

Fair and Efficient Helper-Aided MPC with Cheater Identification

Max Kamps

University of Amsterdam and
Vrije Universiteit Amsterdam, Netherlands
max.kamps@student.uva.nl

Protik Paul*

University of Tartu, Estonia
protik.pmax.paul@gmail.com

Divya Ravi

University of Amsterdam
d.ravi@uva.nl

Abstract

Secure multi-party computation (MPC) enables privacy-preserving tasks such as auctions and voting. In practice, more than half of the parties may collude, making security against a dishonest majority desirable. Unfortunately, dishonest-majority MPC suffers from high communication and can achieve only abort security. For instance, in an auction, an adversary may abort after learning the outcome and rerun the protocol with a higher bid.

Helper-aided MPC mitigates these limitations by introducing a semi-honest, non-colluding helper, a model that naturally fits settings with a central governing or coordinating entity and provides strong guarantees even when up to $n - 1$ parties are malicious. The model assumes that the adversary can either semi-honestly corrupt the helper or maliciously corrupt up to $n - 1$ parties (excluding the helper). While existing helper-aided protocols are fair i.e. either all parties receive the output or none do; they still allow the adversary to abort without any penalty and deny output to all parties. This behavior is undesirable in practice, as parties may incur the cost of computation without receiving the output.

In this work, we design efficient helper-aided MPC protocols that achieve *identifiable fairness*: either all parties obtain the output, or—if an abort occurs—no party receives the output, and the honest parties identify at least one *common cheater*. We propose two protocols, Alhena for synchronous networks and Wasat for asynchronous networks which provide the stronger guarantee of identifiable fairness compared to the state-of-the-art fair protocols Asterisk (Kamarkar et al. IEEE S&P 2024) and Castor (Kamarkar et al. IEEE S&P 2026) in the respective network settings. Our experiments show that, in the online phase, Alhena and Wasat achieves speedups of $4 - 12.5\times$ and $2.5 - 5\times$ over Asterisk and Castor respectively. Furthermore, in the preprocessing phase, Wasat exhibits speedup of $\approx 3\times$ over Castor. Notably, our protocols are designed in such a way that the online performance remains unaffected by active misbehaviour.

1. Introduction

As data processing continues to scale and concerns about data privacy become increasingly prominent, Secure Multi-Party Computation (MPC) [19], [36], [63] has emerged as a key technology for enabling collaborative computation while preserving the privacy of participants' inputs. The most widely studied MPC model assumes that all parties are mutually distrusting and that a centralized adversary corrupts up to a threshold t parties maliciously, meaning these parties can deviate arbitrarily from the protocol guidelines. In its strongest form, this model allows for up to $t = n - 1$ malicious parties among n participants.

While this setting indeed captures a very strong corruption model, it can be overly conservative for many practical applications. Accommodating such a powerful corruption model often leads to significant efficiency overheads and restricts the types of guarantees that can be achieved. For instance, it is well known that desirable properties such as *fairness* (where either all parties receive the output or none do) and *guaranteed output delivery* (a.k.a. GOD, where all honest parties receive the output no matter how the adversary behaves) are impossible when a majority of parties are corrupt [26].

These limitations have motivated an alternative trust model, known as Helper-Aided MPC [47], which incorporates an additional semi-honest (honest-but-curious) party, denoted HP, that acts as a designated helper. The helper typically has no input or output but is trusted to execute the computation correctly while ensuring that participants' data remains

*This work was done while at the ENCRYPTO group, Technical University of Darmstadt, Germany.

confidential, even from the HP itself. The adversary can either semi-honestly corrupt the helper or maliciously corrupt up to $n - 1$ parties (excluding the helper). Karmakar et al. [47] highlight the practical relevance of helper-aided MPC through primary financial applications. They design privacy-preserving dark pools [2], [16], which allow institutional investors to trade massive volumes of financial instruments confidentially. Here, the regulatory body governing the pool’s operational integrity—the Securities and Exchange Commission (SEC) [16]—is modeled as a semi-honest, non-colluding HP. Another relevant application is privacy-preserving inventory matching [57], where buyers and sellers of financial stocks are paired so that related assets can be traded without disrupting market prices for either client. In this case, the central institution matching buyers and sellers (such as the bank) can act as the non-colluding HP.

Prior work [47] has shown that, in this helper-aided MPC model, it is possible to design fair MPC protocols in the dishonest-majority setting that achieve substantial efficiency improvements over standard dishonest-majority protocols. In a follow-up work, [46] designed efficient asynchronous MPC protocols that are better suited to real-world networks, such as the Internet, where message delays are inherent. Their results demonstrate that the helper-aided MPC model enables strong guarantees such as fairness as well as significantly improved corruption resilience compared to asynchronous MPC in the standard model by improving the tolerated number of corruptions from $n/3$ to $n - 1$.

In existing helper-aided MPC protocols [46], [47], the primary security goal is fairness. However, a corrupt adversary can deliberately cause the computation to abort, preventing all parties (including the adversary itself) from receiving the output. While this does not violate fairness, it enables denial-of-service attacks where malicious parties can disrupt the computation without any consequences whatsoever. This is undesirable in practical applications as the honest parties have to bear the cost of the computation without getting the output. This approach is wasteful and may demotivate honest parties from participating. The most desirable goal would be to guarantee output regardless of adversarial behaviour. Towards this, [46], [47] discuss an extension of their fair protocols to GOD where, if the HP is identified as honest, it is treated as a trusted third party to which all parties simply send their inputs in the clear. Though this does not violate the security guarantees of MPC (which protect honest parties’ inputs from the adversary but not from other honest parties), this solution is rather unsatisfactory. It goes against the core goal of MPC of eliminating the need for a fully trusted third party.

