ell 1$, *then* a *is congruent mod* 2^ℓ *to either* $1, -1, -1+2^{\ell-1}, 1+2^{\ell-1}$. Thus, the doubly shared bit generation protocol of [29] proceeds as follows. Generate a^2 for $\mathsf{a} \in \mathbb{Z}_{2^{\ell+2}}$ such that $\mathsf{a}^2 \equiv_{\ell+2} 1$, and compute its smallest root c mod $2^{\ell+2}$. Compute $(\mathsf{c}^{-1}\mathsf{a})$, and by lemma 4.1 of [29] it follows that $\mathsf{c}^{-1}\mathsf{a} \in \{\pm 1, \pm 1 + 2^{\ell+1}\}$. That is, $(\mathsf{c}^{-1}\mathsf{a})$ is congruent to ± 1 modulo $2^{\ell+1}$. Thus, $\mathsf{d} = \mathsf{c}^{-1}\mathsf{a} + 1$ is congruent to 0 or 2 modulo $2^{\ell+1}$ with equal probability. Hence, setting $\mathsf{b} = \mathsf{d}/2$ outputs bit $\mathsf{b} = 0$ or bit $\mathsf{b} = 1$ with equal probability. Observe that the computation has to be performed over $\mathbb{Z}_{2^{\ell+2}}$. Hence, in the protocol description, we use $\ell + 2$ in the superscript to distinguish shares of x over $\mathbb{Z}_{2^{\ell+2}}$ from its shares over $\mathbb{Z}_{2^\ell}$.

The main change in Π_{dsBits} from that of the protocol in [29] is that to generate $\langle\langle \mathsf{r} \rangle\rangle, \langle\langle \mathsf{r}^d \rangle\rangle$ Π_{dsBits} generates ℓ random doubly shared bits $\mathsf{b}_0, \ldots, \mathsf{b}_{\ell-1} \in \mathbb{Z}_2$ instead of a single one, and composes these ℓ bits to generate r, and composes the higher $\ell - d$ bits to generate r^d, as follows.

$$(\langle\langle \mathsf{r} \rangle\rangle, \langle\langle \mathsf{r}^d \rangle\rangle) = \left(\sum_{i=0}^{\ell-1} 2^i \langle\langle \mathsf{b}_i^{\mathsf{R}} \rangle\rangle, \sum_{i=d}^{\ell-1} 2^{i-d} \langle\langle \mathsf{b}_i^{\mathsf{R}} \rangle\rangle \right) \tag{3}$$

Looking ahead, Π_{dsBits} can also be used only to generate a single doubly shared random bit, which finds use in other building blocks such as bit to arithmetic conversion and arithmetic to Boolean conversion. Thus, to distinguish the case when $(\langle\langle \mathsf{r} \rangle\rangle, \langle\langle \mathsf{r}^d \rangle\rangle)$ has to be generated versus when only a single doubly shared bit is to be generated, Π_{dsBits} takes a bit isTr as input and gives as output a doubly shared bit $\langle\langle \mathsf{b}^{\mathsf{R}} \rangle\rangle, \langle\langle \mathsf{b} \rangle\rangle^{\mathbf{B}}$ if $\mathsf{isTr} = 0$, and $(\langle\langle \mathsf{r} \rangle\rangle, \langle\langle \mathsf{r}^d \rangle\rangle)$ otherwise. The protocol appears in Fig. 22.

A final thing to note is that the computation in Π_{dsBits} proceeds over secret-shared data. Thus, to generate shares of the doubly shared bit b, one should be able to divide each share of d by 2, which necessitates d and its shares to be even. This holds true since $\langle \mathsf{d} \rangle^{\ell+2} = \mathsf{c}^{-1} \langle \mathsf{a} \rangle^{\ell+2} + 1 = \mathsf{c}^{-1} \left(2\langle \mathsf{u} \rangle^{\ell+2} + 1 \right) + 1 = 2\mathsf{c}^{-1} \langle \mathsf{u} \rangle^{\ell+2} + \mathsf{c}^{-1} + 1$. Here, $2\mathsf{c}^{-1} \langle \mathsf{u} \rangle^{\ell+2}$ is even due to multiplication by 2, while $\mathsf{c}^{-1} + 1$ is even since c^{-1} is odd by definition.

c) Dot product: As described before, a dot product can be viewed as n instances of multiplication such that the communication for all the instances is aggregated and performed in a single shot to eliminate the vector-size dependency. Consequently, the dot product protocol follows along the lines of the multiplication, and the formal details appear in Fig. 23.

Protocol $\Pi_{\mathsf{dp}}(\mathcal{P}, \langle\langle \vec{x} \rangle\rangle, \langle\langle \vec{y} \rangle\rangle)$

Preprocessing:

1. Invoke Π_{rand} to generate $\langle \mathsf{r} \rangle$ where $\mathsf{r} \in \mathbb{Z}_{2^\ell}$, followed by $\Pi_{\langle\cdot\rangle \to [\cdot]}$ to generate $[\mathsf{r}]$.

2. Invoke $\Pi_{\langle\cdot\rangle\cdot\langle\cdot\rangle \to [\cdot]}$ on $\langle \lambda_{\mathsf{x}_k} \rangle, \langle \lambda_{\mathsf{y}_k} \rangle$ to generate $[\Lambda_{\mathsf{x}_k \mathsf{y}_k}]$ for $k \in \{1, \ldots, \mathsf{n}\}$, and compute $\left[\sum_{k=1}^{\mathsf{n}} \Lambda_{\mathsf{x}_k \mathsf{y}_k} - \mathsf{r}\right] = \sum_{k=1}^{\mathsf{n}} [\Lambda_{\mathsf{x}_k \mathsf{y}_k}] - [\mathsf{r}]$.

3. $P_i \in \mathcal{E}$ invokes $\Pi_{\langle\cdot\rangle \to \mathcal{E}[\cdot]}$ on $\langle \lambda_{\mathsf{x}_k} \rangle, \langle \lambda_{\mathsf{y}_k} \rangle$ to generate ${}^{\mathcal{E}}[\lambda_{\mathsf{x}_k}]_i$, ${}^{\mathcal{E}}[\lambda_{\mathsf{y}_k}]_i$, respectively, for $k \in \{1, \ldots, \mathsf{n}\}$.

4. $P_i \in \mathcal{D}$ sends $\left[\sum_{k=1}^{\mathsf{n}} \Lambda_{\mathsf{x}_k \mathsf{y}_k} - \mathsf{r}\right]_i$ to P_{king}, who sets $\mathsf{D} = \sum_{i:P_i \in \mathcal{D}} \left[\sum_{k=1}^{\mathsf{n}} \Lambda_{\mathsf{x}_k \mathsf{y}_k} - \mathsf{r}\right]_i$.

Online:

1. $P_i \in \mathcal{E}$ computes and sends ${}^{\mathcal{E}}[\zeta]_i = \sum_{k=1}^{\mathsf{n}} \left(-\mathsf{m}_{\mathsf{x}_k} {}^{\mathcal{E}}[\lambda_{\mathsf{y}_k}]_i - \mathsf{m}_{\mathsf{y}_k} {}^{\mathcal{E}}[\lambda_{\mathsf{x}_k}]_i \right) + \left[\sum_{k=1}^{\mathsf{n}} \Lambda_{\mathsf{x}_k \mathsf{y}_k} - \mathsf{r}\right]_i$ to P_{king}.

2. P_{king} computes $\mathsf{E} = \sum_{k=1}^{\mathsf{n}} \mathsf{M}_{\mathsf{x}_k \mathsf{y}_k} + \sum_{i:P_i \in \mathcal{E}} {}^{\mathcal{E}}[\zeta]_i$ and sends $\mathsf{z} - \mathsf{r} = \mathsf{D} + \mathsf{E}$ to all parties in $\mathcal{E}$.

3. Invoke $\Pi_{\cdot \to \langle\langle\cdot\rangle\rangle}$ on $\mathsf{z} - \mathsf{r}$ to generate $\langle\langle \mathsf{z} - \mathsf{r} \rangle\rangle$, and compute $\langle\langle \mathsf{z} \rangle\rangle = \langle\langle \mathsf{z} - \mathsf{r} \rangle\rangle + \langle\langle \mathsf{r} \rangle\rangle$.

Fig. 23: Semi-honest: Dot product protocol

d) Multi-input multiplication: The goal of 3-input multiplication (Fig. 24) is to generate $\langle\langle\cdot\rangle\rangle$-sharing of $\mathsf{z} = \mathsf{abc}$ given $\langle\langle\mathsf{a}\rangle\rangle, \langle\langle\mathsf{b}\rangle\rangle, \langle\langle\mathsf{c}\rangle\rangle$, in a single shot. Observe that

$$\begin{aligned}\mathsf{z}-\mathsf{r} &= \mathsf{abc}-\mathsf{r} = (\mathsf{m_a}-\lambda_\mathsf{a})(\mathsf{m_b}-\lambda_\mathsf{b})(\mathsf{m_c}-\lambda_\mathsf{c})-\mathsf{r}\\ &= \mathsf{M_{abc}} - \mathsf{M_{ac}}\lambda_\mathsf{b} - \mathsf{M_{bc}}\lambda_\mathsf{a} - \mathsf{M_{ab}}\lambda_\mathsf{c}\\ &\quad + \mathsf{m_a}\Lambda_\mathsf{bc} + \mathsf{m_b}\Lambda_\mathsf{ac} + \mathsf{m_c}\Lambda_\mathsf{ab} - \Lambda_\mathsf{abc} - \mathsf{r}\end{aligned}$$

Given that $^{\mathcal{E}}[\Lambda_\mathsf{ab}], ^{\mathcal{E}}[\Lambda_\mathsf{ac}], ^{\mathcal{E}}[\Lambda_\mathsf{bc}], ^{\mathcal{E}}[\Lambda_\mathsf{abc}+\mathsf{r}]$ can be generated in the preprocessing among the parties in $\mathcal{E}$, parties proceed with a similar online phase as in Π_mult to compute the 3-input multiplication without inflating the online cost. With respect to the preprocessing phase,

– For generating $^{\mathcal{E}}[\Lambda_\mathsf{ac}], ^{\mathcal{E}}[\Lambda_\mathsf{bc}]$ parties first compute the respective additive sharings $([\cdot])$ using $\langle\lambda_\mathsf{a}\rangle, \langle\lambda_\mathsf{b}\rangle$ and $\langle\lambda_\mathsf{c}\rangle$ (via two invocations of $\Pi_{\langle\cdot\rangle\cdot\langle\cdot\rangle\to[\cdot]}$, Fig. 20). Following this parties in $\mathcal{D}$ communicate their share of $[\Lambda_\mathsf{ac}]$ and $[\Lambda_\mathsf{bc}]$ to P_king, each masked with a random $[\cdot]$-sharing of 0 (generated using $\Pi_{[0]}$, Fig. 16). This establishes $^{\mathcal{E}}[\Lambda_\mathsf{ac}], ^{\mathcal{E}}[\Lambda_\mathsf{bc}]$ among parties in $\mathcal{E}$.

– For generating $^{\mathcal{E}}[\Lambda_\mathsf{ab}]$, a slightly different approach is taken where parties first generate $\langle\Lambda_\mathsf{ab}\rangle$ using $\langle\lambda_\mathsf{a}\rangle, \langle\lambda_\mathsf{b}\rangle$ (as explained later), followed by non-interactively generating $^{\mathcal{E}}[\Lambda_\mathsf{ab}]$ (via $\Pi_{\langle\cdot\rangle\to\mathcal{T}[\cdot]}$, Fig. 19). The reason for generating $\langle\Lambda_\mathsf{ab}\rangle$ (instead of directly generating $^{\mathcal{E}}[\Lambda_\mathsf{ab}]$) is to facilitate generation of $^{\mathcal{E}}[\Lambda_\mathsf{abc}-\mathsf{r}]$ from $\langle\Lambda_\mathsf{ab}\rangle$, $\langle\lambda_\mathsf{c}\rangle$ and $[\mathsf{r}]$, which closely follows the preprocessing phase of the 2-input multiplication. Specifically, parties can generate $[\Lambda_\mathsf{abc}]$ using $\Pi_{\langle\cdot\rangle\cdot\langle\cdot\rangle\to[\cdot]}$ (Fig. 20) on $\langle\Lambda_\mathsf{ab}\rangle$, $\langle\lambda_\mathsf{c}\rangle$, followed by parties in $\mathcal{D}$ communicating their $[\Lambda_\mathsf{abc}]$ shares masked with $[\cdot]$-sharing of a random r to P_king. This generates $^{\mathcal{E}}[\Lambda_\mathsf{abc}+\mathsf{r}]$-sharing required during online phase.

– Regarding generation of $\langle\Lambda_\mathsf{ab}\rangle$, all parties generate $\langle\cdot\rangle$-sharing of a random $\gamma \in \mathbb{Z}_{2^\ell}$ non-interactively and convert it to $[\gamma]$. Parties then compute $[\Lambda_\mathsf{ab}+\gamma]$ by computing $[\Lambda_\mathsf{ab}]$ from $\langle\lambda_\mathsf{a}\rangle, \langle\lambda_\mathsf{b}\rangle$ followed by summing it up with $[\gamma]$. Parties reconstruct this value towards P_king, who then generates $\langle\Lambda_\mathsf{ab}+\gamma\rangle$, from which parties compute $\langle\Lambda_\mathsf{ab}\rangle = \langle\Lambda_\mathsf{ab}+\gamma\rangle - \langle\gamma\rangle$. On obtaining $\langle\Lambda_\mathsf{ab}\rangle$, parties generate $^{\mathcal{E}}[\Lambda_\mathsf{ab}]$ by invoking $\Pi_{\langle\cdot\rangle\to\mathcal{E}[\cdot]}$.

