Asterisk: Super-fast MPC with a Friend

Banashri Karmakar[1], Nishat Koti[1], Arpita Patra[1], Sikhar Patranabis[2], Protik Paul[1], and Divya Ravi[3]

[1]IISc Bangalore
[1]*{banashrik,kotis,arpita,protikpaul}@iisc.ac.in*
[2]IBM IRL
[2]*sikhar.patranabis@ibm.com*
[3]Aarhus University
[3]*divya@cs.au.dk*

Abstract

Secure multiparty computation (MPC) enables privacy-preserving collaborative computation over sensitive data held by multiple mutually distrusting parties. Unfortunately, in the most natural setting where a majority of the parties are maliciously corrupt (also called the *dishonest majority* setting), traditional MPC protocols incur high overheads and offer weaker security guarantees than are desirable for practical applications. In this paper, we explore the possibility of circumventing these drawbacks and achieving practically efficient dishonest majority MPC protocols with strong security guarantees by assuming an additional semi-honest, non-colluding helper party HP[1]. We believe that this is a more realistic alternative to assuming an honest majority, since many real-world applications of MPC involving potentially large numbers of parties (such as secure auctions and dark pools) are typically enabled by a central entity that can be modeled as the HP.

In the above model, we are the first to design, implement and benchmark a practically-efficient and general multi-party framework, Asterisk[2], which achieves the strong security guarantee of *fairness* (either all parties learn the output or none do), scales to hundreds of parties, outperforms all existing dishonest majority MPC protocols, and is, in fact, competitive with state-of-the-art honest majority MPC protocols. Our experiments show that Asterisk achieves $900-1200\times$ speedup in preprocessing as compared to the best dishonest majority MPC protocols, and supports 100-party evaluation of a circuit with 10^6 multiplication gates in under 2 minutes. We also implement and benchmark practically efficient and highly scalable instances of two applications, namely privacy-preserving secure auctions and dark pools, using Asterisk as the building block. This showcases the effectiveness of Asterisk in enabling real-world privacy-preserving applications with strong efficiency and security guarantees.

[1]We refer to this entity as a friend in the title.

[2]Given a set of common keys among the parties and HP, parties interact only with the HP, and no communication is required among any other parties. Thus, we name our protocol as Asterisk

1 Introduction

In today's digital landscape, vast amounts of user data are being collected and analyzed. This motivates the need for privacy-preserving computation that enable computing on sensitive data without compromising data privacy. Secure multiparty computation (MPC) [50] is one such technology that facilitates computation over sensitive data without disclosing any underlying information. Informally, an MPC protocol allows a set of n distrusting parties with private inputs to jointly evaluate a function on these inputs, while ensuring that a (centralized) adversary corrupting a subset of at most $t < n$ parties learns nothing beyond the function output.

We motivate the practical applicability of MPC with the example use-case of privacy-preserving sealed bid auctions. At a high level, a sealed bid auction involves buying or selling items through a public bidding system, where bidders submit their bids in a 'sealed envelope' to a centralized auctioneer. Eventually, the auctioneer determines the highest bid amount, and the corresponding bidder is said to win the auction. Observe that this simple solution preserves the privacy of a given bidders' inputs against all other bidders, but not against the auctioneer. Concretely, this solution necessitates a trusted centralized entity acting as the auctioneer that: (a) respects the privacy of the bidder's inputs, and (b) executes the computation of the highest bid honestly. MPC, on the other hand, obviates the need for such trust assumptions. The bidders could simply run an MPC protocol amongst themselves (with their bids as their private inputs); the protocol would allow computing the highest bid while guaranteeing that no corrupt subset of $t < n$ bidders can reconstruct any of the honest bidder's inputs (and, in fact, does not learning anything beyond the highest bid and the final winner of the auction).

Honest vs dishonest majority. While this appears to be an ideal solution, one needs to consider more fine-grained aspects, namely the *corruption threshold* that the MPC protocol can withstand, and the *notion of security* that the protocol achieves. In practical applications such as secure auctions, one would expect a majority of the bidders to be corrupt. In the context of MPC, this translates to the adversary corrupting up to $t = (n-1)$ out of n parties, and is termed as the *dishonest majority setting*. The counterpart to this setting is the *honest majority setting*, where the adversary corrupts only a minority of the parties, i.e., $t < n/2$. Our focus in this paper is on MPC in the dishonest majority setting.

Traditional dishonest majority MPC. Unfortunately, traditional MPC protocols in dishonest majority *necessarily* rely on cryptographic hardness assumptions, which renders them practically inefficient in comparison to their honest majority counterparts, and potentially degrades the performance of *real-time, throughput-sensitive* applications (e.g., auctions, dark pools, privacy-preserving machine learning, etc.). Additionally, known impossibility results make it impossible for traditional dishonest majority MPC protocols to provide stronger security notions of fairness or full security/guaranteed output delivery (GOD) in the presence of a malicious[3] adversary [18]. Fairness guarantees that corrupt parties do not gain any unfair advantage in learning the output (i.e., either all parties learn the output of the computation or none do), while GOD provides the stronger guarantee that, irrespective of the corrupt parties' misbehaviour, all parties obtain the output. In the absence of fairness (or GOD), a dishonest majority protocol empowers an adversary to abort the computation at will as well as learn the output. This can be disastrous in applications such as secure auctions, since this allows the adversary to learn the bid of honest bidders, and use this information to its advantage (say, quote a bid which is higher than honest bidder's amount) in the subsequent steps. These drawbacks tend to render traditional dishonest majority MPC protocols insufficient for many practical applications.

Honest majority MPC. While MPC protocols in honest majority setting [8, 9, 28, 40] avoid the above drawbacks and yield practically efficient solutions with strong security guarantees, the very assumption of honest majority may be too strong for certain applications. For example, in case of secure auctions with large number of bidders (say, hundreds), existence of an honest majority is practically infeasible, because this would require assuming that at least half the bidders are honest.

MPC with a friend. The above discussion motivates the need for *alternative MPC models* that could bypass the inherent drawbacks of traditional dishonest majority protocols, while also avoiding the strong corruption threshold assumptions that characterize honest-majority solutions. In this paper, we consider such a model that (we believe) realistically captures corruption behaviors in real-world applications. We dub this model as *MPC with a friend*, where the parties are aided by the presence of an additional helper party HP (which is not trusted but is semi-honest, i.e., does not deviate arbitrarily from the protocol). Observe that, as compared to assuming an honest majority within a (potentially large) set of parties, it seems more practical to identify a single (semi-honest) party, while allowing a majority of the remaining parties to be (maliciously) corrupt. For example, in case of secure auctions, the auctioneer can enact the role of HP. While we would ideally avoid trusting the auctioneer, it is reasonable to assume that the auctioneer is semi-honest since it does not necessarily have an incentive to cheat[4]. A similar argument can be made in case of other applications such as dark pools, where Securities and Exchange Commission can enact the HP.

We note that several works have considered using a "trusted" HP to achieve MPC protocols with stronger security guarantees [25, 26, 31]. However, these works typically assumed that the HP was a small and fully trusted entity (e.g., tamper-proof hardware tokens [6, 11, 15–17, 20, 29, 33]), sometimes doing minimal work. On the other hand, modeling the HP as a semi-honest entity was introduced in a more recent work [32]. As shown in [32], this model circumvents the impossibility of [18], even when the HP is semi-honest, as long as it does not collude with the remaining parties (with a maliciously corrupt majority). It is additionally shown in [32] that this non-collusion assumption is somewhat necessary for strong security notions such as fairness/GOD that we want for practical applications (we expand more on this later). Finally, the authors of [32] propose certain concrete MPC protocols in this setting that achieve GOD; however, these are primarily feasibility results based on strong cryptographic machinery and are practically inefficient.

A related MPC model called Assisted MPC (based on a probabilistically semi-honest, non-colluding helper party) was studied in [44]; however, this model differs from the HP-aided MPC model of [32] in two ways: (a) the authors of [44] assume that the semi-honest behavior of the helper party is *probabilistic* (i.e., it could also be malicious with some finite non-zero probability), and (b) it is not studied in [44] if this model circumvents known impossibilities and enables strong security guarantees such as fairness/GOD (the protocols in [44], while practically efficient, still achieve the weaker notion of abort security – same as traditional dishonest majority MPC protocols). This leads to the question: can we bridge this gap between inefficient protocols with strong security guarantees and efficient protocols with weak security guarantees using a non-colluding, semi-honest HP?

Relation to FaF security. We briefly comment on the non-colluding HP assumption above, and its relation to another notion of security for MPC called friends-and-foes (FaF) security [1], which requires security to hold not only against the adversarial parties (foes), but also against quorums of honest parties (friends). [1] proposed modelling this using a *decentralized* adversary consisting of two different *non-colluding* adversaries—(i) a malicious adversary that corrupts any subset of at most t out of n parties, and (ii) a semi-honest adversary that corrupts any subset of at most $h^\star$ out of the

[3]Malicious adversary may deviate arbitrarily from protocol specification.

[4]Here we assume that the auctioneer is not colluding with any of the bidders; as we discuss later, this is a *necessary* assumption since we run into impossibilities otherwise.

remaining $(n-t)$ parties. Further, the FaF model requires security to hold even when the malicious adversary sends its view to the semi-honest adversary. A natural adaptation of this notion to the HP-aided setting would translate to the malicious adversary corrupting up to $(n-1)$ parties and the semi-honest adversary corrupting the HP (this is clearly weaker than traditional FaF security because the HP is a designated semi-honest party, but stronger than the fully non-colluding HP assumption of [32], since the HP can be given access to the views of the corrupt parties). The impossibility result of [32] does not rule out the possibility of getting stronger security guarantees in this model (their impossibility assumes that the *same centralized* adversary corrupts both the HP and a majority of the remaining parties). Unfortunately, we rule out the possibility of achieving fair MPC protocols in this model (see Section 3), thus establishing that the non-colluding model of HP-aided MPC from [32] is the best we can hope for if we wish to achieve fairness/GOD in dishonest majority. In the rest of the paper, we refer to this setting as the *dishonest majority* HP-*aided setting*. Concretely, we ask the following: *Can we achieve highly efficient MPC protocols with strong security in the dishonest majority* HP-*aided setting?*

1.1 Our contributions

We answer the above question in the affirmative. Our contributions are as described below.

Asterisk. We put forth Asterisk– a highly efficient MPC protocol in the preprocessing paradigm that allows computing arithmetic as well as Boolean circuits over secret-shared inputs in the dishonest majority HP-aided setting (i.e., in the presence of a semi-honest, non-colluding HP) while achieving fairness. Unlike standard dishonest-majority MPC protocols [7, 22, 35] that only achieve abort security due to known impossibilities for achieving fairness [18], Asterisk achieves stronger security (fairness) by leveraging the HP to bypass this impossibility. In particular, Asterisk uses the (semi-honest, non-colluding) HP in both the preprocessing and online phases, and, unlike existing dishonest majority protocols, achieves an extremely fast and lightweight (function-dependent) preprocessing phase, as well as a highly efficient online phase, while maintaining privacy against the (semi-honest) HP. We expand more on this below.

Efficient preprocessing. We note that traditional dishonest-majority MPC protocols [7, 22, 35, 36] crucially rely on cryptographic machinery (such as homomorphic encryption or oblivious transfer) in the preprocessing phase, which leads to greater computational and communication overheads as compared to honest-majority protocols. Asterisk, on the other hand, achieves a highly efficient (function-dependent) preprocessing phase that uses no additional cryptographic machinery, and only requires communicating 3 elements per multiplication gate (this is a significant improvement over the $O(n^2)$ complexity of state-of-the-art dishonest majority protocols, such as [7, 35, 36]).

Protocol	Model	Security	Preprocessing	Online	
			Comm.	Rounds	Comm.
Mascot [35]	DM	Abort	$O(n^2)$	2	$4n$
Overdrive[‡][36]		Abort	$O(n^2)$	2	$4n$
Assisted MPC[†][44]		Abort	$2n$	2	$4n$
ATLAS [28]	HM	Abort	NA	2	$2n$
MPClan [40]		Fairness	$1.5n$	2	$1.5n$
Asterisk	DM with HP	Fairness	3	2	$2n$

‡ For preprocessing, [35] relies on OT and [36] relies on SHE.
† [44] uses an additional helper party in the preprocessing phase.

Table 1: Comparison of Asterisk with the state-of-the-art dishonest majority (DM) and honest majority (HM) protocols.

In fact, Asterisk also outperforms Assisted MPC [44] in terms of preprocessing efficiency. Recall that, like Asterisk, Assisted MPC also relies on an external non-colluding semi-honest party (however, the helper party is only probabilistically semi-honest in the case of Assisted MPC, and the resulting protocol is only abort secure). However, Assisted MPC incurs a communication overhead of $O(n)$ per multiplication gate in its preprocessing phase, which is significantly higher than the requirement of communicating 3 elements per multiplication gate in the preprocessing phase of Asterisk.

Technically, these efficiency gains stem from careful use of the HP in the preprocessing phase of Asterisk, which constitutes a major technical novelty of our work. The (function-dependent) preprocessing phase of Asterisk uses the HP to: (a) produce authenticated additive shares of the masks for all wires in the circuit representation of the function to be computed, and (b) generates shares of multiplication triples. In particular, the preprocessing phase *exclusively* involves the HP generating and sending messages to the parties, while the parties themselves only perform local computations without communicating with each other. This minimizes the communication overhead, while also ensuring that the preprocessing phase is *inherently error-free* (thereby avoiding any additional overheads for verification checks). See Section 4 for details.

Efficient online phase. Asterisk also achieves better efficiency in the online phase as compared to state-of-the-art dishonest majority MPC protocols [35, 36]. We note that a major difference between Asterisk and Assisted MPC [44] is that Asterisk also uses the HP in the online phase, while Assisted MPC does not (thereby incurring greater overheads). Technically, Asterisk uses the HP in the online phase to communicate consistent values to the parties and thereby avoids the reliance on an expensive broadcast channel. Our design of Asterisk thus sheds light on the possibility of improving efficiency by using the HP effectively in the online phase.

Table 1 provides a comparison of Asterisk with other dishonest majority protocols [35, 36, 44] as well as state-of-the-art honest majority protocols [28, 40] in terms of efficiency and security. It is easy to see that Asterisk achieves better

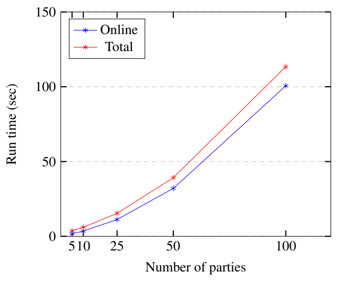

Figure 1: Online and total run time over a LAN network of Asterisk for a circuit of size 10^6 and the depth of the circuit is 10, for varying number of parties.

efficiency (in both the preprocessing and online phases) as well as stronger security guarantees (fairness as opposed to abort security) as compared to all of dishonest majority protocols [35, 36, 44]. In fact, Asterisk is actually closer to the honest majority protocols [28, 40] in terms of efficiency and security guarantees. This is especially notable given that Asterisk is capable of handling a dishonest majority (albeit under the assumption that the HP is semi-honest and non-colluding).

Impossibility of fairness in FaF with HP. We justify our choice of assuming the dishonest majority HP-aided setting (with a semi-honest, *fully* non-colluding HP) for the design of Asterisk by proving that this assumption is, in fact, necessary to achieve dishonest majority MPC protocols satisfying fairness. Concretely, we prove that it is impossible to design a fair protocol when considering the model of FaF security with HP. This result, coupled with the impossibility of attaining fairness using a colluding HP [32], effectively rules out the possibility of designing fair protocols unless the HP is *fully* non-colluding (which is precisely the model we use for designing Asterisk). See Section 3 for more details.

Applications and building blocks. We build upon Asterisk to design secure and efficient protocols in the HP-aided setting for secure auctions and dark pools. Along the way, we design sub-protocols for various granular functionalities, such as equality, comparison, dot product, and shuffle (among others). We believe that these building blocks are of independent interest (with potentially other applications such as privacy-preserving machine learning, anonymous broadcast, etc.) and underline the versatility of Asterisk as a framework for enabling a wide range of privacy-preserving applications in a secure yet efficient manner.

Implementation and benchmarks. We implement and benchmark Asterisk to showcase its efficiency as well as scalability. When evaluating a circuit comprising 10^6 multiplication gates, Asterisk has a response time of around 100 seconds for as many as 100 parties. Fig. 1 summarizes Asterisk's performance for varying number of parties. Fig. 2 presents a comparison between Asterisk and Assisted MPC [44] (both of which use some kind of external helper party). We highlight that Asterisk's preprocessing phase has improvements of up to 28× and 134×, in terms of running time and communication

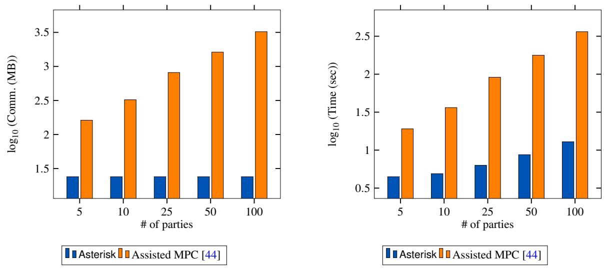

Figure 2: Preprocessing comparison between Assisted MPC [44] and Asterisk for a circuit of size 10^6 (the values are scaled in the logarithmic scale with base 10).

overheads, respectively, as compared to that of Assisted MPC. We present additional experimental results in Section 5 showcasing that, compared to state-of-the-art dishonest majority MPC protocols [35, 36], Asterisk achieves gains of up to 6× and 4× in total and online communication, respectively, as well as gains of up to 1000× in throughput of multiplicative triple-generation in the offline phase.

We also showcase the practicality of Asterisk by benchmarking the applications of secure auctions and dark pools. We observe that in the presence of up to 500 bidders, secure auction can be executed in around half a minute. For dark pools, we observe that for processing a new buy (or sell) request, Asterisk takes around 10 seconds given a sell (buy) list of size 500.

Comparison with FaF-secure protocols. As noted earlier, the dishonest majority HP-aided model bears some resemblance to the FaF-security model [1], with the difference that in the former model, the HP is a designated semi-honest party (there is no such designated party in FaF-security), while in FaF-security, the semi-honest adversary may be given access to the view of the malicious adversary (this is not allowed in the dishonest majority HP-aided model). While [1] presented some feasibility results for FaF-secure MPC without honest majority, more recent works have proposed concretely efficient FaF-secure MPC protocols [30, 38] that achieve stronger-than-abort security, albeit for specific numbers of parties (specifically, 4 parties in [30] and 5 parties in [38]) and specific corruption thresholds (at most one malicious corruption). It is not immediately obvious how one might extend these schemes for any general number of parties n while retaining comparable efficiency and security guarantees. On the other hand, the efficiency and security guarantees of Asterisk hold for any general number of parties (this is particularly suited for the applications we consider in this paper, namely secure auctions and dark pools, where the number of parties could be large in practice). We also note that [30] has a significantly more expensive preprocessing phase as compared to Asterisk, and both [30, 38] cannot tolerate a dishonest majority. Finally, we believe that for the applications that we consider in this paper, the dishonest majority HP-aided model more aptly captures the expected behavior of corrupt parties, which do not seem to have any particular incentive to share their views with the semi-honest HP.

Organization. Section 2 introduces notations and formally defines the dishonest majority HP-aided model. Section 3 justifies this model by presenting our impossibility result for fairness in the FaF model with a semi-honest HP. Section 4 describes our main contribution – the Asterisk protocol. Section 5 presents an outline of how Asterisk is used to realize the target applications (and the building blocks thereof), followed by implementation details and benchmarking results. Finally, Section 6 concludes and discusses some open questions. Certain missing details are deferred to the appendix.

2 Preliminaries

Threat model. We begin with a description of the dishonest majority HP-aided model of MPC that we use throughout the paper. We consider n parties $\mathcal{P} = \{P_1, \dots, P_n\}$, where each party P_i is initialized with input $x_i \in \{0,1\}^*$ and random coins $r_i \in \{0,1\}^*$. These parties interact in synchronous rounds. In every round parties can communicate either over a broadcast channel or a fully connected point-to-point (P2P) network, where we additionally assume all communication to be private and ideally authenticated. Further, we assume that there exists a special party HP, outside $\mathcal{P}$, called a "helper party" (abbreviated henceforth as HP) such that each party P_i can interact with HP via private and authenticated point-to-point channels. The HP does not typically hold any inputs, and also does not obtain any output at the end of the protocol. Depending on whether we allow the HP to keep any state in between its invocations (where an invocation corresponds to all parties interacting with the HP in a single round) or not, we refer to the HP as being 'stateful' or 'stateless' respectively.

We consider a non-colluding adversary who either corrupts up to $n-1$ among the n parties maliciously or the HP in a semi-honest manner. We prove the security of our protocols based on the standard real world / ideal world paradigm [10]. Essentially, the security of a protocol is analyzed by comparing what an adversary can do in the real execution of the protocol to what it can do in an ideal execution, that is considered secure by definition (in the presence of an incorruptible trusted party). For a detailed description of the security model of non-colluding, colluding and a related notion of friends-and-foes (FaF), refer to Appendix A.

Notations. We also introduce some notations that we use throughout the paper.

- *Data representation:* We consider secure computation over finite fields $\mathbb{F}$. $\mathbb{F}$ can be either a binary field $\mathbb{F}_2$ or a prime field $\mathbb{F}_p$ where p is a k-bit prime. For computation, we consider signed values, that is, where the boolean representation of a value $\mathsf{x} \in \mathbb{F}_p$ is in the 2's complement form. A positive value $\mathsf{x} \in [0, \frac{p}{2}]$ has the most significant bit (MSB) as 0 and a negative value $\mathsf{x} \in [-\frac{p}{2}, 0)$ has 1 in the MSB.
- $[1,n]$ denotes the set $\{1,\dots,n\}$.
- $(\mathsf{b})^{\mathsf{A}}$ is used to denote arithmetic equivalent of bit b.
- $\mathsf{b} = 1(\mathsf{x} \overset{?}{=} 0)$ denotes $\mathsf{b} = 1$ if $\mathsf{x} = 0$, otherwise $\mathsf{b} = 0$.
- $\mathsf{b} = 1(\mathsf{x} \overset{?}{\leq} 0)$ denotes $\mathsf{b} = 1$ if $\mathsf{x} \leq 0$, otherwise $\mathsf{b} = 0$.
- $\mathcal{F}_{\mathsf{Rand}}$ is a random coin-tossing functionality. It allows parties and HP to sample common random values such that a corrupt party guesses this value with negligible probability.

3 Impossibility of fairness in FaF with HP

In this section, we show that it is impossible to achieve fairness with FaF security in our HP-aided setting. The impossibility result, at a high level, follows similarly to the impossibility of attaining fairness for $n = 3, t = 1, h^* = 1$ with FaF security, as described in [1]. Note that in the standard definition of $(t, h^\star)$-FaF security, *any* subset of $h^\star$ parties can be semi-honestly corrupt. On the other hand, when considering FaF security with HP, it is only a designated entity (namely the HP) which is semi-honest (see Appendix A.1 for the formal definition). In this way, the standard notion of FaF security is stronger than FaF security with HP. Hence, the impossibility of FaF security with HP does not follow directly from the impossibility of [1]. We show how the proof of [1] can be modified to extend to the notion of FaF security with HP.

The main difference in our proof and that of [1], is that in our case, the semi-honest party (HP) does not have any input, while the semi-honest party in the case of [1] does. Further, the proof of [1] relied on this input being unchanged while the simulator invokes the ideal functionality. To make this proof work for our setting, we first modify the functionality to account for the fact that HP does not have an input. Further, instead of relying on the input of the semi-honest party remaining unchanged, we rely on the fact that the honest party's input to the ideal functionality remains unchanged. We next provide a high-level overview of the proof.