Thus, prior work [46], [47] posed the question: *can we design a GOD protocol without leaking private information to HP?* Our work aims to answer this open question. One possible approach is to rerun the protocol until the output is obtained. However, this does not work directly with existing protocols, as the adversary can repeatedly abort the protocol, resulting in non-termination. This issue could be addressed by designing a protocol that allows honest parties to identify the cheaters (or at least one common cheater) responsible for an abort. Identifying cheating parties would enable countermeasures such as penalties or exclusion from future computations, thereby deterring misbehavior. In the standard MPC setting, this notion is formalized as identifiable abort [3], [43].

It is important to note that fairness and identifiable abort are orthogonal properties. Under identifiable abort, the adversary may still learn the output while honest parties do not; in contrast, fairness guarantees that either all parties receive the output or none do, but provides no mechanism to identify cheaters in the event of an abort.

In practical applications, identifiable abort is useful for detecting cheating parties, but it has limitations. For example, consider an MPC protocol for auctions: if a protocol run aborts but the adversary learns the highest bid, this information could be exploited in a subsequent run on behalf of another corrupt party that was not caught in the earlier execution. Thus, even if the security guarantees of prior protocols were strengthened to identifiable abort and common cheaters were eliminated after each abort, significant challenges would remain in achieving GOD. Firstly, the cost of the protocol increases substantially due to multiple reruns. Second, rerunning an identifiable-abort protocol is not secure, as the output may be revealed to the adversary. Therefore, along with identifiable abort, fairness is also a necessary requirement.

This naturally raises the question: *Can we design practically efficient helper-aided MPC protocols that maintain fairness while also identifying cheating parties in the event of an abort?*

1.1. Our Contribution

We consider the same helper-aided MPC model as in [46], [47] where the adversary can either corrupt up to $n - 1$ among the n regular parties maliciously or corrupt the helper party HP in a semi-honest fashion. Recall that in this model, the HP is assumed to be non-colluding with the parties.

In this work, we answer the question affirmatively by designing helper-aided MPC protocols that achieve *identifiable fairness* (referred to as fairness with identifiable abort in [29]) in both synchronous and asynchronous network settings. Identifiable fairness strengthens standard fairness: not only is fairness preserved, but in the event of an abort, all honest regular parties also agree on the identity of at least one cheating party. Notably, our notion requires only the regular parties—not the helper HP—to identify the misbehaving party¹. This is particularly appropriate in scenarios where the helper has no input or output and serves solely to assist in the computation.

1. This notion is in a similar vein to the notion of identifiable secret sharing introduced in [41] for the outsourced setting with a dishonest majority of clients and an external server (analogous to the HP). In that setting, it is impossible for the server to distinguish cheaters from honest clients, whereas the honest clients are able to agree on which clients are corrupted.

Our constructions follow the preprocessing–online paradigm and achieve identifiable fairness. This can be upgraded to GOD by rerunning the protocol after eliminating the corrupt parties identified in each abort, as shown by the compiler of [29].

- In the synchronous network setting, we propose the protocol Alhena, which achieves GOD whenever the offline phase completes successfully, and if the preprocessing fails, all honest parties identify at least one corrupt party. Compared to the state-of-the-art Asterisk protocol [47], which ensures fairness (without cheater identification), Alhena provides the stronger guarantee of identifiable fairness and has a faster online phase. Our experiments show that Alhena outperforms Asterisk by a factor of $4 - 12.5\times$ in the online phase. The overhead of cheater identification is incurred only during the preprocessing phase.
- In the asynchronous network setting, we propose the protocol Wasat, which achieves identifiable fairness. Compared to the state-of-the-art asynchronous MPC protocol Castor [46] which achieves fairness (without cheater identification), Wasat provides identifiable fairness and has both a faster preprocessing and online phase. In the online phase, Wasat outperforms Castor by a factor of $2.5 - 5\times$, while a speedup of $\approx 3\times$ is achieved during the preprocessing. Unlike Alhena, Wasat suffers from the problem of input deprivation, which is inherent in asynchronous network (also in helper-aided setting, as shown in [46]). This means that the computation may ignore upto t honest parties’ inputs. However, similar to the asynchronous protocols in [46], the design of our protocol is such that all honest parties’ inputs whose messages were delivered in the first round would be included in the computation. Therefore, in practice, one could imagine the protocol specifying a generous waiting time just for the first round, after which parties wait only for $n - t$ messages in each subsequent round.

Lastly, we note that, to the best of our knowledge, existing asynchronous MPC protocols either provide security with abort or guaranteed output delivery, but are not specifically designed to achieve cheater identification as a primary goal (although some relevant building blocks exist, such as asynchronous secret sharing with identified abort developed in [6]). We find it noteworthy that the presence of a semi-honest helper can be leveraged to introduce identifiability into asynchronous protocols: because parties can afford to wait for and trust messages from the helper, reaching agreement on the identity of a cheater becomes significantly easier than in standard asynchronous settings where messages can be arbitrarily delayed.