Similarly, for the 4-input multiplication, to obtain $\langle\langle\cdot\rangle\rangle$-sharing of $\mathsf{z} = \mathsf{abcd}$ given the $\langle\langle\cdot\rangle\rangle$-sharing of $\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}$, we can write $\mathsf{z}+\mathsf{r}$ as

$$\begin{aligned}\mathsf{z}-\mathsf{r} &= (\mathsf{m_a}-\lambda_\mathsf{a})(\mathsf{m_b}-\lambda_\mathsf{b})(\mathsf{m_c}-\lambda_\mathsf{c})(\mathsf{m_d}-\lambda_\mathsf{d})-\mathsf{r} \qquad (4)\\ &= \mathsf{M_{abcd}} - \mathsf{M_{bcd}}\lambda_\mathsf{a} - \mathsf{M_{acd}}\lambda_\mathsf{b} - \mathsf{M_{abd}}\lambda_\mathsf{c} - \mathsf{M_{abc}}\lambda_\mathsf{d}\\ &\quad + \mathsf{M_{ab}}\Lambda_\mathsf{cd} + \mathsf{M_{ac}}\Lambda_\mathsf{bd} + \mathsf{M_{ad}}\Lambda_\mathsf{bc} + \mathsf{M_{bc}}\Lambda_\mathsf{ad} + \mathsf{M_{bd}}\Lambda_\mathsf{ac}\\ &\quad + \mathsf{M_{cd}}\Lambda_\mathsf{ab} - \mathsf{m_a}\Lambda_\mathsf{bcd} - \mathsf{m_b}\Lambda_\mathsf{acd} - \mathsf{m_c}\Lambda_\mathsf{abd} - \mathsf{m_d}\Lambda_\mathsf{abc}\\ &\quad + \Lambda_\mathsf{abcd} - \mathsf{r}\end{aligned}$$

Here, parties need to generate the $^{\mathcal{E}}[\cdot]$-sharing of $\Lambda_\mathsf{ab}, \Lambda_\mathsf{ac}, \Lambda_\mathsf{ad}, \Lambda_\mathsf{bc}, \Lambda_\mathsf{bd}, \Lambda_\mathsf{cd}, \Lambda_\mathsf{abc}, \Lambda_\mathsf{abd}, \Lambda_\mathsf{acd}, \Lambda_\mathsf{bcd}, \Lambda_\mathsf{abcd}$. Generation of $^{\mathcal{E}}[\cdot]$-sharing of $\Lambda_\mathsf{ac}, \Lambda_\mathsf{ad}, \Lambda_\mathsf{bc}, \Lambda_\mathsf{bd}$ can proceed similar to generation of $^{\mathcal{E}}[\Lambda_\mathsf{ac}]$ in $\Pi_{\mathsf{3\text{-}mult}}$. Generation of $^{\mathcal{E}}[\cdot]$-sharing of $\Lambda_\mathsf{ab}, \Lambda_\mathsf{cd}$ is carried out by first generating its $\langle\cdot\rangle$-sharing. This enables generation of $^{\mathcal{E}}[\cdot]$-sharing of $\Lambda_\mathsf{abc}, \Lambda_\mathsf{abd}, \Lambda_\mathsf{acd}, \Lambda_\mathsf{bcd}$ following steps similar to generation of $^{\mathcal{E}}[\Lambda_\mathsf{ac}]$ in $\Pi_{\mathsf{3\text{-}mult}}$. Finally, $^{\mathcal{E}}[\Lambda_\mathsf{abcd}-\mathsf{r}]$ is generated similar to generating $^{\mathcal{E}}[\Lambda_\mathsf{abc}+\mathsf{r}]$ in $\Pi_{\mathsf{3\text{-}mult}}$. We omit formal details of 4-input multiplication protocol, $\Pi_{\mathsf{4\text{-}mult}}$, as it is very close to $\Pi_{\mathsf{3\text{-}mult}}$.

Protocol $\Pi_{\mathsf{3\text{-}mult}}(\mathcal{P}, \langle\langle\mathsf{a}\rangle\rangle, \langle\langle\mathsf{b}\rangle\rangle, \langle\langle\mathsf{c}\rangle\rangle)$

Preprocessing:

1. Invoke Π_rand to generate $\langle\mathsf{r}\rangle$ and $\langle\gamma\rangle$ where $\mathsf{r}, \gamma \in \mathbb{Z}_{2^\ell}$. Invoke $\Pi_{\langle\cdot\rangle\to[\cdot]}$ to generate $[\mathsf{r}], [\gamma]$.

2. Invoke $\Pi_{[0]}$ to generate two different $[\cdot]$-shares of 0: $[0_1]$, $[0_2]$.

3. Generation of $^{\mathcal{E}}[\Lambda_\mathsf{ac}], ^{\mathcal{E}}[\Lambda_\mathsf{bc}]$.
- Invoke $\Pi_{\langle\cdot\rangle\cdot\langle\cdot\rangle\to[\cdot]}$ on $\langle\lambda_\mathsf{a}\rangle, \langle\lambda_\mathsf{c}\rangle$ to generate $[\Lambda_\mathsf{ac}]_i$, and compute $[\Lambda_\mathsf{ac}+0_1]_i = [\Lambda_\mathsf{ac}]_i + [0_1]_i$.
- $P_i \in \mathcal{D}$ sends $[\Lambda_\mathsf{ac}+0_1]_i$ to $P_\mathsf{king}(= P_{t+1})$.
- Analogous steps are carried out to generate $[\Lambda_\mathsf{bc}+0_2]$.
- $P_i \in \mathcal{E}\setminus P_{t+1}$ sets $^{\mathcal{E}}[\Lambda_\mathsf{bc}]_i = [\Lambda_\mathsf{bc}+0_2]_i$ and $^{\mathcal{E}}[\Lambda_\mathsf{ac}]_i = [\Lambda_\mathsf{ac}+0_1]_i$.
- P_{t+1} sets $^{\mathcal{E}}[\Lambda_\mathsf{bc}]_{t+1} = [\Lambda_\mathsf{bc}+0_2]_{t+1} + \sum_{i:P_i\in\mathcal{D}}[\Lambda_\mathsf{bc}+0_2]_i$ and $^{\mathcal{E}}[\Lambda_\mathsf{ac}]_{t+1} = [\Lambda_\mathsf{ac}+0_1]_{t+1} + \sum_{i:P_i\in\mathcal{D}}[\Lambda_\mathsf{ac}+0_1]_i$.

4. Generation of $^{\mathcal{E}}[\Lambda_\mathsf{ab}]$.
- Invoke $\Pi_{\langle\cdot\rangle\cdot\langle\cdot\rangle\to[\cdot]}$ on $\langle\lambda_\mathsf{a}\rangle, \langle\lambda_\mathsf{b}\rangle$ to generate $[\Lambda_\mathsf{ab}]_i$, set $[\Lambda_\mathsf{ab}+\gamma]_i = [\Lambda_\mathsf{ab}]_i + [\gamma]_i$, and send $[\Lambda_\mathsf{ab}+\gamma]_i$ to P_king.
- P_king reconstructs $\Lambda_\mathsf{ab}+\gamma$, and sends $\Lambda_\mathsf{ab}+\gamma$ to $P_i \in \mathcal{E}$. Parties non-interactively generate $\langle\Lambda_\mathsf{ab}+\gamma\rangle$ via $\Pi_{\cdot\to\langle\langle\cdot\rangle\rangle}$ and $\Pi_{\langle\langle\cdot\rangle\rangle\to\langle\cdot\rangle}$.
- Compute $\langle\Lambda_\mathsf{ab}\rangle = \langle\Lambda_\mathsf{ab}+\gamma\rangle - \langle\gamma\rangle$ and invoke $\Pi_{\langle\cdot\rangle\to\mathcal{E}[\cdot]}$ on $\langle\Lambda_\mathsf{ab}\rangle$ to generate $^{\mathcal{E}}[\Lambda_\mathsf{ab}]$.

5. Generation of $^{\mathcal{E}}[\Lambda_\mathsf{abc}+\mathsf{r}]$.
- Invoke $\Pi_{\langle\cdot\rangle\cdot\langle\cdot\rangle\to[\cdot]}$ on $\langle\Lambda_\mathsf{ab}\rangle, \langle\lambda_\mathsf{c}\rangle$ to generate $[\Lambda_\mathsf{abc}]_i$, and compute $[\Lambda_\mathsf{abc}+\mathsf{r}]_i = [\Lambda_\mathsf{abc}]_i + [\mathsf{r}]_i$.
- $P_i \in \mathcal{D}$ sends $[\Lambda_\mathsf{abc}+\mathsf{r}]_i$ to P_king.
- $P_i \in \mathcal{E}\setminus P_{t+1}$ sets $^{\mathcal{E}}[\Lambda_\mathsf{abc}+\mathsf{r}]_i = [\Lambda_\mathsf{abc}+\mathsf{r}]_i$.
- P_{t+1} sets $^{\mathcal{E}}[\Lambda_\mathsf{abc}+\mathsf{r}]_{t+1} = [\Lambda_\mathsf{abc}+\mathsf{r}]_{t+1} + \sum_{i:P_i\in\mathcal{D}}[\Lambda_\mathsf{abc}+\mathsf{r}]_i$.

6. $P_i \in \mathcal{E}$ invoke $\Pi_{\langle\cdot\rangle\to\mathcal{E}[\cdot]}$ on $\langle\lambda_\mathsf{a}\rangle, \langle\lambda_\mathsf{b}\rangle$ and $\langle\lambda_\mathsf{c}\rangle$ to generate $^{\mathcal{E}}[\lambda_\mathsf{a}]_i, ^{\mathcal{E}}[\lambda_\mathsf{b}]_i, ^{\mathcal{E}}[\lambda_\mathsf{c}]_i$, respectively.

Online:

1. $P_i \in \mathcal{E}$ computes and sends $^{\mathcal{E}}[\zeta]_i = -\mathsf{M_{ac}}{}^{\mathcal{E}}[\lambda_\mathsf{b}]_i - \mathsf{M_{bc}}{}^{\mathcal{E}}[\lambda_\mathsf{a}]_i - \mathsf{M_{ab}}{}^{\mathcal{E}}[\lambda_\mathsf{c}]_i + \mathsf{m_a}{}^{\mathcal{E}}[\Lambda_\mathsf{bc}]_i + \mathsf{m_b}{}^{\mathcal{E}}[\Lambda_\mathsf{ac}]_i + \mathsf{m_c}{}^{\mathcal{E}}[\Lambda_\mathsf{ab}]_i - {}^{\mathcal{E}}[\Lambda_\mathsf{abc}+\mathsf{r}]_i$ to P_king.

2. P_king computes and sends $\mathsf{z}-\mathsf{r} = \mathsf{M_{abc}} + \sum_{i:P_i\in\mathcal{E}}{}^{\mathcal{E}}[\zeta]_i$ to $P_i \in \mathcal{E}$.

3. Invoke $\Pi_{\cdot\to\langle\langle\cdot\rangle\rangle}$ on $\mathsf{z}-\mathsf{r}$ to generate $\langle\langle\mathsf{z}-\mathsf{r}\rangle\rangle$, and compute $\langle\langle\mathsf{z}\rangle\rangle = \langle\langle(\mathsf{z}-\mathsf{r})\rangle\rangle + \langle\langle\mathsf{r}\rangle\rangle$.

Fig. 24: Semi-honest: 3-input multiplication protocol

B. Malicious protocols

a) Input sharing: This protocol ($\Pi^\mathsf{M}_\mathsf{Sh}(P_s, \mathsf{a})$) is similar to the semi-honest one, where to enable P_s to generate $\langle\langle\mathsf{a}\rangle\rangle$, parties generate $\langle\lambda_\mathsf{a}\rangle$ such that P_s learns λ_a, followed by P_s sending the masked value $\mathsf{m_a} = \mathsf{a}+\lambda_\mathsf{a}$ to all. However, note that a corrupt P_s can cause inconsistency among the honest parties by sending different masked values. To ensure the same value is received by all, parties perform a hash-based consistency check, denoted by Π_agree (§II), where each party sends a hash of the received masked value(s) to every other party and `aborts` if it receives inconsistent hashes. Note that this check for all the inputs can be combined, thereby amortizing the cost.

Protocol $\Pi^{\mathsf{M}}_{\mathsf{Sh}}(P_s, \mathsf{a})$

Preprocessing: Invoke $\Pi_{\mathsf{pRand}}(P_s)$ to generate $\langle \lambda_{\mathsf{a}} \rangle$, with P_s learning λ_{a} where $\lambda_{\mathsf{a}} \in \mathbb{Z}_{2^\ell}$.

Online: P_s computes and sends $\mathsf{m}_{\mathsf{a}} = \mathsf{a} + \lambda_{\mathsf{a}}$ to all $P_i \in \mathcal{P}$.

Verification: Invoke Π_{agree} on $\{\mathsf{m}_{\mathsf{a}}\}$.

Fig. 25: Malicious: Input sharing protocol

b) Reconstruction: To reconstruct $\langle\!\langle \cdot \rangle\!\rangle$-shared value a towards $P_s \in \mathcal{P}$, observe that each share that P_s misses is held by $t+1$ other parties. Each of these parties sends the missing share to P_s. If the received values for a share are consistent, P_s uses this value to perform reconstruction, and `aborts` otherwise. As an optimization, one party can send the missing share while reconstructing several values, and t others can send its hash.

Protocol $\Pi^{\mathsf{fair}}_{\mathsf{Rec}}(\langle\!\langle \mathsf{z} \rangle\!\rangle)$

Preprocessing:

1. Invoke Π_{rand} to generate $\langle \lambda_{\mathsf{z}} \rangle$ where $\lambda_{\mathsf{z}} \in_R \mathbb{Z}_{2^\ell}$.