First, we work out the proof for two parties in which one will be maliciously corrupt and in addition there is a semi-honestly corrupt HP. The argument for n parties and one HP follows from player partitioning technique [43]. Consider 2 parties A and B, and a HP. We show that there exists a function F (which takes input from A and B), that cannot have a fair protocol in the considered model of FaF security with HP. To prove the above claim, we proceed via contradiction. That is, we show that if there exists a fair protocol, then there exists a polynomial time algorithm that can invert a one-way permutation. For this, we define the function F to be Swap_κ using a one-way permutation f_κ (κ is the security parameter). To be specific, we use the following functionality

$$\mathsf{Swap}_\kappa((a, y_B), (b, y_A)) = \begin{cases} (a,b) \text{ if } f_\kappa(a) = y_A \text{ and } f_\kappa(b) = y_B \\ 0^\kappa \text{ otherwise} \end{cases}$$

Each party inputs a pair of values $(a, y_B), (b, y_A)$, where a, b is in the domain of f_κ, and y_A, y_B is in the range of f_κ. Swap_κ

then outputs (a,b) if $f_\kappa(b) = y_A$ and $f_\kappa(a) = y_B$. Let there exist a fair protocol to compute Swap_κ that comprises r rounds, where $r \leq p(\kappa)$, for some polynomial $p(\cdot)$. We show that there exists a round $i \leq r$ such that either (A, HP) gain an advantage in learning the output than B or it is the case that (B, HP) gain an advantage in learning the output than A. Without loss of generality, consider the case that A is malicious. We then show that this advantage of the pair (A, HP) over (B, HP) cannot be simulated, thereby breaking the fairness property. For this, we show that if this can be simulated, then there exists a polynomial time algorithm which can break the one-wayness property of the one-way permutation.

To showcase that the advantage of (A, HP) over (B, HP) cannot be simulated, consider the strategy of a malicious A. A acts honestly (using the original input it held) until this round i, after which it aborts. A then sends its entire view to HP. Because of the aforementioned claim on the existence of round i, receiving A's view allows the HP to recover the correct output (corresponding to an all-honest execution) with a significantly higher probability than B. If there exists a simulator for such an (A, HP) pair, then we can construct a polynomial time algorithm M, which can be used to break the one-wayness of the underlying one-way permutation. Elaborately, M is the same algorithm performed by HP to obtain the output at the end of round i when A aborts. Observe that M takes as input the view of HP (which includes the view of corrupt A as well as its input y_B), and generates the output of Swap_κ, which comprises the inverse of a one-way permutation (b). Hence, using M, we can construct a polynomial-time algorithm to invert a one-way permutation.

The formal theorem appears below and detailed proof in Appendix B.

Theorem 3.1. *There exists an n-party functionality such that no protocol computes it with fairness in the FaF security model with* HP.

4 Efficient n-PC with HP

Here, we describe the MPC protocol of Asterisk, which is designed in the preprocessing paradigm. Computation begins by executing a one-time shared key setup phase, where common PRF keys are established between the parties to facilitate the non-interactive generation of common random values among the parties during the protocol execution. Next, the preprocessing phase kicks in, where the necessary preprocessing (input-independent) data is generated. This is followed by the online phase once the input is available.

4.1 Shared key setup

Let $F : \{0,1\}^\kappa \times \{0,1\}^\kappa \to \mathbb{F}$ be a secure pseudo-random function (PRF). Parties rely on a one-time setup [2, 3, 40, 41] to establish the following PRF keys among different subsets of the parties (including the HP) which allows them to non-interactively generate common random values among themselves. Specifically, the set of keys established between the parties and HP for our protocol is as follows:

- Every party $P_i \in \mathcal{P}$ holds a common key with HP, denoted by k_i, $i \in [1,n]$.
- $\mathcal{P}$ hold a common key $\mathsf{k}_\mathcal{P}$, unknown to HP.
- $\mathcal{P} \cup \{\mathrm{HP}\}$ hold a common key $\mathsf{k}_{\mathsf{all}}$.

The ideal functionality for the same appears in Fig. 3. Discussion on how to instantiate $\mathcal{F}_{\mathsf{Setup}}$ is deferred to Appendix C.1.

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

$\mathcal{F}_{\mathsf{Setup}}$ picks random keys $\{\{\mathsf{k}_i\}_{i\in[1,n]}, \mathsf{k}_\mathcal{P}, \mathsf{k}_{\mathsf{all}}\}$ for $\{\{P_i, \mathrm{HP}\}_{i\in[1,n]}, \mathcal{P}, \mathcal{P} \cup \{\mathrm{HP}\}\}$ respectively.

- Set $\mathbf{y}_i = \{\mathsf{k}_i, \mathsf{k}_\mathcal{P}, \mathsf{k}_{\mathsf{all}}\}$.
- Set $\mathbf{y}_{\mathrm{HP}} = \{\{\mathsf{k}_i\}_{i\in[1,n]}, \mathsf{k}_{\mathsf{all}}\}$.

Output: Send $(\mathsf{Output}, \mathbf{y}_s)$ to $P_s \in \mathcal{P}$ & $(\mathsf{Output}, \mathbf{y}_{\mathrm{HP}})$ to HP.

Figure 3: Ideal functionality for shared-key setup

4.2 Sharing semantics

To achieve (malicious) security in the dishonest majority setting, Asterisk uses *authenticated* additive secret-sharing of the inputs. Let $\mathsf{v} \in \mathbb{F}$ be a secret. We say that v is additively shared among parties in $\mathcal{P}$ if there exist $\mathsf{v}_i \in \mathbb{F}$ such that $\mathsf{v} = \sum_{i=1}^{n} \mathsf{v}_i$ and $P_i \in \mathcal{P}$ holds v_i. For authentication, we use an information-theoretic (IT) MAC (message authentication code) under a global MAC key $\Delta \in \mathbb{F}$. The authentication tag t for a value v is defined as $\mathsf{t} = \Delta \cdot \mathsf{v}$, where the MAC key Δ and the tag t are additively shared among the parties in $\mathcal{P}$. The following are the various sharing semantics used by Asterisk.

1. $[\cdot]$-*sharing:* A value $\mathsf{v} \in \mathbb{F}$ is said to be $[\cdot]$-shared (additively shared) among parties in $\mathcal{P}$ if the ith party P_i holds $[\mathsf{v}]_i \in \mathbb{F}$ such that $\mathsf{v} = \sum_{i=1}^{n} [\mathsf{v}]_i$.

2. $\langle\cdot\rangle$-*sharing:* A value $\mathsf{v} \in \mathbb{F}$ is said to be $\langle\cdot\rangle$-shared among parties in $\mathcal{P}$ if, the ith party P_i holds $\langle \mathsf{v} \rangle_i = \{[\mathsf{v}]_i, [\mathsf{t}_\mathsf{v}]_i\}$, where $\mathsf{t}_\mathsf{v} = \Delta \cdot \mathsf{v}$ is the tag of v, as discussed above. Along with the share of value v and the tag t_v, parties also hold an additive sharing of the MAC key Δ, i.e., P_i holds $[\Delta]_i$.

3. $[\![\cdot]\!]$-*sharing:* A value $\mathsf{v} \in \mathbb{F}$ is said to be $[\![\cdot]\!]$-shared if
   - there exists a mask $\delta_\mathsf{v} \in \mathbb{F}$ that is $\langle\cdot\rangle$-shared among the parties in $\mathcal{P}$, and
   - there exists a masked value $\mathsf{m}_\mathsf{v} = \mathsf{v} + \delta_\mathsf{v}$ that is held by each $P_i \in \mathcal{P}$.

   We denote P_i's $[\![\cdot]\!]$-share of v as $[\![\mathsf{v}]\!]_i = \{\mathsf{m}_\mathsf{v}, \langle \delta_\mathsf{v} \rangle_i\}$.

Note that all these sharing schemes are linear, i.e., given shares of values $a_1, \ldots, a_m$ and public constants $c_1, \ldots, c_m$, shares of $\sum_{i=1}^{m} c_i a_i$ can be computed non-interactively for an integer m. Further, Boolean sharing of a secret bit is analogous to arithmetic sharing as described above, with the difference that addition/subtraction operations are replaced by XOR ($\oplus$). We use the superscript $\mathbf{B}$ to denote Boolean sharing of a bit. The Boolean world is analogous to the arithmetic world where addition/subtraction operations are replaced with XORs, and multiplication is replaced with AND ($\wedge$).

4.3 Design of Asterisk

We proceed to the design of Asterisk in the preprocessing model. We describe the phases below.

Function-dependent preprocessing phase. In this phase, parties obtain $\langle\cdot\rangle$-shares of the masks associated with all the wires in the circuit representing the function to be computed, while HP gets them all in clear. Specifically, the HP begins by sampling the MAC key Δ and generates its $[\cdot]$-shares. For an input wire with value v where P_d is the input provider, parties generate $\langle\cdot\rangle$-shares of a random mask δ_{v} such that δ_{v} is known to P_d and HP. For wires that are the output of an addition gate, HP and all parties locally add the $\langle\cdot\rangle$-shares of masks of the input wires of the addition gate to obtain the $\langle\cdot\rangle$-shares of the mask for the output wire. With respect to a multiplication gate, let x, y be the inputs to this gate and z be the output. Parties generate $\langle\cdot\rangle$-shares of a random mask δ_{z} such that it is known to the HP. Moreover, HP computes $\delta_{\mathsf{xy}} = \delta_{\mathsf{x}} \cdot \delta_{\mathsf{y}}$, where $\delta_{\mathsf{x}}, \delta_{\mathsf{y}}$ are the masks corresponding to x and y, respectively. Then it generates $\langle\cdot\rangle$-share of δ_{xy}. Looking ahead, δ_{xy} will be used to generate the masked value for the output of the multiplication gate. Finally, for an output gate where v is the output, HP generates and stores the mask δ_{v}. During the online phase, it will send δ_{v} to all the parties to facilitate output reconstruction. The preprocessing phase is described in Fig. 7 which uses several sub-protocols for generating the $\langle\cdot\rangle$-shares of different values. We start with the latter.

The preprocessing phase involves generation of $\langle\cdot\rangle$-sharing of the following kind of secrets– (i) a random mask that is known by a designated party and the HP, (ii) a random mask that is known only to the HP, (iii) a secret value (not necessarily uniformly random) known to the HP. Note that to generate $\langle\cdot\rangle$-shares of a value v, the HP can sample $n-1$ shares $[\mathsf{v}]_i$ for $i \in \{1,\ldots,n-1\}$, followed by computing $[\mathsf{v}]_n = \mathsf{v} - \sum_{i=1}^{n-1}[\mathsf{v}]_i$. Similarly, it can compute the tag $\mathsf{t_v}$, then sample $n-1$ shares for $\mathsf{t_v}$ followed by computing the last share for $\mathsf{t_v}$. The HP can the send the ith share of v and $\mathsf{t_v}$ to P_i. This would require $2n$ elements of communication. We improve this cost by using the common keys established during the setup phase.

$\Pi_{\langle\cdot\rangle\text{-Sh}}(P_d, \mathsf{Rand})$ (Fig. 4) indicates generating a $\langle\cdot\rangle$-sharing of a random value v such that the dealer, P_d, and HP know the value in clear. This requires communication of 2 elements.

Protocol $\Pi_{\langle\cdot\rangle\text{-Sh}}(P_d, \mathsf{Rand})$

Input: $\forall i \in [1,n]$, P_i's input k_i, and HP's input $\{\{\mathsf{k}_i\}_{i\in[1,n]}, \Delta\}$.
Output: P_d's output v and $\langle\mathsf{v}\rangle_d$ and $P_i (i \neq d)$ gets $\langle\mathsf{v}\rangle_i$.
Protocol:
- HP and P_d sample a random value v using the k_d.
- For all $i \in \{1,\ldots,n-1\}$, HP and P_i sample a random share $[\mathsf{v}]_i$ and a random share of the tag $[\mathsf{t_v}]_i$ using the k_i.
- HP computes $[\mathsf{v}]_n = \mathsf{v} - \sum_{i\in[n-1]}[\mathsf{v}]_i$ and $[\mathsf{t_v}]_n = (\Delta \cdot \mathsf{v}) - \sum_{i\in[n-1]}[\mathsf{t_v}]_i$.
- HP sends $([\mathsf{v}]_n, [\mathsf{t_v}]_n)$ to P_n.

Figure 4: Generating $\langle\cdot\rangle$ of a random value known to P_d

$\Pi_{\langle\cdot\rangle\text{-Sh}}(\mathsf{HP}, \mathsf{v})$ (Fig. 5) indicates generating a $\langle\cdot\rangle$-sharing of a random value v sampled by HP. This requires communication of 1 element.

Protocol $\Pi_{\langle\cdot\rangle\text{-Sh}}(\mathsf{HP}, \mathsf{Rand})$

Input: $\forall i \in [1,n]$, P_i's input k_i, and HP's input $\{\{\mathsf{k}_i\}_{i\in[1,n]}, \Delta\}$.
Output: P_i gets $\langle\mathsf{v}\rangle_i$ for all $i \in [1,n]$.
Protocol:
- For all $i \in \{1,\ldots,n-1\}$, HP and P_i sample a random share $[\mathsf{v}]_i$ and a random share of the tag $[\mathsf{t_v}]_i$ using k_i.
- HP and P_n sample a random value $[\mathsf{v}]_n$ using k_n.
- HP computes $\mathsf{v} = \sum_{i=1}^{n}[\mathsf{v}]_i$ and $[\mathsf{t_v}]_n = (\Delta \cdot \mathsf{v}) - \sum_{i\in[n-1]}[\mathsf{t_v}]_i$.
- HP sends $[\mathsf{t_v}]_n$ to P_n.

Figure 5: Generating $\langle\cdot\rangle$ of a random value known only to HP

$\Pi_{\langle\cdot\rangle\text{-Sh}}(\mathsf{HP}, \mathsf{v})$ (Fig. 6) indicates generating a $\langle\cdot\rangle$-sharing of v held by HP. This requires communication of 2 elements.

Protocol $\Pi_{\langle\cdot\rangle\text{-Sh}}(\mathsf{HP}, \mathsf{v})$

Input: $\forall i \in [1,n]$, P_i's input k_i, and HP's input $\{\{\mathsf{k}_i\}_{i\in[1,n]}, \Delta, \mathsf{v}\}$.
Output: P_i gets $\langle\mathsf{v}\rangle_i$ for all $i \in [1,n]$.
Protocol:
- For all $i \in \{1,\ldots,n-1\}$, HP and P_i sample a random share $[\mathsf{v}]_i$ and a random share of the tag $[\mathsf{t_v}]_i$ using k_i.
- HP computes $[\mathsf{v}]_n = \mathsf{v} - \sum_{i\in[n-1]}[\mathsf{v}]_i$ and $[\mathsf{t_v}]_n = (\Delta \cdot \mathsf{v}) - \sum_{i\in[n-1]}[\mathsf{t_v}]_i$.
- HP sends $([\mathsf{v}]_n, [\mathsf{t_v}]_n)$ to P_n.

Figure 6: Generating $\langle\cdot\rangle$ of a value v known to HP

We emphasize that the complete preprocessing for a circuit C can be performed at once. Only one round of communication is required from HP to P_n such that all the parties obtain the preprocessing data for the circuit C. For the ith party, $\mathsf{preproc}_i$ be the preprocessing data. $\mathsf{preproc}_i$ consists of $\langle\cdot\rangle$-sharing of all wires of C, additionally, for multiplication gates, it contains $\langle\cdot\rangle$-sharing of the product of the masks of the input wires for the corresponding gate. HP's $\mathsf{preproc}_{\mathsf{HP}}$ only contains the mask of the output wires, which is used for the output reconstruction. We describe the preprocessing phase of our protocol in Fig. 7.

Protocol Π_{Prep}

Input: For all $i \in [1,n]$, P_i has input the PRF key k_i and the circuit C. HP has $\{\mathsf{k}_i\}_{i\in[1,n]}$ and C as input.
Output: P_i's output $\mathsf{preproc}_i$ and HP's output $\mathsf{preproc}_{\mathsf{HP}}$.
Protocol:

- *KeyGen:* For all $i \in \{1,\ldots,n\}$, P_i and HP sample $[\Delta]_i$ using their common key k_i. HP computes $\Delta = \sum_{i=1}^{n} [\Delta]_i$.
- *Input gates:* For a gate with input v and input provider P_d, all parties and HP execute $\Pi_{\langle\cdot\rangle\text{-}\mathsf{Sh}}(P_d, \mathsf{Rand})$ to enable generation of mask δ_{v} for v in clear among HP and P_d, and generation of its $\langle\cdot\rangle$-shares among the parties in $\mathcal{P}$.
- *Addition gates:* For a gate with input x and y, HP and P_i locally compute $\langle\delta_{\mathsf{x+y}}\rangle_i = \langle\delta_{\mathsf{x}}\rangle_i + \langle\delta_{\mathsf{y}}\rangle_i$, where $\langle\delta_{\mathsf{x}}\rangle_i + \langle\delta_{\mathsf{y}}\rangle_i = ([\delta_{\mathsf{x}}]_i + [\delta_{\mathsf{y}}]_i, [\mathsf{t_x}]_i + [\mathsf{t_y}]_i)$.
- *Multiplication gates:* For a gate with input x and y, all parties and HP execute $\Pi_{\langle\cdot\rangle\text{-}\mathsf{Sh}}(\mathsf{HP}, \mathsf{Rand})$ to generate $\langle\cdot\rangle$-shares of the mask δ_{z} for the output wire $\mathsf{z} = \mathsf{x}\cdot\mathsf{y}$, where δ_{z} is known in clear to HP. HP computes $\delta_{\mathsf{xy}} = \delta_{\mathsf{x}} \cdot \delta_{\mathsf{y}}$. All parties and HP execute $\Pi_{\langle\cdot\rangle\text{-}\mathsf{Sh}}(\mathsf{HP}, \delta_{\mathsf{xy}})$ to generate $\langle\delta_{\mathsf{xy}}\rangle$.

Figure 7: The function-dependent preprocessing phase of Asterisk for a circuit C

Note that the preprocessing phase exclusively involves the (semi-honest) HP generating and sending certain values to the parties. All other parties' tasks are limited to local computation, and they are not involved in the communication. Hence, *the preprocessing phase is inherently error-free*, and we do not need any additional verification checks.

Online phase. Given that $\{\{\mathsf{preproc}_i\}_{i\in[1,n]}\}$ and $\mathsf{preproc}_{\mathsf{HP}}$ is generated in the preprocessing phase, once the input is available, the online phase involves generating the masked values for all the wires in the circuit, as per the $[\![\cdot]\!]$-sharing semantics. Recall that the masked value $\mathsf{m_v}$ for a $[\![\cdot]\!]$-shared value v is defined as $\mathsf{m_v} = \mathsf{v} + \delta_{\mathsf{v}}$ where $\langle\delta_{\mathsf{v}}\rangle$ is generated in the preprocessing phase. At a high level, to generate the masked value $\mathsf{m_v}$, each party is required to perform local computations to generate its $[\cdot]$-share of $\mathsf{m_v}$. The parties are then expected to send their $[\cdot]$-share of $\mathsf{m_v}$ consistently to every other party. Each party uses the received $[\cdot]$-shares of $\mathsf{m_v}$ to reconstruct it. Note that a malicious party may send inconsistent $[\cdot]$-shares to the other parties. To ensure consistency of the communicated messages, we resort to performing all communication via the HP. Since the HP is semi-honest and will not introduce any errors, communicating via the HP emulates a perfectly secure broadcast channel. Note that this communication pattern via the HP is similar to the $\mathsf{P_{King}}$ based approach [23], where parties in $\mathcal{P}$ send their $[\cdot]$-shares of a value to the $\mathsf{P_{King}}$, followed by $\mathsf{P_{King}}$ sending messages to all the parties. Observe, however, that the HP receives all the $[\cdot]$-shares of $\mathsf{m_v}$ from all the parties, which empowers it to learn $\mathsf{m_v}$ in the clear. Moreover, since the HP also knows the mask δ_{v}, this allows it to learn the underlying secret $\mathsf{v} = \mathsf{m_v} - \delta_{\mathsf{v}}$, thereby breaching privacy. To avoid such leakage, all messages that are communicated via the HP are further masked with random values which are known only to the parties in $\mathcal{P}$. In this way, it is ensured that parties receive consistent messages via the HP, while the HP does not learn any additional information. While the formal details of the online phase for evaluating the circuit representing the function to be computed appear in Fig. 11, we next give an overview of the same.

Input gates. For an input gate $\mathsf{g_{in}}$, let P_d be the designated party, referred to as the dealer, providing the input v. Observe that, from the preprocessing phase, P_d holds a random value δ_{v} as a mask for v, while the i-th party P_i holds $\langle\delta_{\mathsf{v}}\rangle_i$. All parties, excluding the HP, sample a common random value r using their common PRF key $\mathsf{k}_{\mathcal{P}}$. P_d computes $q_{\mathsf{v}} = \mathsf{v} + \delta_{\mathsf{v}} + r$ and sends q_{v} to HP. HP sends q_{v} to all the parties. Each party locally computes $\mathsf{m_v} = q_{\mathsf{v}} - r$, thereby generating its $[\![\cdot]\!]$-share of v as $[\![\mathsf{v}]\!]_i = \{\mathsf{m_v}, \langle\delta_{\mathsf{v}}\rangle_i\}$. The formal details of input sharing are provided in Fig. 8.

Protocol $\Pi_{[\![\cdot]\!]\text{-}\mathsf{Sh}}$

Input:

- Every input provider/dealer P_d, has input v and the mask δ_{v}.
- For all $i \in [1,n]$, P_i has the input gate $\mathsf{g_{in}}$, $\langle\delta_{\mathsf{v}}\rangle_i$ generated in Fig. 7 for $\mathsf{g_{in}}$ and common key $\mathsf{k}_{\mathcal{P}}$ generated in the setup phase.

Output: For all $i \in [1,n]$, P_i's output $[\![\mathsf{v}]\!]_i = (\mathsf{m_v}, \langle\delta_{\mathsf{v}}\rangle_i)$ where $\mathsf{m_v} = \mathsf{v} + \delta_{\mathsf{v}}$.
Protocol:

- All parties, excluding HP, sample a random value r using their common key $\mathsf{k}_{\mathcal{P}}$.
- P_d computes $q_{\mathsf{v}} = \mathsf{v} + \delta_{\mathsf{v}} + r$ and sends it to HP.
- HP sends q_{v} to all the parties.
- For all $i \in [1,n]$, P_i sets $\mathsf{m_v} = q_{\mathsf{v}} - r$ and outputs $[\![\mathsf{v}]\!]_i = \{\mathsf{m_v}, \langle\delta_{\mathsf{v}}\rangle_i\}$.

Figure 8: The online phase of input sharing of Asterisk

Addition gates. For an addition gate $\mathsf{g_{add}}$, the parties locally add the $[\![\cdot]\!]$-shares of the input wires to obtain the $[\![\cdot]\!]$-sharing for the output wire. Parties complete this step by locally adding the masked values of the input wires of $\mathsf{g_{add}}$ and the addition of the masks is already done in preprocessing.

Multiplication gates. For a multiplication gate $\mathsf{g_{mul}}$, let x and y be the inputs, and let z be the output. To generate $[\![\cdot]\!]$-shares of z, parties need to generate $\mathsf{m_z}$, and $\langle\cdot\rangle$-shares of the mask δ_{z}. Recall that $\langle\cdot\rangle$-shares of δ_{z} are generated in the preprocessing phase. With respect to $\mathsf{m_z}$, it can be computed as follows.

$$\begin{aligned}\mathsf{m_z} = \mathsf{z} + \delta_{\mathsf{z}} = \mathsf{xy} + \delta_{\mathsf{z}} &= (\mathsf{m_x} - \delta_{\mathsf{x}})(\mathsf{m_y} - \delta_{\mathsf{y}}) + \delta_{\mathsf{z}} \\ &= \mathsf{m_x m_y} - \mathsf{m_x}\delta_{\mathsf{y}} - \mathsf{m_y}\delta_{\mathsf{x}} + \delta_{\mathsf{xy}} + \delta_{\mathsf{z}}\end{aligned}$$

Given the $\langle\cdot\rangle$-shares of $\delta_{\mathsf{x}}, \delta_{\mathsf{y}}, \delta_{\mathsf{z}}$ and δ_{xy}, which are generated in the preprocessing, and relying on the linearity of $\langle\cdot\rangle$-sharing, parties can locally generate $\langle\cdot\rangle$-shares of $\mathsf{m_z}$ using the above equation. To reconstruct $\mathsf{m_z}$, a straightforward approach is for all parties to send their share of $\mathsf{m_z}$ to the HP. However, this would leak information about $\mathsf{z} = \mathsf{m_z} - \delta_{\mathsf{z}}$ to the semi-honest HP, thus leading to a privacy breach. In order to achieve privacy against the HP, all parties excluding HP sample a random vector $\boldsymbol{r} = (r_1, \ldots, r_n)$ using their common PRF key $\mathsf{k}_{\mathcal{P}}$ such that P_i uses r_i to pad its share of $\mathsf{m_z}$. Elaborately, since $\langle\cdot\rangle$-share of $\mathsf{m_z}$ comprises of $[\cdot]$-shares of $\mathsf{m_z}$ and its tag $\mathsf{t_{m_z}}$, parties compute $[q_\mathsf{z}]_i = r_i + [\mathsf{m_z}]_i$, and $[\mathsf{t}_{q_\mathsf{z}}]_i = -\mathsf{m_x} \cdot [\mathsf{t_y}]_i - \mathsf{m_y} \cdot [\mathsf{t_x}]_i + [\mathsf{t_{xy}}]_i + [\mathsf{t_z}]_i + [\Delta]_i \cdot (\mathsf{m_x} \cdot \mathsf{m_y} + \sum_{j=1}^{n} r_j)$. Each party P_i then sends $[q_\mathsf{z}]_i$ to the HP, which in turn reconstructs q_z and sends the reconstructed value back to all the parties. Each party locally computes $\mathsf{m_z} = q_\mathsf{z} - \sum_{j=1}^{n} r_j$, and thus obtains $[\![\mathsf{z}]\!]$. Formal details of multiplication protocol appear in Fig. 9.