1.2. Technical Overview

We begin by highlighting the challenges involved in identifying cheaters in existing helper-aided MPC protocols. At a high level, the fair helper-aided protocol Asterisk [47] follows the preprocessing–online paradigm and proceeds as follows: In the preprocessing phase, a helper HP generates random masks for each wire and authenticates them using a single global MAC key Δ . Each party receives additive shares of the masks, additive shares of the corresponding MAC tags and additive shares of the key Δ . During the online phase, parties secret-share their inputs and evaluate the circuit gate-by-gate on masked values. For each gate, they locally compute additive shares of the masked intermediate values, which are then reconstructed by a designated party P_{King} . Correctness is enforced via a final MAC verification on these reconstructed masked values using the parties’ additive shares of the corresponding tags. However, because authentication applies only to the reconstructed masked values and not to the individual additive shares contributed by each party, a MAC failure detects misbehavior but does not identify the deviating party. Consequently, the protocol does not provide cheater identification. Likewise, the helper-aided protocols in [46] do not support cheater identification, as they also authenticate values instead of individual shares. Authenticating each additive share, as done in MPC protocols in the standard model (without a helper) [8] would enable cheater identification, but at the cost of significant communication and computational overhead.

Instead, we explore whether the presence of the helper can be leveraged more effectively. Our key idea is to switch to a two-party sharing structure in which all regular parties hold a common authenticated share, while the complementary share is held by the helper HP. Since one share is held by HP, it does not require verification due to the semi-honest nature of HP. As a result, only a single share needs to be verified. At a high-level, cheater detection relies on the two key concepts: (i) redundancy of the share among the parties, which ensures any misbehaviour will get detected, (ii) honest behaviour of the HP, which helps all the honest parties identify the misbehaving party. We elaborate on this idea below.

Since all our protocol steps are deterministic and all regular parties are supposed to do identical computation using their common share, every honest party sends the same message to the HP. In case inconsistent versions are received, the HP can simply relay this information. This suffices for honest parties to detect corrupt parties, as they can identify the parties who sent inconsistent messages as being corrupt. Notably, this idea extends naturally to the asynchronous setting. In this case, the HP relays the first message it receives from any party, including the sender’s identity. If this message is inconsistent, honest parties simply detect the sender as corrupt.

We note that this approach provides several additional advantages. First, it allows us to avoid techniques like triple sacrifice, used in Castor [46], to verify the correctness of preprocessing data, which incurs substantial communication overhead. Second, this approach enables guaranteed output delivery in the online phase. For each intermediate gate, the HP receives from the parties the (common) share of the masked intermediate value together with its corresponding tag. The HP

reconstructs the masked intermediate value only after successfully verifying the tag. The unforgeability of the MAC ensures that the reconstructed value is correct. Since the HP is guaranteed to receive at least one correct share from the honest party, it can safely wait until a share with a verifying tag is received; ensuring both correctness and robust computation at every step in the online phase.

1.3. Related Work

As mentioned previously, prior works in the helper-aided MPC model include [47] and [46], which design efficient fair MPC protocols in synchronous and asynchronous networks, respectively. Feasibility results in this model (related to minimal use of the HP) were investigated in [41], [44].

Alternative models assume the presence of an external trusted or semi-trusted entity (e.g., hardware tokens), as explored in works such as [4], [15], [17], [18], [20], [21], [32], [38], [48], among others. The work of [53] studies an assisted MPC computation model in which the helper’s semi-honest behavior is probabilistic, allowing the helper to be malicious with non-zero probability.

In the standard model, MPC with identifiable abort was introduced in [3]. Since then, there has been a rich line of work focused on improving the efficiency of MPC protocols with cheater identification, including (but not limited to) [7], [8], [9], [28], [42], [58]. Arithmetic circuit based identifiable abort protocols [8] require authenticating each individual share of a secret rather than authenticating the secret as a whole, which incurs significant communication overhead. To mitigate this cost, [7] introduces pairwise MACs to enable cheater identification. These works are designed for the synchronous setting and tolerate a dishonest majority of corruptions. In contrast, identifiability in asynchronous networks remains far less explored. Notably, [6] studies asynchronous secret sharing with identified abort.

On the other hand, the works of [6], [22], [51], [54] focus on improving the efficiency of asynchronous MPC protocols in the standard model under the corruption threshold $t < n/3$, which is known to be optimal. The literature also contains a broad range of asynchronous MPC (AMPC) protocols. Several works [1], [12], [25], [33], [37], [45], [55], [56] provide information-theoretic constructions. Among these, [1] achieves linear communication of $O(n)$ elements per multiplication gate, at the cost of an additive $O(n^5)$ -element overhead.

Other approaches [13], [23], [24], [27], [39], [40], [60] reduce communication by relying on cryptographic assumptions, including additive homomorphic encryption, homomorphic commitments, somewhat homomorphic encryption, and fully homomorphic encryption. In particular, [23] achieves linear communication using additive homomorphic encryption under the atomic-send model for $t < (1 - \epsilon)n/3$. More recently, [52] presents an AMPC protocol with worst-case communication of $O(n^2)$ elements per multiplication gate and $O(n)$ elements in the optimistic case, using only lightweight cryptography. Similarly, [5] proposes a lightweight AMPC protocol with linear communication per multiplication gate. All of these protocols, however, require an honest-majority threshold of $t < n/3$.