2. For $j \in \{1, \ldots, \mathsf{q}\}$:
- Each $P_i \in \mathcal{T}_j$ generates commitments on $\langle \lambda_{\mathsf{z}} \rangle_{\mathcal{T}_j}$ using the common randomness, and sends to all other parties.
- $P_i \notin \mathcal{T}_j$ `aborts` if commitments for $\langle \lambda_{\mathsf{z}} \rangle_{\mathcal{T}_j}$ are inconsistent.

Online:

1. Parties broadcast an alive bit, indicating that they did not `abort`.

2. If all parties are alive, $P_i \in \mathcal{P}$ sends the decommitment to the shares in $\langle \lambda_{\mathsf{z}} \rangle_i$ to the respective parties.

3. Parties use the valid decommitment to obtain the missing share of λ_{z}, reconstruct λ_{z}, and compute $\mathsf{z} = \mathsf{m}_{\mathsf{z}} - \lambda_{\mathsf{z}}$.

Fig. 26: Fair: Reconstruction protocol

Fairness is a stronger security notion than security with abort, where, during reconstruction, either all parties learn the output or none do. For fair reconstruction, we extend the techniques in [74] to the n-party setting, where commitments are generated on each share of the mask (required to reconstruct z) by $t+1$ parties in the preprocessing phase. During the online phase, these are decommitted towards the respective parties if all parties are alive (did not `abort`). Since there is at least one honest party among every set of $t+1$ parties, if all honest parties are alive, then parties are guaranteed to obtain the correct decommitment of the missing share from the honest party, and all honest parties can reconstruct the output. Else, none of the parties will obtain the output.

c) Multiplication: The maliciously secure multiplication protocol ($\Pi^{\mathsf{M}}_{\mathsf{mult}}$) appears in Fig. 27.

Overcoming the privacy breach described in [47] We elaborate on the privacy breach that arises due to deferring the correctness check and how it is overcome in our case. We first explain the attack that a malicious adversary can launch if reconstruction towards P_{king} is performed by relying on RSS (or Shamir sharing) naively and further justify why it gets bypassed in our protocol. Consider a circuit with two sequential multiplication gates with the output of the first gate, say a, going as input to the second gate. Let b denote the other input to the second multiplication gate, and z denote its output. In a P_{king} based approach for multiplication, t parties send their respective (RSS/Shamir) share of a masked value to P_{king}. In particular, for the first multiplication gate in the circuit mentioned above, t parties send their corresponding share of $\mathsf{a} - \mathsf{r}_{\mathsf{a}}$ to P_{king}, who reconstructs it and sends it back to all. Delaying the verification allows a malicious P_{king} to send an inconsistent value of $\mathsf{a} - \mathsf{r}_{\mathsf{a}}$ to the parties, using which it can learn the private input b, as follows. Suppose P_{king} sends the correct $\mathsf{a} - \mathsf{r}_{\mathsf{a}}$ to all but one out of the remaining t online parties, to which it sends $\mathsf{a} - \mathsf{r}_{\mathsf{a}} + \delta$. Owing to this, for the next multiplication gate P_{king} receives the shares of $\mathsf{z} - \mathsf{r}_{\mathsf{z}}$ from the former $t-1$ parties and a share of $(\mathsf{a}+\delta)\mathsf{b} - \mathsf{r}_{\mathsf{z}} = \mathsf{z} + \delta\mathsf{b} - \mathsf{r}_{\mathsf{z}}$ from the latter party. Having obtained these and additionally using the shares of $\mathsf{z} - \mathsf{r}_{\mathsf{z}}$ and $\mathsf{z} + \delta\mathsf{b} - \mathsf{r}_{\mathsf{z}}$ corresponding to the t corrupt parties including itself, a malicious P_{king} can reconstruct $\mathsf{z} - \mathsf{r}_{\mathsf{z}}$ as well as $\mathsf{z} + \delta\mathsf{b} - \mathsf{r}_{\mathsf{z}}$, thus learning b in clear. The crux of this attack lies in the fact that a malicious adversary corrupting t parties including P_{king} already possesses t shares each of $\mathsf{z} - \mathsf{r}_{\mathsf{z}}$ and $\mathsf{z} + \delta\mathsf{b} - \mathsf{r}_{\mathsf{z}}$. Thus, an additional share of these obtained from the online parties allows it to carry out the attack successfully. However, the same does not hold for the case of additive (${}^{\mathcal{E}}[\cdot]$) sharing.

Protocol $\Pi^{\mathsf{M}}_{\mathsf{mult}}(\mathcal{P}, \langle\!\langle \mathsf{a} \rangle\!\rangle, \langle\!\langle \mathsf{b} \rangle\!\rangle, \mathsf{isTr})$

$\mathsf{isTr} = 1$ denotes that truncation is required and $\mathsf{isTr} = 0$ denotes otherwise.

Preprocessing:

1. If $\mathsf{isTr} = 0$: invoke Π_{rand} to generate $\langle \mathsf{r} \rangle$ where $\mathsf{r} \in \mathbb{Z}_{2^\ell}$. Invoke $\Pi_{\langle\cdot\rangle \to [\cdot]}$ and $\Pi_{\langle\cdot\rangle \to \langle\!\langle\cdot\rangle\!\rangle}$ on $\langle \mathsf{r} \rangle$ to generate $[\mathsf{r}]$ and $\langle\!\langle \mathsf{r} \rangle\!\rangle$, respectively.

2. Else, invoke $\Pi^{\mathsf{M}}_{\mathsf{dsBits}}(\mathcal{P}, 1)$ (Fig. 22) to generate $\langle\!\langle \mathsf{r} \rangle\!\rangle$, $\langle\!\langle \mathsf{r}^d \rangle\!\rangle$, and $\Pi_{\langle\!\langle\cdot\rangle\!\rangle \to [\cdot]}$ on $\langle\!\langle \mathsf{r} \rangle\!\rangle$ to generate $[\mathsf{r}]$.

3. Invoke Π_{multPre} on $\langle \lambda_{\mathsf{a}} \rangle, \langle \lambda_{\mathsf{b}} \rangle$ to generate $\langle \Lambda_{\mathsf{ab}} \rangle$.

4. $P_i \in \mathcal{E}$ invokes $\Pi_{\langle\cdot\rangle \to {}^{\mathcal{E}}[\cdot]}$ on $\langle \Lambda_{\mathsf{ab}} \rangle$, $\langle \lambda_{\mathsf{a}} \rangle$, $\langle \lambda_{\mathsf{b}} \rangle$ and $\langle \mathsf{r} \rangle$ to generate ${}^{\mathcal{E}}[\Lambda_{\mathsf{ab}}]$, ${}^{\mathcal{E}}[\lambda_{\mathsf{a}}]$, ${}^{\mathcal{E}}[\lambda_{\mathsf{b}}]$ and ${}^{\mathcal{E}}[\mathsf{r}]$, respectively.

Online:

1. $P_i \in \mathcal{E}$ computes ${}^{\mathcal{E}}[\zeta]_i = -\mathsf{m}_{\mathsf{a}} {}^{\mathcal{E}}[\lambda_{\mathsf{b}}]_i - \mathsf{m}_{\mathsf{b}} {}^{\mathcal{E}}[\lambda_{\mathsf{a}}]_i + {}^{\mathcal{E}}[\Lambda_{\mathsf{ab}} - \mathsf{r}]_i$, and sends ${}^{\mathcal{E}}[\zeta]_i$ to P_{king}.

2. P_{king} reconstructs ζ, computes and sends $\mathsf{z} - \mathsf{r} = \zeta + \mathsf{M}_{\mathsf{ab}}$ to all parties[a].

3. If $\mathsf{isTr} = 0$: invoke $\Pi_{\cdot \to \langle\!\langle\cdot\rangle\!\rangle}$ on $\mathsf{z} - \mathsf{r}$ to generate $\langle\!\langle \mathsf{z} - \mathsf{r} \rangle\!\rangle$, and compute $\langle\!\langle \mathsf{z} \rangle\!\rangle = \langle\!\langle \mathsf{z} - \mathsf{r} \rangle\!\rangle + \langle\!\langle \mathsf{r} \rangle\!\rangle$.

4. Else, invoke $\Pi_{\cdot \to \langle\!\langle\cdot\rangle\!\rangle}$ on $(\mathsf{z} - \mathsf{r})^d$ to generate $\langle\!\langle (\mathsf{z} - \mathsf{r})^d \rangle\!\rangle$, and compute $\langle\!\langle \mathsf{z}^d \rangle\!\rangle = \langle\!\langle (\mathsf{z} - \mathsf{r})^d \rangle\!\rangle + \langle\!\langle \mathsf{r}^d \rangle\!\rangle$.

Verification for all multiplication gates: Invoke Π_{Vrfy} on $\langle\!\langle\cdot\rangle\!\rangle$-shares of $(\mathsf{a}_1, \mathsf{b}_1, \mathsf{z}_1), \ldots, (\mathsf{a}_m, \mathsf{b}_m, \mathsf{z}_m)$ which denote the inputs and outputs of the m multiplication gates whose correctness is to be verified.

[a] $\mathsf{z} - \mathsf{r}$ is sent to parties in $\mathcal{E}$ during the online phase computation whereas it is sent to parties in $\mathcal{D}$ in a single shot before verification begins.

Fig. 27: Malicious: Multiplication protocol

Notice that in our protocol, during reconstruction towards P_{king}, any redundancy due to $\langle\langle\cdot\rangle\rangle$-sharing is eliminated with parties switching to ${}^{\mathcal{E}}[\cdot]$-sharing (additive sharing among parties in $\mathcal{E}$). Due to this, even if P_{king} sends inconsistent values to the parties, the ${}^{\mathcal{E}}[\cdot]$-share of $\mathsf{z}-\mathsf{r}_\mathsf{z}$ or $\mathsf{z}+\delta\mathsf{b}-\mathsf{r}_\mathsf{z}$ that it receives, corresponds to an additive share defined with respect to parties in $\mathcal{E}$. Hence, this additionally received additive share cannot be combined with the shares held by the t corrupt parties to perform the reconstruction. Thus, the earlier strategy of P_{king} of using these additional shares in conjunction with the t corrupt shares to reconstruct $\mathsf{z}-\mathsf{r}_\mathsf{z}$ and $\mathsf{z}+\delta\mathsf{b}-\mathsf{r}_\mathsf{z}$ does not hold. The primary reason which prevents the attack is the elimination of redundancy in the sharing scheme by switching to $(t+1)$-out-of-$(t+1)$ additive sharing (${}^{\mathcal{E}}[\cdot]$-sharing) for the set of parties in $\mathcal{E}$, which is known to withstand this attack [47].

Discussion about [38] The above attack can be circumvented by making P_{king} broadcast the reconstructed value to all the parties, as discussed in [38]. To further optimize the protocol by requiring only $t+1$ parties to be active in the online phase, they rely on broadcast with abort, which comprises two phases–(i) *send*: where P_{king} sends the value to the recipients, and (ii) *verification*: where the recipients exchange hash of the received value among themselves, and abort in case of inconsistency. However, for amortization, they defer the verification (even with respect to broadcast) towards the end of the protocol, thus making their protocol susceptible to the aforementioned attack. We observe that one fix is to perform the verification with respect to broadcast after each level in the circuit. This, however, requires all the parties to be online. An optimization to let only the $t+1$ parties in the online phase to perform this verification after each level, thereby allowing the remaining t parties to be shut off. Specifically, this involves performing *verification* where the online parties exchange the hash of the received value and abort in case of inconsistency. When the remainder t (offline) parties come online towards the end of the protocol for verifying the correctness of the multiplication gates, this verification should be preceded by first verifying the consistency of the values broadcast by P_{king} to the offline parties (and involves participation of all n parties). Since the online phase involves broadcasting the reconstructed value to t other online parties, this amounts to an exchange of $\mathcal{O}(t^2)$ hashes after each level, thereby incurring a circuit depth-dependent overhead in the communication cost as well as the rounds. In order for the communication cost to get amortized, it is required that the circuit has $\mathcal{O}(t^2)$ gates at each level. However, the overhead in terms of number of rounds persists.

Multiplication with truncation – Instantiating $\mathcal{F}^{\mathsf{M}}_{\mathsf{TrGen}}$ with maliciously secure doubly shared bits generation protocol: As mentioned earlier, $\mathcal{F}^{\mathsf{M}}_{\mathsf{TrGen}}$ (Fig. 28) can be realized using the maliciously secure variant of Π_{dsBits}, denoted as $\Pi^{\mathsf{M}}_{\mathsf{dsBits}}$. This protocol is similar to the semi-honest protocol except with the following differences to account for malicious behaviour. The $\langle\cdot\rangle$-shares of $\mathsf{e}_i=\mathsf{a}^2$ are generated by invoking Π_{multPre} instead of relying on $\Pi_{\langle\cdot\rangle\cdot\langle\cdot\rangle\to[\cdot]}$. This ensures generation of correct $\langle\cdot\rangle$-shares of e_i, and malicious behaviour, if any, will lead to an `abort`. Following this, e_i is either correctly reconstructed towards all or parties `abort`. This ensures that an adversary cannot lead to reconstruction of an incorrect e_i. Concretely, for reconstruction, similar to multiplication, every party sends its $\langle\cdot\rangle$-share to every other party, and `aborts` in case of inconsistencies in the received values[5]. The rest of the protocol steps (which are non-interactive) remain unchanged, and hence a formal protocol is omitted.

Functionality $\mathcal{F}^{\mathsf{M}}_{\mathsf{TrGen}}$

$\mathcal{F}^{\mathsf{M}}_{\mathsf{TrGen}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$.

Input: $\mathcal{F}^{\mathsf{M}}_{\mathsf{TrGen}}$ optionally receives a special command, $(\mathtt{abort},\mathsf{P})$, from $\mathcal{S}$ indicating that honest parties in $\mathcal{P}$ with indices in P should `abort`.