Protocol Π_{mult}

Input:

- For all $i \in [1,n]$, P_i has the multiplication gate $\mathsf{g_{mul}}$. It has the $\langle\cdot\rangle$ of the following values generated in Fig. 7 for $\mathsf{g_{mul}}$ $\delta_\mathsf{x}, \delta_\mathsf{y}, \delta_\mathsf{xy}, \delta_\mathsf{z}$.
- For all $i \in [1,n]$, P_i has the common key $\mathsf{k}_{\mathcal{P}}$ generated in the setup phase.

Output: For all $i \in [1,n]$, P_i's output $\mathsf{m_z} = \mathsf{z} + \delta_\mathsf{z}$, where $\mathsf{z} = \mathsf{x} \cdot \mathsf{y}$.

Protocol:

- For all $i \in [1,n]$, P_i holds $[\![\mathsf{x}]\!]_i, [\![\mathsf{y}]\!]_i$ and $\langle\delta_\mathsf{xy}\rangle_i, \langle\delta_\mathsf{z}\rangle_i$.
- All parties excluding HP sample a random vector $\boldsymbol{r}$ of length n using their common key $\mathsf{k}_{\mathcal{P}}$.
- Each party P_i computes $[\mathsf{t}_{q_\mathsf{z}}]_i = -\mathsf{m_x} \cdot [\mathsf{t_y}]_i - \mathsf{m_y} \cdot [\mathsf{t_x}]_i + [\mathsf{t_{xy}}]_i + [\mathsf{t_z}]_i + [\Delta]_i \cdot (\mathsf{m_x} \cdot \mathsf{m_y} + \sum_{j=1}^{n} r_j)$.
- P_1 computes $[q_\mathsf{z}]_1 = \mathsf{m_x} \cdot \mathsf{m_y} + \boldsymbol{r}_1 - \mathsf{m_x} \cdot [\delta_\mathsf{y}]_1 - \mathsf{m_y} \cdot [\delta_\mathsf{x}]_1 + [\delta_\mathsf{xy}]_1 + [\delta_\mathsf{z}]_1$ and sends it to HP.
- For all $i \in [1,n] \setminus \{1\}$, P_i computes $[q_\mathsf{z}]_i = \boldsymbol{r}_i - \mathsf{m_x} \cdot [\delta_\mathsf{y}]_i - \mathsf{m_y} \cdot [\delta_\mathsf{x}]_i + [\delta_\mathsf{xy}]_i + [\delta_\mathsf{z}]_i$ and sends it to HP.
- HP reconstructs q_z and sends it to all the parties.
- For all $i \in [1,n]$, P_i sets $\mathsf{m_z} = q_\mathsf{z} - \sum_{i=1}^{n} \boldsymbol{r}_i$ and outputs $[\![\mathsf{z}]\!] = (\mathsf{m_z}, \langle\delta_\mathsf{z}\rangle_i)$.

Figure 9: The online phase of multiplication of Asterisk

Verification. What remains to check is whether a party P_i has sent the correct shares of $\mathsf{m_z}$ to the HP. To check for this, it suffices to check if $[\mathsf{t}_{q_\mathsf{z}}] - q_\mathsf{z} \cdot [\Delta]$ is a sharing of 0. To attain an efficient realization of this check, we combine the verification with respect to multiple multiplication gates into a single check, and this check is performed at the end of the computation, before output reconstruction. This allows us to amortize the cost due to this verification check across many multiplication gates. To combine the checks corresponding to m multiplication gates into one check, parties proceed as follows.

- All parties (including the HP) sample a random vector $\boldsymbol{\rho} = (\rho_0, \rho_1, \ldots, \rho_m) \leftarrow \mathbb{F}^{m+1}$ using the common coin functionality $\mathcal{F}_{\mathsf{Rand}}$. Recall that the $\mathcal{F}_{\mathsf{Rand}}$ functionality allows parties and HP to sample common random values such that a corrupt party can guess the sampled value with negligible probability. To realize $\mathcal{F}_{\mathsf{Rand}}$, HP samples a key $\mathsf{k_{ver}}$ and sends it to all the parties in $\mathcal{P}$. Parties can evaluate a PRF with key $\mathsf{k_{ver}}$ on a common input ctr. Note that, we can not use $\mathsf{k_{all}}$ for $\mathcal{F}_{\mathsf{Rand}}$ since an adversarial party knows the key beforehand and can compute $\boldsymbol{\rho}$. Thus the soundness of the verification check does not hold. Since the key $\mathsf{k_{ver}}$ is not known to any party, a malicious party can guess $\boldsymbol{\rho}$ with negligible probability. Therefore, it can cheat and pass the verification with negligible probability.
- All parties (excluding the HP) sample a random sharing of 0, i.e., P_i gets α_i such that $\sum_{i=1}^{n} \alpha_i = 0$, using their common key $\mathsf{k}_{\mathcal{P}}$.

Let $\{q_{\mathsf{z}_1}, \ldots, q_{\mathsf{z}_m}\}$ be the reconstructed values sent by HP to all the parties, and let $\{[\mathsf{t}_{q_{\mathsf{z}_1}}]_i, \ldots, [\mathsf{t}_{q_{\mathsf{z}_m}}]_i\}$ be the shares of the tags held by party P_i. For $j \in \{1, \ldots, m\}$, P_i computes $[\omega_{\mathsf{z}_j}]_i = [\mathsf{t}_{q_{\mathsf{z}_j}}]_i - q_{\mathsf{z}_j} \cdot [\Delta]_i$, and sends the following to the HP: $[\omega_\mathsf{z}]_i = \rho_0 \cdot \alpha_i + \sum_{j \in [m]} \rho_j \cdot [\omega_{\mathsf{z}_j}]_i$. Finally, the HP checks if $\sum_{i=1}^{n} [\omega_\mathsf{z}]_i = 0$, and sends the outcome to all the parties. The details of the batched MAC verification are given in Fig. 10.

Protocol Π_{verify}

Input:

- For every $i \in [1,n]$, P_i has the reconstructed values $\{q_{\mathsf{z}_1}, q_{\mathsf{z}_2}, \ldots, q_{\mathsf{z}_m}\}$.
- For every $i \in [1,n]$, P_i has $[\mathsf{t}_{q_{\mathsf{z}_j}}]_i$ as the MAC share of q_{z_j} and $[\Delta]_i$ as the MAC key share for the ith party.

Output: All parties and HP output 1 if the verification passes, else 0.

Protocol:

- HP samples a new key $\mathsf{k_{ver}}$ and send it to all the parties in $\mathcal{P}$.
- For all $i \in [1,n]$, P_i computes $[\omega_{\mathsf{z}_j}]_i = [\mathsf{t}_{q_{\mathsf{z}_j}}]_i - q_{\mathsf{z}_j} \cdot [\Delta]_i$.
- For all $i \in [1,n]$, P_i samples $\boldsymbol{\rho} = (\rho_0, \rho_1, \ldots, \rho_m) \leftarrow \mathbb{F}^{m+1}$ using the key $\mathsf{k_{ver}}$.
- All parties (excluding the HP) sample a random sharing of 0, i.e., P_i gets α_i such that $\sum_{i=1}^{n} \alpha_i = 0$, using the common key $\mathsf{k}_{\mathcal{P}}$.
- For all $i \in [1,n]$, P_i computes $[\omega_\mathsf{z}]_i = \rho_0 \cdot \alpha_i + \sum_{j=1}^{m} \rho_j \cdot [\omega_{\mathsf{z}_j}]_i$ and sends to HP.
- HP checks if $\sum_{i=1}^{n} [\omega_\mathsf{z}]_i = 0$. If the check fails, HP sends 0 to all, else 1.

Figure 10: Amortized MAC-Verification of Asterisk

Output reconstruction. If the parties pass the above verification, then HP sends the masks for the output wires. Let v be an output, and δ_v be the mask generated in the preprocessing phase. HP sends δ_v to all the parties. Each party, holding $\mathsf{m_v}$ computes $\mathsf{v} = \mathsf{m_v} - \delta_\mathsf{v}$ and gets the output.

Remark. Note that HP sends the masks of the output wires to all the parties only if the verification is passed. Therefore if a malicious party deviates from the protocol, resulting in abort, the malicious party fails to learn the output of the protocol. Hence, our protocol achieves the fairness security guarantee. Since HP stores the masks of the output wires, this requires the HP to be a stateful machine that stores data in memory. However, we can also adapt our protocol to use a stateless

HP instead using known techniques from [27], which we elaborate upon in Appendix D.

Protocol Π_{Online}

Input:

- For every input provider/dealer P_d, it has input v.
- For all $i \in [1,n]$, P_i has the circuit C, the preprocessing data $\mathsf{preproc}_i$ generated in Fig. 7 and common keys $\{\mathsf{k}_i, \mathsf{k}_{\mathcal{P}}, \mathsf{k}_{\mathsf{all}}\}$ generated in the setup phase.
- HP's input $\mathsf{preproc}_{\mathsf{HP}}$.

Output: For all $i \in [1,n]$, P_i's output $\mathsf{C}(\mathsf{x}_1, \mathsf{x}_2, \ldots, \mathsf{x}_n)$.

Protocol:

- *Input gates:* Parties and HP execute $\Pi_{[\![\cdot]\!]\text{-}\mathsf{Sh}}$ (Fig. 8) for every input gates of C.
- *Addition gates:* Let $\mathsf{g}_{\mathsf{add}}$ be an addition gate, where x and y are the inputs.
  - For all $i \in [1,n]$, P_i holds $[\![\mathsf{x}]\!]_i, [\![\mathsf{y}]\!]_i$.
  - For all $i \in [1,n]$, P_i locally computes $[\![\mathsf{x}+\mathsf{y}]\!]_i = [\![\mathsf{x}]\!]_i + [\![\mathsf{y}]\!]_i$, where $[\![\mathsf{x}]\!]_i + [\![\mathsf{y}]\!]_i = ((\mathsf{m}_\mathsf{x} + \mathsf{m}_\mathsf{y}), (\langle \delta_\mathsf{x} \rangle_i + \langle \delta_\mathsf{y} \rangle_i))$.
- *Multiplication gates:* Parties and HP execute Π_{mult} (Fig. 9) for every multiplication gates of C.
- *Amortized verification:* At the end of all the multiplication gates, all parties and HP execute Π_{verify}(Fig. 10). If parties output 1, then $\mathsf{continue}$, else output $\perp$.
- *Output reconstruction:* Let g be an output wire and v be an output. HP holds the mask δ_v of v. HP sends δ_v to all the parties. Parties get the output $\mathsf{v} = \mathsf{m}_\mathsf{v} - \delta_\mathsf{v}$.

Figure 11: The online phase of Asterisk for circuit C

4.4 Proof of security

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

- **Initialize:** On input $(\mathsf{Init}, \mathbb{F})$ from parties and HP, store $\mathbb{F}$.
- **Input:** On input $(\mathsf{Input}, P_i, \mathsf{id}, \mathsf{x})$ from P_i and $(\mathsf{Input}, P_i, \mathsf{id})$ from all other parties, with a fresh identifier id and $\mathsf{x} \in \mathbb{F}$, store $(\mathsf{id}, \mathsf{x})$.
- **Add:** On command $(\mathsf{Add}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$ from all parties (where $\mathsf{id}_1, \mathsf{id}_2$ are present in memory), retrieve $(\mathsf{id}_1, \mathsf{x})$, $(\mathsf{id}_2, \mathsf{y})$ and store $(\mathsf{id}_3, \mathsf{x}+\mathsf{y})$.
- **Multiply:** On command $(\mathsf{Mult}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$ from all parties (where $\mathsf{id}_1, \mathsf{id}_2$ are present in memory), retrieve $(\mathsf{id}_1, \mathsf{x})$, $(\mathsf{id}_2, \mathsf{y})$ and store $(\mathsf{id}_3, \mathsf{x} \cdot \mathsf{y})$.
- **Output:** On input $(\mathsf{Output}, \mathsf{id})$ from all the parties (where id is present in the memory), output id to the adversary. Wait for an input from the adversary; if this is $\mathsf{Deliver}$ then retrieve $(\mathsf{id}, \mathsf{y})$ and send it to all the parties, otherwise output $\perp$ to all the parties including the adversary.

Figure 12: Ideal functionality for evaluating f with fairness

Lemma 4.1. *Assuming existence of a PRF, protocol* Asterisk *designed in the preprocessing(Fig. 7)-online(Fig. 11) realizes $\mathcal{F}_{\mathsf{MPC}}$ (Fig. 12) with computational non-colluding security in the presence of a semi-honest* HP.

Below we briefly discuss the proof of Lemma 4.1 and the formal proof is given in Appendix C.2.

Case I: Consider a malicious adversary corrupting all but one party from $\mathcal{P}$. In the preprocessing, the adversary performs only local computation thus the preprocessing is error-free. The online phase is similar to the online phase of [7]. Before the output reconstruction, the verification check ensures that the adversary cannot cheat (except with negligible probability) by sending incorrect values during the reconstruction phase which is executed by HP. Finally, to obtain the output, HP sends the masks of the output wires only if the verification check is passed. Thus, either all parties get the output or none which ensures fairness.

Case II: Consider the HP to be semi-honest. In the preprocessing phase, it generates the preprocessing data to evaluate the circuit C. In the online phase, all the values sent to HP are padded with a randomly sampled value. Therefore, to a semi-honest HP, the messages it received in the online phase are indistinguishable from randomly sampled messages. Hence, our protocol is secure against the semi-honest HP.

5 Applications and benchmarks

In this section, we describe how we use Asterisk to realize two applications, namely secure auctions and privacy-preserving dark pools, in a practically efficient yet secure manner. We then describe a prototype implementation of Asterisk and compare its performance other MPC protocols [35, 36, 44]. Finally, we implement and benchmark the aforementioned applications built on top of our Asterisk.

5.1 Building blocks for applications

For the desired applications, we realize several building blocks, which are essentially adaptations of Asterisk for certain specific (sub-)functionalities. These sub-protocols are then used by the overall protocols realizing the applications. While these building blocks have been studied in the traditional MPC settings [13, 19, 38, 46], we present novel and highly efficient instantiations of these building blocks in the HP-aided setting.

Table 2 summarizes the various sub-protocols that we use as building blocks for our applications, as well as the corresponding overheads. Due to lack of space in the body of the paper, we defer the full details of these building blocks to Appendix E, and the details of how to use these building blocks to realize the desired applications to Appendix F. Conceptually, our realizations of these sub-protocols follow the same design paradigm as our multiplication protocol described in Sec: 4.3, where all of the communication happens via the HP, and the correctness of the computation is verified before

Protocol	Input	Output	Preprocessing	Online Comm	Online Round
Π_{mult}	$\llbracket \mathsf{a} \rrbracket, \llbracket \mathsf{b} \rrbracket$	$\llbracket \mathsf{x} \rrbracket : \mathsf{x} = \mathsf{a} \cdot \mathsf{b}$	3	$2n$	2
Π_{mult3}	$\llbracket \mathsf{a} \rrbracket, \llbracket \mathsf{b} \rrbracket, \llbracket \mathsf{c} \rrbracket$	$\llbracket \mathsf{y} \rrbracket : \mathsf{y} = \mathsf{a} \cdot \mathsf{b} \cdot \mathsf{c}$	9	$2n$	2
Π_{mult4}	$\llbracket \mathsf{a} \rrbracket, \llbracket \mathsf{b} \rrbracket, \llbracket \mathsf{c} \rrbracket, \llbracket \mathsf{d} \rrbracket$	$\llbracket \mathsf{z} \rrbracket : \mathsf{z} = \mathsf{a} \cdot \mathsf{b} \cdot \mathsf{c} \cdot \mathsf{d}$	23	$2n$	2
$\Pi_{\mathsf{PrefixAND}}$	$\llbracket \mathsf{x}_1 \rrbracket^{\mathsf{B}}, \ldots, \llbracket \mathsf{x}_k \rrbracket^{\mathsf{B}}$	$\llbracket \mathsf{y}_1 \rrbracket^{\mathsf{B}}, \ldots, \llbracket \mathsf{y}_k \rrbracket^{\mathsf{B}} : \mathsf{y}_i = \wedge_{j=1}^{i} \mathsf{x}_j$	$\frac{35}{4} \log_4 k$	$\frac{3}{2} n \log_4 k$	$2 \log_4 k$
$\Pi_{k\text{-}\mathsf{mult}}$	$\llbracket \mathsf{x}_1 \rrbracket^{\mathsf{B}}, \ldots, \llbracket \mathsf{x}_k \rrbracket^{\mathsf{B}}$	$\llbracket \mathsf{b} \rrbracket^{\mathsf{B}} : \mathsf{b} = \wedge_{i=1}^{k} \mathsf{x}_i$	8	$\frac{2n}{3}$	$2 \log_4 k$
Π_{EQZ}	$\llbracket \mathsf{x} \rrbracket$	$\llbracket \mathsf{b} \rrbracket^{\mathsf{B}} : \mathsf{b} = 1(\mathsf{x} \overset{?}{=} 0)$	12	$\frac{2n}{3}$	$2 \log_4 k$
Π_{BitA}	$\llbracket \mathsf{b} \rrbracket^{\mathsf{B}}$	$\llbracket \mathsf{b} \rrbracket$	3	$2n$	2
Π_{LTZ}	$\llbracket \mathsf{x} \rrbracket$	$\llbracket \mathsf{b} \rrbracket : \mathsf{b} = 1(\mathsf{x} \overset{?}{\leq} 0)$	$7 + \frac{35}{4} \log_4 k$	$2n + \frac{2n}{k} + \frac{3n}{2} \log_4 k$	$2 \log_4 k + 4$
Π_{DotP}	$\{\llbracket \mathsf{x}_s \rrbracket, \llbracket \mathsf{y}_s \rrbracket\}_{s \in [\mathsf{N}]}$	$\llbracket \sum_{s \in [\mathsf{N}]} \mathsf{x}_s \cdot \mathsf{y}_s \rrbracket$	3	$2n$	2
Π_{shuffle}	$\llbracket \mathsf{x}_1 \rrbracket, \ldots, \llbracket \mathsf{x}_{\mathsf{N}} \rrbracket$	$\llbracket \mathsf{x}_{\pi(1)} \rrbracket, \ldots, \llbracket \mathsf{x}_{\pi(\mathsf{N})} \rrbracket$	$2\mathsf{N}^2 + 3\mathsf{N}$	$2n\mathsf{N}$	2
Π_{sel}	$\llbracket \mathsf{b} \rrbracket^{\mathsf{B}}, \llbracket \mathsf{x}_0 \rrbracket, \llbracket \mathsf{x}_1 \rrbracket$	$\llbracket \mathsf{x}_{1-\mathsf{b}} \rrbracket$	6	$4n$	4

Note: k is the number of bits to represent an element of $\mathbb{F}$; N is the vector-length in Π_{DotP} and Π_{shuffle}.

Table 2: Summary of the various building blocks that we design for our applications

output reconstruction via a verification phase. We refer to Appendix E for a more detailed exposition.

5.2 Prototype implementation

We now describe a prototype implementation of Asterisk, which we use for benchmarking and comparison.

Environment. Benchmarks are performed over LAN and WAN using n1-highmem-64 instance of Google Cloud. The machine is equipped with a 2.3GHz n1-highmem-64, 64 core, Intel® Xeon® E5-2696V3 processor, with 416GB RAM memory. Each party is run as a process on a single machine, and we use interprocess communication for emulating the actual communication. To simulate the distributed environments, we use the Linux tc command from the network emulation package netem to modify the bandwidth and latency. To simulate the LAN network we consider a bandwidth of 1 Gbps and 0.5 ms of latency, and for the WAN network, we consider a bandwidth of 100 Mbps and 50 ms of latency.

Implementation of Asterisk. We implement Asterisk in C++17 using the codebase of QuadSquad [30] and the EMP toolkit [49]. We use computational security parameter $\kappa = 128$ and statistical security of at least 2^{-40}. We use AES-NI in the CTR mode to realize the PRFs, and use the NTL library [48] for all $\mathbb{Z}_p$ operations (p being a 64-bit prime)[5].

Implementation of other protocols. For comparison with [35] and [36], we rely on publicly available implementations from the widely used MP-SPDZ [34] framework. We use our own implementation for [44], since it does not have a publicly available implementation.

[5]We will open-source our code upon acceptance. Our code is developed for benchmarking and is not optimized for industry-grade use. We believe that incorporating state-of-the-art code optimizations like GPU-assisted computing can further enhance the efficiency of our protocols.

5.3 Performance benchmarks for Asterisk

We evaluate Asterisk on synthetic circuits with a total 1M multiplication gates. We consider circuits of depth $d = 10$, and 100 (with 100000 and 10000 multiplication gates at each level, resp.), as well as varying number of parties n ranging from $n = 5$ to $n = 100$, in both LAN and WAN network settings. The corresponding run times and communication overheads are reported in Table 3. We present additional results for $d = 1000$ (with 100 multiplication gates at each level) in Appendix F, Fig. 24.

Observe that the overheads for the preprocessing phase are independent of d and n. This is precisely as expected (recall that the offline phase of Asterisk is a one-shot circuit-independent message of size 24MB from the HP to P_n) and validates the high practical efficiency of the offline phase. For the online phase, the time taken and communication overheads increase with d and n resp., which is again as expected (since the number of communication rounds increases with d, and each round requires communicating messages of size 8MB between the HP and all n parties). The overall communication, however, only scales with n and the *total number* of gates (and hence, is independent of d).

Comparison with offline phase of [44]. We compare the offline phase of Assisted MPC [44] with that of Asterisk in Table 4 (Fig. 2) over a WAN network for varying number of parties n and for the same synthetic circuit with 10^6 multiplication gates[6] (the online phase of [44] is identical to that of [36], and the corresponding comparison with Asterisk is presented subsequently in(Table 6). Unlike Asterisk, Assisted MPC requires the HP to communicate with all the parties in the offline phase. Hence, the communication cost of Assisted MPC increases linearly with the n, whereas it remains con-

[6]For a fair comparison, we let the helper in [44] be honest because [44] achieves its best efficiency in the presence of an honest helper.

Depth	n	Total comm. (MB)	LAN time (sec)		WAN time (sec)	
			Preproc.	Online	Preproc.	Online
10	5	104.00	2.01	1.70	4.47	20.16
	10	184.00	2.54	3.50	4.95	39.90
	25	424.00	4.12	11.26	6.50	103.29
	50	824.00	7.16	32.03	8.44	214.31
	100	1624.00	12.73	100.66	12.61	466.98
100	5	104.00	1.96	2.03	4.21	55.57
	10	184.00	2.54	4.15	5.01	111.78
	25	424.00	4.04	13.30	6.32	281.83
	50	824.00	6.22	36.84	8.60	574.94
	100	1624.00	11.79	114.10	12.58	1460.14

Table 3: Asterisk's total communication cost and run time over LAN and WAN networks for circuits with 10^6 multiplication gates, for varying depth ($d \in \{10, 100\}$) and varying numbers of parties ($n \in \{5, 10, 25, 50, 100\}$).

# of parties	Assisted MPC [44]		**Asterisk**	
	Comm. (MB)	Time (sec)	Comm. (MB)	Time (sec)
5	161	18.9	24	4.2
10	322	36.4	24	5.0
25	804	90.8	24	6.3
50	1608	178.3	24	8.6
100	3216	361.3	24	12.6

Table 4: Offline phase: Asterisk vs Assisted MPC [44].

Protocol	Communication (KB)	Throughput (/sec $\times 10^3$)
Mascot	8.45	0.19
Overdrive	7.35	0.24
Assisted MPC	0.161	52.91
Asterisk	0.024	222.22

Table 5: Offline phase overheads (per multiplication triple generation for 5 parties): Mascot [35] vs Overdrive [36] vs Assisted MPC [44] vs Asterisk.

stant for Asterisk. This allows Asterisk to save up to 134× and 28× in communication and run time overheads, respectively.