It is well known (see, e.g., [46]) that even computationally secure MPC protocols providing only abort security must satisfy $n \geq 3t + 1$. The work of [10] circumvents this bound by assuming an input-distribution oracle (realized via a single synchronous broadcast round) and employing threshold homomorphic encryption. Building on the techniques of [31], their protocol achieves MPC with $n \geq 2t + 1$. This relaxation comes at a cost: the circuit must be evaluated n times—once with each party acting as a distinguished party (analogous to the P_{King} of [34])—and the protocol incurs significant computational overhead due to threshold encryption. These limitations motivate the search for MPC protocols that achieve identifiable abort/fairness, that can be used to achieve GOD [29], in both synchronous as well as asynchronous setting while avoiding honest-majority assumptions, strong setup, or prohibitive communication overhead.

In Table 1, we provide an asymptotic comparison with state-of-the-art protocols. We consider the dishonest-majority protocol TurboSpeedz [11], which achieves an improved online phase via function-dependent preprocessing; however, it only offers abort security. Baum et al. [7] present the state-of-the-art identifiable abort construction, which incurs a significantly higher communication cost in both the online and preprocessing phases compared to ours. Furthermore, in contrast to prior helper-aided protocols, we improve the security guarantees and/or the communication costs in the same setting.

2. Preliminaries

2.1. Security Definitions

We prove the security of our protocols in the standard real world/ideal world paradigm [14], [35] where 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 which is secure by definition (in the presence of an incorruptible trusted party).

2.1.1. The real world. An n -party protocol π with parties $\mathcal{P} = \{P_1, \dots, P_n\}$ is an n -tuple of probabilistic polynomial-time (PPT) interactive Turing machines (ITMs). We let $\mathcal{A}$ denote a special PPT ITM that represents the adversary and is initialized

Protocol	Network	Guarantee	Preprocessing cost	Online cost
TurboSpeedz [11]	Synchronous	Abort	$\mathcal{O}(n^2) \times \text{OLE}$	$2n$
Asterisk [47]	Synchronous	Fair	3	$2n$
Pollux [46]	Asynchronous	Fair	$2 \times (n - t)$	$2n$
Castor [46]	Asynchronous	Fair	$12n \times \text{OLE}$	$2n$
[7]	Synchronous	Id-Abort	$n^2 \times \text{OLE} + \mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Alhena	Synchronous	Id-Fair	$4 \times \text{OLE}$	$2n$
Wasat	Asynchronous	Id-Fair	$4n \times \text{OLE}$	$2n$

The preprocessing of [11] can be instantiated by OLE [49] or somewhat homomorphic encryption (SHE) [50]. Comparison purpose we consider OT-based preprocessing of [11].

TABLE 1: Comparing efficient MPC protocols. Preprocessing cost reflects the cost per multiplication in the preprocessing phase, in terms of building blocks plus total communication in field elements.

with an input that contains the identities of the corrupt parties, their respective private inputs, and an auxiliary input. During the execution of the protocol over authenticated secure point-to-point communication channels, maliciously corrupt parties receive arbitrary instructions from adversary $\mathcal{A}$, while honest parties and semi-honestly corrupt parties faithfully follow the instructions of the protocol. We consider the adversary $\mathcal{A}$ to be static and rushing, that is, 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, 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 2.1 (Real-world execution). *Let π be an n -party protocol amongst $\{P_1, \dots, P_n\}$ computing an n -party function $f : (\{0, 1\}^*)^n \rightarrow (\{0, 1\}^*)^n$ and let $C \subset [1, n] \cup \{HP\}$ denote the set of indices of the corrupted parties. The execution of π under $(\mathcal{A}, C)$ in the real world, on input vector $\vec{x} = (x_1, \dots, x_n)$, auxiliary input aux and security parameter κ , denoted $\text{real}_{\pi, C, \mathcal{A}(\text{aux})}(\vec{x}, \kappa)$, is defined as the output vector of $P_1, \dots, P_n$ and $\mathcal{A}(\text{aux})$ resulting from the protocol interaction.*

2.1.2. The ideal world. We describe ideal world execution with identifiable fairness below. The ideal functionalities for identifiable abort and guaranteed output delivery appear in Appendix A.

Definition 2.2 (Ideal Computation with Identifiable Fairness). *Let $f : (\{0, 1\}^*)^n \rightarrow (\{0, 1\}^*)^n$ be an n -party function and let $C \subset [1, n] \cup \{HP\}$ denote the set of all corrupt parties. Then, the joint ideal execution of f under $(\mathcal{S}, C)$ on input vector $\vec{x} = (x_1, \dots, x_n)$, auxiliary input aux to $\mathcal{S}$ and security parameter κ , denoted $\text{ideal}_{f, C, \mathcal{S}(\text{aux})}(\vec{x}, \kappa)$, is defined as the output vector of $P_1, \dots, P_n$ 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) If there exists an $i \in C$ such that $x'_i = \perp$ or outside the domain, then set $y_j = (\text{abort}, i)$ for all $j \in [n]$. Else compute $(y_1, \dots, y_n) = f(x'_1, \dots, x'_n)$.
- 3) Trusted party sends y_j to every P_j , for all $j \in [1, n]$.
- 4) Outputs: Honest parties always output the message received from the trusted party, while corrupt parties output nothing. The simulator $\mathcal{S}$ outputs an arbitrary function of the initial inputs $\{x_i\}_{i \in C}$, the messages received by the corrupt parties from the trusted party and its auxiliary input.