$\mathcal{F}^{\mathsf{M}}_{\mathsf{TrGen}}$ proceeds as follows.
- Samples random $\mathsf{r}\in\mathbb{Z}_{2^\ell}$, and computes $\mathsf{r}^d=\mathsf{r}/2^d$.
- Generates $\langle\langle\cdot\rangle\rangle$-shares of r,r^d.
- Let y_s denote the $\langle\langle\cdot\rangle\rangle$-shares of r,r^d for party $P_s\in\mathcal{P}$. If received $(\mathtt{abort},\mathsf{P})$ from $\mathcal{S}$, set $\mathsf{y}_s=\mathtt{abort}$ for P_s, where $s\in\mathsf{P}$.

Output: Send $(\mathsf{Output},\mathsf{y}_s)$ to every $P_s\in\mathcal{P}$.

Fig. 28: Ideal functionality $\mathcal{F}^{\mathsf{M}}_{\mathsf{TrGen}}$

d) Multi-input multiplication: The malicious variant of multi-input multiplication protocol, at a high level, can be viewed as an amalgamation of the semi-honest multi-input multiplication and the malicious multiplication protocol. For the case of 3-input multiplication, recall that the semi-honest protocol to compute $\langle\langle\mathsf{z}\rangle\rangle$ given $\langle\langle\mathsf{a}\rangle\rangle,\langle\langle\mathsf{b}\rangle\rangle$ and $\langle\langle\mathsf{c}\rangle\rangle$ where $\mathsf{z}=\mathsf{abc}$ requires parties to obtain ${}^{\mathcal{E}}[\Lambda_{\mathsf{ab}}],{}^{\mathcal{E}}[\Lambda_{\mathsf{ac}}],{}^{\mathcal{E}}[\Lambda_{\mathsf{bc}}]$ and ${}^{\mathcal{E}}[\Lambda_{\mathsf{abc}}]$ in the preprocessing phase, which is then used to reconstruct m_z in the online phase.

Since parties in $\mathcal{E}$ are required to hold the correct ${}^{\mathcal{E}}[\cdot]$-sharings before the online phase begins, as in the case of multiplication, the techniques from semi-honest protocol fail in this setting. Hence, our protocol uses 4 instances of a maliciously secure multiplication protocol in the preprocessing phase, one each to compute $\langle\Lambda_{\mathsf{ab}}\rangle,\langle\Lambda_{\mathsf{ac}}\rangle,\langle\Lambda_{\mathsf{bc}}\rangle$ and $\langle\Lambda_{\mathsf{abc}}\rangle$. Each of the $\langle\cdot\rangle$-sharing is further converted to ${}^{\mathcal{E}}[\cdot]$-sharing using $\Pi_{\langle\cdot\rangle\to\mathcal{E}[\cdot]}$ to ensure active participation of only $t+1$ parties in the online phase for reconstruction of $\mathsf{z}-\mathsf{r}$. Further, to detect malicious behaviour during reconstruction of $\mathsf{z}-\mathsf{r}$, a verification check similar to the multiplication protocol is performed such that parties `abort` if the check fails.

For 4-input multiplication, parties obtain $\langle\langle\cdot\rangle\rangle$-sharing of $\mathsf{z}=\mathsf{abcd}$ using $\mathsf{z}-\mathsf{r}=(\mathsf{m}_\mathsf{a}-\lambda_\mathsf{a})(\mathsf{m}_\mathsf{b}-\lambda_\mathsf{b})(\mathsf{m}_\mathsf{c}-\lambda_\mathsf{c})(\mathsf{m}_\mathsf{d}-\lambda_\mathsf{d})-\mathsf{r}$. The protocol proceeds in a similar manner as the 3-input case by delegating the computation of product terms to the preprocessing phase.

e) Dot product: To generate $\langle\langle\mathsf{z}\rangle\rangle$ for $\mathsf{z}=\vec{\mathsf{x}}\odot\vec{\mathsf{y}}$ where $\vec{\mathsf{x}}$ and $\vec{\mathsf{y}}$ are vectors of size n and are $\langle\langle\cdot\rangle\rangle$-shared, the protocol proceeds similar to the semi-honest variant. That is, in the preprocessing phase parties in $\mathcal{E}$ obtain ${}^{\mathcal{E}}[\cdot]$-shares of $\sigma=\sum_{k=1}^{\mathsf{n}}\lambda_{\mathsf{x}_k}\lambda_{\mathsf{y}_k}$ and $\lambda_{\mathsf{x}_k},\lambda_{\mathsf{y}_k}$ for $k\in\{1,\ldots,\mathsf{n}\}$. Although the latter two can be computed by parties locally with an invocation of $\Pi_{\langle\cdot\rangle\to\mathcal{E}[\cdot]}$ (Fig. 19), computation of the former differs significantly from the semi-honest protocol. For this, we extend the ideas from SWIFT [56] and generate σ, by executing a maliciously secure dot product protocol Π_{dotPre} (abstracted as a functionality $\mathcal{F}_{\mathsf{DotPPre}}$ in Fig. 29). Specifically,

[5]This can be optimized similar to the online phase of multiplication, where the value is first reconstructed towards P_{king} who sends the reconstructed value to all, followed by verifying its correctness via the verification check.

parties invoke Π_{dotPre} on $\langle\cdot\rangle$-shares of $\vec{\lambda_{\mathsf{x}}} = (\lambda_{\mathsf{x}_1}, \ldots, \lambda_{\mathsf{x}_n})$ and $\vec{\lambda_{\mathsf{y}}} = (\lambda_{\mathsf{y}_1}, \ldots, \lambda_{\mathsf{y}_n})$ to compute $\langle\sigma\rangle$, followed by an invocation of $\Pi_{\langle\cdot\rangle\to\mathcal{E}[\cdot]}$ to obtain ${}^{\mathcal{E}}[\sigma]$. Having computed the necessary preprocessing data, the online phase proceeds similarly to the semi-honest protocol where parties reconstruct $\mathsf{z}-\mathsf{r}$ via P_{king}. To account for misbehaviour, the protocol is augmented with a verification phase similar to that in malicious multiplication.

Observe that a trivial realization of $\mathcal{F}_{\mathsf{DotPPre}}$ can be reduced to n instances of multiplication. However, we extend the ideas from [14], [15], [56] to eliminate the vector-size dependency in the preprocessing phase. For this, we instantiate Π_{dotPre} using a semi-honest dot product protocol [48] whose cost matches that of semi-honest multiplication [31], followed by a verification phase where the cost of verification can be amortized away for multiple dot products, thereby resulting in vector-size independent preprocessing.

Elaborately, the semi-honest dot product [48] protocol takes as input $\langle\vec{x}\rangle, \langle\vec{y}\rangle$ where $\vec{x}, \vec{y}$ are vectors of size n, and outputs $\langle\mathsf{z}\rangle = \langle\vec{x}\odot\vec{y}\rangle$. For this, parties invoke $\Pi_{\langle\cdot\rangle\cdot\langle\cdot\rangle\to[\cdot]}$ on each element in $\vec{x}, \vec{y}$ and sum these up to generate $[\rho] = [\vec{x}\odot\vec{y}]$. These shares are randomized by summing with $[\mathsf{r}]$ (converted from $\langle\mathsf{r}\rangle$) for a random r, and the sum $\mathsf{z}+\mathsf{r} = (\vec{x}\odot\vec{y})+\mathsf{r}$ is reconstructed towards P_{king}, who sends the reconstructed $\mathsf{z}+\mathsf{r}$ to parties in $\mathcal{E}$. All parties then non-interactively generate $\langle\mathsf{z}+\mathsf{r}\rangle$ by setting one of its share as $\mathsf{z}+\mathsf{r}$ and the others as 0. Given $\langle\mathsf{z}+\mathsf{r}\rangle, \langle\mathsf{r}\rangle$, parties can compute $\langle\mathsf{z}\rangle = \langle\mathsf{z}+\mathsf{r}\rangle - \langle\mathsf{r}\rangle$. Observe that communication of $[\mathsf{z}+\mathsf{r}]$ to P_{king} requires $2t$ elements, while communicating $\mathsf{z}+\mathsf{r}$ to parties in $\mathcal{E}$ requires t elements, resulting in a matching cost of $3t$ elements as that required for semi-honest multiplication [31].

Functionality $\mathcal{F}_{\mathsf{DotPPre}}$

$\mathcal{F}_{\mathsf{DotPPre}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$. Let $\mathcal{T}_i$ be the set of the honest parties.

Input: $\mathcal{F}_{\mathsf{DotPPre}}$ receives the $\langle\cdot\rangle$-shares of the vectors $\vec{\mathsf{a}} = (\mathsf{a}_1, \ldots, \mathsf{a}_n)$ and $\vec{\mathsf{b}} = (\mathsf{b}_1, \ldots, \mathsf{b}_n)$ from the parties. $\mathcal{F}_{\mathsf{DotPPre}}$ also receives $\langle\cdot\rangle$-shares of $\mathsf{z} = \vec{\mathsf{a}}\odot\vec{\mathsf{b}}$ of corrupt parties from $\mathcal{S}$. $\mathcal{S}$ is also allowed to send a special command, $(\mathtt{abort}, \mathsf{P})$, which indicates that parties in $\mathcal{P}$ with indices in P should $\mathtt{abort}$.

$\mathcal{F}_{\mathsf{DotPPre}}$ proceeds as follows.

– Reconstruct $\mathsf{a}_k, \mathsf{b}_k$ for $k \in \{1, \ldots, \mathsf{n}\}$ using the shares received from honest parties and compute $\mathsf{z} = \sum_{k=1}^{\mathsf{n}} \mathsf{a}_k \cdot \mathsf{b}_k$.

– Compute the $\langle\cdot\rangle$-share of z to be held by the set of honest parties as the difference between z and the sum of $\langle\cdot\rangle$-shares of z received from corrupt parties.

– Let y_s denote the $\langle\cdot\rangle$-shares of z for party $P_s \in \mathcal{P}$. If received $(\mathtt{abort}, \mathsf{P})$ from $\mathcal{S}$, set $\mathsf{y}_s = \mathtt{abort}$ for P_s, where $s \in \mathsf{P}$.

Output: Send $(\mathsf{Output}, \mathsf{y}_s)$ to every $P_s \in \mathcal{P}$.

Fig. 29: Ideal functionality for Π_{dotPre}

To verify the correctness of this dot product computation, we extend the verification technique for multiplication in [15], to verify the correctness of dot product. We give a high level idea of how the verification of m dot product triples $(\vec{x_1}, \vec{y_1}, \mathsf{z}_1), \ldots, (\vec{x_m}, \vec{y_m}, \mathsf{z}_m)$, can be performed. For this, correctness of the dot product triples can be verified by taking a random linear combination,

$$\beta = \sum_{k=1}^{m} \theta_k \cdot \left(\mathsf{z}_k - \sum_{j=1}^{\mathsf{n}} \mathsf{x}_{kj} \cdot \mathsf{y}_{kj} \right)$$

where $\{\theta_k\}_{k=1}^m$ is randomly chosen by all the parties and checking if $\beta = 0$. Given $\langle\cdot\rangle$-shares of $\vec{x}_k, \vec{y}_k, \mathsf{z}_k$ for $k \in \{1, \ldots, m\}$, parties can compute an additive share ($[\cdot]$-share) of β by invoking $\Pi_{\langle\cdot\rangle\cdot\langle\cdot\rangle\to[\cdot]}$. However, since $[\cdot]$-sharing does not allow for robust reconstruction, the approach is to generate $\langle\beta\rangle$ and then robustly reconstruct it and check equality with 0. To generate $\langle\beta\rangle$, parties first $\langle\cdot\rangle$-share (via $\Pi_{\langle\cdot\rangle}$, §A-A) their $[\cdot]$-share of

$$\psi = \sum_{k=1}^{m} \theta_k \cdot \sum_{j=1}^{\mathsf{n}} \mathsf{x}_{kj} \cdot \mathsf{y}_{kj}.$$

Let ψ^i denote the $[\cdot]$-share of ψ held by P_i. Given $\langle\psi^i\rangle$ for $i \in \{1, \ldots, n\}$, parties can compute

$$\langle\beta\rangle = \sum_{k=1}^{m} \theta_k \cdot \langle\mathsf{z}_k\rangle - \sum_{i=1}^{n} \langle\psi^i\rangle$$

and reconstruct β. It is, however, required to ensure that every party P_i $\langle\cdot\rangle$-shares the correct ψ^i. To check the correctness of ψ^i, parties need to verify if

$$\psi^i - \sum_{k=1}^{m} \theta_k \left(\sum_{j=1}^{\mathsf{n}} \mathsf{x}^i_{kj} \cdot \mathsf{y}^i_{kj} \right) = 0 \tag{5}$$

where $\mathsf{x}^i_{kj}, \mathsf{y}^i_{kj}$ denote the $\langle\cdot\rangle$-share of $\mathsf{x}_{kj}, \mathsf{y}_{kj}$ held by P_i. Note that following along the lines of $\Pi_{\to\langle\langle\cdot\rangle\rangle}$, parties can generate these $\langle\cdot\rangle$-share of $\mathsf{x}^i_{kj}, \mathsf{y}^i_{kj}$ from $\langle\cdot\rangle$-shares of $\mathsf{x}_{kj}, \mathsf{y}_{kj}$, non-interactively. Now, setting $\mathsf{a}_{kj} = \theta_k \mathsf{x}^i_{kj}, \mathsf{b}_{kj} = \mathsf{y}^i_{kj}, \mathsf{c} = \psi^i$, for $k \in \{1, \ldots, m\}$, Eq. (5), can re-written as

$$\begin{aligned} &\mathsf{c} - \sum_{k=1}^{m} \sum_{j=1}^{\mathsf{n}} \mathsf{a}_{kj} \mathsf{b}_{kj} = 0 \\ \implies &\mathsf{c} - \sum_{l=1}^{m\mathsf{n}} \tilde{\mathsf{a}}_l \tilde{\mathsf{b}}_l = 0 \end{aligned} \tag{6}$$

The correctness of Eq. (6) can be verified by invoking $\mathcal{F}^{\mathtt{abort}}_{\mathsf{proveDeg2Rel}}$ (see section 3 of [15] for the definition and its instantiation), which takes as input $\langle\cdot\rangle$-shares of $\tilde{\mathsf{a}}_l, \tilde{\mathsf{b}}_l, \mathsf{c}$ for $l \in \{1, \ldots, m\mathsf{n}\}$, which are known in clear to party P_i, and verifies if Eq. (6) holds. The protocol realizing $\mathcal{F}^{\mathtt{abort}}_{\mathsf{proveDeg2Rel}}$ for all n parties requires communicating $\mathcal{O}(n\log(m\mathsf{n})+n)$ extended ring elements per party. Further, since steps other than $\mathcal{F}^{\mathtt{abort}}_{\mathsf{proveDeg2Rel}}$ require sharing and reconstructing one element, it adds a small constant cost, resulting in the communication cost for verifying m dot products for vector size n being $\mathcal{O}(n\log(m\mathsf{n})+n)$ extended ring elements per party.