Comparison with offline phase of [35,36]. In Table 5, we compare the offline phases of Asterisk with that of [35,36] (we also include a data-point for Assisted MPC [44] for completeness) in terms of the cost of generating a single multiplication triple coupled with the throughput (number of multiplication triples that can be generated per second). We opt for such a comparison since [35, 36] use function-independent offline phases which generate batches of multiplication triples. As expected, Asterisk provides a much better throughput, where the gain is up to 1168×, 925× and 4×, respectively, in comparison to [35], [36], and [44], respectively.

# of parties	Dishonest Majority		**Asterisk**	
	Comm. (MB)	Time (sec)	Comm. (MB)	Time (sec)
5	256	29.2	80	55.6
10	576	54.0	160	111.8
25	1536	142.0	400	281.8
50	3137	291.9	800	574.9
100	6338	982.4	1600	1460.1

Table 6: Online phase: Asterisk vs [35,36,44].

Comparison with online phase of [35,36,44]. In Table 6, we compare the online phase of Asterisk with the other dishonest majority protocols [35,36,44] (since all of these have identical online phases, we only report the results for [35]) over a WAN for a circuit with 10^6 multiplication gates with depth 100, and for varying numbers of parties n. Asterisk achieves communication savings upto 4×. The relatively poor running time does not follows the asymptotic analysis (see Table 1, where Asterisk is clearly better than [35,36,44]), and is a consequence of the vast gap in terms of optimizations between our implementation (which is purely for benchmarking) and that of [35] in the MP-SPDZ library (which is highly optimized but does not support running Asterisk for a fair comparison due to the inherently function-dependent of its preprocessing framework). For a fair comparison of the running time, one should use a similarly optimized implementation of Asterisk, which we leave as an interesting future work. Finally, we also provide a comparison of the online phase of Asterisk with that of [28] in Appendix F.

5.4 Benchmarks for applications

We now benchmark the two target applications of Asterisk, namely secure auctions and dark pools in the HP-aided setting. We refer to Appendix E and Appendix F for the detailed applications of the building blocks and the overall application protocols that we have benchmarked. Here, we do not compare Asterisk with other MPC protocols, since we expect precisely the same performance savings as in Section 5.3.

Secure auctions. We benchmark the public bidding-based secure auction protocol from Appendix F.1 (which builds upon Asterisk), where the centralized auctioneer acts as the HP and the bidders act as the parties (where each bidder inputs its own bid), and the protocol outputs the highest bid amount and, thereby, the highest bidder. Table 7 shows the performance of our auction protocol for varying numbers of bidders. We observe that our protocol performs auctions with up to 100 bidders within a few minutes (less than 7 mins) over a WAN network, with a communication overhead of 14 MB. Due to the resource-constrained nature of our implementation platform, we were unable to benchmark auctions when the number of bidders is more than 100. This is not a limitation

# of bidders	Preproc.		Online	
	Comm. (KB)	Time (sec)	Comm. (KB)	Time (sec)
5	3.361	0.01	4.88	8.65
10	9.012	0.02	28.72	22.31
25	26.458	0.07	242.4	68.09
50	85.926	0.22	1779.6	161.68
100	303.166	0.79	13649.6	375.14

Table 7: Communication and run time for secure auction where each bidder participates in the computation.

# of bids	Preproc.		Online	
	Comm. (MB)	Time (sec)	Comm. (MB)	Time (sec)
100	0.30	0.16	0.68	19.54
250	1.13	0.34	2.67	22.58
500	4.36	1.41	10.59	27.12

Table 8: Communication and run time for secure auction running with $n = 5$ for a larger number of bids.

of our protocol (which works generally for any n), but a limitation of our experimental setup, which could not support more than 100 parallel processes simulating the bidders in the protocol. Hence, to capture the scenario where number of bidders are more than 100, we also consider benchmarking auctions in the outsourced setting, where we assume that a set of 5 servers are hired to carry out the MPC protocol for secure auction on the bids that are secret-shared among them by the bidders. The results of performing auctions when the number of bidders varies between 100 to 500 appears in Table 8. As can be observed, in the outsourced setting, secure auction in the presence of up to 500 bidders takes roughly 27 seconds in the online phase.

Secure dark pools. We now benchmark the secure dark pool protocol described in Appendix F.2, which builds upon Asterisk and uses a HP-based centralized dark pool operator that is responsible for matching the submitted requests from the participants in the MPC protocol. We consider two popular algorithms—continuous double auctions (CDA) and volume matching (VM)—that are used for matching buy requests with appropriate sell requests in dark pools. As in the case of secure auctions, we could not scale to more than $n = 100$ parties due to the constraints of our implementation platform. Hence, we choose to focus on the outsourced setting with $n = 5$ and showcase the scalability of the protocol to larger-sized lists.

Table 9 demonstrates the performance of our dark pool protocol. Since the performance of the dark pool algorithm varies with the number of buy requests (stored in a buy list) and sell requests (stored in a sell list) already present in the system, we vary the number of these requests to analyze the performance of Asterisk when executing these algorithms. For processing an incoming trade request, Asterisk executes

$N = M$	Algorithm	Preproc.		Online		Throughput (/min)
		Comm. (KB)	Time (sec)	Comm. (KB)	Time (sec)	
50	CDA	53.36	0.16	60.92	8.26	7.26
	VM	61.69	0.22	41.2	7.18	835.94
100	CDA	114.12	0.32	120.92	8.28	7.25
	VM	122.97	0.43	81.2	7.26	1653.76
250	CDA	284.39	0.80	300.92	8.85	6.78
	VM	306.79	1.11	201.2	8.07	3719.61
500	CDA	568.17	1.60	600.92	9.83	6.10
	VM	613.17	2.13	401.2	9.66	6210.67

Table 9: Communication, run time, and throughput (per minute) for dark pool CDA and VM algorithms for $n = 5$ and varying buy list size (N), and sell list size (M).

the dark pool algorithms within a few seconds (up to 12 seconds), even when the buy list size (N) and sell list size (M) is as large as 500. The total amount of communication is less than 1.5 MB. We also report the throughput, which captures the number of incoming orders that can be processed in a minute. The throughput is computed [12] as throughput (/min) = 60/online run time, and $(M + N) * 60$/online run time for CDA and VM, respectively.

6 Conclusion and open questions

This work studies a new model, HP-aided MPC, which considers an additional semi-honest party. The presence of this semi-honest party helps to attain a security guarantee (fairness) which is known to be impossible, in general, for a standard dishonest majority setting. In this model, we build a framework, Asterisk, that achieves the following: (a) fairness security guarantee, (b) much better efficiency in comparison to dishonest majority protocols, (c) comparable efficiency with honest majority protocols, (d) scalability for a large number of parties. Moreover, the existence of the HP is natural in various applications, such as auctioneer in the auctions, and Securities and Exchange Commission in the dark pool, which makes the model practically relevant.

Our work gives rise to many interesting open questions. It is worthwhile to check if one can extend the security of Asterisk from fairness to GOD *while preserving privacy against the semi-honest* HP. Note that if we allow the HP to learn the output, then such an extension is immediate: if the protocol aborts (implying at least one malicious corruption in $\mathcal{P}$), the HP must be honest (this follows immediately from our model), and can be used by the parties as a trusted third party (TTP) to compute the output. It is interesting to explore alternative practically motivated models that would allow for circumventing and improving the de facto security and efficiency bottleneck of dishonest majority MPC. One can explore the use of Asterisk for building additional privacy-preserving applications (e.g., federated cloud computing, where the assumption of a semi-honest HP is naturally justified).

References

[1] Bar Alon, Eran Omri, and Anat Paskin-Cherniavsky. MPC with friends and foes. In *CRYPTO*, 2020.

[2] Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, and Or Weinstein. Optimized honest-majority mpc for malicious adversaries—breaking the 1 billion-gate per second barrier. In *IEEE S&P*, 2017.

[3] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In *ACM CCS*, 2016.

[4] Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin, and Hikaru Tsuchida. Secure graph analysis at scale. In *ACM CCS*, 2021.

[5] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. Optorama: Optimal oblivious ram. In *ASIACRYPT*, 2020.

[6] Saikrishna Badrinarayanan, Abhishek Jain, Rafail Ostrovsky, and Ivan Visconti. Non-interactive secure computation from one-way functions. In *ASIACRYPT*, 2018.

[7] Aner Ben-Efraim, Michael Nielsen, and Eran Omri. Turbospeedz: Double your online spdz! improving spdz using function dependent preprocessing. In *ACNS*, 2019.

[8] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear pcps. In *CRYPTO*, 2019.

[9] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Efficient fully secure computation via distributed zero-knowledge proofs. In *ASIACRYPT*, 2020.

[10] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, 2001.

[11] Ran Canetti and Marc Fischlin. Universally composable commitments. In *CRYPTO*, 2001.

[12] John Cartlidge, Nigel P Smart, and Younes Talibi Alaoui. Mpc joins the dark side. In *ACM ASIACCS*, 2019.

[13] Octavian Catrina and Sebastiaan De Hoogh. Improved primitives for secure multiparty integer computation. In *Security and Cryptography for Networks*, 2010.

[14] T-H Hubert Chan, Jonathan Katz, Kartik Nayak, Antigoni Polychroniadou, and Elaine Shi. More is less: Perfectly secure oblivious algorithms in the multi-server setting. In *ASIACRYPT*, 2018.

[15] Nishanth Chandran, Wutichai Chongchitmate, Rafail Ostrovsky, and Ivan Visconti. Universally composable secure computation with corrupted tokens. In *CRYPTO*, pages 432–461, 2019.

[16] Nishanth Chandran, Vipul Goyal, and Amit Sahai. New constructions for uc secure computation using tamper-proof hardware. In *EUROCRYPT*, 2008.

[17] Seung Geol Choi, Jonathan Katz, Dominique Schröder, Arkady Yerukhimovich, and Hong-Sheng Zhou. (efficient) universally composable oblivious transfer using a minimal number of stateless tokens. In *TCC*, 2014.

[18] Richard Cleve. Limits on the security of coin flips when half the processors are faulty. In *ACM STOC*, 1986.

[19] Geoffroy Couteau. New protocols for secure equality test and comparison. In *ACNS*, 2018.

[20] Dana Dachman-Soled, Tal Malkin, Mariana Raykova, and Muthuramakrishnan Venkitasubramaniam. Adaptive and concurrent secure computation from new adaptive, non-malleable commitments. In *ASIACRYPT*, 2013.

[21] Anders Dalskov, Daniel Escudero, and Marcel Keller. Fantastic four: Honest-majority four-party secure computation with malicious security. In *USENIX Security*, 2020.

[22] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P Smart. Practical covertly secure mpc for dishonest majority–or: breaking the spdz limits. In *ESORICS*, 2013.

[23] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In *CRYPTO*, 2007.

[24] Saba Eskandarian and Dan Boneh. Clarion: Anonymous communication from multiparty shuffling protocols. In *NDSS*, 2022.

[25] Matthias Fitzi, Juan A. Garay, Ueli M. Maurer, and Rafail Ostrovsky. Minimal complete primitives for secure multi-party computation. In *CRYPTO*, 2001.

[26] S. Dov Gordon, Yuval Ishai, Tal Moran, Rafail Ostrovsky, and Amit Sahai. On complete primitives for fairness. In *TCC*, 2010.

[27] S. Dov Gordon, Yuval Ishai, Tal Moran, Rafail Ostrovsky, and Amit Sahai. On complete primitives for fairness. In *TCC*, 2010.

[28] Vipul Goyal, Hanjun Li, Rafail Ostrovsky, Antigoni Polychroniadou, and Yifan Song. Atlas: Efficient and scalable mpc in the honest majority setting. In *CRYPTO*, 2021.

[29] Carmit Hazay, Antigoni Polychroniadou, and Muthuramakrishnan Venkitasubramaniam. Composable security in the tamper-proof hardware model under minimal complexity. In *TCC*, 2016.

[30] Aditya Hegde, Nishat Koti, Varsha Bhat Kukkala, Shravani Patil, Arpita Patra, and Protik Paul. Attaining god beyond honest majority with friends and foes. In *ASIACRYPT*, 2022.

[31] Yuval Ishai, Rafail Ostrovsky, and Hakan Seyalioglu. Identifying cheaters without an honest majority. In *TCC*, 2012.

[32] Yuval Ishai, Arpita Patra, Sikhar Patranabis, Divya Ravi, and Akshayaram Srinivasan. Fully-secure MPC with minimal trust. In *TCC*, 2022.

[33] Jonathan Katz. Universally composable multi-party computation using tamper-proof hardware. In *EUROCRYPT*, 2007.

[34] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In *ACM CCS*, 2020.

[35] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Mascot: faster malicious arithmetic secure computation with oblivious transfer. In *ACM CCS*, 2016.

[36] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: making spdz great again. In *EUROCRYPT*, 2018.

[37] Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal, et al. Find thy neighbourhood: Privacy-preserving local clustering. *PETS*, 2023.

[38] Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, and Bhavish Raj Gopal. Pentagod: Stepping beyond traditional god with five parties. In *ACM CCS*, 2022.

[39] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. Swift: Super-fast and robust privacy-preserving machine learning. In *USENIX Security*, 2021.

[40] Nishat Koti, Shravani Patil, Arpita Patra, and Ajith Suresh. Mpclan: Protocol suite for privacy-conscious computations. *Journal of Cryptology*, 2023.

[41] Nishat Koti, Arpita Patra, Rahul Rachuri, and Ajith Suresh. Tetrad: Actively Secure 4PC for Secure Training and Inference. In *NDSS*, 2022.

[42] Donghang Lu and Aniket Kate. Rpm: Robust anonymity at scale. *Cryptology ePrint Archive*, 2022.

[43] Nancy A Lynch. *Distributed algorithms*. Elsevier, 1996.

[44] Philipp Muth and Stefan Katzenbeisser. Assisted mpc. *Cryptology ePrint Archive*, 2022.

[45] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. Graphsc: Parallel secure computation made easy. In *IEEE S&P*, 2015.

[46] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. Aby2. 0: Improved mixed-protocol secure two-party computation. In *USENIX Security*, 2021.

[47] A Pranav Shriram, Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal, and Somya Sangal. Ruffle: Rapid 3-party shuffle protocols. *PETS*, 2023.

[48] Victor Shoup. NTL: A Library for doing Number Theory. https://libntl.org/, 2021.

[49] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

[50] Andrew C Yao. Protocols for secure computations. In *FOCS*, 1982.

A Security model

The real world. An n-party protocol Π with n parties $\mathcal{P} = \{P_1, \ldots, P_n\}$ is an n-tuple of probabilistic polynomial-time (PPT) interactive Turing machines (ITMs), where each party P_i is initialized with input $x_i \in \{0,1\}^*$ and random coins $r_i \in \{0,1\}^*$. These parties interact in synchronous rounds. In every round parties can communicate either over a broadcast channel or a fully connected point-to-point (P2P) network, where we additionally assume all communication to be private and ideally authenticated. Further, we assume that there exists a special party HP called a "trusted party" (abbreviated henceforth as HP) such that each party P_i can interact with HP via private and authenticated point-to-point channels. The HP does not typically hold any inputs, and also does not obtain any output at the end of the protocol. Depending on whether we allow the HP to keep any state in between its invocations (where an invocation corresponds to all parties interacting with the HP in a single round) or not, we refer to the HP as being 'stateful' or 'stateless' respectively.

We let $\mathcal{A}$ denote a special PPT ITM that represents the adversary and that is initialized with input that contains the identities of the corrupt parties, their respective private inputs, and an auxiliary input. During the execution of the protocol, the maliciously corrupt parties (sometimes referred to as 'active') receive arbitrary instructions from the adversary $\mathcal{A}$, while the honest parties and the semi-honestly corrupt (sometimes referred to as 'passive') parties faithfully follow the instructions of the protocol. We consider the adversary $\mathcal{A}$ to be rushing, i.e., during every round the adversary can see the messages the honest parties sent before producing messages from actively corrupt parties.

At the end of the protocol execution, the honest parties produce output, the corrupt parties produce no output, and the adversary outputs an arbitrary function of its view. The view of a party during the execution consists of its input, random coins and the messages it sees during the execution.

Definition A.1 (Real-world execution)**.** Let Π be an n-party protocol amongst $\{P_1,\ldots,P_n\}$ computing an n-party function $f : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$ and let $\mathcal{C} \subset [1,n] \cup \{\mathrm{HP}\}$ denote the set of indices of the corrupted parties. The execution of Π under $(\mathcal{A},\mathcal{C})$ in the real world, on input vector $\vec{x} = (x_1,\ldots,x_n)$, auxiliary input aux and security parameter κ, denoted $\mathsf{real}_{\Pi,\mathcal{C},\mathcal{A}(\mathsf{aux})}(\vec{x},\kappa)$, is defined as the output vector of $P_1,\ldots,P_n$ and $\mathcal{A}(\mathsf{aux})$ resulting from the protocol interaction.

The ideal world. We describe ideal world executions with unanimous abort ($\mathsf{un\text{-}abort}$), identifiable abort ($\mathsf{id\text{-}abort}$), fairness ($\mathsf{fairness}$) and full security aka. guaranteed output delivery (full).

Definition A.2 (Ideal Computation)**.** Consider $\mathsf{type} \in \{\mathsf{un\text{-}abort},\mathsf{id\text{-}abort},\mathsf{fairness},\mathsf{full}\}$. Let $f : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$ be an n-party function and let $\mathcal{C} \subset [1,n] \cup \{\mathrm{HP}\}$ denotes the set of all corrupt parties. Then, the joint ideal execution of f under $(\mathcal{S},\mathcal{C})$ on input vector $\vec{x} = (x_1,\ldots,x_n)$, auxiliary input aux to $\mathcal{S}$ and security parameter κ, denoted $\mathsf{ideal}^{\mathsf{type}}_{f,\mathcal{C},\mathcal{S},(\mathsf{aux})}(\vec{x},\kappa)$, is defined as the output vector of $P_1,\ldots,P_n$ and $\mathcal{S}$ resulting from the following ideal process.

1. *Parties send inputs to trusted party*: An honest party P_i sends its input x_i to the trusted party. The simulator $\mathcal{S}$ may send to the trusted party arbitrary inputs for the actively corrupt parties. Let x'_i be the value actually sent as the input of party P_i.

2. *Trusted party speaks to simulator*: The trusted party computes $(y_1,\ldots,y_n) = f(x'_1,\ldots,x'_n)$. If there are no corrupt parties or $\mathsf{type} = \mathsf{full}$, proceed to step 4.

   (a) *If* $\mathsf{type} \in \{\mathsf{un\text{-}abort},\mathsf{id\text{-}abort}\}$*:* The trusted party sends $\{y_i\}_{i\in\mathcal{C}}$ to $\mathcal{S}$.

   (b) *If* $\mathsf{type} = \mathsf{fairness}$*:* The trusted party sends `ready` to $\mathcal{S}$.

3. *Simulator $\mathcal{S}$ responds to trusted party*:

   (a) *If* $\mathsf{type} \in \{\mathsf{un\text{-}abort},\mathsf{fairness}\}$*:* The simulator can send abort to the trusted party.

   (b) *If* $\mathsf{type} = \mathsf{id\text{-}abort}$*:* If it chooses to abort, the simulator $\mathcal{S}$ can select an actively corrupt party $i^* \in \mathcal{C}$ who will be blamed, and send (abort, i^*) to the trusted party.

4. *Trusted party answers parties*:

   (a) If the trusted party got abort from the simulator $\mathcal{S}$,

      i. It sets the abort message $\mathsf{abortmsg}$, as follows:

      - if $\mathsf{type} \in \{\mathsf{un\text{-}abort},\mathsf{fairness}\}$, we let $\mathsf{abortmsg} = \perp$.
      - if $\mathsf{type} = \mathsf{id\text{-}abort}$, we let $\mathsf{abortmsg} = (\perp, i^*)$.

      ii. The trusted party then sends $\mathsf{abortmsg}$ to every party P_j, $j \in [n] \setminus \mathcal{C}$.

      Note that, if $\mathsf{type} = \mathsf{full}$, we will never be in this setting, since $\mathcal{S}$ was not allowed to ask for an abort.

   (b) Otherwise, it sends y_j to every P_j, $j \in [1,n]$.

5. *Outputs*: Honest parties always output the message received from the trusted party while the corrupt parties output nothing. The simulator $\mathcal{S}$ outputs an arbitrary function of the initial inputs $\{x_i\}_{i\in\mathcal{C}}$, the messages received by the corrupt parties from the trusted party and its auxiliary input.

In this work, we primarily consider the non-colluding security notion of [32], which is described below.

Non-colluding security. Informally, protocols with this notion of security are secure against a non-colluding adversary that corrupts *either* any subset of the n parties $\{P_1,\ldots,P_n\}$ maliciously *or* the HP passively (i.e. in a semi-honest manner). The formal definition appears below.

Definition A.3 (Non-colluding security)**.** Consider $\mathsf{type} \in \{\mathsf{un\text{-}abort},\mathsf{id\text{-}abort},\mathsf{fairness},\mathsf{full}\}$. Let $f : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$ be an n-party function. A protocol Π securely computes the function f in the *non-colluding model* with type security if for any adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that for every $\mathcal{C}$ where either (a) $\mathcal{C} \subset [1,n]$ malicious corruptions or (b) $\mathcal{C} = \mathrm{HP}$ semi-honest corruption, we have

$$\left\{\mathsf{real}_{\Pi,\mathcal{C},\mathcal{A}(\mathsf{aux})}(\vec{x},\kappa)\right\}_{\vec{x}\in(\{0,1\}^*)^n,\kappa\in\mathbb{N}} \equiv \left\{\mathsf{ideal}^{\mathsf{type}}_{f,\mathcal{C},\mathcal{S}(\mathsf{aux})}(\vec{x},\kappa)\right\}_{\vec{x}\in(\{0,1\}^*)^n,\kappa\in\mathbb{N}}.$$

Note that the corruption is non-simultaneous. Therefore we need the above indistinguishability to hold in both the corruption cases.

A protocol achieves computational security, if the above distributions are computationally close in the presence of the parties, $\mathcal{A},\mathcal{S}$ that are PPT. A protocol achieves statistical (resp. perfect) security if the distributions are statistically close (resp. identical).

A.1 Alternative corruption models

As mentioned above, we consider a non-colluding adversary who either corrupts up to $n-1$ among the n parties maliciously or the HP in a semi-honest manner. One might also consider the following alternative corruption models.

A.1.1 Collluding security.

Informally, in this corruption model, the adversary can *simultaneously* corrupt the majority of the parties maliciously as well as the HP in a semi-honest manner. More formally, security is defined similar to non-colluding security (defined above), except that the indistinguishability must hold for every $\mathcal{C} \subset [1,n] \cup \{\text{HP}\}$, where the corruptions in $[1,n]$ are malicious and corruption of HP is semi-honest.

A.1.2 Friends and foes (FaF) security with HP.

Alon et. al. [1] proposed the security notion of MPC with Friends and Foes (FaF). This notion requires the inputs of the honest parties to be protected against not only the adversary (foes), but also from quorums of other honest parties (friends). This is modelled by a decentralized adversary which comprises two different *non-colluding* adversaries—(i) a malicious adversary that corrupts any subset of at most t out of n parties, and (ii) a semi-honest adversary that corrupts any subset of at most $h^\star$ out of the remaining $n-t$ parties. A protocol secure against such adversaries is said to be $(t,h^\star)$-FaF secure. Further, the FaF model requires security to hold even when an adversary sends its view to other parties.