In this work, we consider the non-colluding security notion of [44], 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 t -size subset of the n parties $\mathcal{P} = \{P_1, \dots, P_n\}$ maliciously *or* the HP passively (i.e. in a semi-honest manner). The formal definition appears below.

Definition 2.3 (Non-colluding security). *Let $f : (\{0, 1\}^*)^n \rightarrow (\{0, 1\}^*)^n$ be an n -party function. A protocol Π securely computes the function f in the non-colluding model with identifiable abort / identifiable fairness / guaranteed output delivery security if for any adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that for every C where either (a) $C \subset [1, n]$ and $\|C\| = t$ malicious corruptions or (b) $C = \text{HP}$ semi-honest corruption, we have*

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

In the above, $\text{ideal}_{f, C, S(\text{aux})}(\vec{x}, \kappa)$ refers to the ideal computation defined, according to the corresponding guarantee (i.e., identifiable abort, identifiable fairness, or guaranteed output delivery). Note that the corruption is non-simultaneous. Therefore we need the above indistinguishability to hold in both the corruption cases. Finally, 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).

2.2. Notation and Background

Shared key setup. Our protocols require a shared key setup. We denote by k_{all} the PRF key shared among all parties in $\mathcal{P}$, k_i the PRF key shared between P_i and HP, and by $k_{\mathcal{P}}$ the PRF key shared among parties in $\mathcal{P}$ but hidden from HP. We refer to [47] for details of realization of such a setup for the synchronous setting and [46] for the asynchronous setting.

Information-Theoretic MAC. Our constructions use a one-time information-theoretic message authentication code. This is defined by a secret key $\Delta \leftarrow \mathbb{F}$ (where $\mathbb{F}$ denotes a field). Given a message $m \in \mathbb{F}$, a tag is generated by sampling $r \leftarrow \mathbb{F}$ and computing $t := \Delta \cdot m - r$. Verification accepts (m, t, r) if and only if $t + r = \Delta \cdot m$. This MAC is one-time unforgeable with information-theoretic security, provided that r is freshly sampled for each message.

Sharing Semantics. Our protocols use a 2-party sharing structure, where one of the shares is held by (all) regular parties and the complementary share is held by the HP. Below we present the notations for secret sharing.

- $[\cdot]$: We define secret sharing of a value v denoted by $[v]$ as follows. Let $v = [v]_0 + [v]_1$, where a party P'_i ($i \in [n]$) share is $[v]_0$, and the HP share is $[v]_1$.
- $\langle \cdot \rangle$: We consider the authenticated secret sharing denoted by $\langle \cdot \rangle = (\langle \cdot \rangle_{\mathcal{P}}, \langle \cdot \rangle_{\text{HP}})$, which is generated in the preprocessing phase and defined as follows. Let $([v]_0, [v]_1)$ be an $[\cdot]$ -sharing of v . We define the MAC on $[v]_0$ as $t_{[v]_0} = \Delta \cdot [v]_0 - r_{[v]_0}$, where Δ is a global MAC key known only to the HP. We set the share $\langle v \rangle_{\mathcal{P}}$ of the $\langle v \rangle$ -sharing held by each P_i to be $\langle v \rangle_{\mathcal{P}} = ([v]_0, t_{[v]_0})$. The share $\langle v \rangle_{\text{HP}}$ of the $\langle v \rangle$ -sharing held by the HP is defined as $\langle v \rangle_{\text{HP}} = ([v]_1, r_{[v]_0})$.
- $\llbracket \cdot \rrbracket$: We define $\llbracket v \rrbracket = (\llbracket v \rrbracket_{\mathcal{P}}, \llbracket v \rrbracket_{\text{HP}})$ as the authenticated masked sharing of a secret v . Let $\langle \delta_v \rangle = (\langle \delta_v \rangle_{\mathcal{P}}, \langle \delta_v \rangle_{\text{HP}})$ be an authenticated sharing of the mask δ_v generated in the preprocessing phase. Let $m_v = v + \delta_v$ be the masked secret computed in the online phase. All parties including the HP hold m_v in the clear. Let $\llbracket v \rrbracket_{\mathcal{P}} = (m_v, \langle \delta_v \rangle_{\mathcal{P}})$ be the share of each P_i and let $\llbracket v \rrbracket_{\text{HP}} = (m_v, \langle \delta_v \rangle_{\text{HP}})$ be the share of the HP.

Note: When we write that a value is $\langle \cdot \rangle$ -shared or $\llbracket \cdot \rrbracket$ -shared, we mean that the parties hold $\langle \delta_v \rangle_{\mathcal{P}}$ or $\llbracket v \rrbracket_{\mathcal{P}}$, while the HP holds $\langle \delta_v \rangle_{\text{HP}}$ or $\llbracket v \rrbracket_{\text{HP}}$, respectively.

Oblivious Linear Evaluation. Our constructions rely on the building block of Oblivious Linear Evaluation (OLE), which is a 2-party functionality where the sender S holds inputs a and c , and the receiver R holds input b . S receives no output, while R obtains $d = a \cdot b - c$. As shown in SPDZ2k [30], this can be efficiently realized using oblivious transfer.