Appendix C
Building Blocks

For completeness, we discuss the building blocks used in our framework. These blocks are known from the literature [57], [73] and we show how these can be extended to n-party setting.

A. Semi-honest building blocks

a) Bit to arithmetic : Given Boolean shares $\langle\langle \mathsf{b} \rangle\rangle^{\mathbf{B}}$ of bit b, protocol Π_{bit2A} generates its arithmetic shares, $\langle\langle \mathsf{b}^{\mathsf{R}} \rangle\rangle$ over $\mathbb{Z}_{2^\ell}$ (Fig. 30). Here, b^{R} denotes the arithmetic values of b over the ring $\mathbb{Z}_{2^\ell}$. The idea is to generate a randomized version, $\zeta = \mathsf{b} \oplus \mathsf{r}$ of b, and then recover arithmetic shares of b as $\mathsf{b} = \zeta \oplus \mathsf{r}$ by performing arithmetic equivalent of XOR ($\mathsf{x} \oplus \mathsf{y} = \mathsf{x}^{\mathsf{R}} + \mathsf{y}^{\mathsf{R}} - 2\mathsf{x}^{\mathsf{R}}\mathsf{y}^{\mathsf{R}}$).

b) Bit injection: Π_{BitInj} facilitates generation of $\langle\langle \mathsf{b}^{\mathsf{R}} \cdot \mathsf{v} \rangle\rangle$ given $\langle\langle \mathsf{b} \rangle\rangle^{\mathbf{B}}, \langle\langle \mathsf{v} \rangle\rangle$ for $\mathsf{b} \in \mathbb{Z}_2$ and $\mathsf{v} \in \mathbb{Z}_{2^\ell}$. As seen in [57],

$$\begin{aligned}\mathsf{b}^{\mathsf{R}}\mathsf{v} &= (\mathsf{m}_{\mathsf{b}} \oplus \lambda_{\mathsf{b}})^{\mathsf{R}}(\mathsf{m}_{\mathsf{v}} - \lambda_{\mathsf{v}}) \\ &= \mathsf{m}_{\mathsf{b}}^{\mathsf{R}}\mathsf{m}_{\mathsf{v}} - \mathsf{m}_{\mathsf{b}}^{\mathsf{R}}\lambda_{\mathsf{v}} + (2\mathsf{m}_{\mathsf{b}}^{\mathsf{R}} - 1)(\lambda_{\mathsf{b}}^{\mathsf{R}}\lambda_{\mathsf{v}} - \mathsf{m}_{\mathsf{v}}\lambda_{\mathsf{b}}^{\mathsf{R}})\end{aligned}$$

Given ${}^{\mathcal{E}}[\cdot]$-shares of $\lambda_{\mathsf{v}}, \lambda_{\mathsf{b}}^{\mathsf{R}}, \lambda_{\mathsf{b}}^{\mathsf{R}}\lambda_{\mathsf{v}}, \mathsf{r}$ and $\langle\langle \mathsf{r} \rangle\rangle$ for $\mathsf{r} \in \mathbb{Z}_{2^\ell}$, and the knowledge that $\mathsf{m}_{\mathsf{v}}, \mathsf{m}_{\mathsf{b}}^{\mathsf{R}}$ is held by all parties in $\mathcal{E}$, parties can compute ${}^{\mathcal{E}}\left[\mathsf{b}^{\mathsf{R}}\mathsf{v} + \mathsf{r}\right]$, reconstruct it via P_{king} and generate $\langle\langle \mathsf{b}^{\mathsf{R}}\mathsf{v} + \mathsf{r} \rangle\rangle$. $\langle\langle \mathsf{b}^{\mathsf{R}}\mathsf{v} \rangle\rangle$ can then be computed as $\langle\langle \mathsf{b}^{\mathsf{R}}\mathsf{v} \rangle\rangle = \langle\langle \mathsf{b}^{\mathsf{R}}\mathsf{v} + \mathsf{r} \rangle\rangle - \langle\langle \mathsf{r} \rangle\rangle$. To facilitate this, in the preprocessing phase parties generate ${}^{\mathcal{E}}[\cdot]$-shares of $\mathsf{r}, \lambda_{\mathsf{v}}, \lambda_{\mathsf{b}}^{\mathsf{R}}, \lambda_{\mathsf{b}}^{\mathsf{R}}\lambda_{\mathsf{v}}$, and $\langle\langle \mathsf{r} \rangle\rangle$. Here, ${}^{\mathcal{E}}[\mathsf{r}], {}^{\mathcal{E}}[\lambda_{\mathsf{v}}]$ and $\langle\langle \mathsf{r} \rangle\rangle$ are generated as in the preprocessing of multiplication, ${}^{\mathcal{E}}\left[\lambda_{\mathsf{b}}^{\mathsf{R}}\right]$ is generated via Π_{bit2A} followed by $\Pi_{\langle\langle\cdot\rangle\rangle \to \mathcal{E}[\cdot]}$ (§II), and ${}^{\mathcal{E}}\left[\lambda_{\mathsf{b}}^{\mathsf{R}}\lambda_{\mathsf{v}}\right]$ is generated as in the preprocessing of multiplication.

Protocol $\Pi_{\mathsf{bit2A}}(\mathcal{P}, \langle\langle \mathsf{b} \rangle\rangle^{\mathbf{B}})$

Preprocessing:

1. Invoke Π_{dsBits} to generate $\langle\langle \mathsf{r}^{\mathsf{R}} \rangle\rangle, \langle\langle \mathsf{r} \rangle\rangle^{\mathbf{B}}$ for $\mathsf{r} \in \mathbb{Z}_2$.

2. Invoke preprocessing phase of Π_{mult}.

Online:

1. Compute $\langle\langle \zeta \rangle\rangle^{\mathbf{B}} = \langle\langle \mathsf{b} \rangle\rangle^{\mathbf{B}} \oplus \langle\langle \mathsf{r} \rangle\rangle^{\mathbf{B}}$.

2. $P_i \in \mathcal{E}$ invokes $\Pi^{\mathbf{B}}_{\langle\langle\cdot\rangle\rangle \to \mathcal{E}[\cdot]}$ to generate ${}^{\mathcal{E}}[\zeta]_i^{\mathbf{B}}$ and sends ${}^{\mathcal{E}}[\zeta]_i^{\mathbf{B}}$ to P_{king}, who reconstructs ζ and generates $\langle\langle \zeta^{\mathsf{R}} \rangle\rangle$.

3. Invoke online phase of Π_{mult} to generate $\langle\langle \zeta^{\mathsf{R}}\mathsf{r}^{\mathsf{R}} \rangle\rangle$, and compute $\langle\langle \mathsf{b}^{\mathsf{R}} \rangle\rangle = \langle\langle \zeta^{\mathsf{R}} \rangle\rangle + \langle\langle \mathsf{r}^{\mathsf{R}} \rangle\rangle - 2\langle\langle \zeta^{\mathsf{R}}\mathsf{r}^{\mathsf{R}} \rangle\rangle$.

Fig. 30: Semi-honest: Bit to arithmetic

c) Arithmetic to Boolean sharing: Extending the techniques from [57], protocol Π_{A2B} generates $\langle\langle \mathsf{x} \rangle\rangle^{\mathbf{B}}$ from $\langle\langle \mathsf{x} \rangle\rangle$ for $\mathsf{x} \in \mathbb{Z}_{2^\ell}$. For this, given arithmetic and Boolean shares of $\mathsf{r} \in \mathbb{Z}_{2^\ell}$, Boolean shares of x are computed as $(\mathsf{x} + \mathsf{r}) - \mathsf{r}$ by evaluating a parallel prefix adder (PPA) circuit [73], [68]. The PPA circuit inputs two Boolean values ($\mathsf{x} + \mathsf{r}$, $-\mathsf{r}$ in this case) and outputs their sum. The protocol appears in Fig. 31. Looking ahead, Π_{A2B} is used in the preprocessing phase in the applications considered. Hence, we rely on PPA circuit from [68] as it provides a good trade-off between rounds and communication as opposed to the circuit from [73] which is optimized to provide a fast online phase at the expense of a higher preprocessing cost (yielding higher total cost than [68]).

d) Boolean to arithmetic sharing: This protocol generates $\langle\langle \mathsf{x} \rangle\rangle$ from $\langle\langle \mathsf{x} \rangle\rangle^{\mathbf{B}}$ where $\mathsf{x} \in \mathbb{Z}_{2^\ell}$. Inspired from [57], [56], observe that $\mathsf{x} = \sum_{i=0}^{\ell-1} 2^i(\mathsf{x}[i])^{\mathsf{R}}$. Thus, we invoke Π_{bit2A} on $\mathsf{x}[i]$ for $i \in \{0, \ldots, \ell-1\}$ to generate $\langle\langle \mathsf{x}[i]^{\mathsf{R}} \rangle\rangle$ followed by locally combining it as per the above equation to generate $\langle\langle \mathsf{x} \rangle\rangle$. Optimizations in [57] carry forward to our setting as well.

Protocol $\Pi_{\mathsf{A2B}}(\mathcal{P}, \langle\langle \mathsf{x} \rangle\rangle)$

Preprocessing:

1. Invoke $\Pi_{\mathsf{dsBits}}(\mathcal{P}, 0)$ to generate $\langle\langle (\mathsf{r}[i])^{\mathsf{R}} \rangle\rangle$ and $\langle\langle \mathsf{r}[i] \rangle\rangle^{\mathbf{B}}$ where $\mathsf{r}[i] \in \mathbb{Z}_2$ for $i \in \{0, \ldots, \ell-1\}$, and set $\langle\langle \mathsf{r} \rangle\rangle = \sum_{i=0}^{\ell-1} 2^i \langle\langle (\mathsf{r}[i])^{\mathsf{R}} \rangle\rangle$.

2. Execute the preprocessing phase for the PPA circuit which computes $\langle\langle \mathsf{x} \rangle\rangle^{\mathbf{B}} = \langle\langle \mathsf{x} + \mathsf{r} \rangle\rangle^{\mathbf{B}} - \langle\langle \mathsf{r} \rangle\rangle^{\mathbf{B}}$.

Online:

1. Compute $\langle\langle \mathsf{x} + \mathsf{r} \rangle\rangle = \langle\langle \mathsf{x} \rangle\rangle + \langle\langle \mathsf{r} \rangle\rangle$

2. Parties in $\mathcal{E}$ invoke $\Pi_{\langle\cdot\rangle \to \mathcal{E}[\cdot]}$ on $\langle\langle \mathsf{x} + \mathsf{r} \rangle\rangle$ to generate ${}^{\mathcal{E}}[\mathsf{x} + \mathsf{r}]$ and send their share to P_{king}.

3. P_{king} reconstructs and sends $\mathsf{x} + \mathsf{r}$ to all parties in $\mathcal{E}$.

4. Invoke $\Pi^{\mathbf{B}}_{\cdot \to \langle\langle\cdot\rangle\rangle}$ to generate $\langle\langle \mathsf{x} + \mathsf{r} \rangle\rangle^{\mathbf{B}}$, and execute the online phase of PPA circuit to compute $\langle\langle \mathsf{x} \rangle\rangle^{\mathbf{B}} = \langle\langle \mathsf{x} + \mathsf{r} \rangle\rangle^{\mathbf{B}} - \langle\langle \mathsf{r} \rangle\rangle^{\mathbf{B}}$.

Fig. 31: Semi-honest: Arithmetic to Boolean

e) Comparison: To compare $\mathsf{x}, \mathsf{y} \in \mathbb{Z}_{2^\ell}$ in FPA, we extend the technique of [68], [74], [56], [22], [57], [73], where checking $\mathsf{x} < \mathsf{y}$ is equivalent to checking if the most significant bit (msb) of $\mathsf{v} = \mathsf{x} - \mathsf{y}$ is 1. To extract the msb from $\langle\langle \mathsf{v} \rangle\rangle$, we rely on Π_{bitext} which takes as input $\langle\langle \mathsf{v} \rangle\rangle$ and outputs the $\langle\langle\cdot\rangle\rangle^{\mathbf{B}}$-share of the msb of v, denoted as $\langle\langle \mathsf{msb}(\mathsf{v}) \rangle\rangle^{\mathbf{B}}$. The optimized bit extraction circuit from [73] is used for computing the msb whose inputs are two $\langle\langle\cdot\rangle\rangle^{\mathbf{B}}$-shared values and output is the $\langle\langle\cdot\rangle\rangle^{\mathbf{B}}$-shared msb of the sum of these two inputs. Observe that, given $\langle\langle \mathsf{v} \rangle\rangle$, v can be written as $\mathsf{v} = \mathsf{m}_{\mathsf{v}} - \lambda_{\mathsf{v}}$, and hence $\langle\langle\cdot\rangle\rangle^{\mathbf{B}}$-shares of m_{v} and λ_{v} constitute the two inputs to the circuit. While $\langle\langle \mathsf{m}_{\mathsf{v}} \rangle\rangle^{\mathbf{B}}$ can be generated non-interactively by invoking $\Pi^{\mathbf{B}}_{\cdot \to \langle\langle\cdot\rangle\rangle}$ in the online phase, $\langle\langle \lambda_{\mathsf{v}} \rangle\rangle^{\mathbf{B}}$ is generated by performing an arithmetic to boolean conversion in the preprocessing phase. Evaluation of bit extraction circuit then gives $\langle\langle \mathsf{msb}(\mathsf{v}) \rangle\rangle^{\mathbf{B}}$.