When proving security in the FaF model, there is the additional requirement of simulating the view of any subset of uncorrupted (or semi-honest) parties, in addition to simulating the view of the maliciously corrupt parties. This necessitates the use of two simulators in the ideal world– one for the malicious adversary and one for the semi-honest adversary. Further, the malicious adversary is allowed to send its entire view to the semi-honest adversary in the ideal world to capture the behaviour where the malicious adversary may send non-protocol messages to uncorrupted parties in the real world. Elaborately, let $\mathcal{A}$ denote the probabilistic polynomial time (PPT) malicious adversary in the real-world corrupting t parties in $I \subset \mathcal{P}$, and $\mathcal{S}_\mathcal{A}$ denote the corresponding ideal-world simulator. Similarly, let $\mathcal{A}_\mathcal{H}$ denote the (PPT) semi-honest adversary corrupting $h^\star$ parties in $\mathcal{H} \subset \mathcal{P} \setminus I$ in the real-world, and $\mathcal{S}_{\mathcal{A}_\mathcal{H}}$, be the ideal-world simulator. Note that in the classical definition of ideal-world, $\mathcal{H} = \phi$. Let $\mathcal{F}$ be the ideal-world functionality. Let $\mathsf{VIEW}^{\mathsf{REAL}}_{\mathcal{A},\Pi}(1^\lambda, z_\mathcal{A})$ be the malicious adversary's $(\mathcal{A})$ view and $\mathsf{OUT}^{\mathsf{REAL}}_{\mathcal{A},\Pi}(1^\lambda, z_\mathcal{A})$ denote the output of the uncorrupted parties (in $\mathcal{P} \setminus I$) during a random execution of Π, where $z_\mathcal{A}$ is the auxiliary input of $\mathcal{A}$. Similarly, let $\mathsf{VIEW}^{\mathsf{REAL}}_{\mathcal{A},\mathcal{A}_\mathcal{H},\Pi}(1^\lambda, z_\mathcal{A}, z_{\mathcal{A}_\mathcal{H}})$ be the semi-honest adversary's $(\mathcal{A}_\mathcal{H})$ view during an execution of Π running alongside $\mathcal{A}$, where $z_{\mathcal{A}_\mathcal{H}}$ is the auxiliary input of $\mathcal{A}_\mathcal{H}$. Note that $\mathsf{VIEW}^{\mathsf{REAL}}_{\mathcal{A},\mathcal{A}_\mathcal{H},\Pi}(1^\lambda, z_\mathcal{A}, z_{\mathcal{A}_\mathcal{H}})$ consists of the non-prescribed messages sent by the malicious adversary to the semi-honest parties. Correspondingly, let $\mathsf{VIEW}^{\mathsf{IDEAL}}_{\mathcal{S}_\mathcal{A},\mathcal{F}}(1^\lambda, z_\mathcal{A})$ be the malicious adversary's simulated view with $\mathcal{S}_\mathcal{A}$ corrupting parties in I and $\mathsf{OUT}^{\mathsf{IDEAL}}_{\mathcal{S}_\mathcal{A},\mathcal{F}}(1^\lambda, z_\mathcal{A})$ denote the output of the uncorrupted parties (in $\mathcal{P} \setminus I$) during a random execution of ideal-world functionality $\mathcal{F}$. Similarly, let $\mathsf{VIEW}^{\mathsf{IDEAL}}_{\mathcal{S}_\mathcal{A},\mathcal{S}_{\mathcal{A}_\mathcal{H}},\mathcal{F}}(1^\lambda, z_\mathcal{A}, z_{\mathcal{A}_\mathcal{H}})$ be the semi-honest adversary's simulated view with $\mathcal{S}_{\mathcal{A}_\mathcal{H}}$ corrupting parties in $\mathcal{H}$ during an execution of $\mathcal{F}$ running alongside $\mathcal{A}$. A protocol Π is said to compute $\mathcal{F}$ with (weak) computational $(t,h^\star)$-FaF-security if

$$\left(\mathsf{VIEW}^{\mathsf{IDEAL}}_{\mathcal{S}_\mathcal{A},\mathcal{F}}(1^\lambda, z_\mathcal{A}), \mathsf{OUT}^{\mathsf{IDEAL}}_{\mathcal{S}_\mathcal{A},\mathcal{F}}(1^\lambda, z_\mathcal{A})\right) \equiv \left(\mathsf{VIEW}^{\mathsf{REAL}}_{\mathcal{A},\Pi}(1^\lambda, z_\mathcal{A}), \mathsf{OUT}^{\mathsf{REAL}}_{\mathcal{A},\Pi}(1^\lambda, z_\mathcal{A})\right),$$

$$\left(\mathsf{VIEW}^{\mathsf{IDEAL}}_{\mathcal{S}_\mathcal{A},\mathcal{S}_{\mathcal{A}_\mathcal{H}},\mathcal{F}}(1^\lambda, z_\mathcal{A}, z_{\mathcal{A}_\mathcal{H}}), \mathsf{OUT}^{\mathsf{IDEAL}}_{\mathcal{S}_\mathcal{A},\mathcal{F}}(1^\lambda, z_\mathcal{A})\right) \equiv \left(\mathsf{VIEW}^{\mathsf{REAL}}_{\mathcal{A},\mathcal{A}_\mathcal{H},\Pi}(1^\lambda, z_\mathcal{A}, z_{\mathcal{A}_\mathcal{H}}), \mathsf{OUT}^{\mathsf{REAL}}_{\mathcal{A},\Pi}(1^\lambda, z_\mathcal{A})\right).$$

With the above background, we move on to customize FaF-security to our scenario in the presence of a HP. Unlike the FaF-security model, where security is required not only against t maliciously corrupt parties, but also against *any* subset of $h^\star$ semi-honest parties, FaF-security with HP is a special case. Here, we require semi-honest security to be provided only against the single party designated as the HP. Elaborately, as defined in Section A.1.2, let $\mathcal{A}$ be an adversary corrupting at most $n-1$ parties maliciously, and let $\mathcal{A}_{\text{HP}}$ be a semi-honest adversary that corrupts the HP. We say a protocol, Π, is FaF-secure with HP if

$$\left(\mathsf{VIEW}^{\mathsf{IDEAL}}_{\mathcal{A},\mathcal{F}}(1^\lambda, z_\mathcal{A}), \mathsf{OUT}^{\mathsf{IDEAL}}_{\mathcal{A},\mathcal{F}}(1^\lambda, z_\mathcal{A})\right) \equiv \left(\mathsf{VIEW}^{\mathsf{REAL}}_{\mathcal{A},\Pi}(1^\lambda, z_\mathcal{A}), \mathsf{OUT}^{\mathsf{REAL}}_{\mathcal{A},\Pi}(1^\lambda, z_\mathcal{A})\right),$$

$$\left(\mathsf{VIEW}^{\mathsf{IDEAL}}_{\mathcal{A},\mathcal{A}_{\text{HP}},\mathcal{F}}(1^\lambda, z_\mathcal{A}, z_{\mathcal{A}_{\text{HP}}}), \mathsf{OUT}^{\mathsf{IDEAL}}_{\mathcal{A},\mathcal{F}}(1^\lambda, z_\mathcal{A})\right) \equiv \left(\mathsf{VIEW}^{\mathsf{REAL}}_{\mathcal{A},\mathcal{A}_{\text{HP}},\Pi}(1^\lambda, z_\mathcal{A}, z_{\mathcal{A}_{\text{HP}}}), \mathsf{OUT}^{\mathsf{REAL}}_{\mathcal{A},\Pi}(1^\lambda, z_\mathcal{A})\right).$$

Here, $\mathsf{VIEW}^{\mathsf{IDEAL}}_{\mathcal{A},\mathcal{A}_{\text{HP}},\mathcal{F}}(1^\lambda, z_\mathcal{A}, z_{\mathcal{A}_{\text{HP}}})$ is $\mathcal{A}_{\text{HP}}$'s simulated view during an execution of $\mathcal{F}$ alongside $\mathcal{A}$, and $\mathsf{VIEW}^{\mathsf{REAL}}_{\mathcal{A},\mathcal{A}_{\text{HP}},\Pi}(1^\lambda, z_\mathcal{A}, z_{\mathcal{A}_{\text{HP}}})$ consists of non-prescribed messages sent by $\mathcal{A}$ to $\mathcal{A}_{\text{HP}}$.

B Proof of Theorem 3.1

We present the proof for a two-party functionality. The argument can be extended for any number of parties using the player-partitioning argument. We refer to the two parties as A, B.

Lemma B.1. *Assume that one-way permutation exists. Then there exists a* 2*-party functionality that no protocol computes with fairness in the FaF security model with* HP *(refer to precise the definition given in Section A.1.2).*

Proof of Lemma B.1 Let $f = \{f_\kappa : \{0,1\}^\kappa \to \{0,1\}^\kappa\}_{\kappa\in\mathbb{N}}$ be a one-way permutation. Define the symmetric 2-party functionality $\mathsf{Swap} = \{\mathsf{Swap}_\kappa : \{0,1\}^{2\kappa} \times \{0,1\}^{2\kappa} \to \{0,1\}^{2\kappa}\}_{\kappa\in\mathbb{N}}$ as follows. Parties A and B each hold string (a, y_B) and (b, y_A) respectively. The output is then defined to be

$$\mathsf{Swap}_\kappa((a,y_B),(b,y_A)) = \begin{cases} (a,b) \text{ if } f_\kappa(a) = y_A \text{ and } f_\kappa(b) = y_B \\ 0^\kappa \text{ otherwise} \end{cases}$$

For the sake of contradiction, suppose a protocol Π securely computes the Swap functionality with fairness FaF security with HP model. We fix a security parameter κ and let r denote the number of rounds in Π. Consider an evaluation of Swap with the output being (a,b). Formally, we consider the following distribution over the inputs.

- a,b are each selected from $\{0,1\}^\kappa$ uniformly at random and independently.
- $y_A = f_\kappa(a)$ and $y_B = f_\kappa(b)$.

For $i \in \{0,\ldots,r\}$ let a_i be the final output of A assuming that B aborted after sending ith round messages. Similarly, for $i \in \{0,\ldots,r\}$ we define b_i to be the final output of B assuming that A aborted after sending ith round messages. Observe that a_r (b_r) is the output of A (B) when B (A) sends messages in all the rounds. We first claim that there exists a round where either A and HP jointly or B and HP jointly, gain an advantage in computing the correct output.

Claim B.2. Either there exists $i \in \{0,\ldots,r\}$ such that

$$\Pr[a_i = (a,b)] - \Pr[b_i = (a,b)] \geq \frac{1 - neg(\kappa)}{2r+1},$$

or there exists $i \in [r]$ such that

$$\Pr[b_i = (a,b)] - \Pr[a_{i-1} = (a,b)] \geq \frac{1 - neg(\kappa)}{2r+1}.$$

The probabilities above are taken over the choice of inputs and of random coins for the parties.

Proof. The proof follows by the following averaging argument. By correctness and the fact that f_κ is one-way, it follows that

$$\begin{aligned} &1 - neg(\kappa) \\ \leq &\Pr[a_r = (a,b)] - Pr[b_0 = (a,b)] \\ = &\sum_{i=0}^{r} \left(\Pr[a_i = (a,b)] - \Pr[b_i = (a,b)]\right) \\ + &\sum_{i=1}^{r} \left(\Pr[b_i = (a,b)] - \Pr[a_{i-1} = (a,b)]\right) \end{aligned}$$

Since there are $2r+1$ summands, there must exist an i for which one of the differences is at least $\frac{1-neg(\kappa)}{2r+1}$. ☐

Assume without loss of generality that there exists an $i \in [1,r]$ such that the former equality in Claim B.2 holds (the other case is done analogously). Define a malicious adversary $\mathcal{A}$ as follows. For the security parameter κ, it receives the round i as auxiliary input. Now, $\mathcal{A}$ corrupts A and makes it act honestly (using the party's original input a) up to and including round i. After receiving the i-th message, the adversary instructs A to abort. Finally, the adversary sends its entire view to HP. Note that this is allowed as per the FaF security notion as nothing stops a malicious adversary from sending its view to the semi-honest HP in the real-world.

Next, we show that no pair of simulators $\mathcal{S}_\mathcal{A}$ and $\mathcal{S}_{\mathcal{A},\mathrm{HP}}$ can produce views for A and HP so that the ideal view and the real view are indistinguishable. For that, we assume towards contradiction that such simulators do exist. Let $(a^*, y_B^*) \in \{0,1\}^{2\kappa}$ be the input that $\mathcal{S}_\mathcal{A}$ sent to the ideal functionality. Additionally, denote $q = \Pr[b_i = (a,b)]$.

Now depending upon $\Pr[(a^*,y_B^*) = (a,y_B)]$, two scenarios are possible. We will analyze them case by case.

Case I : $\Pr[(a^*,y_B^*) = (a,y_B)] \geq q + \frac{1}{p(\kappa)}$ for some polynomial $p(\cdot)$ for infinitely many κ's. Let $\mathsf{OUT}_B^{\mathsf{IDEAL}}$ be the output of B in the ideal world. Since f_κ is a permutation we have that

$$\Pr[\mathsf{OUT}_B^{\mathsf{IDEAL}} = (a,b)] = \Pr[(a^*,y_B^*) = (a,y_B)] \geq q + \frac{1}{p(\kappa)}$$

Thus, by comparing the output of B to (a,b) it is possible to distinguish the real from the ideal with advantage at least $1/p(\kappa)$.

Case II : $\Pr[(a^*,y_B^*) = (a,y_B)] \leq q + neg(\kappa)$. Here we show how to distinguish between the view of HP in the real world from its ideal world counterpart. Recall that in the real world $\mathcal{A}$ sent its view to HP. Let M be the algorithm specified by the protocol, that A and HP use to compute their output assuming B has aborted. Namely, M outputs a_i in the real world. By Claim B.2 it holds that $\Pr[a_i = (a,b)] \geq q + \frac{1-neg(\kappa)}{2r+1}$. We next consider the ideal world. Let V be the view generated by $\mathcal{S}_{\mathcal{A},\mathrm{HP}}$. We claim that

$$\Pr[M(V) = (a,b) \wedge (a^*,y_B^*) \neq (a,y_B)] \leq neg(\kappa).$$

Note that

$$\Pr[M(V) = (a,b) \wedge a^* \neq a] \leq neg(\kappa).$$

Since f_κ is a permutation and B does not change the input it sends to the ideal functionality, (i.e. B sends its input (b,y_A)) the output computed by the ideal functionality will be 0^κ. Moreover, as f_κ is one-way, it follows that if $M(V)$ did output (a,b), then it can be used to break the security of f_κ. At a high level, this is because M has computed $f_\kappa^{-1}(y_B)$. Elaborately, this can be done by sampling $a \leftarrow \{0,1\}^\kappa$, computing $f_\kappa(a)$, and finally, computing a view V using the simulators and applying M to it (if a^* computed by $\mathcal{S}_\mathcal{A}$ equals to a then abort).

On the other hand,

$$\Pr[M(V) = (a,b) \wedge y_B^* \neq y_B] \leq neg(\kappa).$$

Since B does not change its input b to the ideal functionality, thus $f_\kappa(b) = y_B \neq y_B^*$, due to the well-definedness of the function f_κ. Therefore, the output computed by the ideal functionality will be 0^κ.

Hence,

$$\begin{aligned}&\Pr[M(V)=(a,b)\wedge(a^*,y_B^*)\neq(a,y_B)]\\ \leq&\Pr[M(V)=(a,b)\wedge a^*\neq a]+\Pr[M(V)=(a,b)\wedge y_B^*\neq y_B].\end{aligned}$$

Thus,

$$\Pr[M(V)=(a,b)\wedge(a^*,y_B^*)\neq(a,y_B)]\leq neg(\kappa)$$

Hence, we conclude that

$$\begin{aligned}&\Pr[M(V)=(a,b)]\\ =&\Pr[M(V)=(a,b)\wedge(a^*,y_B^*)=(a,y_B)]\\ &+\Pr[M(V)=(a,b)\wedge(a^*,y_B^*)\neq(a,y_B)]\\ \leq&\Pr[(a^*,y_B^*)=(a,y_B)]+neg(\kappa)\leq q+neg(\kappa).\end{aligned}$$

Therefore, by applying M to the view it is possible to distinguish with advantage at least $\frac{1-neg(\kappa)}{2r+1}-neg(\kappa)$.

C Asterisk

C.1 Shared key setup

Here we provide details about how $\mathcal{F}_{\mathsf{Setup}}$ (Fig. 3) can be instantiated. Note that $\{\mathsf{k}_i\}_{i\in[1,n]}$ and $\mathsf{k}_{\mathsf{all}}$ can be sampled by HP and sent to the corresponding parties. However, generating $\mathsf{k}_{\mathcal{P}}$ is slightly challenging since this key should be generated while ensuring that all parties agree on the key despite misbehaviour of corrupt parties. We rely on the following abort secure protocol for generating this key. To generate $\mathsf{k}_{\mathcal{P}}$, each party sends a randomly sampled value to all other parties. Upon receiving values from all parties, each party takes the sum of the received value and sets that as the common key. To verify that all parties have agreed upon a common key, they rely on the HP as follows. All parties evaluate the PRF on a common counter (ctr) value using the common key and send the output to HP. If all the received values at the HP match, then HP sends continue to all the parties; else, it sends abort. If parties receive abort, they terminate the protocol. Otherwise, the protocol continues with $\mathsf{k}_{\mathcal{P}}$ as the common key among the parties. Note that, though the above-mentioned procedure provides only abort security, while running it inside Asterisk it does not impact the fairness guarantee of Asterisk. This is because aborting during key setup which is an input independent phase does not allow the adversary to learn any information about the output.

C.2 Security proof

The formal security proof of Asterisk is provided here.

Simulator $\mathcal{S}_{\mathcal{A}}$

Malicious Let $\mathcal{A}$ be a malicious adversary corrupting up to $n-1$ parties among the computing parties $\mathcal{P}=\{P_1,\ldots,P_n\}$. Therefore there is at least one honest party, say P_h. In this case, recall that HP is also honest. Let $\mathcal{S}_{\mathcal{A}}$ denote the simulator for this case.

- **Preprocessing:** $\mathcal{S}_{\mathcal{A}}$ simulates the preprocessing phase of the protocol by acting on behalf of HP as per the protocol specifications.
- **Input:** Let P_d be the input provider.
- Case I: $d\neq h$. In the online phase, $\mathcal{S}_{\mathcal{A}}$ receives q_{v} from P_d (since $P_d\neq P_h$, thus corrupted by $\mathcal{A}$) and sends q_{v} to all the parties. $\mathcal{S}_{\mathcal{A}}$ extracts P_d's input $\mathsf{v}=q_{\mathsf{v}}-r-\delta_{\mathsf{v}}$. Note that r is the common random pad known to P_h and δ_{v} is known to HP from the preprocessing phase.
- Case II: $d=h$. $\mathcal{S}_{\mathcal{A}}$ performs the preprocessing phase according to the protocol on behalf of HP. In the online phase, $\mathcal{S}_{\mathcal{A}}$ samples a random value $\tilde{q}_{\mathsf{v}}$ and sends it to all the parties.
- **Multiplication:** In the online phase, $\mathcal{S}_{\mathcal{A}}$ receives at most $n-1$ shares of q_{z}. $\mathcal{S}_{\mathcal{A}}$ samples a random $\tilde{q}_{\mathsf{z}}$ and sends it to all the parties.
- **Verification:** $\mathcal{S}_{\mathcal{A}}$ simulates $\mathcal{F}_{\mathsf{Rand}}$ and obtains $\boldsymbol{\rho}$. Further, it simulates $\mathcal{F}_{\mathsf{Setup}}$ and obtains a sharing of zero, $\{\alpha_1,\alpha_2,\ldots,\alpha_n\}$ such that $\sum_{i=1}^{n}\alpha_i=0$. For all parties P_i corrupted by $\mathcal{A}$, $\mathcal{S}_{\mathcal{A}}$ receives $[\omega_{\mathsf{z}}]_i$. $\mathcal{S}_{\mathcal{A}}$ checks if $[\omega_{\mathsf{z}}]_i=\rho_0\cdot\alpha_i+\sum_{j\in[m]}\rho_j\cdot(\left[\mathsf{t}_{q_{\mathsf{z}_j}}\right]_i-q'_{\mathsf{z}_j}\cdot[\Delta]_i)$, where q'_{z_j} be the opened value. Note that $\left[\mathsf{t}_{q_{\mathsf{z}_j}}\right]_i=-\mathsf{m}_{\mathsf{x}_j}\cdot\left[\mathsf{t}_{\mathsf{y}_j}\right]_i-\mathsf{m}_{\mathsf{y}_j}\cdot\left[\mathsf{t}_{\mathsf{x}_j}\right]_i+\left[\mathsf{t}_{\mathsf{x}_j\mathsf{y}_j}\right]_i+\left[\mathsf{t}_{\mathsf{z}_j}\right]_i+[\Delta]_i\cdot(\mathsf{m}_{\mathsf{x}_j}\cdot\mathsf{m}_{\mathsf{y}_j}+\sum_{s=1}^{n}r_{sj})$, where $\mathsf{m}_{\mathsf{x}_j},\mathsf{m}_{\mathsf{y}_j}$ and r_{sj} for all $s\in[1,n]$ and $j\in[1,m]$ are held by the honest party P_h and $\left[\mathsf{t}_{\mathsf{x}_j}\right],\left[\mathsf{t}_{\mathsf{y}_j}\right],\left[\mathsf{t}_{\mathsf{x}_j\mathsf{y}_j}\right],\left[\mathsf{t}_{\mathsf{z}_j}\right]$ are held by HP.
- **Output:** If the above verification fails, $\mathcal{S}_{\mathcal{A}}$ invokes $\mathcal{F}_{\mathsf{MPC}}$ and sends $\perp$. Else, $\mathcal{S}_{\mathcal{A}}$ invokes $\mathcal{F}_{\mathsf{MPC}}$ with P_i's input for all corrupt P_i. (Recall that $\mathcal{S}_{\mathcal{A}}$ had extracted corrupt P_i's input during the input stage.) $\mathcal{F}_{\mathsf{MPC}}$ gives y as the output, then it computes $\delta_{\mathsf{y}}=\mathsf{m}_{\mathsf{y}}-\mathsf{y}$ and sends on behalf of HP.

Figure 13: Simulator $\mathcal{S}$ for Π_{mpc} for a malicious adversary corrupting up to $n-1$ parties among n parties

Indistinguishability.

Case I: Fig. 13 describes the simulation steps when a malicious adversary corrupts up to $n-1$ parties among n parties. In this scenario HP and party P_h is honest. Consider $\mathcal{A}$ be a malicious adversary corrupting up to $n-1$ parties (i.e. all parties excluding P_h) among the party set $\mathcal{P}$.

For input gates, if P_d is corrupt, then $\mathcal{S}_{\mathcal{A}}$ does not send any messages. Thus, the view is indistinguishable trivially. If P_d is honest then $\mathcal{S}_{\mathcal{A}}$ sends a random value $\tilde{q}_{\mathsf{v}}$ to $\mathcal{A}$. In the real protocol, P_d sends q_{v} to HP, which is forwarded to all the parties. Here $q_{\mathsf{v}}=\mathsf{m}_{\mathsf{v}}+r$, where $\mathsf{m}_{\mathsf{v}}=\mathsf{v}+\delta_{\mathsf{v}}$. Since δ_{v} is sampled uniformly, therefore q_{v} is distributed uniformly. Hence the distribution of q_{v} and $\tilde{q}_{\mathsf{v}}$ are identical.

For multiplication gates, $\mathcal{S}_{\mathcal{A}}$ sends $\tilde{q}_{\mathsf{z}}$ to $\mathcal{A}$. In the protocol, $\mathcal{A}$ receives q_{z} from HP. q_{z} and $\tilde{q}_{\mathsf{z}}$ are uniformly distributed thus the distribution is identical.

Next, we analyze the difference between the verification check carried out by the simulator $\mathcal{S}_{\mathcal{A}}$ and the verification check in the real-world protocol. In the verification step, $\mathcal{A}$ can introduce an error in q_{z_j}. However, it can obtain a sharing of the corresponding tag with negligible probability $\left(\frac{1}{|\mathbb{F}|}\right)$. Thus, consider a $j \in [1,m]$ such that $\left[\mathsf{t}_{q_{z_j}}\right]_i - q'_{z_j} \cdot [\Delta]_i \neq 0$, however $\sum_{i=1}^{n} \left[\rho_0 \cdot \alpha_i + \sum_{j\in[m]} \rho_j \cdot (\left[\mathsf{t}_{q_{z_j}}\right]_i - q'_{z_j} \cdot [\Delta]_i)\right] = 0$. This is possible for some choice of $\boldsymbol{\rho}$. To elaborate, consider $\xi_0 = \sum_{i=1}^{n} \alpha_i$ and $\xi_j = \sum_{i=1}^{n}(\left[\mathsf{t}_{q_{z_j}}\right]_i - q'_{z_j} \cdot [\Delta]_i)$ for all $j \in [1,m]$, set $\boldsymbol{\xi} = (\xi_0, \xi_1, \ldots, \xi_m)$. Then $T_{\boldsymbol{\xi}}(\boldsymbol{\rho}) = \sum_{j=0}^{m} \xi_j \cdot \rho_j$. $T_{\boldsymbol{\xi}}$ is a non-zero linear map, then $dim(ker(T_{\boldsymbol{\xi}})) \leq m$. Therefore, the probability that a randomly sampled $\boldsymbol{\rho}$ belongs to $ker(T_{\boldsymbol{\xi}}) \leq \frac{|\mathbb{F}|^m}{|\mathbb{F}|^{m+1}} = \frac{1}{|\mathbb{F}|}$. $\mathcal{S}_{\mathcal{A}}$ gets $[\omega_z]_i$ from all parties P_i corrupted by $\mathcal{A}$. In the real protocol, $[\omega_z]_i \neq \rho_0 \cdot \alpha_i + \sum_{j\in[m]} \rho_j \cdot (\left[\mathsf{t}_{q_{z_j}}\right]_i - q'_{z_j} \cdot [\Delta]_i)$. $\mathcal{A}$ can introduce an error ε in the $[\omega_z]_i$ term such that $\sum_{i=1}^{n} [\omega_z]_i + \varepsilon = 0$. In that case, in the real protocol, $\mathcal{A}$ cheats successfully. In the simulated world, the simulator aborts. However, $\mathcal{A}$ can find such an ε with probability $\frac{1}{|\mathbb{F}|}$. Therefore, $\mathcal{A}$ can successfully cheat with probability at most $\frac{3}{|\mathbb{F}|}$ which is negligible.