3. Synchronous Protocol Alhena

In this section, we describe our synchronous protocol Alhena that achieves identifiable fairness in the helper-aided setting, where either $n - 1$ among the n regular parties are maliciously corrupt or HP is semi-honest corrupt. The protocol is in the preprocessing-online paradigm, where the preprocessing phase achieves identifiable abort and the online phase achieves GOD. Overall, the protocol achieves identifiable fairness as the preprocessing phase does not involve any computation related to parties' inputs; therefore any abort during preprocessing preserves fairness.

Below, we first introduce an important building block used in our preprocessing phase, and then describe the preprocessing and online phases.

3.1. OLE with Identifiable Abort

Functionality $\mathcal{F}_{\text{MS-OLE-IA}}$

- **Initialize:** On input $(\text{Init}, S_i, \mathbb{F})$ from S_i (for all $i \in [n]$) and $(\text{Init}, R, \mathbb{F})$ from R , store $\mathbb{F}$.
- **Input:** On input (Input, a_i, c_i) from S_i , for all $i \in [n]$ and (Input, b) from R : store $(a = a_\ell, b, c = c_\ell)$ where ℓ is the lexicographically smallest index in $\mathcal{P} \setminus C$.
- **Output:** On input (Output) from all the parties, send **ready** to the adversary. Wait for an input from the adversary
 - If this is 'deliver', then send $(a \cdot b - c)$ to R , and send **(success)** to each $S_i \in \mathcal{P} \setminus C$.

- Else, if it received ‘(abort, i)’ where $i \in C$, then send $\perp$ to R and $(\perp, i)$ to all $S_i \in \mathcal{P} \setminus C$.

Figure 1: Ideal functionality for Multi-Sender OLE with Identifiable Abort

One of the main building blocks we use in our protocol is a Multi-Sender Oblivious Linear Evaluation with Identifiable Abort (MS-OLE-IA). Informally, MS-OLE-IA is a variant of OLE adapted to the helper-aided setting where the helper HP acts as the receiver R (with input b) and the parties in $\mathcal{P} = \{P_1, \dots, P_n\}$ act as senders with common input (a, c) . The receiver HP either obtains the correct output $(a \cdot b - c)$ or $\perp$. In the latter case, it is guaranteed that the honest parties in $\mathcal{P}$ learn the identity of at least one corrupt party. We present the corresponding ideal functionality $\mathcal{F}_{\text{MS-OLE-IA}}$ below.

Next, we present a protocol $\Pi_{\text{MS-OLE-IA}}$ that compiles a 2-party maliciously secure OLE protocol Π_{OLE} to a multi-party protocol that realizes $\mathcal{F}_{\text{MS-OLE-IA}}$. At a high level, a standard two-party OLE instance is executed between a designated party P_d , acting as the sender, and HP acting as the receiver. By requiring P_d to derive its randomness from the shared key $k_{\mathcal{P}}$, the resulting protocol transcript becomes verifiable by the honest senders, enabling detection and identification of any cheating behavior.

In the implementation, we optimize communication by making the HP send a hash of the sender’s messages. This suffices for the verification.

Protocol $\Pi_{\text{MS-OLE-IA}}$

Input:

- $\forall i \in [n]$, P_i has input a, c . let P_d denote a designated party.
- HP has input b .

Output: HP receives $a \cdot b - c$ (in case of no abort).

Building Block: 2-party Π_{OLE} (instantiated using [62]) and shared key setup generated.

Protocol:

- The designated party P_d and HP participate in an instance of Π_{OLE} : P_d participates as sender using its input (a, c) and randomness derived from $k_{\mathcal{P}}$. HP participates as receiver using input b . Let $\tau = \tau_S || \tau_R$ denote the transcript of this instance, where τ_S comprises of the messages sent by P_d and τ_R comprises of the messages sent by HP. Let y denote the output of Π_{OLE} obtained by HP.
- HP sends τ to all parties $P_i \in \mathcal{P} \setminus P_d$.
- Each party P_i ($i \in [n] \setminus d$) does the following:
 - The transcript received $\tau = \tau_S || \tau_R$ is first parsed into messages corresponding to individual communication rounds.
 - Recompute the round-by-round transcript τ'_S comprising of sender messages in Π_{OLE} using input (a, c) , randomness derived from $k_{\mathcal{P}}$ and τ_R received from HP. More specifically, the sender’s message in Round r is recomputed using the receiver messages from all prior rounds, i.e. $\tau_R^{<r}$ as per the protocol specification of Π_{OLE} .
 - During recomputation, if any round of τ'_S does not match the corresponding round in τ_S , send abort to HP and terminate with output $(\perp, d)$ identifying P_d as cheater. Else send success to HP.
- HP does the following: If there exists $i^* \in [n]$ from whom HP received abort, output $\perp$ and send (abort, i^*) to each $P_i \in \mathcal{P}$. Else output y .
- Each $P_i \in \mathcal{P}$ does the following: Output $(\perp, i^*)$ identifying P_{i^*} as cheater if received (abort, i^*) from HP.

Figure 2: Protocol realizing $\mathcal{F}_{\text{MS-OLE-IA}}$

Lemma 3.1. $\Pi_{\text{MS-OLE-IA}}$ realizes $\mathcal{F}_{\text{MS-OLE-IA}}$.

Proof. To prove this, we must argue that if $\Pi_{\text{MS-OLE-IA}}$ results in HP outputting $\perp$, then the honest parties agree on a corrupt party as the cheater. Else, the output obtained by the HP must be correct. We have the following two cases based on if the designated sender P_d misbehaves or not.