Protocol $\Pi_{\mathsf{Eq}}(\mathcal{P}, \langle\langle \mathsf{x} \rangle\rangle, \langle\langle \mathsf{y} \rangle\rangle)$

Preprocessing:

1. Perform preprocessing phase of Π_{A2B} and the preprocessing of 4-input multiplications.

Online:

1. Compute $\langle\langle \mathsf{v} \rangle\rangle = \langle\langle \mathsf{x} \rangle\rangle - \langle\langle \mathsf{y} \rangle\rangle$ and invoke Π_{A2B} to generate $\langle\langle \mathsf{v} \rangle\rangle^{\mathbf{B}}$.

2. Generate $\langle\langle \bar{\mathsf{v}} \rangle\rangle^{\mathbf{B}}$ by setting $\mathsf{m}_{\bar{\mathsf{v}}} = 1 \oplus \mathsf{m}_{\mathsf{v}}$ and $\lambda_{\bar{\mathsf{v}}} = \lambda_{\mathsf{v}}$.

3. Perform AND of all the bits in $\bar{\mathsf{v}}$ following the tree based approach by invoking the online phase of 4-input multiplication to generate $\langle\langle \mathsf{b} \rangle\rangle^{\mathbf{B}}$.

Fig. 32: Semi-honest: Equality check protocol

f) Equality Check: Given $\langle\langle\cdot\rangle\rangle$-shared $\mathsf{x}, \mathsf{y} \in \mathbb{Z}_{2^\ell}$, this protocol outputs a $\langle\langle\cdot\rangle\rangle^{\mathbf{B}}$-shared bit, which is set to 1 if $\mathsf{x} = \mathsf{y}$, and 0 otherwise. The approach is to obtain the bit decomposition of $\mathsf{v} = \mathsf{x} - \mathsf{y}$ by performing Π_{A2B}, and check if all bits of v are 0. For this, parties non-interactively obtain 1's complement of the bits of v, denoted as $\bar{\mathsf{v}}$, by setting the corresponding $\mathsf{m}_{\bar{\mathsf{v}}} = 1 \oplus \mathsf{m}_{\mathsf{v}}$ and $\lambda_{\bar{\mathsf{v}}} = \lambda_{\mathsf{v}}$. Parties proceed to compute an AND of all the bits in $\bar{\mathsf{v}}$ following the standard-tree based approach where we use the 4-input multiplication to save on rounds and communication. If $\mathsf{v} = 0$, then the AND outputs 1 else it outputs a 0. The protocol appears in Fig. 32.

Building Block	Semi-honest			Malicious		
	Communication		Rounds Online	Communication		Rounds Online
	Preprocessing	Online		Preprocessing	Online	
Sharing	–	$(t+1)\ell$	1	–	$2t\ell$	1
Reconstruction[a]	–	$3t\ell$	2	–	$n(\mathsf{q}-\mathsf{g})\ell$	1
Multiplication	$t\ell$	$2t\ell$	2	$3t\ell$	$3t\ell$	2
3-input multiplication	$6t\ell$	$2t\ell$	2	$12t\ell$	$3t\ell$	2
4-input multiplication	$15t\ell$	$2t\ell$	2	$33t\ell$	$3t\ell$	2
Doubly shared bits	$4t(\ell+2)$	–	–	$6t(\ell+2)$	–	–
Multiplication with truncation	$4t(\ell+2)\ell+t\ell$	$2t\ell$	2	$3t\ell+6t(\ell+2)\ell$	$3t\ell$	2
Dot product	$t\ell$	$2t\ell$	2	$3t\ell$	$3t\ell$	2
Bit to arithmetic	$4t(\ell+2)+t\ell$	$4t\ell$	4	$6t(\ell+2)+3t\ell$	$6t\ell$	4
Bit injection	$4t(\ell+2)+6t\ell$	$2t\ell$	2	$6t(\ell+2)+12t\ell$	$3t\ell$	2
Arithmetic to Boolean	$4t(\ell+2)\ell$ $+t\ell\log_2\ell$	$2t\ell(1+\log_2\ell)$	$2+2\log_2\ell$	$6t(\ell+2)\ell$ $+3t\ell\log_2\ell$	$3t\ell(1+\log_2\ell)$	$2+2\log_2\ell$
Boolean to arithmetic	$4t(\ell+2)\ell$	$2t\ell$	2	$6t(\ell+2)\ell$	$3t\ell$	2
Comparison[b]	$\mathsf{u}_1+4t(\ell+2)\ell+$ $3t\ell\log_2\ell+2t\ell$	$2t\mathsf{u}_2$	$2\log_4\ell$	$6t(\ell+2)\ell+6t\ell\log_2\ell$ $+3t\ell+\mathsf{u}_1$	$3t\mathsf{u}_2$	$2\log_4\ell$

ℓ - size of ring in bits.
[a]Accounts for reconstruction towards all; $\mathsf{q}=\binom{n}{h}$, $\mathsf{g}=\binom{n-1}{h-1}$. [b]$\mathsf{u}_1=3t\mathsf{n}_2+12t\mathsf{n}_3+33t\mathsf{n}_4$, $\mathsf{u}_2=\mathsf{n}_2+\mathsf{n}_3+\mathsf{n}_4$, $\mathsf{n}_2=41, \mathsf{n}_3=27, \mathsf{n}_4=47$ denote the number of AND gates in the bit extraction circuit of ABY2 [73] with $2,3,4$ inputs, respectively.

TABLE IX: Communication and round complexity of protocols: semi-honest and malicious

g) Maxpool / Minpool: Maxpool allows parties to compute $\langle\!\langle\cdot\rangle\!\rangle$-share of the maximum value $\mathsf{x}_{\max}$ among a vector of values $\vec{x}=(\mathsf{x}_1,\ldots,\mathsf{x}_\mathsf{n})$. For this, we proceed along the lines of [57]. Observe that the maximum among two values $\mathsf{x}_i,\mathsf{x}_j$ can be computed by first using the secure comparison protocol to obtain $\langle\!\langle\mathsf{b}\rangle\!\rangle^{\mathbf{B}}$ such that $\mathsf{b}=0$ if $\mathsf{x}_i\geq\mathsf{x}_j$ and 1 otherwise. Following this, parties can compute $\mathsf{b}(\mathsf{x}_j-\mathsf{x}_i)+\mathsf{x}_i$ using the bit injection protocol, to obtain the maximum value as the output. To compute the maximum among a vector of values, parties follow the standard binary tree-based approach where consecutive pairs of values are compared in a level-by-level manner. We refer to the resulting protocol as $\Pi_{\max}$. A protocol $\Pi_{\min}$ for minpool can be worked out similarly.

h) ReLU: The ReLU function, $\mathsf{ReLU}(\mathsf{v})=\max(0,\mathsf{v})$, can be written as $\mathsf{ReLU}(\mathsf{v})=\overline{\mathsf{b}}\cdot\mathsf{v}$, where bit $\mathsf{b}=1$ if $\mathsf{v}<0$ and 0 otherwise. Here $\overline{\mathsf{b}}$ denotes the complement of b. Given $\langle\!\langle\mathsf{v}\rangle\!\rangle$, parties invoke Π_{bitext} on $\langle\!\langle\mathsf{v}\rangle\!\rangle$ to obtain $\langle\!\langle\mathsf{b}\rangle\!\rangle^{\mathbf{B}}$. $\langle\!\langle\cdot\rangle\!\rangle^{\mathbf{B}}$-sharing of $\overline{\mathsf{b}}$ is then computed, non-interactively, by setting $\mathsf{m}_{\overline{\mathsf{b}}}=1\oplus\mathsf{m}_\mathsf{b}$. Given $\langle\!\langle\overline{\mathsf{b}}\rangle\!\rangle^{\mathbf{B}}$ and $\langle\!\langle\mathsf{v}\rangle\!\rangle$, ReLU is computed using Π_{BitInj}.

B. Malicious blocks

Note that the malicious variants for the building blocks such as bit to arithmetic, Boolean to arithmetic, and arithmetic to Boolean conversion, bit extraction, secure comparison, secure equality check, ReLU, maxpool, and convolutions, follow along similar lines to that of the semi-honest protocols with the difference that the underlying protocols used are replaced with their maliciously secure variants. Moreover, for steps that involve opening values via P_{king}, the reconstructed values are sent to all and are accompanied by a verification check similar to the one in the multiplication protocol.

C. Communication cost

Table IX summarises communication cost and online round complexity of semi-honest and maliciously secure protocols.

APPENDIX D
ADDITIONAL BENCHMARKS

A. Deep NN and GNN

a) NN architecture: Among NNs, the first, NN-1, is a 3-layered fully connected network with ReLU activation after each layer, as considered in [68], [74], [56]. The second, NN-2, is LeNet [60] architecture, which contains two convolutional layers and two fully connected layers with ReLU activation after each layer. Additionally, for convolutional layers, this is followed by maxpool operation. Finally, NN-3 is VGG16 [82] architecture that comprises 16 layers in total, which includes fully connected, convolutional, ReLU activation, and maxpool layers. Last 2 NNs were considered in [85].

b) GNN architecture: The goal of spectral-based GNNs [35], [55] is to learn a function of signals $\vec{\mathsf{x}_1},\ldots,\vec{\mathsf{x}_m}$ each of length n, on a graph $\mathsf{G}=(\mathsf{V},\mathsf{E},\mathbf{M})$, where V is the set of n vertices of the graph, E is the set of edges and $\mathbf{M}$ is the the graph description in terms of an $n\times n$ adjacency matrix. The j^{th} component of every signal $\vec{\mathsf{x}_i}$ corresponds to j^{th} node of the graph. Training data is used to compute graph description $\mathbf{M}$, which is common for all signals considered.

The approximation of graph filters using a truncated expansion in terms of Chebyshev polynomials was put forth in [35]. Chebyshev polynomials are recursively defined as follows:

$$T_k(x)=\begin{cases}1 & \text{if } k=0\\ x & \text{if } k=1\\ 2xT_{k-1}(x)-T_{k-2}(x) & \text{otherwise}\end{cases}$$

and the inference phase for a $n\times c$ signal matrix $\mathbf{X}$ with f feature maps, where c represents the dimension of feature vector for each node, with a K-localized filter matrix Θ_k can be performed as $\mathbf{Y}=\sum_{k=0}^{K-1}T_k(\tilde{\mathbf{L}})\mathbf{X}\Theta_k$. Here, $\tilde{\mathbf{L}}=\frac{2}{\lambda_{max}}\cdot\mathbf{L}-\mathbf{I}\cdot\lambda_{max}$, and λ_{max} is the largest eigenvalue of the normalized graph Laplacian $\mathbf{L}$, $\mathbf{Y}$ is an $n\times f$ dimensional matrix and the trainable parameter for the k^{th} layer Θ_k is of dimension $c\times f$.

We use the simplified architecture of [35] given in [81]. The GNN architecture in the latter uses one graph convolution layer without pooling operation instead of the original model with two graph convolution layers, each of which is followed by a pooling operation. Further, K is set to 5 instead of 25. This architecture is shown to achieve an accuracy of more than 99% on MNIST classification in [81].

– Graph convolution layer:
  - *Input:* $T_k(\tilde{\mathbf{L}})$ with dimensions 784×784, Θ_k with dimensions 1×32, for $k \in \{0, \ldots, K-1\}$, and 28×28 image transformed into a vector $\vec{x}$ of dimension 784.
  - *Output:* $\sum_{k=0}^{K-1} T_k(\tilde{\mathbf{L}})\vec{x}\Theta_k$ with dimensions 784×32.
– ReLU activation: Calculates the ReLU for each input.
– Fully connected layer (FC): with 10 nodes.

Benchmarks for the semi-honest and maliciously secure protocol appear in Table X, Table XI.

NN Type	Ref.	n	Online			End-to-end		
			Comm	Time	TP[a]	Comm	Time	Cost[b]
NN-1	DN07	5	0.16	18.55 ±.4	211.69	3.41	21.46	0.06
		7	0.24		202.48	5.11	22.29	0.10
		9	0.33		202.49	6.81	22.31	0.13
	This	5	0.02	4.61 ±.02	832.61 ±.04	3.41	11.09 ±.02	0.02
		7	0.03			5.11		0.03
		9	0.05			6.81		0.04
NN-2	DN07	5	15.58	46.20	83.12	269.23	56.44	0.21
		7	23.39	48.39	79.35	403.85	59.60	0.32
		9	31.18	48.40	79.35	538.47	59.61	0.42
	This	5	1.92	11.08 ±.02	346.60 ±.3	269.50	25.34 ±.03	0.10
		7	2.88			404.25		0.14
		9	3.84			539.00		0.18
NN-3	DN07	5	228.07	152.95	25.11	4288.26	213.77	1.34
		7	342.24	160.10	23.99	6432.39	227.28	1.96
		9	456.33	160.14	23.99	8576.52	227.33	2.56
	This	5	29.70	36.92 ±.02	104.01 ±.04	4292.06	104.09 ±.03	0.91
		7	44.55			6438.09		1.08
		9	59.40			8584.12		1.71
GNN	DN07	5	20.14	7.26	528.66	956.21	17.00	0.20
		7	30.22	7.54	509.38	1434.31	17.54	0.29
		9	40.29	7.56	509.38	1912.41	17.57	0.38
	This	5	5.34	2.16 ±.02	1777.78 ±.06	956.46	8.97 ±.02	0.17
		7	8.00			1434.69		0.26
		9	10.67			1912.92		0.33

Communication in MB and time in seconds.
[a]TP denotes throughput [b]monetary cost in USD

TABLE X: Semi-honest: Benchmarks for deep NN and GNN.