Thus the real world view is indistinguishable from the simulated view.

Simulator $\mathcal{S}_{\mathrm{HP}}$

Semi-Honest Let HP be semi-honest. Thus all the computational parties are honest. Let $\mathcal{S}_{\mathrm{HP}}$ be the simulator. Since HP does not have any input, $\mathcal{S}_{\mathrm{HP}}$ works as follows on input y where y is the output of the functionality.

- **Preprocessing:** $\mathcal{S}_{\mathrm{HP}}$ acts on behalf of the parties in $\mathcal{P}$ and receives the values from HP.
- **Input:** $\mathcal{S}_{\mathrm{HP}}$ samples a random value $\tilde{q}_{\mathsf{v}}$ for an input wire on behalf of an input provider. Then it receives the same $\tilde{q}_{\mathsf{v}}$ from HP.
- **Multiplication:** $\mathcal{S}_{\mathrm{HP}}$ samples n random values $\{[\tilde{q}_{\mathsf{z}}]_i\}_{i\in[1,n]}$ and sends these to HP. Then it receives $\tilde{q}_{\mathsf{z}}$ from HP.
- **Verification:** $\mathcal{S}_{\mathrm{HP}}$ samples a random sharing of zero, $[\tilde{\omega}]_i$ and sends it to HP.

Figure 14: Simulator $\mathcal{S}_{\mathrm{HP}}$ for Π_{mpc} for a semi-honest HP

Case II: Fig. 14 describes the simulation steps when the HP is semi-honest. In this case, all parties in $\mathcal{P}$ are honest.

Since HP does not have any input to the protocol, the simulated view of the preprocessing step of the protocol is identical to the real view of the preprocessing phase.

For an input gate, $\mathcal{S}_{\mathrm{HP}}$ samples and sends a randomly sampled $\tilde{q}_{\mathsf{v}}$. In the real protocol, HP receives $q_{\mathsf{v}} = \mathsf{m}_{\mathsf{v}} + r$, where r is uniformly sampled. Therefore, q_{v} is also uniformly distributed. Thus, the distribution of q_{v} and $\tilde{q}_{\mathsf{v}}$ are identical.

For a multiplication gate, $\mathcal{S}_{\mathrm{HP}}$ samples and sends $[\tilde{q}_{\mathsf{z}i}]$ to HP on behalf of P_i. In the real protocol, P_i sends $[q_{\mathsf{z}}]_i$ to HP, where $[q_{\mathsf{z}}]_i = [\mathsf{m}_{\mathsf{z}}]_i + r_i$ where r_i is a randomly sampled value. Therefore, $[q_{\mathsf{z}}]_i$ is also uniformly distributed. Hence, for all $i \in [1,n]$, $[\tilde{q}_{\mathsf{z}i}]$ and $[q_{\mathsf{z}}]_i$ are identically distributed.

In the verification step, $\mathcal{S}_{\mathrm{HP}}$ samples a random sharing of zero $[\tilde{\omega}]_i$. In the real protocol, P_i sends $[\omega]_i$ such that $\sum_{i=1}^{n} [\omega]_i = 0$, since all the parties follow the protocol. Also, $[\omega]_i = \rho_0 \cdot \alpha_i + \sum_{j=1}^{m} \rho_j \left[\omega_{\mathsf{z}_j}\right]_i$, where $\{\alpha_i\}_{i\in[1,n]}$ is a random sharing of zero. Therefore $[\omega_i]$ is uniformly distributed such that $\sum_{i=1}^{n} [\omega]_i = 0$. Therefore, $[\omega]_i$ and $[\tilde{\omega}]_i$ are identically distributed.

D Achieving fairness with stateless HP

In this section, we elaborate upon how our protocol could be modified to rely on a stateless HP. Recall that the only information we required the HP to store was the mask for output wire, say δ_{v}. Suppose we tweak our protocol to not use this mask and essentially assume $\delta_{\mathsf{v}} = 0$ for output wire by default. Then, it may so happen that the adversary obtains the output (as $\mathsf{m}_{\mathsf{v}} = v$) but makes the honest parties abort by misbehaving during the verification check. This would result in a protocol achieving security with abort (notion where the adversary learns the output and subsequently gets to choose whether to deliver the output to honest parties or not), which is a weaker notion than fairness.

The above shows that our efficient protocol can be tweaked to obtain achieving security with abort and relying on a stateless HP (instead of stateful). We can now upgrade this to fairness by using another call to stateless HP as follows (this is based on the technique of [27] which we recall for completeness):

- Run an instance of the MPC protocol with abort to compute additive shares of the output, rather than the output directly. In more detail, let Π' denote an instance of the abort protocol run for the following circuit which takes as input the inputs of parties $(x_1, \ldots, x_n)$ and does the following: computes the output, say $y = f(x_1, \ldots, x_n)$ and generate an additive sharing of y, namely $(y_1, \ldots, y_n)$. Sample the signing and verification key for a digital signature scheme and authenticate the additive shares. The output given to a party P_i comprises of the following values: (a) y_i (b) signature on y_i (c) verification key.
- An honest party aborts if it does not obtain its output at the end of Π' . Else, it continues to the next stage where parties invoke the stateless HP with their output of Π' (which comprises of the authenticated additive share and verification key).
- The HP checks if all parties sent the same verification key. If so, it proceeds to check if the shares are valid (i.e. the signatures on the additive shares verify). If all the shares verify successfully, the HP reconstructs the output y using all the n additive shares and returns y to all parties. Else it sends $\perp$ to all parties.

Intuitively, the above protocol maintains fairness because (a) if the adversary aborts during Π' and does not let honest parties obtain their output, then it learns at most $n-1$ additive shares of the output, which reveals no information about the function output $y = f(x_1, \ldots, x_n)$. (b) If the adversary misbehaves during the last call to the stateless HP (by either not sending its share or sending an incorrect share / verification key), no one gets the output; maintaining fairness.

E Building blocks

We design various building blocks required for the considered applications. While these building blocks have been studied in the literature [13, 19, 38, 46], the novelty lies in adapting these to the HP-aided setting considered in this work while keeping the efficiency of the constructions at the centre stage. Secure realization of protocols for various primitives such as comparison, equality check, and boolean to arithmetic conversion requires a heavy preprocessing, specifically for a large number of parties. We provide efficient construction with a very lightweight preprocessing by utilizing the presence of HP. Note that all our constructions follow along the lines of the multiplication protocol described in Sec: 4.3, where all communication happens via the HP, and the correctness of the computation is verified before output reconstruction via a verification phase. The building blocks considered are as follows. For the application of secure auctions, where the goal is to identify the maximum value in a set of N secret-shared values, we design secure protocols for comparison and oblivious selection between two secret-shared values. The comparison protocol further relies on other primitives such as prefixOR and multi-input multiplication, for which we provide efficient realizations in our considered setting. For the next application of dark pools, we design secure solutions for two most popular algorithms that are used in dark pools—continuous double auction (CDA) and volume matching (VM). In addition to comparison and oblivious select, dark pool algorithms require other building blocks such as equality and bit to arithmetic conversion, which are also designed in our work. The equality check protocol further relies on (i) k-mult—a primitive that enables the multiplication of k-inputs, and (ii) multi-input multiplication—a primitive that enables multiplying 3 and 4 inputs in a single shot at the same online cost as that of 2-input multiplication. We also design protocols for (i) and (ii). Finally, we also design secure protocols for dot product and shuffle, where the former finds widespread use in privacy-preserving machine learning, while the latter is used in various graph-based algorithms, anonymous communication, secure sorting, to name a few.

E.1 Multi-input multiplication

Several primitives such as prefixOR, and equality, to name a few, require multiplication of several inputs. The standard method to multiply, say k inputs, is to follow the tree-based approach. Here, in each of the $\log_2(k)$ rounds, every consecutive pair of inputs is multiplied using the multiplication protocol, which takes 2 inputs. The online complexity of this approach can be improved if one has access to multiplication protocols which allows multiplying more than 2 inputs in a single shot, also referred to as multi-input multiplication [46]. Specifically, we design 3-input and 4-input multiplication protocols that allow multiplying 3 and 4 inputs, respectively, while having the same online complexity as that of 2-input multiplication. The use of multi-input multiplication allows attaining an improvement of at least $2\times$ in the online complexity. Let $\mathsf{mult3}$ be a multiplication gate that takes 3 values say $\mathsf{a}, \mathsf{b}, \mathsf{c}$ as inputs and outputs $\mathsf{y} = \mathsf{a} \cdot \mathsf{b} \cdot \mathsf{c}$. Similarly, $\mathsf{mult4}$ takes $\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}$ as inputs and outputs $\mathsf{y} = \mathsf{a} \cdot \mathsf{b} \cdot \mathsf{c} \cdot \mathsf{d}$. The protocols to evaluate $\mathsf{mult3}$ and $\mathsf{mult4}$ are discussed next.

Π_{mult3}**:** The protocol takes as input $[\![\cdot]\!]$-shares of $\mathsf{a}, \mathsf{b}, \mathsf{c}$. To generate $[\![\mathsf{y}]\!]$, where $\mathsf{y} = \mathsf{a} \cdot \mathsf{b} \cdot \mathsf{c}$, in the preprocessing phase, HP generates $\langle \cdot \rangle$-shares of a random mask δ_{y} for the output wire. In the online phase, parties need to compute m_{y}, which can be written as

$$\begin{aligned}
\mathsf{m}_{\mathsf{y}} =& \mathsf{y} + \delta_{\mathsf{y}} = \mathsf{abc} + \delta_{\mathsf{y}} \\
=& (\mathsf{m}_{\mathsf{a}} - \delta_{\mathsf{a}})(\mathsf{m}_{\mathsf{b}} - \delta_{\mathsf{b}})(\mathsf{m}_{\mathsf{c}} - \delta_{\mathsf{c}}) + \delta_{\mathsf{y}} \\
=& \mathsf{m}_{\mathsf{a}}\mathsf{m}_{\mathsf{b}}\mathsf{m}_{\mathsf{c}} - \mathsf{m}_{\mathsf{a}}\mathsf{m}_{\mathsf{b}}\delta_{\mathsf{c}} - \mathsf{m}_{\mathsf{b}}\mathsf{m}_{\mathsf{c}}\delta_{\mathsf{a}} - \mathsf{m}_{\mathsf{c}}\mathsf{m}_{\mathsf{a}}\delta_{\mathsf{b}} \\
&+ \mathsf{m}_{\mathsf{a}}\delta_{\mathsf{b}}\delta_{\mathsf{c}} + \mathsf{m}_{\mathsf{b}}\delta_{\mathsf{c}}\delta_{\mathsf{a}} + \mathsf{m}_{\mathsf{c}}\delta_{\mathsf{a}}\delta_{\mathsf{b}} - \delta_{\mathsf{a}}\delta_{\mathsf{b}}\delta_{\mathsf{c}} + \delta_{\mathsf{y}} \\
=& \mathsf{m}_{\mathsf{a}}\mathsf{m}_{\mathsf{b}}\mathsf{m}_{\mathsf{c}} - \mathsf{m}_{\mathsf{a}}\mathsf{m}_{\mathsf{b}}\delta_{\mathsf{c}} - \mathsf{m}_{\mathsf{b}}\mathsf{m}_{\mathsf{c}}\delta_{\mathsf{a}} - \mathsf{m}_{\mathsf{c}}\mathsf{m}_{\mathsf{a}}\delta_{\mathsf{b}} \\
&+ \mathsf{m}_{\mathsf{a}}\delta_{\mathsf{bc}} + \mathsf{m}_{\mathsf{b}}\delta_{\mathsf{ca}} + \mathsf{m}_{\mathsf{c}}\delta_{\mathsf{ab}} - \delta_{\mathsf{abc}} + \delta_{\mathsf{y}}
\end{aligned}$$

As done in the multiplication protocol Sec: 4, to generate m_{y}, the approach is for parties to generate $\langle \cdot \rangle$-shares of m_{y}, followed by reconstructing it via the HP. To enable this, in the preprocessing phase, HP generates $\langle \cdot \rangle$-shares of of $\delta_{\mathsf{ab}} = \delta_{\mathsf{a}}\delta_{\mathsf{b}}, \delta_{\mathsf{bc}} = \delta_{\mathsf{b}}\delta_{\mathsf{c}}, \delta_{\mathsf{ca}} = \delta_{\mathsf{c}}\delta_{\mathsf{a}}, \delta_{\mathsf{abc}} = \delta_{\mathsf{a}}\delta_{\mathsf{b}}\delta_{\mathsf{c}}$. In the online phase, parties generate $\langle \cdot \rangle$-shares of m_{y} using the $\langle \cdot \rangle$-shares of $\delta_{\mathsf{ab}}, \delta_{\mathsf{bc}}, \delta_{\mathsf{ca}}, \delta_{\mathsf{abc}}$. To reconstruct m_{y}, parties send their shares to HP. However, to provide privacy against HP, as done in the case of 2-input multiplication, parties generate shares of q_{y} which is m_{y} masked with a random pad where the pad is known to all the parties except HP. Parties reconstruct q_{y} via the HP, from which each party obtains m_{y} by locally subtracting the pad.

Note that naively multiplying 3 inputs using the 2-input multiplication would require communicating 6 elements in the preprocessing phase, and $4n$ elements in the online phase with 4 rounds of interaction. On the other hand, relying on Π_{mult3} for the same requires communication of 9 elements in the preprocessing and $2n$ elements in the online phase with 2 rounds of interaction. In this way, Π_{mult3} allows attaining an improvement of $2\times$ in the online complexity.

Protocol Π_{mult3}

Preprocessing:

- HP locally computes $\delta_{\mathsf{ab}} = \delta_{\mathsf{a}} \cdot \delta_{\mathsf{b}}, \delta_{\mathsf{bc}} = \delta_{\mathsf{b}} \cdot \delta_{\mathsf{c}}, \delta_{\mathsf{ca}} = \delta_{\mathsf{a}} \cdot \delta_{\mathsf{c}}$, $\delta_{\mathsf{abc}} = \delta_{\mathsf{a}} \cdot \delta_{\mathsf{b}} \cdot \delta_{\mathsf{c}}$ and $\sigma_{\mathsf{ab}} = \Delta \cdot \delta_{\mathsf{ab}}, \sigma_{\mathsf{bc}} = \Delta \cdot \delta_{\mathsf{bc}}, \sigma_{\mathsf{ca}} = \Delta \cdot \delta_{\mathsf{ca}}$, $\sigma_{\mathsf{abc}} = \Delta \cdot \delta_{\mathsf{abc}}$
- Parties and HP invoke $\Pi_{\langle\cdot\rangle\text{-}\mathsf{Sh}}(\mathsf{HP}, \mathsf{v})$ for $\mathsf{v} \in \{\delta_{\mathsf{ab}}, \delta_{\mathsf{bc}}, \delta_{\mathsf{ca}}, \delta_{\mathsf{abc}}\}$.
- Parties and HP invoke $\Pi_{\langle\cdot\rangle\text{-}\mathsf{Sh}}(\mathsf{HP}, \mathsf{Rand})$ to generate $\langle\cdot\rangle$-shares of a random δ_{y}.

Online:

- All parties excluding HP sample a random vector $\boldsymbol{r}$ of length n using their common key $\mathsf{k}_{\mathcal{P}}$.
- P_1 computes $\langle q_{\mathsf{y}} \rangle_1 = \mathsf{m_a m_b m_c} + \boldsymbol{r}_1 - \mathsf{m_a m_b} \langle \delta_{\mathsf{c}} \rangle_1 - \mathsf{m_b m_c} \langle \delta_{\mathsf{a}} \rangle_1 - \mathsf{m_c m_a} \langle \delta_{\mathsf{b}} \rangle_1 + \mathsf{m_a} \langle \delta_{\mathsf{bc}} \rangle_1 + \mathsf{m_b} \langle \delta_{\mathsf{ca}} \rangle_1 + \mathsf{m_c} \langle \delta_{\mathsf{ab}} \rangle_1 - \langle \delta_{\mathsf{abc}} \rangle_1 + \langle \delta_{\mathsf{y}} \rangle_1$.
- For all $i \neq 1$, P_i computes $\langle q_{\mathsf{y}} \rangle_i = \boldsymbol{r}_i - \mathsf{m_a m_b} \langle \delta_{\mathsf{c}} \rangle_i - \mathsf{m_b m_c} \langle \delta_{\mathsf{a}} \rangle_i - \mathsf{m_c m_a} \langle \delta_{\mathsf{b}} \rangle_i + \mathsf{m_a} \langle \delta_{\mathsf{bc}} \rangle_i + \mathsf{m_b} \langle \delta_{\mathsf{ca}} \rangle_i + \mathsf{m_c} \langle \delta_{\mathsf{ab}} \rangle_i - \langle \delta_{\mathsf{abc}} \rangle_i + \langle \delta_{\mathsf{y}} \rangle_i$.
- For each $i \in [1, n]$, P_i sends $[q_{\mathsf{y}}]_i$ to HP.
- HP reconstructs and send $q_{\mathsf{y}} = \sum_{i=1}^{n} [q_{\mathsf{y}}]_i$ to all parties.
- Each P_i locally computes $\mathsf{m_y} = q_{\mathsf{y}} - \sum_{i=1}^{n} \boldsymbol{r}_i$.
- For each $i \in [1, n]$, P_i outputs $[\![\mathsf{y}]\!]_i = (\mathsf{m_y}, \langle \delta_{\mathsf{y}} \rangle_i)$.

Figure 15: Secure evaluation of fan-in 3 multiplication gate

Π_{mult4}**:** Similar to Π_{mult3}, we consider multiplication gates with 4 inputs. In this case too, $\mathsf{m_y}$ can be expressed as a combination of the masked values and masks. Here, the preprocessing phase involves the HP generating $\langle\cdot\rangle$-shares of $\{\delta_{\mathsf{ab}}, \delta_{\mathsf{bc}}, \delta_{\mathsf{ca}}, \delta_{\mathsf{ad}}, \delta_{\mathsf{bd}}, \delta_{\mathsf{cd}}, \delta_{\mathsf{abc}}, \delta_{\mathsf{acd}}, \delta_{\mathsf{abd}}, \delta_{\mathsf{bcd}}, \delta_{\mathsf{abcd}}\}$. In the online phase, parties generate $\mathsf{m_y}$ similarly as described for Π_{mult3}. This protocol requires communicating 23 elements in the preprocessing phase, and $2n$ elements of communication in the online phase with 2 rounds of interaction.

E.2 PrefixOR and PrefixAND

Given an array of k bits $\{a_1, a_2, \ldots, a_k\}$, where each bit is secret-shared, prefixOR outputs a sharing of an array of k bits $\{b_1, b_2, \ldots, b_k\}$ such that $b_i = \vee_{j=1}^{i} a_j$. Observe that, $b_i = 1 \oplus \wedge_{j=1}^{i} (1 \oplus a_i)$. Hence, to realize prefixOR it suffices to design a protocol for PrefixAND, that takes k bits $\{c_1, c_2, \ldots, c_k\}$ as inputs and outputs k bits $\{d_1, d_2, \ldots, d_k\}$ such that $d_i = \wedge_{j=1}^{i} c_i$. A naive way of computing prefixAND via 2-input multiplication requires $2k-2$ rounds of communication in the online phase. An optimized solution is presented in [37], which relies on multi-input multiplication. We rely on the protocol described in [37] to design a protocol for prefixAND which allows us to attain a round complexity to $2\log_4 k$. The resulting protocol requires communicating $\frac{3}{2} nk \log_4 k$ bits in the online phase and $\frac{35}{4} k \log_4 k$ bits in the preprocessing phase. At a high-level, given just 4 bits of input, their prefixAND can be obtained in two rounds by computing the 2-input, 3-input and 4-input multiplication. The prefixAND of more than four bits can be computed via a tree-based approach by computing the 2,3,4-input multiplication with respect to every consecutive group of four bits in each level of the tree, and repeating this process at every level to obtain the final result. In this way, prefixAND of up to 16 bits can be computed in 4 rounds and up to 64 bits in 6 rounds. The protocol steps appear in Fig. 16 and an illustration for $k = 16$ bits appears in Fig. 17.

Protocol $\Pi_{\mathsf{PrefixAND}}(a_1, \ldots, a_k)$

```
• For $j = 1$ to $k$ set $a_j^{(1)} = a_j$
• For $l = 1$ to $\log_4 k$
○ For $j = 1$ to $k/4^l$
  • Set $p = 4^l * (j-1)$,
    $q = 4^l * (j-1) + 4^{l-1}$,
    $r = 4^l * (j-1) + 2 * 4^{l-1}$,
    $s = 4^l * (j-1) + 3 * 4^{l-1}$
  • for $i = 1$ to $4^{l-1}$
  • $[\![a_{p+i}^{(l+1)}]\!]^{\mathsf{B}} = [\![a_{p+i}^{(l)}]\!]^{\mathsf{B}}$
  • $[\![a_{q+i}^{(l+1)}]\!]^{\mathsf{B}} = \Pi_{\mathsf{mult}}([\![a_q^{(l)}]\!]^{\mathsf{B}}, [\![a_{q+i}^{(l)}]\!]^{\mathsf{B}})$
  • $[\![a_{r+i}^{(l+1)}]\!]^{\mathsf{B}} = \Pi_{\mathsf{mult3}}([\![a_q^{(l)}]\!]^{\mathsf{B}}, [\![a_r^{(l)}]\!]^{\mathsf{B}}, [\![a_{r+i}^{(l)}]\!]^{\mathsf{B}})$
  • $[\![a_{s+i}^{(l+1)}]\!]^{\mathsf{B}} = \Pi_{\mathsf{mult4}}([\![a_q^{(l)}]\!]^{\mathsf{B}}, [\![a_r^{(l)}]\!]^{\mathsf{B}}, [\![a_s^{(l)}]\!]^{\mathsf{B}}, [\![a_{s+i}^{(l)}]\!]^{\mathsf{B}})$
Output: $([\![a_1^{(m)}]\!]^{\mathsf{B}}, \ldots, [\![a_k^{(m)}]\!]^{\mathsf{B}})$ where $m = \log_4 k$.
```

Figure 16: Computing PrefixAND of k boolean bits

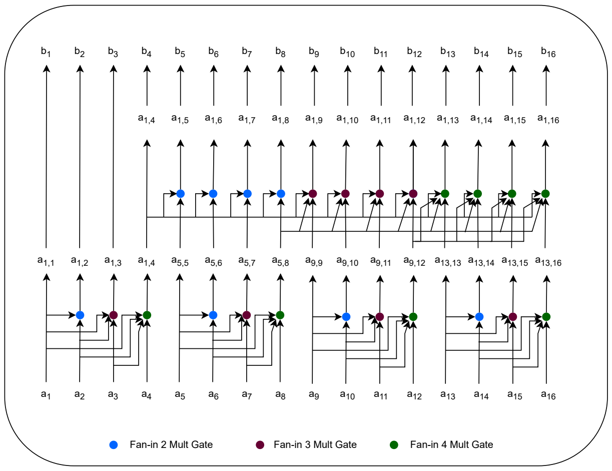
b1 b2 b3 b4 b5 b6 b7 b8 b9 b10 b11 b12 b13 b14 b15 b16
a1,4 a1,5 a1,6 a1,7 a1,8 a1,9 a1,10 a1,11 a1,12 a1,13 a1,14 a1,15 a1,16
a1,1 a1,2 a1,3 a1,4 a5,5 a5,6 a5,7 a5,8 a9,9 a9,10 a9,11 a9,12 a13,13 a13,14 a13,15 a13,16
a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 a11 a12 a13 a14 a15 a16
● Fan-in 2 Mult Gate
● Fan-in 3 Mult Gate
● Fan-in 4 Mult Gate

Figure 17: PrefixAND

E.3 k-mult

Let $\boldsymbol{x} = \{\mathsf{x}_1, \ldots, \mathsf{x}_k\}$ be an array of k values. The k-mult primitive takes $[\![\cdot]\!]$-shares of $\boldsymbol{x}$ as input and outputs $[\![\cdot]\!]$-shares of

$y = \prod_{i=1}^{k} \mathsf{x}_i$, which is the multiplication of the k elements in $\boldsymbol{x}$. Naively realizing this via the multiplication protocol requires $2\log_2 k$ rounds. We design a protocol $\Pi_{k\text{-mult}}$ for k-mult using Π_{mult4} such that the round complexity reduces to $2log_4k$. For ease of presentation, we consider k to be a power of 4. The protocol proceeds in iterations where in each iteration, we invoke Π_{mult4} on every consecutive set of 4 inputs to this iteration. The outputs generated by Π_{mult4} in this iteration constitute the input to the next iteration. In this way, in each iteration, the number of elements to be multiplied reduces by a factor of 4. Thus, at the end of $\log_4 k$ iterations, the desired output is generated. Note that invoking $\Pi_{k\text{-mult}}$ requires communication of $\frac{23}{3}(k-1)$ elements in the preprocessing phase. The communication cost in the online phase is $\frac{2}{3}n(k-1)$ elements, and it requires a total $2\log_4 k$ rounds of interaction. the formal protocol steps appear in Fig. 18.