- 1) Suppose P_d follows the protocol specifications honestly during Π_{OLE} . Then, τ'_S recomputed by honest sender will be identical to τ_S received from the HP. Therefore, none of the honest parties will send abort to HP. We now have two possibilities based on whether any corrupt party P_{i^*} sends abort to the HP or not. If this occurs, then all honest parties agree on the corrupt P_{i^*} as the cheater. Else, if no party sends abort to the HP, the HP obtains the correct output.
- 2) Suppose P_d does not follow the protocol specifications. In this case, τ'_S recomputed by honest sender will not be consistent with τ_S received from the HP. Subsequently, all honest parties terminate identifying the corrupt P_d as the cheater.

3.2. Preprocessing Phase

We describe the preprocessing phase of our protocol Alhena, which provides security with identifiable abort: either the preprocessing completes successfully, or the parties in $\mathcal{P}$ agree on a common cheater. We first describe sub-protocols before presenting the final protocol $\Pi_{\text{Alhena}}^{\text{Preproc}}$ in Fig. 7

Generating the shares of a random value v . To generate the sharing $[v]$ of a random value v , the parties and HP use the shared key setup in the following way.

Protocol $\Pi_{\text{RandRSS}}^{\text{Alhena}}$

Input: $\forall i \in [n], P_i$ holds a PRF key $k_{\mathcal{P}}$.

Output: The parties output common $[v]_{\mathcal{P}}$, and HP outputs $[v]_{\text{HP}}$.

Protocol:

- All $P_i \in \mathcal{P}$ sample and output $[v]_{\mathcal{P}}$ using $k_{\mathcal{P}}$.
- HP randomly samples and outputs $[v]_{\text{HP}}$.

Figure 3: Generating replicated shares of a random value

Generating the shares of a random value v known to P_d . With the help of the pairwise keys between each party and HP, parties can create a sharing $[v]$ of a random value v that is revealed to a specific party. This sub-protocol $\Pi_{\text{RSS}}^{\text{Alhena}}$ appears below.

Authenticating a shared value v . After generating a sharing $[v]$ of a random value v (Fig. 3), the parties and the HP jointly generate an authentication tag $t_{[v]_{\mathcal{P}}} = \Delta \cdot [v]_{\mathcal{P}} - r_{[v]_{\mathcal{P}}}$ on the parties' share $[v]_{\mathcal{P}}$. The tag $t_{[v]_{\mathcal{P}}}$ and $(\Delta, r_{[v]_{\mathcal{P}}})$ is known to all parties (except HP) and HP respectively. Note, that Δ stays the same for all tags, whereas $r_{[v]_{\mathcal{P}}}$ is freshly randomly sampled for each individual tag. The tag is generated using $\mathcal{F}_{\text{MS-OLE-IA}}$ as follows: The parties act as sender with common input $([v]_{\mathcal{P}}, t_{[v]_{\mathcal{P}}})$ where $t_{[v]_{\mathcal{P}}}$ is derived using the shared key $k_{\mathcal{P}}$; while HP acts as receiver with input (Δ) . Note that this enables HP to receive $r_{[v]_{\mathcal{P}}} = \Delta \cdot [v]_{\mathcal{P}} - t_{[v]_{\mathcal{P}}}$ as output of the OLE, as required. Note that based on the guarantees of $\mathcal{F}_{\text{MS-OLE-IA}}$, if HP does not receive the output, then parties in $\mathcal{P}$ would identify a common cheater. The formal protocol appears below.

Protocol $\Pi_{\text{RSS}}^{\text{Alhena}}$

Input: $\forall i \in [n], P_i$ holds PRF keys $\{k_{\mathcal{P}}, k_i\}$, and HP holds $\{k_i\}_{i \in [n]}$.

Output: The parties output common $[v]_{\mathcal{P}}$, HP outputs $[v]_{\text{HP}}$ and P_d outputs v

Protocol:

- All $P_i \in \mathcal{P}$ sample and output $[v]_{\mathcal{P}}$ using $k_{\mathcal{P}}$.
- HP samples and outputs $[v]_{\text{HP}}$ using k_d .
- P_d samples $[v]_{\mathcal{P}}$ using $k_{\mathcal{P}}$ and $[v]_{\text{HP}}$ using k_d and computes $v = [v]_{\mathcal{P}} + [v]_{\text{HP}}$.

Figure 4: Generating replicated shares of a value known to P_d

Protocol $\Pi_{\text{SSAuth}}^{\text{Alhena}}$ in $\mathcal{F}_{\text{MS-OLE-IA}}$ -hybrid model

Input: $\forall i \in [n], P_i$ holds $[v]_{\mathcal{P}}$, and HP holds $[v]_{\text{HP}}$.

Output: The parties output $\langle v \rangle_{\mathcal{P}}$, and HP outputs $\langle v \rangle_{\text{HP}}$.

Protocol:

If $P = \text{HP}$, it does the following:

- Invoke $\mathcal{F}_{\text{MS-OLE-IA}}$ as receiver with input Δ .
- Receive $\perp$ or $\langle v \rangle_{\text{HP}} = ([v]_{\text{HP}}, r_{[v]_{\mathcal{P}}})$ as output.