Compared to our semi-honest variant for evaluating NNs, the malicious variant incurs a 2× higher online communication cost for NN-1 and NN-2. However, this difference closes in with deeper NNs, with the communication being 1.5× for NN-3. The drop in the difference can be attributed to the one-time cost of verification required in the malicious variant, which gets amortized over deeper circuits. Due to the same reason, in comparison to the semi-honest case, the malicious variant has an overhead of around 1 second in the online run-time, which in turn reflects in the reduced throughput. Similar to the semi-honest evaluation of NNs, the overall communication is an order of magnitude higher than the online communication due to the cost incurred for truncation during preprocessing. Also, analogous to the trend observed for synthetic circuits, the overhead in overall run-time is approximately 11 seconds owing to the distributed zero-knowledge proof verification required in the preprocessing phase. For GNN, the trend follows closely to that of NN-3,where malicious variant incurs 1.5× higher communication than its semi-honest counterpart.

NN Type	n	Online			End-to-end		
		Comm	Time	TP[a]	Comm	Time	Cost[b]
NN-1	5	0.04	5.44 ±.02	706.40 ±.04	3.59	22.96 ±.02	0.07
	7	0.06			5.39		0.10
	9	0.08			7.20		0.11
NN-2	5	2.88	11.93 ±.03	322.63 ±.2	286.18	37.71 ±.04	0.15
	7	4.32			429.28		0.22
	9	5.77			571.98		0.27
NN-3	5	44.56	37.91 ±.02	101.27 ±.04	4535.95	124.54	1.04
	7	66.84			6804.06	126.69	1.53
	9	89.12			9066.43	129.42	1.94
GNN	5	8.01	3.02 ±.02	1275.75	977.65	22.39 ±.03	0.23
	7	12.01		1267.35	1466.49		0.34
	9	16.02		1267.32	1954.95		0.42

Communication in MB and time in seconds.
[a]TP denotes throughput [b]monetary cost in USD

TABLE XI: Malicious: Benchmarks for deep NN and GNN.

B. Biometric Matching

Given a database of m biometric samples $(\vec{s}_1, \ldots, \vec{s}_m)$ each of size n, and a user holding its sample $\vec{u}$, the goal of biometric matching is to identify the sample from the database that is "closest" to $\vec{u}$. The notion of "closeness" can be formalized by various distance metrics, of which Euclidean Distance (**EuD**) is the most widely used. Following the general trend, we reduce our biometric matching problem to that of finding the sample from the database which has the least **EuD** with the user's sample $\vec{u}$. We follow [70], [73] where **EuD** between two vectors $\vec{x}, \vec{y}$ of length n is given as

$$\mathbf{EuD}_{\vec{x}\vec{y}} = \sum_{i=1}^{i=\mathsf{n}} (\mathsf{x}_i - \mathsf{y}_i)^2 = \vec{z} \odot \vec{z} \tag{7}$$

where $\vec{z} = ((\mathsf{x}_1 - \mathsf{y}_1), \ldots, (\mathsf{x}_\mathsf{n} - \mathsf{y}_\mathsf{n}))$.

#seq	Ref.	n	Online			End-to-end		
			Comm	Time	TP[a]	Comm	Time	Cost[b]
4096	DN07	5	2.51	66.51	57.73	27.70	79.85	0.24
		7	3.76	69.81	55.00	41.55	83.24	0.36
		9	5.02	69.87	54.97	55.40	83.30	0.48
	This (semi)	5	0.35	15.07 ±.02	254.86 ±.03	27.74	17.35 ±.03	0.04
		7	0.53			41.61		0.06
		9	0.70			55.49		0.08
	This (mal)	5	0.53	15.89 ±.02	241.66 ±.1	30.43	29.27 ±.02	0.09
		7	0.80			45.65		0.13
		9	1.07			60.81		0.16
16384	DN07	5	10.03	77.51	49.54	110.83	93.47	0.29
		7	15.06	81.35	47.20	166.24	97.70	0.45
		9	20.07	81.36	47.14	221.66	97.71	0.59
	This (semi)	5	1.41	17.53 ±.02	219.16 ±.04	110.99	20.24 ±.05	0.06
		7	2.11			166.49		0.09
		9	2.81			221.99		0.11
	This (mal)	5	2.11	18.35 ±.02	209.24 ±.06	121.71	32.30 ±.03	0.11
		7	3.17			182.58		0.16
		9	4.23			243.19		0.20

Communication in MB and time in seconds.
[a]TP denotes throughput [b]monetary cost in USD

TABLE XII: Benchmarks for biometric matching for varying number of sequences in the database.

To achieve this goal of performing biometric matching securely, each $\vec{s}_i$, for all $i \in \{1, \ldots, m\}$ in the database is $\langle\!\langle \cdot \rangle\!\rangle$-shared among the n parties participating in the computation. Specifically, each component $\vec{s}_{i_j}$, for all $j \in \{1, \ldots, \mathsf{n}\}$ is

$\langle\!\langle \cdot \rangle\!\rangle$-shared among all the parties. Similarly, the user also $\langle\!\langle \cdot \rangle\!\rangle$-shares its sample $\vec{u}$. The parties compute a $\langle\!\langle \cdot \rangle\!\rangle$-shared distance vector $\mathbf{DV}$ of size m, where the i^{th} component corresponds to the $\mathbf{EuD}$ between $\vec{u}$ and $\vec{s}_i$. For this, each party locally obtains $\langle\!\langle \vec{z}_i \rangle\!\rangle = \langle\!\langle \vec{s}_i \rangle\!\rangle - \langle\!\langle \vec{u} \rangle\!\rangle$ and computes $\langle\!\langle \mathbf{DV}_i \rangle\!\rangle$ according to Eq. 7 using the dot product operation. The final step is then to identify the minimum of these m components of $\mathbf{DV}$, which can be performed using the protocol $\Pi_{\min}$ for minpool operation. Table XII tabulates the benchmarks when the database has 4096 and 16384 samples.

The trend observed for 4096 and 16384 samples adheres to that observed from Table VIII for the case of 1024 and 65536 samples. Specifically, these settings also enjoy a 4.6× improvement in online run-time and throughput and around 83% saving in the monetary cost compared to DN07. Moreover, similar to the prior case, the malicious variant incurs a minimal overhead of 4% in the online throughput and 9.5% in the total communication compared to our semi-honest setting.

C. Genome Sequence Matching

Given a genome sequence as a query, genome matching aims to identify the most similar sequence from a database of sequences. This task is also commonly referred to as Similar Sequence Query (SSQ) identification and has implications in the advancing field of medical science. An SSQ algorithm on two sequences s and q, requires the computation of Edit Distance (ED), which quantifies how different two sequences are by identifying the minimum number of additions, deletions, and substitutions needed to transform one sequence to the other. To compute the ED, we extend the (2-party) protocol from [79] which builds on top of the approximation from [5], to the n-party setting. We describe high-level idea of the approximation algorithm for ED computation for a query sequence q against a database of sequences $\{\mathsf{s}_1, \ldots, \mathsf{s}_m\}$.

Protocol $\Pi_{\mathsf{ED}}(\mathcal{P}, \langle\!\langle \mathsf{LUT}_\mathsf{s} \rangle\!\rangle, \langle\!\langle \mathsf{q} \rangle\!\rangle)$

1. For $i = 1$ to ω
- For $j = 1$ to m
  - Invoke Π_{Eq} on $\langle\!\langle \mathsf{LUT}_\mathsf{s}[i][j] \rangle\!\rangle$ and $\langle\!\langle \mathsf{q}[i] \rangle\!\rangle$ to generate $\langle\!\langle \mathsf{b}_j \rangle\!\rangle^{\mathbf{B}}$.
  - Invoke Π_{bit2A} on $\langle\!\langle \mathsf{b}_j \rangle\!\rangle^{\mathbf{B}}$ and generate $\langle\!\langle \mathsf{b}_j \rangle\!\rangle$.
- Let $\vec{\mathsf{b}} = \{\mathsf{b}_1, \ldots, \mathsf{b}_m\}$. Compute $\langle\!\langle \mathsf{d}_i \rangle\!\rangle = \Pi_{\mathsf{dp}}(\langle\!\langle \vec{\mathsf{b}} \rangle\!\rangle, \langle\!\langle \mathsf{LUT}_\mathsf{s}[i][\cdot] \rangle\!\rangle)$.

2. Compute $\langle\!\langle \mathsf{d} \rangle\!\rangle = \sum_{i=1}^{\omega} \langle\!\langle \mathsf{d}_i \rangle\!\rangle$.

Fig. 33: Edit distance between query q and sequence s with respect to a database of m sequences and ω blocks

The ED approximation algorithm has a non-interactive phase, during which the database owner with the sequences $\mathsf{s}_1, \ldots, \mathsf{s}_m$, generates a Look-Up-Table (LUT) for each sequence. These LUTs are then secret-shared among all the parties. To generate the LUT, the sequences in the database are aligned with respect to a common reference genome sequence (using the Wagner-Fischer algorithm [86]), and divided into blocks of a fixed, predetermined size. Based on the most frequently occurring block sequences in the database, an LUT is constructed consisting of these block values and their distance from each other. Specifically, for a database of m sequences $\{\mathsf{s}_1, \ldots, \mathsf{s}_m\}$, each of length ω blocks, an LUT_i is constructed for each s_i. Each LUT has m columns, one corresponding to each s_i in the database, and ω rows, one corresponding to each block of a sequence, where $\mathsf{LUT}_\mathsf{s}[i][j]$ corresponds to the ED between block i of the sequence s and s_j . This completes the non-interactive phase of the ED approximation algorithm.

Given the LUTs, when a new query q has to be processed, its ED must be computed from every sequence s in the database. For this, similar to the non-interactive phase, the query is first aligned with the reference sequence and broken down into blocks of the same fixed size. Then, the i^{th} block from the query is matched with the i^{th} block of each sequence in the LUT for a sequence s. If the block values match, then the precomputed distance is taken as the output for that block; otherwise, the output is taken to be 0. Finally, the resultant sum of distances for all the blocks is taken to be the approximated ED between q and the sequence s. Computing the ED to all such sequences s in the database then allows the identification of the most similar sequence for the query using the minpool operation. Algorithms for ED computation between two sequences, and SSQ appear in Fig. 33, Fig. 34, respectively, where accuracy and correctness follow from [5].

Protocol $\Pi_{\mathsf{SSQ}}(\mathcal{P}, \{\langle\!\langle \mathsf{LUT}_\mathsf{s} \rangle\!\rangle\}_{s=1}^{m}, \langle\!\langle \mathsf{q} \rangle\!\rangle)$

1. For $\mathsf{s} = 1$ to m
- Invoke Π_{ED} on $\langle\!\langle \mathsf{LUT}_\mathsf{s} \rangle\!\rangle$ and $\langle\!\langle \mathsf{q} \rangle\!\rangle$ to generate $\langle\!\langle \mathsf{d}_\mathsf{s} \rangle\!\rangle$.

2. Invoke $\Pi_{\min}$ on $\langle\!\langle \mathsf{d}_1 \rangle\!\rangle, \ldots, \langle\!\langle \mathsf{d}_m \rangle\!\rangle$ to generate $\langle\!\langle \mathsf{s}_{\min} \rangle\!\rangle \in \{\mathsf{d}_1, \ldots, \mathsf{d}_m\}$.

Fig. 34: Similar sequence queries

Since the generation of LUTs happens non-interactively, we only focus on the computation of ED with respect to the new query q, which requires interaction, and benchmark the same. Table XIII provides the benchmarks when the database consists of $m = 1000, 4000$ for block length $\omega = 25, 35$ respectively. As expected, the observations tabulated for the varying sequence lengths follows closely to the ones for the case of $m = 2000$ and $\omega = 30$ given in Table VII.

m, ω	Ref.	n	Online			End-to-end		
			Comm[a]	Time	TP[b]	Comm[c]	Time	Cost[d]
$m = 1000$ $\omega = 25$	DN07	5	10.85	60.58	63.39	0.17	74.13	0.25
		7	16.28	63.60	60.38	0.25	77.76	0.37
		9	21.71	63.62	60.37	0.33	77.79	0.50
	This (semi)	5	6.42	16.12 ±.01	236.21 ±.15	0.17	19.08 ±.02	0.07
		7	9.63			0.25		0.10
		9	12.84			0.33		0.13
	This (mal)	5	9.51	16.8 ±.1	228.71 228.44 226.82	0.18	31.21 ±.08	0.12
		7	14.14			0.27		0.16
		9	18.40			0.36		0.21
$m = 4000$ $\omega = 35$	DN07	5	59.87	72.08	53.27	0.92	92.04	0.43
		7	89.86	75.65	50.76	1.39	98.90	0.64
		9	119.81	75.67	50.72	1.85	98.93	0.84
	This (semi)	5	35.87	19.34 ±.02	198.55 ±.35	0.92	25.97 ±.03	0.21
		7	53.80			1.39		0.31
		9	71.74			1.85		0.39
	This (mal)	5	53.11	20.11 ±.06	190.95 ±.4	0.99	41.83 ±.06	0.29
		7	78.96			1.48		0.42
		9	102.68			1.97		0.52

Time in seconds.
[a]communication in MB [b]TP denotes throughput [c]communication in GB [d]monetary cost in USD

TABLE XIII: Benchmarks for genome sequence matching for varying number of sequences (m) and block length (ω).