Protocol $\Pi_{k\text{-mult}}([\![\mathsf{x}_1]\!], \ldots, [\![\mathsf{x}_k]\!]$

- Set $\mathsf{x}_j^{(1)} = \mathsf{x}_j$ for all $j \in [1,k]$.
- For $l = 1$ to $\log_4 k$
- For $j = 1$ to $k/4^l$
- $[\![\mathsf{x}_j^{(l+1)}]\!] = \Pi_{\mathsf{mult4}}([\![\mathsf{x}_{4j-3}^{(l)}]\!], [\![\mathsf{x}_{4j-2}^{(l)}]\!], [\![\mathsf{x}_{4j-1}^{(l)}]\!], [\![\mathsf{x}_{4j}^{(l)}]\!])$.

Output: P_i outputs $[\![\mathsf{x}_1^{(\log_4 k)}]\!]_i$.

Figure 18: Computing k-mult of k secret shared values

E.4 Equality check

The equality check protocol takes two secret shared values x and y as inputs and outputs 1 if $\mathsf{x} = \mathsf{y}$, else 0. Note that, checking $\mathsf{x} = \mathsf{y}$ reduces to checking $\mathsf{z} = 0$ where $\mathsf{z} = \mathsf{x} - \mathsf{y}$. Therefore $\Pi_{\mathsf{EQ}}([\![\mathsf{x}]\!], [\![\mathsf{y}]\!]) = \Pi_{\mathsf{EQZ}}([\![\mathsf{x} - \mathsf{y}]\!])$. Thus we design a protocol Π_{EQZ} that takes a secret shared value $[\![\mathsf{x}]\!]$ and outputs a boolean shared bit $[\![\mathsf{b}]\!]^{\mathsf{B}}$ such that $\mathsf{b} = 1$ if $\mathsf{x} = 0$ and $\mathsf{b} = 0$ if $\mathsf{x} \neq 0$.

At a high level, the protocol proceeds as follows. To check if $\mathsf{x} = 0$ where x is $[\![\cdot]\!]$-shared, the idea is to reconstruct $\mathsf{d} = \mathsf{x} + r$ where r is a random value not known to all parties, excluding HP. Observe that if $\mathsf{x} = 0$, then $\mathsf{d} = r$. Hence, to check if $\mathsf{x} = 0$, parties compute $e_j = \mathsf{d}_j \oplus r_j$, for all $j \in [0, k-1]$, where d_j and r_j are the jth bits of d and r respectively and boolean representation of d and r require k-bit to represent. If $\mathsf{x} = 0$, then $e_j = 0$ and $1 \oplus e_j = 1$ for all $j \in [0, k-1]$. Thus, $\mathsf{EQZ}(\mathsf{x}) = \wedge_{j=0}^{k-1}(1 \oplus e_j)$, which can be computed using k-mult on $\{(1 \oplus e_0), \ldots, (1 \oplus e_{k-1})\}$ to get the desired output. Note that performing the above computation requires shares of r as well as its bits, which can be generated by the HP. In [13], parties generate a sharing of r and bits of r, then parties open $\mathsf{d} = \mathsf{x} + r$ to all the parties. However, we optimize it by allowing parties to sample a random value c using $\mathsf{k}_{\mathsf{all}}$. Then HP generates sharing of $r = \mathsf{c} + \delta_{\mathsf{x}}$ as well as bits of r. In the online phase, parties obtain $\mathsf{d} = \mathsf{x} + r = \mathsf{c} + \mathsf{m}_{\mathsf{x}}$ without any interaction. This approach avoids performing reconstruction of $\mathsf{x} + r$ and saves communication of n elements. The formal protocol steps appear in Fig. 19. One instance of Π_{EQZ} requires communication of 12 elements in the preprocessing phase and $\frac{2n}{3}$ elements in the online phase. It requires $2\log_4 k$ rounds of interaction in the online phase.

Protocol $\Pi_{\mathsf{EQZ}}([\![\mathsf{x}]\!])$

Preprocessing:

- HP and all parties sample a random value c using the common key $\mathsf{k}_{\mathsf{all}}$.
- Execute $\Pi_{\langle\cdot\rangle\text{-}\mathsf{Sh}}(\mathrm{HP}, r)$ where $r = \mathsf{c} + \delta_{\mathsf{x}}$.
- HP sets $(r_{k-1}, \ldots, r_0) = \mathsf{BitDecompose}(r)$.
- Execute $\Pi_{\langle\cdot\rangle^{\mathsf{B}}}(\mathrm{HP}, r_j)$, for all $j \in [0, k-1]$.
- Execute preprocessing phase of $\Pi_{k\text{-mult}}$ (Fig. 18) using $(\langle r_0\rangle^{\mathsf{B}}, \ldots, \langle r_{k-1}\rangle^{\mathsf{B}})$.

Online:

- Parties set $\mathsf{d} = \mathsf{c} + \mathsf{m}_{\mathsf{x}}$.
- Parties locally set $(d_{k-1}, \ldots, d_0) = \mathsf{BitDecompose}(\mathsf{d})$.
- Parties set $[\![e_j]\!]^{\mathsf{B}} = (1 \oplus d_j, \langle r_j\rangle^{\mathsf{B}})$ that is, $\mathsf{m}_{e_j} = 1 \oplus d_j$ and $\delta_{e_j} = r_j$, for $j \in \{0, \ldots, k-1\}$.
- Parties and HP execute the online phase of $\Pi_{k\text{-mult}}([\![e_0]\!]^{\mathsf{B}}, \ldots, [\![e_{k-1}]\!]^{\mathsf{B}})$ and get output $[\![\mathsf{b}]\!]^{\mathsf{B}}$.

Figure 19: Equality Test

E.5 Bit to arithmetic

Given a boolean shares ($[\![\cdot]\!]^{\mathsf{B}}$-shares) of a bit b, this protocol generates its arithmetic shares. For this, observe that $[\![\mathsf{b}]\!]^{\mathsf{B}} = (\mathsf{m}_{\mathsf{b}}, \langle\delta_{\mathsf{b}}\rangle^{\mathsf{B}})$ where $\mathsf{b} = \mathsf{m}_{\mathsf{b}} \oplus \delta_{\mathsf{b}}$. If we let $(\mathsf{m}_{\mathsf{b}})^{\mathsf{A}}, (\delta_{\mathsf{b}})^{\mathsf{A}}$ denote the arithmetic equivalent of the bits m_{b} and δ_{b}, then the arithmetic equivalent of b is given as $(\mathsf{b})^{\mathsf{A}} = (\mathsf{m}_{\mathsf{b}})^{\mathsf{A}} + (\delta_{\mathsf{b}})^{\mathsf{A}} - 2(\mathsf{m}_{\mathsf{b}})^{\mathsf{A}}(\delta_{\mathsf{b}})^{\mathsf{A}}$. Thus we generate arithmetic shares of $(\mathsf{b})^{\mathsf{A}}$ by generating arithmetic shares of $(\mathsf{m}_{\mathsf{b}})^{\mathsf{A}}$ and $(\delta_{\mathsf{b}})^{\mathsf{A}}$. Since $(\mathsf{m}_{\mathsf{b}})^{\mathsf{A}}$ is available to all the parties (excluding HP), its $[\![\cdot]\!]$-shares can be generated as $[\![(\mathsf{m}_{\mathsf{b}})^{\mathsf{A}}]\!] = ((\mathsf{m}_{\mathsf{b}})^{\mathsf{A}}, \langle 0\rangle)$ where each component of $\langle 0\rangle$ is 0. However, δ_{b} is shared among the parties. Thus generating $[\![(\delta_{\mathsf{b}})^{\mathsf{A}}]\!]$ cannot be done non-interactively. In the standard (non-HP) setting, generating $\langle\cdot\rangle$-shares of $(\delta_{\mathsf{b}})^{\mathsf{A}}$ from its $\langle\cdot\rangle^{\mathsf{B}}$-shares is expensive. On the other hand, in our setting, HP holds the δ_{b} in clear, and therefore, it can generate $\langle(\delta_{\mathsf{b}})^{\mathsf{A}}\rangle$ in the preprocessing phase. Parties can then set $[\![(\delta_{\mathsf{b}})^{\mathsf{A}}]\!] = (0, -\langle(\delta_{\mathsf{b}})^{\mathsf{A}}\rangle)$. Note that the computation of $(\mathsf{b})^{\mathsf{A}}$ also has a term $(\mathsf{m}_{\mathsf{b}})^{\mathsf{A}}(\delta_{\mathsf{b}})^{\mathsf{A}}$. This requires performing multiplication, for which we rely on Π_{mult}. The formal protocol steps appear in Fig. 20. The communication cost of Π_{BitA} protocol is $2n$ elements, and it requires 2 rounds of interaction in the online phase. Preprocessing phase requires 1 instance of $\Pi_{\langle\cdot\rangle\text{-Sh}}(\text{HP}, \delta_\mathsf{b})$ and $\Pi_{\langle\cdot\rangle\text{-Sh}}(\text{HP}, \mathsf{Rand})$. In total, this requires communication of 3 elements in the preprocessing phase.

Protocol $\Pi_{\mathsf{BitA}}(\llbracket \mathsf{b} \rrbracket^{\mathsf{B}})$

Preprocessing:

- Execute $\Pi_{\langle\cdot\rangle\text{-Sh}}(\text{HP}, (\delta_\mathsf{b})^\mathsf{A})$.
- Execute $\Pi_{\langle\cdot\rangle\text{-Sh}}(\text{HP}, \mathsf{Rand})$ for a randomly sampled value δ_w.

Online:

- Each party P_i sets $\llbracket(\mathsf{m_b})^\mathsf{A}\rrbracket_i = ((\mathsf{m_b})^\mathsf{A}, \langle 0\rangle_i)$ where $\langle 0\rangle_i = (0,0)$ and $\llbracket(\delta_\mathsf{b})^\mathsf{A}\rrbracket_i = (0, -\langle(\delta_\mathsf{b})^\mathsf{A}\rangle_i)$.
- Parties excluding HP sample a random vector $\boldsymbol{r}$.
- For all $i \in [1,n]$, P_i sets $\langle q_\mathsf{w}\rangle_i = -(\mathsf{m_b})^\mathsf{A} \cdot \langle(\delta_\mathsf{b})^\mathsf{A}\rangle_i + \langle\delta_\mathsf{w}\rangle_i + \boldsymbol{r}_i$.
- For all $i \in [1,n]$, P_i sends $\langle q_\mathsf{w}\rangle_i$ to HP.
- HP sends q_w to all the parties.
- Each party P_i locally obtain $\mathsf{m_w} = q_\mathsf{w} - \sum_{i\in[n]} \boldsymbol{r}_i$.
- P_i outputs $\llbracket(\mathsf{b})^\mathsf{A}\rrbracket_i = (\mathsf{m}_{(\mathsf{b})^\mathsf{A}}, \langle\delta_{(\mathsf{b})^\mathsf{A}}\rangle_i)$ where $\mathsf{m}_{(\mathsf{b})^\mathsf{A}} = (\mathsf{m_b})^\mathsf{A} - 2\mathsf{m_w}$ and $\langle\delta_{(\mathsf{b})^\mathsf{A}}\rangle_i = -\langle(\delta_\mathsf{b})^\mathsf{A}\rangle_i - 2\langle\delta_\mathsf{w}\rangle_i$.

Figure 20: Bit to Arithmetic conversion

E.6 Comparison

The comparison protocol takes two secret shared values x and y as inputs and outputs 1 if x < y, 0 otherwise. This can be reduced to checking if $\mathsf{z} < 0$ where $\mathsf{z} = \mathsf{x} - \mathsf{y}$. Thus we design a protocol Π_{LTZ} that takes one secret shared value $\llbracket \mathsf{x} \rrbracket$ as input and outputs $\llbracket 1 \rrbracket$ if $\mathsf{x} < 0$ and $\llbracket 0 \rrbracket$ otherwise. For this, we adapt the approach of [13] to work with our setting. Informally, the protocol proceeds as follows. To check if $\mathsf{x} < 0$, it is sufficient to check the most significant bit (MSB) of x is a 1. This is because MSB is a 1 if $\mathsf{x} < 0$; else, it is 0. To extract the MSB of x the idea is to compute x', which is the lower $k-1$ bits of x, and compute $\text{MSB}(\mathsf{x}) = (\mathsf{x} - \mathsf{x}') * 2^{-(k-1)}$. The problem now boils down to computing x' which are the lower $k-1$ bits of x. The approach is to reconstruct $\mathsf{d} = \mathsf{x} + r \mod 2^{k-1}$ where r is a random value not known to any party. Given d, x' can be computed as $\mathsf{x}' = \mathsf{d} - r + \mathsf{u}2^{k-1}$ where u denotes if $\mathsf{d} < r$, i.e., $\mathsf{u} = 1$ if $\mathsf{d} < r$, else $\mathsf{u} = 0$. To compute u, parties proceed as follows. Observe that if $s \in [0, k-2]$ denotes the highest bit position where the bits of d and r differ, then u can be written as $\mathsf{u} = \mathsf{d}_s \oplus 1$. To compute d_s, the approach is to compute the XOR of the corresponding bits in d and r, followed by computing a prefixOR of the output of the XOR. This results in a vector of bits which comprises of 0s for all $j > s$ and 1s for all $j \le s$. Elaborately, let $\mathsf{y}_j = \mathsf{d}_j \oplus r_j$ for $j \in [0, k-2]$, where d_j, r_j represent the j^{th} bit of d, r, respectively. Parties execute prefixOR on $(\mathsf{y}_{k-2}, \ldots, \mathsf{y}_0)$ to generate $(\mathsf{z}_{k-2}, \ldots, \mathsf{z}_0)$ such that $\mathsf{z}_j = 0$ for $j > s$ and $\mathsf{z}_j = 1$ for $j \le s$. To identify the position s where d and r first differ, parties generate $\mathsf{w}_j = \mathsf{z}_j \oplus \mathsf{z}_{j+1}$ for $j \in [0, k-3]$ and set $\mathsf{z}_{k-2} = \mathsf{w}_{k-2}$. Observe that performing this XOR of the consecutive bits in $(\mathsf{z}_{k-2}, \ldots, \mathsf{z}_0)$ ensures that only $\mathsf{w}_s = 1$ while $\mathsf{w}_j = 0$ for all $j \ne s$. Thus, taking a dot product of $(\mathsf{w}_{k-2}, \ldots, \mathsf{w}_0)$ with the bits in d, i.e., $(\mathsf{d}_{k-2}, \ldots, \mathsf{d}_0)$, allows to obtain d_s, from which $\mathsf{u} = \mathsf{d}_s \oplus 1$ can be obtained. Note that except for d, all the other values described above are shared, and hence, the computation proceeds on these shared values. The formal protocol steps appear in Fig. 21.

Protocol $\Pi_{\mathsf{LTZ}}(\llbracket \mathsf{x} \rrbracket)$

Preprocessing:

- HP and all parties sample a random value c using their common key.
- Execute $\Pi_{\langle\cdot\rangle\text{-Sh}}(\text{HP}, r)$ where $r = c + \delta_\mathsf{x}$.
- HP computes $(r_{k-1}, \ldots, r_0) = \mathsf{BitDecompose}(r)$.
- Execute $\Pi_{\langle\cdot\rangle^\mathsf{B}}(\text{HP}, r_j)$ for all $j \in [0, k-2]$.
- Parties and HP set $\langle\delta_{\mathsf{y}_j}\rangle^\mathsf{B} = \langle r_j\rangle^\mathsf{B}$ for $j \in [0, k-2]$.
- Execute preprocessing of PrefixOR on inputs $(\langle\delta_{\mathsf{y}_{k-2}}\rangle^\mathsf{B}, \ldots, \langle\delta_{\mathsf{y}_0}\rangle^\mathsf{B})$ and obtain $(\langle\delta_{\mathsf{z}_{k-2}}\rangle^\mathsf{B}, \ldots, \langle\delta_{\mathsf{z}_0}\rangle^\mathsf{B})$.
- Set $\langle\delta_{\mathsf{w}_j}\rangle^\mathsf{B} = \langle\delta_{\mathsf{z}_j}\rangle^\mathsf{B} \oplus \langle\delta_{\mathsf{z}_{j+1}}\rangle^\mathsf{B}$ where for all $j \in [0, k-2]$ and $\langle\delta_{\mathsf{z}_{k-1}}\rangle^\mathsf{B} = 0$.
- Execute $\Pi_{\langle\cdot\rangle^\mathsf{B}}(\text{HP}, \mathsf{Rand})$ for a randomly sampled bit δ_u.
- Perform preprocessing of Π_{BitA} for the input mask $\langle\delta_\mathsf{u}\rangle^\mathsf{B}$.

Online:

- Parties locally set $\mathsf{d} = (\mathsf{m_x} + \mathsf{c}) \mod 2^{k-1}$
- Set $(\mathsf{d}_{k-2}, \ldots, \mathsf{d}_0) = \mathsf{BitDecompose}(\mathsf{d})$
- Parties set $\mathsf{m}_{\mathsf{y}_j} = \mathsf{d}_j$ for all $j \in [0, k-2]$.
- $(\llbracket \mathsf{z}_{k-2} \rrbracket^\mathsf{B}, \ldots, \llbracket \mathsf{z}_0 \rrbracket^\mathsf{B}) = \Pi_{\mathsf{PrefixOR}}(\llbracket \mathsf{y}_{k-2} \rrbracket^\mathsf{B}, \ldots, \llbracket \mathsf{y}_0 \rrbracket^\mathsf{B})$
- Set $\mathsf{m}_{\mathsf{w}_j} = \mathsf{m}_{\mathsf{z}_j} \oplus \mathsf{m}_{\mathsf{z}_{j+1}}$ for all $j \in [0, k-2]$ where $\mathsf{m}_{\mathsf{z}_{k-1}} = 0$.
- All parties excluding HP sample random vector of bits $\boldsymbol{r}$.
- P_1 computes $\langle q_\mathsf{u}\rangle_1^\mathsf{B} = \oplus_{j=0}^{k-2} \mathsf{m}_{\mathsf{w}_j} \cdot \mathsf{d}_j \oplus_{j=0}^{k-2} \mathsf{d}_j \langle\delta_{\mathsf{w}_j}\rangle_1^\mathsf{B} \oplus \langle\delta_\mathsf{u}\rangle_1^\mathsf{B} \oplus \boldsymbol{r}_1$.
- For all $i \ne 1$, P_i computes $\langle q_\mathsf{u}\rangle_i^\mathsf{B} = \oplus_{j=0}^{k-2} \mathsf{d}_j \langle\delta_{\mathsf{w}_j}\rangle_i^\mathsf{B} \oplus \langle\delta_\mathsf{u}\rangle_i^\mathsf{B} \oplus \boldsymbol{r}_i$.
- For all $i \in [1,n]$, P_i sends $\langle q_\mathsf{u}\rangle_i^\mathsf{B}$ to HP.
- HP reconstructs and sends q_u to all the parties.
- All parties obtain $\mathsf{m_u} = q_\mathsf{u} \oplus (\oplus_{i\in[n]} \boldsymbol{r}_i)$.
- Parties and HP perform the online phase of BitA on input $(1 \oplus \mathsf{m_u}, \langle\delta_\mathsf{u}\rangle^\mathsf{B})$ and output $\llbracket \mathsf{v} \rrbracket$.
- $\llbracket \mathsf{x}' \rrbracket = (\mathsf{d} + 2^{k-1}\mathsf{m_v}, 2^{k-1}\langle\delta_\mathsf{v}\rangle + \langle r\rangle)$.
- Parties output $\llbracket \mathsf{b} \rrbracket = (\llbracket \mathsf{x} \rrbracket - \llbracket \mathsf{x}' \rrbracket) * 2^{-(k-1)}$

Figure 21: Less Than Zero Test

The communication cost of Π_{LTZ} is $7 + \frac{35}{4}\log_4 k$ elements in the preprocessing phase and $2n + \frac{2n}{k} + \frac{3n}{2}\log_4 k$ elements in the online phase. In the online phase, $2\log_4 k + 4$ rounds of interaction are required.

E.7 Oblivious selection

The oblivious selection protocol, Π_{sel} takes the $[\![\cdot]\!]$-shares of the values x_0 and x_1 along with $[\![\cdot]\!]^{\mathsf{B}}$-shares of a bit b. It gives as output $[\![\cdot]\!]$-shares of x_{1-i} if $\mathsf{b} = i$. A naive way of executing oblivious selection can be done by securely computing $\mathsf{b} \cdot \mathsf{x}_0 + (1 - \mathsf{b}) \cdot \mathsf{x}_1$. However, this requires performing 2 secure multiplications in parallel. This can be optimized by computing $\mathsf{b} \cdot (\mathsf{x}_0 - \mathsf{x}_1) + \mathsf{x}_1$. This requires performing a single multiplication. Note that since the bit b is Boolean shared, the parties first perform Π_{BitA} to generate $[\![\mathsf{b}]\!]$ followed by the multiplication.

Below we additionally discuss how the primitive operations of dot product and shuffle can be realized securely. While dot product forms a crucial primitive in applications such as privacy-preserving machine learning [21, 30, 39, 41], shuffle is also extensively used in various applications such as anonymous broadcast [24, 47], oblivious RAM [5, 14], the graphSC paradigm [4, 45], to name a few.

E.8 Dot product

Let $\boldsymbol{a}$ and $\boldsymbol{b}$ are two vectors of length N, and let c be the output. Corresponding to every component of $\boldsymbol{a}$ and $\boldsymbol{b}$, a party P_i holds the authenticated sharing of the masks, and HP holds the complete masks in the preprocessing phase. HP computes the dot product of the masks of $\boldsymbol{a}$ and masks of $\boldsymbol{b}$. Let $\delta_{\boldsymbol{ab}}$ be the dot product of the masks of $\boldsymbol{a}$ and the masks of $\boldsymbol{b}$. HP generates an authenticated additive sharing of $\delta_{\boldsymbol{ab}}$. Finally, HP generates an authenticated additive sharing of a random value δ_{c}. In the online phase, all parties excluding HP, sample a random vector $\boldsymbol{r}$. P_1 computes $\langle q_{\mathsf{c}} \rangle_1 = \boldsymbol{r}_1 + \sum_{j=1}^{N} (\mathsf{m}_{\boldsymbol{a}_j}\mathsf{m}_{\boldsymbol{b}_j} - \mathsf{m}_{\boldsymbol{a}_j}\langle \delta_{\boldsymbol{b}_j} \rangle_1 - \mathsf{m}_{\boldsymbol{b}_j}\langle \delta_{\boldsymbol{a}_j} \rangle_1) + \langle \delta_{\boldsymbol{ab}} \rangle_1 + \langle \delta_{\mathsf{c}} \rangle_1$ and for all $i \neq 1$, P_i computes $\langle q_{\mathsf{c}} \rangle_i = \boldsymbol{r}_i + \sum_{j=1}^{\mathsf{N}} (-\mathsf{m}_{\boldsymbol{a}_j}\langle \delta_{\boldsymbol{b}_j} \rangle_i - \mathsf{m}_{\boldsymbol{b}_j}\langle \delta_{\boldsymbol{a}_j} \rangle_i) + \langle \delta_{\boldsymbol{ab}} \rangle_i + \langle \delta_{\mathsf{c}} \rangle_i$. Each P_i sends $\langle q_{\mathsf{c}} \rangle_i$ to HP. HP reconstructs and sends q_{c} to all. Finally, all parties locally obtain $\mathsf{m}_{\mathsf{c}} = q_{\mathsf{c}} - \sum_{i=1}^{n} \boldsymbol{r}_i$.