APPENDIX E
SECURITY PROOFS

Security proofs are given in the real-world/ideal-world simulation-based paradigm [63]. Let $\mathcal{A}^{\mathsf{sh}}, \mathcal{A}^{\mathsf{mal}}$ denote the real-world semi-honest, malicious adversary, respectively, corrupting at most t parties in $\mathcal{P}$, denoted by $\mathcal{C}$. Let $\mathcal{S}^{\mathsf{sh}}, \mathcal{S}^{\mathsf{mal}}$ denote the corresponding ideal world semi-honest, malicious adversary, respectively. Security proofs are given in the $\mathcal{F}_{\mathsf{setup}}, \mathcal{F}_{\mathsf{TrGen}}$-hybrid (and $\mathcal{F}^{\mathsf{M}}_{\mathsf{TrGen}}, \mathcal{F}_{\mathsf{MulPre}}, \mathcal{F}_{\mathsf{DotPPre}}$-hybrid for malicious setting) model. For modularity, we provide simulation steps for each protocol separately.

The following is the strategy for simulating the computation of function f (represented by a circuit ckt). The simulator $\mathcal{S}^{\mathsf{sh}}$ knows the input and output of the adversary $\mathcal{A}^{\mathsf{sh}}$, and sets the inputs of the honest parties to be 0. $\mathcal{S}^{\mathsf{sh}}$ emulates $\mathcal{F}_{\mathsf{setup}}$ and gives the respective keys to the $\mathcal{A}^{\mathsf{sh}}$. Knowing all the inputs and randomness, $\mathcal{S}^{\mathsf{sh}}$ can compute all the intermediate values for each building block in the clear. Thus, $\mathcal{S}^{\mathsf{sh}}$ proceeds to simulate each building block in topological order using the aforementioned values (input and output of $\mathcal{A}^{\mathsf{sh}}$, randomness and intermediate values). We provide the simulation steps for each of the sub-protocols separately for modularity. When carried out in the respective order, these steps result in the simulation steps for the entire computation. To distinguish the simulators for various protocols, we use the corresponding protocol name as the subscript of $\mathcal{S}^{\mathsf{sh}}$.

a) Sharing and Reconstruction: Simulation for input sharing (Fig. 21) and reconstruction appears in Fig. 35, Fig. 36, respectively.

Simulator $\mathcal{S}^{\mathsf{sh}}_{\mathsf{Sh}}$

Preprocessing:

– Emulate $\mathcal{F}_{\mathsf{setup}}$ and give the respective shared keys to $\mathcal{A}^{\mathsf{sh}}$.

– Samples shares of λ_{a} commonly held with $\mathcal{A}^{\mathsf{sh}}$ using the respective PRF keys while other values are sampled randomly.

Online:

– If $P_s \in \mathcal{C}$, receive $\mathsf{m_a}$ from $\mathcal{A}^{\mathsf{sh}}$ on behalf of honest parties in $\mathcal{E}$. Else, set $\mathsf{a} = 0$, $\mathsf{m_a} = \lambda_{\mathsf{a}}$ and sends $\mathsf{m_a}$ to $\mathcal{A}^{\mathsf{sh}}$ on behalf of P_s if there exists a corrupt party in $\mathcal{E}$.

Fig. 35: Semi-honest: Simulation for $\Pi_{\mathsf{Sh}}(P_s, \mathsf{a})$

Simulator $\mathcal{S}^{\mathsf{sh}}_{\mathsf{Rec}}$

– If $P_{\mathsf{king}} \in \mathcal{C}$, use the output a, and $\mathsf{m_a}$ and ${}^{\mathcal{E}}[\lambda_{\mathsf{a}}]_j$ held by corrupt $P_j \in \mathcal{C} \cap \mathcal{E}$ to compute the shares ${}^{\mathcal{E}}[\lambda_{\mathsf{a}}]_i$ of each honest $P_i \in \mathcal{E}$ such that $\mathsf{m_a} - \mathsf{a} = \sum_{P_i \in \mathcal{E} \setminus \mathcal{C}} {}^{\mathcal{E}}[\lambda_{\mathsf{a}}]_i + \sum_{P_j \in \mathcal{C} \cap \mathcal{E}} {}^{\mathcal{E}}[\lambda_{\mathsf{a}}]_j$. Send the shares of the honest parties in $\mathcal{E}$ to $\mathcal{A}^{\mathsf{sh}}$.

– If P_{king} is honest, send output a to $\mathcal{A}^{\mathsf{sh}}$ on behalf of P_{king}.

Fig. 36: Semi-honest: Simulation for reconstruction

b) Multiplication: Simulation steps for multiplication (Fig. 4) are provided in Fig. 37. Observe that the adversary's view in the simulation is indistinguishable from its view in the real world since it only receives random value in each step of the protocol.

Simulator $\mathcal{S}^{\mathsf{sh}}_{\mathsf{mult}}$

Preprocessing:

– If $\mathsf{isTr} = 0$: Sample $\langle\cdot\rangle$-shares of r commonly held with $\mathcal{A}^{\mathsf{sh}}$ using the respective shared keys while other values are sampled randomly.

– Else if $\mathsf{isTr} = 1$: Emulate $\mathcal{F}_{\mathsf{TrGen}}$ to generate $\langle\langle \mathsf{r} \rangle\rangle, \langle\langle \mathsf{r}^d \rangle\rangle$.

– On behalf of every honest $P_i \in \mathcal{D}$, send a random value for $[\Lambda_{\mathsf{ab}} - \mathsf{r}]_i$ to $\mathcal{A}^{\mathsf{sh}}$ if $P_{\mathsf{king}} \in \mathcal{C}$.

Online:

– If $P_{\mathsf{king}} \in \mathcal{C}$, send random value for ${}^{\mathcal{E}}[\zeta]_i$ to $\mathcal{A}^{\mathsf{sh}}$ on behalf of the honest $P_i \in \mathcal{E}$.

– If $P_{\mathsf{king}} \notin \mathcal{C}$, send a random $\mathsf{z} - \mathsf{r}$ to $\mathcal{A}^{\mathsf{sh}}$, if there exists a corrupt party in $\mathcal{E}$.

Fig. 37: Semi-honest: Simulation for Π_{mult}

c) Other building blocks: Simulation steps for the remaining building blocks can be obtained analogously by simulating the steps for the respective underlying protocols in their order of invocations.

A. Malicious security

The following is the strategy for simulating the computation of function f (represented by a circuit ckt). The simulator emulates $\mathcal{F}_{\mathsf{setup}}$ and gives the respective keys to the malicious adversary, $\mathcal{A}^{\mathsf{mal}}$. This is followed by the input sharing phase in which $\mathcal{S}^{\mathsf{mal}}$ extracts the input of $\mathcal{A}^{\mathsf{mal}}$, using the known keys, and sets the inputs of the honest parties to be 0. Knowing all the inputs, $\mathcal{S}^{\mathsf{mal}}$ can compute all the intermediate values for each building block in the clear. Further, $\mathcal{S}^{\mathsf{mal}}$ invokes $\mathcal{F}^{\mathsf{mal}}_{\mathsf{n-PC}}$ and obtains the function output on clear. $\mathcal{S}^{\mathsf{mal}}$ proceeds to simulate each building block in topological order using the aforementioned values (inputs of $\mathcal{A}^{\mathsf{mal}}$, intermediate values, and function output). As before, we provide the simulation steps for each of the sub-protocols separately for modularity. When carried out in the respective order, these steps result in the simulation steps for the entire computation. To distinguish the simulators for various protocols, the corresponding protocol name appears as the subscript of $\mathcal{S}^{\mathsf{mal}}$.

a) Sharing: Simulation for sharing appears in Fig. 38.

Simulator $\mathcal{S}^{\mathsf{mal}}_{\mathsf{Sh}}$

Preprocessing:

– Emulate $\mathcal{F}_{\mathsf{setup}}$ and give the respective shared keys to $\mathcal{A}^{\mathsf{mal}}$.

– Samples shares of λ_{a} commonly held with $\mathcal{A}^{\mathsf{mal}}$ using the respective PRF keys while other values are sampled randomly.

Online:

– For $P_s \in \mathcal{C}$, receive $\mathsf{m_a}$ from $\mathcal{A}^{\mathsf{mal}}$ on behalf of honest parties in $\mathcal{E}$, and obtain $\mathsf{a} = \mathsf{m_a} - \lambda_{\mathsf{a}}$ (since $\mathcal{S}^{\mathsf{mal}}$ knows all the PRF keys, it knows λ_{a}). Invoke $\mathcal{F}^{\mathsf{mal}}_{\mathsf{n-PC}}$ with $(\mathsf{Input}, \mathsf{a})$ on behalf of $\mathcal{A}^{\mathsf{mal}}$.

– On behalf of the honest parties, set its input $\mathsf{a} = 0$, $\mathsf{m_a} = \lambda_{\mathsf{a}}$ and send $\mathsf{m_a}$ to $\mathcal{A}^{\mathsf{mal}}$ if there exists a corrupt party in $\mathcal{E}$.

Verification: Send $\mathsf{H}(\mathsf{m_a})$ to $\mathcal{A}^{\mathsf{mal}}$ on behalf of the honest parties. If inconsistent $\mathsf{m_a}$s were received with respect to a corrupt party, invoke $\mathcal{F}^{\mathsf{mal}}_{\mathsf{n-PC}}$ with $(\mathsf{Signal}, \mathsf{abort})$.

Fig. 38: Malicious: Simulation for $\Pi^{\mathsf{M}}_{\mathsf{Sh}}(P_s, \mathsf{a})$

b) Reconstruction: Simulation for reconstruction (with `abort`) appears in Fig. 39.

Simulator $\mathcal{S}^{\mathsf{mal}}_{\mathsf{Rec}}$

– Use output a obtained from $\mathcal{F}^{\mathsf{mal}}_{\mathsf{n-PC}}$, $\mathsf{m_a}$ and $\langle \lambda_\mathsf{a} \rangle_j$ held by corrupt $P_j \in \mathcal{C}$ to compute the shares $\langle \lambda_\mathsf{a} \rangle_i$ of each honest $P_i \in \mathcal{E}$ such that $\mathsf{m_a} - \mathsf{a} = \lambda_\mathsf{a}$. Send the shares of the honest parties in $\mathcal{E}$ to $\mathcal{A}^{\mathsf{mal}}$, and receive shares from $\mathcal{A}^{\mathsf{mal}}$ on behalf of honest parties.

– If any honest party P_i is unable to reconstruct the output, add P_i to set P. Send $(\mathsf{Signal}, \mathtt{abort}, P)$ to $\mathcal{F}^{\mathsf{mal}}_{\mathsf{n-PC}}$.

Fig. 39: Malicious: Simulation for reconstruction

c) Multiplication: Simulation steps for multiplication (Fig. 27) are provided in Fig. 40.

Simulator $\mathcal{S}^{\mathsf{mal}}_{\mathsf{mult}}$

Preprocessing:

– If $\mathsf{isTr} = 0$: Sample $\langle \cdot \rangle$-shares of r commonly held with $\mathcal{A}^{\mathsf{mal}}$ using the respective shared keys while other values are sampled randomly.

– Else if $\mathsf{isTr} = 1$: Emulate $\mathcal{F}^{\mathsf{M}}_{\mathsf{TrGen}}$ to generate $\langle\langle \mathsf{r} \rangle\rangle, \langle\langle \mathsf{r}^d \rangle\rangle$.

– Emulate $\mathcal{F}_{\mathsf{MulPre}}$ to generate $\langle \cdot \rangle$-shares of Λ_{ab}.

Online:

– If $P_{\mathsf{king}} \in \mathcal{C}$, send random value for ${}^{\mathcal{E}}[\zeta]_i$ to $\mathcal{A}^{\mathsf{mal}}$ on behalf of the honest $P_i \in \mathcal{E}$.

– If $P_{\mathsf{king}} \notin \mathcal{C}$, send a random $\mathsf{z} - \mathsf{r}$ to $\mathcal{A}^{\mathsf{mal}}$.

Verification:

– Send $\mathsf{H}(\mathsf{z}_1 - \mathsf{r}_1 || \ldots || \mathsf{z}_m - \mathsf{r}_m)$ with respect to m multiplications, to $\mathcal{A}^{\mathsf{mal}}$ on behalf of the honest parties. If the hash values received from $\mathcal{A}^{\mathsf{mal}}$ are inconsistent, invoke $\mathcal{F}^{\mathsf{mal}}_{\mathsf{n-PC}}$ with $(\mathsf{Signal}, \mathtt{abort})$.

– If $\mathcal{A}^{\mathsf{mal}}$ has sent incorrect $\mathsf{z}-\mathsf{r}$ for any multiplication ($\mathcal{S}^{\mathsf{mal}}$ can detect this since it knows all inputs and randomness that should be used by $\mathcal{A}^{\mathsf{mal}}$), generate random shares for Ω and simulate reconstruction steps of $\mathcal{S}^{\mathsf{mal}}_{\mathsf{Rec}}$. Invoke $\mathcal{F}^{\mathsf{mal}}_{\mathsf{n-PC}}$ with $(\mathsf{Signal}, \mathtt{abort})$.

– Else, if $\mathcal{A}^{\mathsf{mal}}$ has behaved honestly throughout, simulate reconstruction of $\Omega = 0$ using steps from $\mathcal{S}^{\mathsf{mal}}_{\mathsf{Rec}}$. Invoke $\mathcal{F}^{\mathsf{mal}}_{\mathsf{n-PC}}$ with $(\mathsf{Signal}, \mathtt{abort})$.

Fig. 40: Malicious: Simulation for $\Pi^{\mathsf{M}}_{\mathsf{mult}}$

Observe that since $\mathcal{A}^{\mathsf{mal}}$ sees random shares in both the real-world protocol and in the simulation, indistinguishability of the simulation follows.

d) Other building blocks: Simulations for the remaining building blocks can be obtained analogously and using the steps for the underlying protocols.