Protocol $\Pi_{\mathsf{DotP}}([\![\boldsymbol{a}]\!], [\![\boldsymbol{b}]\!])$

Preprocessing:

- HP generates $\langle \cdot \rangle$ of $\delta_{\boldsymbol{ab}}$ where $\delta_{\boldsymbol{ab}} = \sum_{j \in [\mathsf{N}]} \delta_{\boldsymbol{a}_j} \cdot \delta_{\boldsymbol{b}_j}$.
- HP samples δ_{c} and generates $\langle \cdot \rangle$ of δ_{c}.

Online:

- All parties excluding HP sample a random vector $\boldsymbol{r}$ using their common key.
- Party P_1 computes $\langle q_{\mathsf{c}} \rangle_1 = \boldsymbol{r}_1 + \sum_{j=1}^{\mathsf{N}} (\mathsf{m}_{\boldsymbol{a}_j}\mathsf{m}_{\boldsymbol{b}_j} - \mathsf{m}_{\boldsymbol{a}_j}\langle \delta_{\boldsymbol{b}_j} \rangle_1 - \mathsf{m}_{\boldsymbol{b}_j}\langle \delta_{\boldsymbol{a}_j} \rangle_1) + \langle \delta_{\boldsymbol{ab}} \rangle_1 + \langle \delta_{\mathsf{c}} \rangle_1$.
- For all $i \neq 1$, P_i computes $\langle q_{\mathsf{c}} \rangle_i = \boldsymbol{r}_i + \sum_{j=1}^{\mathsf{N}} (-\mathsf{m}_{\boldsymbol{a}_j}\langle \delta_{\boldsymbol{b}_j} \rangle_i - \mathsf{m}_{\boldsymbol{b}_j}\langle \delta_{\boldsymbol{a}_j} \rangle_i + \langle \delta_{\boldsymbol{ab}} \rangle_i + \langle \delta_{\mathsf{c}} \rangle_i$.
- For all $i \in [1, n]$, P_i sends $\langle q_{\mathsf{c}} \rangle_i$ to HP.
- HP reconstructs q_{c} and sends it to all.
- Each party locally obtains $\mathsf{m}_{\mathsf{c}} = q_{\mathsf{c}} - \sum_{i \in [n]} \boldsymbol{r}_i$.

Figure 22: Dot Product

We emphasize that the communication cost for the dot product is independent of the size of the vectors both in the preprocessing and online phases. A dot product requires communication of 3 elements in the preprocessing phase and $2n$ elements in the online phase. The round complexity is 2.

E.9 Shuffle

Let $\boldsymbol{x}$ be a vector of length N. The shuffle functionality generates a vector $\boldsymbol{y}$ such that $\boldsymbol{y}_i = \boldsymbol{x}_{\pi(i)}$ for a random permutation $\pi : \mathbb{Z}_{\mathsf{N}} \to \mathbb{Z}_{\mathsf{N}}$. In the secure shuffle protocol, parties start with a secret shared vector $\boldsymbol{x}$, and at the end of the protocol, parties obtain secret sharing of the vector $\boldsymbol{y}$. Note that the permutation π remains hidden from all the parties.

Protocol $\Pi_{\mathsf{shuffle}}([\![\mathsf{x}_1]\!], \ldots, [\![\mathsf{x}_{\mathsf{N}}]\!])$

Preprocessing:

- HP sample a random permutation $\pi : \mathbb{Z}_{\mathsf{N}} \to \mathbb{Z}_{\mathsf{N}}$, and generates M_π.
- Execute $\Pi_{\langle\cdot\rangle\text{-}\mathsf{Sh}}(\mathrm{HP}, M_\pi(i, j))$ for $i, j \in [1, \mathsf{N}]$.
- for $i \in [1, \mathsf{N}]$,
  - perform preprocessing of Π_{DotP} where the inputs are $(\langle M_\pi(i,1) \rangle, \ldots, \langle M_\pi(i,\mathsf{N}) \rangle)$ and $(\langle \delta_{\mathsf{x}_1} \rangle, \ldots, \langle \delta_{\mathsf{x}_{\mathsf{N}}} \rangle)$.

Online:

- For $i \in [1, \mathsf{N}]$,
  - Set $\mathsf{m}_{M_\pi(i,j)} = 0$, for all $i, j \in [1, \mathsf{N}]$.
  - Perform online phase of Π_{DotP} on inputs $([\![M_\pi(i,1)]\!], \ldots, [\![M_\pi(i,\mathsf{N})]\!])$ and $([\![\mathsf{x}_1]\!], \ldots, [\![\mathsf{x}_{\mathsf{N}}]\!])$. Let the output be $[\![\mathsf{z}_i]\!]$

Output: $([\![\mathsf{z}_1]\!], \ldots, [\![\mathsf{z}_{\mathsf{N}}]\!])$.

Figure 23: Secure shuffle for a length N vector

A permutation π on N length vector can be represented as a matrix M_π of size $\mathsf{N} \times \mathsf{N}$. M_π is a binary matrix where each row and each column of M_π contains exactly one 1, and the rest are 0. Further note that, $\boldsymbol{y} = \pi(\boldsymbol{x}) = M_\pi \boldsymbol{x}$. HP will sample a random permutation π and it will generate authenticated sharing of M_π. HP performs $\Pi_{\langle\cdot\rangle\text{-}\mathsf{Sh}}(\mathrm{HP}, M_\pi(i, j))$. where $M_\pi(i, j)$ is the jth element in the ith row of the matrix $M_\pi(i, j)$. This step is performed in the preprocessing phase, and it requires total $2 \times \mathsf{N}^2$ elements to be communicated. Thus a secure shuffle protocol can be executed by performing securely multiplying a matrix multiplication with a vector if the matrix and the vector are shared. Multiplication of a matrix and vector can be done by performing dot products between the rows of the matrix with the vector. The protocol for dot product E.8 is described above in Fig: 22. Therefore the total cost of the shuffle protocol is $2\mathsf{N}^2 + 3\mathsf{N}$ elements in the preprocessing phase and $2n\mathsf{N}$ elements in the online. The online round complexity is 2.

Our construction is similar to a construction presented in [42]. However, the generation of the permutation matrix in [42] is substantially expensive than our construction due the presence of HP.

E.10 Complexity analysis

Multiplication with 3 and 4 inputs The protocol Π_{mult3} (Fig. 15) has preprocessing cost 9 elements, it requires 4 instances of $\Pi_{\langle\cdot\rangle\text{-}\mathsf{Sh}}(\mathsf{HP}, \mathsf{v})$ and 1 instance of $\Pi_{\langle\cdot\rangle\text{-}\mathsf{Sh}}(\mathsf{HP}, \mathsf{Rand})$. The online communication is $2n$ elements and 2 rounds. The protocol Π_{mult4} has preprocessing cost 23 elements this is due to 11 instances of $\Pi_{\langle\cdot\rangle\text{-}\mathsf{Sh}}(\mathsf{HP}, \mathsf{v})$ and 1 instance of $\Pi_{\langle\cdot\rangle\text{-}\mathsf{Sh}}(\mathsf{HP}, \mathsf{Rand})$. The online communication is $2n$ elements and 2 rounds.

PrefixOR and PrefixAND The protocol $\Pi_{\mathsf{PrefixAND}}$ (Fig. 16) requires to evaluate a boolean circuit which is of depth $\log_4 k$, which has $k/4$ mult, $k/4$ mult3 and $k/4$ mult4 in every level. In total, it has preprocessing cost $\frac{35}{4}\log_4 k$ elements and online communication $\frac{3n}{2}\log_4 k$ elements and $2\log_4 k$ rounds.

Multiplication of k bits The protocol $\Pi_{k\text{-}\mathsf{mult}}$ (Fig. 18) requires to evaluate a circuit which has 4^{level} mult4 gates in every level where level varies from 1 to $\log_4 k$. Therefore, it has preprocessing cost $\frac{23}{3}$ elements and online communication $\frac{2n}{3}$ elements and $2\log_4 k$ rounds.

Equality check The protocol Π_{EQZ} (Fig. 19) has preprocessing cost 12 elements and online communication $\frac{2n}{3}$ elements and $2\log_4 k$ rounds. This follows directly from the multiplication of k bits.

Bit to arithmetic conversion The protocol Π_{BitA} (Fig. 20) has preprocessing cost 3 elements and online communication $2n$ elements and 2 rounds.

Comparison The protocol Π_{LTZ} (Fig. 21) has preprocessing cost $7 + \frac{35}{4}\log_4 k$ elements and online communication $2n + \frac{2n}{k} + \frac{3n}{2}\log_4 k$ elements and $2\log_4 k + 4$ rounds. This follows directly from the prefixOR.

Oblivious selection The Oblivious selection protocol has preprocessing cost 6 elements and online communication $4n$ elements and 4 rounds.

# of parties	Atlas		Asterisk	
	Comm. (MB)	Time (sec)	Comm. (MB)	Time (sec)
5	128	31.2	80	55.6
10	320	42.8	160	111.8
25	768	109.4	400	281.8
50	1600	441.5	800	574.9

Table 10: Comparison of communication and run time for Atlas [28] in the online phase.

F Applications and additional benchmarks

Below we provide an additional comparison of run time for our protocol when running over LAN and WAN settings. We consider a circuit of size 10^6, which has 1000 multiplication gates in every level and 1000 depth. When we execute the protocol by varying the number of parties in $\{4, 8, 30, 40, 60\}$, we observe that our protocol performs up to $52\times$ faster than the execution over a WAN when it is executed over a LAN network. A visual representation of the same appears in Fig. 24.

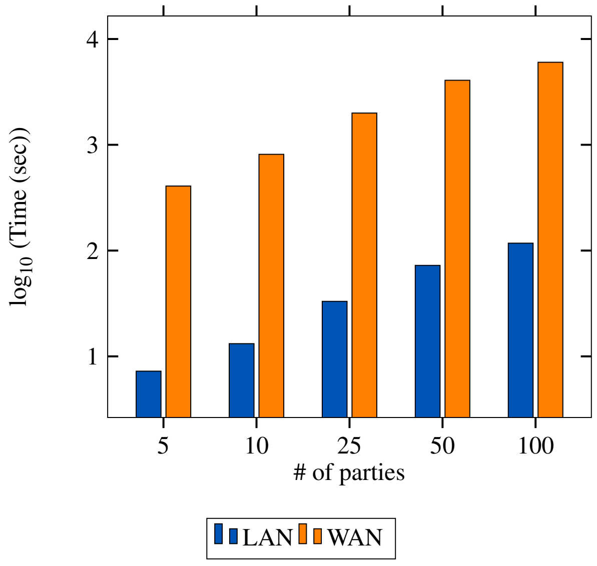

Figure 24: Comparison of the run time of Asterisk when running over LAN and WAN networks (the values are scaled in the logarithmic scale with base 10).

Additionally, we also compare against the state-of-the-art honest majority protocol Atlas [28] implemented in [34]. We note that the implementation allows running only the online phase of [28], which is reported in Table 10. We observe that our protocol has up to $2\times$ improvement in online communication and a comparable run time in comparison to [28], despite the fact that the implementation of [28] is highly optimized.

F.1 Secure auctions

The protocol for secure auctions proceeds as follows. Let $\boldsymbol{b}$ denote a N-sized array that comprises the bid amounts and the bidder's names. A simple approach to compute the highest bid among a set of N bids in $\boldsymbol{b}$ is to sequentially perform a secure comparison between $\boldsymbol{b}[i], \boldsymbol{b}[i+1]$ for $i = 1$ to $N-1$. This requires $N-1$ rounds of comparison. This approach can be optimized to obtain the result in $O(\log(N))$ rounds using the tree-based approach. Elaborately, we begin by comparing $\boldsymbol{b}[2i-1], \boldsymbol{b}[2i]$ for $i = 1$ to $N/2$ in parallel. We then identify which among $\boldsymbol{b}[2i-1], \boldsymbol{b}[2i]$ is greater. For this, given the secret-shared output of the comparison, say $\mathsf{b}_i = \mathbf{1}(\mathsf{x}, \mathsf{y})$, where $\mathsf{x} = \boldsymbol{b}[2i-1]$ and $\mathsf{y} = \boldsymbol{b}[2i]$, parties compute $\mathsf{c} = \mathsf{b}(\mathsf{x}-\mathsf{y}) + \mathsf{y}$, where c is the higher value among x, y. This requires performing one oblivious select, which requires 2 rounds of interaction. Following this, we obtain $N/2$ highest elements, and repeat this process until we find the single highest element. This requires $(2\log_4 k + 4)\log_2(N)$ rounds for comparison and $2\log_2(N)$ rounds for oblivious select, for a total of $(2\log_4 k + 6)\log_2(N)$rounds. The protocol appears in Fig. 25.

```
Protocol $\Pi_{\mathsf{Auction}}(\mathcal{P}, \boldsymbol{b})$
Input: Array $\boldsymbol{b}$ of size $N$.
Output: Largest value in $\boldsymbol{b}$.
 – Set $\mathsf{k} = N/2$.
 – While $\mathsf{k} > 1$ do:
   ○ for $i = 1$ to $\mathsf{k}$ do:
     - $[\![\mathsf{b}_i]\!] = \Pi_{\mathsf{LT}}([\![\boldsymbol{b}[2i-1]]\!], [\![\boldsymbol{b}[2i]]\!])$
     - $[\![\mathsf{c}_i]\!] = \Pi_{\mathsf{mult}}(([\![\boldsymbol{b}[2i]]\!] - [\![\boldsymbol{b}[2i-1]]\!]), [\![\mathsf{b}_i]\!]) + [\![\boldsymbol{b}[2i-1]]\!]$
   ○ $\mathsf{k} = \mathsf{k}/2$
   ○ for $i = 1$ to $\mathsf{k}$: $[\![\boldsymbol{b}[i]]\!] = [\![\mathsf{c}_i]\!]$
 – Output $[\![\boldsymbol{b}[1]]\!]$
```

Figure 25: Secure auction protocol

F.2 Secure dark pools

The considered algorithms for dark pools are described below.

F.2.1 Continuous double auction

We begin by explaining the continuous double auction algorithm (CDA) [12], where orders are processed in a continuous manner. Each buy order (or request) comprises the client's name, $name^b$, the units to be bought, b, and the buying price, q. Similarly, a sell order comprises the client's name, $name^s$, the units to be sold, s, and the selling price, p. The algorithm maintains a buy list ($\mathcal{B}$), which comprises of buy orders that are not yet matched, and a sell list ($\mathcal{S}$) of sell orders that are yet to be matched. The algorithm maintains the invariant that orders in the buy list are sorted in descending order as per the buying price, while orders in the sell list are sorted in ascending order as per the selling price.

We next explain the steps involved when processing a new buy order in the system. The steps for processing a sell order are analogous. When a new buy order arrives, the algorithm proceeds to identify matching sell orders. A buy order is said to match a sell order if the following two matching criteria are satisfied—(i) the buying price of the buy order is greater than or equal to the selling price of the sell order (known as the price criteria), and (ii) the units of one order must be able to satisfy the units of the other (known as the volume criteria). Starting with the first sell order in $\mathcal{S}$, the incoming buy order is matched with every order in $\mathcal{S}$ until either the price criteria or volume criteria fails. Thus, the buy order may have more than one matching sell order. When either one of the matching criteria fails, the buy order may either be fully satisfied, i.e., all of its units are exhausted by getting matched to sell orders or, it may be partially satisfied, i.e., some of its units are still unmatched. If the buy order is partially satisfied, then it is inserted in the right position as per its buying price in the sorted $\mathcal{B}$. To ensure that the steps in the matching phase can be run in parallel instead of sequentially identifying the matching sell orders, the work of [38] maintains additional information, which allows for reducing the round complexity of the above-mentioned algorithm.

Thus, the CDA algorithm comprises a matching phase followed by an insertion phase. The formal protocols for CDA, matching phase and insertion phase appear in Fig. 26, Fig. 27, Fig. 28 where N, M denote the size of the sell list and buy list respectively.

```
Protocol $\Pi_{\mathsf{CDA}}(\mathcal{P}, ([\![name_0^b]\!], [\![b_0]\!], [\![q_0]\!]), \mathcal{B}, \mathcal{S})$
 – Invoke $\Pi_{\mathsf{PSL}}$ on $([\![name_0^b]\!], [\![b_0]\!], [\![q_0]\!])$
 – Compute $[\![e]\!]^{\mathsf{B}} = \Pi_{\mathsf{EQ}}([\![b_0]\!], 0)$
 – Compute $[\![q_0]\!] = \Pi_{\mathsf{sel}}([\![q_0]\!], [\![0]\!], [\![e]\!]^{\mathsf{B}})$
 – Invoke $\Pi_{\mathsf{Insert}}$ to insert $([\![name_0^b]\!], [\![b_0]\!], [\![q_0]\!])$ into buy list
```

Figure 26: Overall CDA

```
Protocol $\Pi_{\mathsf{PSL}}(\mathcal{P}, ([\![name_0^b]\!], [\![b_0]\!], [\![q_0]\!]), \mathcal{S})$
 – Set $[\![w_0]\!] = [\![0]\!]$
 – For each $i = 1$ to $N$ do in parallel: $[\![w_i]\!] = \sum_{j=1}^{i} [\![s_j]\!]$
 – For each $i = 1$ to $N$ do in parallel:
   ○ $[\![z_i]\!] = \Pi_{\mathsf{LT}}([\![w_{i-1}]\!], [\![b_0]\!])$, $[\![z'_i]\!] = \Pi_{\mathsf{LT}}([\![p_i]\!], [\![q_0]\!] + 1)$
 – For $i = 1$ to $N$ do in parallel: $[\![f_i]\!] = \Pi_{\mathsf{mult}}([\![z_i]\!], [\![z'_i]\!])$
 – Reconstruct $f_i$'s and set $k = i$ such that $f_i = 1$ and $f_{i+1} =$
   $0$ for $i \in \{1, \ldots, N\}$. Else set $k = 0$.
 – for each $i = 1$ to $k-1$ do in parallel:
   ○ Reconstruct $([\![name_i^s]\!], [\![s_i]\!], [\![p_i]\!])$
 – $[\![s'_k]\!] = \Pi_{\mathsf{sel}}([\![s_k]\!], [\![b_0]\!] - [\![w_{k-1}]\!], [\![z_{k+1}]\!]^{\mathsf{B}})$
```

- Reconstruct $(\llbracket name^s_k \rrbracket, \llbracket s'_k \rrbracket, \llbracket p_k \rrbracket)$, set $\llbracket s_k \rrbracket = \llbracket s_k \rrbracket - \llbracket s'_k \rrbracket$
- Delete first $k-1$ elements from $\mathcal{S}$.

Figure 27: CDA matching phase: processing sell list

Protocol $\Pi_{\mathsf{Insert}}\left(\mathcal{P}, (\llbracket name^b_0 \rrbracket, \llbracket b_0 \rrbracket, \llbracket q_0 \rrbracket), \mathcal{B}\right)$

```
– Insert $(\llbracket 0 \rrbracket, \llbracket 0 \rrbracket, \llbracket 0 \rrbracket)$ to the end of $\mathcal{B}$
– Compute $\llbracket f_0 \rrbracket = \Pi_{\mathsf{LT}}(\llbracket q_0 \rrbracket, \llbracket q_1 \rrbracket))$
– For $i = 1$ to $M+1$ do
    ∘ $\llbracket f_i \rrbracket = \Pi_{\mathsf{LT}}(\llbracket q_0 \rrbracket, \llbracket q_i \rrbracket + 1))$
    ∘ $\llbracket f'_i \rrbracket = \Pi_{\mathsf{mult}}((1 - \llbracket f_i \rrbracket), \llbracket f_{i-1} \rrbracket)$
    ∘ $\llbracket f''_i \rrbracket = \Pi_{\mathsf{mult}}((1 - \llbracket f_i \rrbracket), (1 - \llbracket f'_i \rrbracket))$
– For $i = 1$ to $M+1$ do in parallel
    ∘ $\llbracket name'^b_i \rrbracket = \llbracket f_i \rrbracket \cdot \llbracket name^b_i \rrbracket + \llbracket f'_i \rrbracket \cdot \llbracket name^b_0 \rrbracket + \llbracket f''_i \rrbracket \cdot \llbracket name^b_{i-1} \rrbracket$
    ∘ $\llbracket b'_i \rrbracket = \llbracket f_i \rrbracket \cdot \llbracket b_i \rrbracket + \llbracket f'_i \rrbracket \cdot \llbracket b_0 \rrbracket + \llbracket f''_i \rrbracket \cdot \llbracket b_{i-1} \rrbracket$
    ∘ $\llbracket q'_i \rrbracket = \llbracket f_i \rrbracket \cdot \llbracket q_i \rrbracket + \llbracket f'_i \rrbracket \cdot \llbracket q_0 \rrbracket + \llbracket f''_i \rrbracket \cdot \llbracket q_{i-1} \rrbracket$
```

Figure 28: Obliviously inserting into buy list

F.2.2 Volume matching

Unlike the CDA algorithm, orders are processed at fixed intervals in volume-based matching, where matches are identified only based on the volume. Thus, a buy or sell order only comprises the client's name and the volume to be traded. Let b_i denote the units to be bought by client i and s_i denote the units to be sold. The buy orders and sell orders are stored as separate lists where orders are now sorted based on their time of arrival. Given that matching has to be performed only based on volume, either all the sell orders or all the buy orders are guaranteed to be satisfied. Hence, the algorithm proceeds by computing the total volume to be traded in each list and identifies the list that has a lesser volume. All orders in this list will be satisfied. At the end, the algorithm outputs updated volumes of the orders where b'_i or s'_i in the output indicates the number of units that were traded out of the original b_i or s_i units in the order. This sequential algorithm of [12] is also optimized to allow computations to be performed in parallel by maintaining additional book-keeping information in the work of [38]. The optimized algorithm for volume matching appears in Fig. 29.

Protocol $\Pi_{\mathsf{VM}}\left(\mathcal{P}, \{\llbracket s_i \rrbracket\}_{i=1}^N, \{\llbracket b_j \rrbracket\}_{j=1}^M\right)$

```
1. Compute $\llbracket S \rrbracket = \sum_{i=1}^N \llbracket s_i \rrbracket$ and $\llbracket B \rrbracket = \sum_{j=1}^M \llbracket b_j \rrbracket$
2. Compute $\llbracket f \rrbracket = \Pi_{\mathsf{LT}}(\llbracket S \rrbracket, \llbracket B \rrbracket)$
3. Set $\llbracket T \rrbracket = \Pi_{\mathsf{sel}}(\llbracket S \rrbracket, \llbracket B \rrbracket, \llbracket f \rrbracket)$, $\llbracket s_0 \rrbracket = 0$ and $\llbracket b_0 \rrbracket = 0$
4. For $i$ from 1 to $N$ do in parallel: $\llbracket L^s_i \rrbracket = \llbracket T \rrbracket - \sum_{j=0}^{i-1} \llbracket s_j \rrbracket$
5. For $i$ from 1 to $M$ do in parallel: $\llbracket L^b_i \rrbracket = \llbracket T \rrbracket - \sum_{j=0}^{i-1} \llbracket b_j \rrbracket$
6. For $i$ from 1 to $N$ do in parallel:
    ∘ $\llbracket z_1 \rrbracket = \Pi_{\mathsf{LT}}(\llbracket L^s_i \rrbracket, \llbracket 1 \rrbracket))$ and $\llbracket z_2 \rrbracket = \Pi_{\mathsf{LT}}(\llbracket L^s_i \rrbracket, \llbracket s_i \rrbracket)$
    ∘ $\llbracket s_i \rrbracket = ((\llbracket L^s_i \rrbracket - \llbracket s_i \rrbracket) \cdot \llbracket z_2 \rrbracket + \llbracket s_i \rrbracket) \cdot (1 - \llbracket z_1 \rrbracket)$
7. For $j$ from 1 to $M$ do in parallel:
    ∘ $\llbracket z_1 \rrbracket = \Pi_{\mathsf{LT}}(\llbracket L^b_j \rrbracket, \llbracket 1 \rrbracket)$ and $\llbracket z_2 \rrbracket = \Pi_{\mathsf{LT}}(\llbracket L^b_j \rrbracket, \llbracket b_j \rrbracket)$
    ∘ $\llbracket b_i \rrbracket = ((\llbracket L^b_j \rrbracket - \llbracket b_i \rrbracket) \cdot \llbracket z_2 \rrbracket + \llbracket b_j \rrbracket) \cdot (1 - \llbracket z_1 \rrbracket)$
8. Reconstruct $\llbracket s_i \rrbracket$ and $\llbracket b_j \rrbracket$ for all $i$ and $j$
```

Figure 29: Volume matching