If $P \neq \text{HP}$, it does the following:

- Sample random value $t_{[v]_{\mathcal{P}}}$ using $k_{\mathcal{P}}$.
- Invoke $\mathcal{F}_{\text{MS-OLE-IA}}$ as sender with input $([v]_{\mathcal{P}}, t_{[v]_{\mathcal{P}}})$.
- If $(\perp, i^*)$ was received as output of $\mathcal{F}_{\text{MS-OLE-IA}}$, abort identifying P_{i^*} as cheater. Else output $\langle v \rangle_{\mathcal{P}} = ([v]_{\mathcal{P}}, t_{[v]_{\mathcal{P}}})$.

Figure 5: Generating authenticated replicated shares of $[v]$

Multiplication of two $[\cdot]$ -shared values. Given additively shared inputs $[x]$ and $[y]$ (generated in (Fig. 3)), the target is to compute $[xy] = [x] \cdot [y]$. In the protocol Π_{MultSS} (Fig. 6), the parties and HP jointly compute the four terms: $[x]_{\mathcal{P}} \cdot [y]_{\mathcal{P}}$, $[x]_{\mathcal{P}} \cdot [y]_{\text{HP}}$, $[x]_{\text{HP}} \cdot [y]_{\mathcal{P}}$, $[x]_{\text{HP}} \cdot [y]_{\text{HP}}$ to obtain:

$$[x \cdot y] = [x]_{\mathcal{P}} \cdot [y]_{\mathcal{P}} + [x]_{\mathcal{P}} \cdot [y]_{\text{HP}} + [x]_{\text{HP}} \cdot [y]_{\mathcal{P}} + [x]_{\text{HP}} \cdot [y]_{\text{HP}}$$

where $[x \cdot y]$ is shared among the parties and HP. Each party in $\mathcal{P}$ can locally compute the first term $[x]_{\mathcal{P}} \cdot [y]_{\mathcal{P}}$ without interaction, and similarly, HP can compute $[x]_{\text{HP}} \cdot [y]_{\text{HP}}$ independently. However, to compute the cross terms involving

shares from both HP and the other parties (namely, $[x]_{\mathcal{P}} \cdot [y]_{\text{HP}}$ and $[x]_{\text{HP}} \cdot [y]_{\mathcal{P}}$), we use two instances of $\mathcal{F}_{\text{MS-OLE-IA}}$ for the parties in $\mathcal{P}$ and HP to obtain additive shares of these cross terms.

In more detail, to compute $[x]_{\mathcal{P}} \cdot [y]_{\text{HP}}$, $\mathcal{F}_{\text{MS-OLE-IA}}$ is invoked by parties in $\mathcal{P}$ acting as senders with common input $([x]_{\mathcal{P}}, z^{(u)})$, where $z^{(u)}$ is derived using the shared key $k_{\mathcal{P}}$; while HP acts as the receiver with input $[y]_{\text{HP}}$. This will enable HP to obtain $u = [x]_{\mathcal{P}} \cdot [y]_{\text{HP}} - z^{(u)}$. The other cross term $[x]_{\text{HP}} \cdot [y]_{\mathcal{P}}$ can also be computed similarly via another invocation to $\mathcal{F}_{\text{MS-OLE-IA}}$. The formal details appear below.

Protocol $\Pi_{\text{MultSS}}^{\text{Alhena}}$ in the $\mathcal{F}_{\text{MS-OLE-IA}}$ -hybrid model

Input: $\forall i \in [n]$, P_i has $[x]_{\mathcal{P}}$, $[y]_{\mathcal{P}}$ and common PRF key $k_{\mathcal{P}}$ generated in the setup phase. The HP has $[x]_{\text{HP}}$ and $[y]_{\text{HP}}$.

Output: $\forall i \in [n]$, P_i and HP output $[z]$, where $z = x \cdot y$.

Protocol:

- If $P = \text{HP}$, then it does the following:
 - Invoke $\mathcal{F}_{\text{MS-OLE-IA}}$ as receiver with input $[x]_{\text{HP}}$. Receive as output from $\mathcal{F}_{\text{MS-OLE-IA}}$ either $\perp$ or w .
 - Invoke $\mathcal{F}_{\text{MS-OLE-IA}}$ as receiver with input $[y]_{\text{HP}}$. Receive as output from $\mathcal{F}_{\text{MS-OLE-IA}}$ either $\perp$ or u .
 - If $\perp$ was received in any of the above, output $\perp$. Else output $[z]_{\text{HP}} = [x]_{\text{HP}} \cdot [y]_{\text{HP}} + u + w$.
- If $P \neq \text{HP}$, then it does the following:
 - All parties excluding HP sample random values $z^{(u)}$, $z^{(w)}$ using $k_{\mathcal{P}}$.
 - Invoke $\mathcal{F}_{\text{MS-OLE-IA}}$ as sender with inputs $([x]_{\mathcal{P}}, z^{(u)})$.
 - Invoke $\mathcal{F}_{\text{MS-OLE-IA}}$ as sender with inputs $([y]_{\mathcal{P}}, z^{(w)})$.
 - If $(\perp, i^*)$ is received as output from any of the above invocations, abort identifying i^* as the cheater. Else, compute $[z]_{\mathcal{P}} = [x]_{\mathcal{P}} \cdot [y]_{\mathcal{P}} + z^{(u)} + z^{(w)}$.

Figure 6: Multiplication of shared values

Authenticating the product $[xy]$ of two shares. To authenticate the sharing $[xy]$ obtained by $\Pi_{\text{MultSS}}^{\text{Alhena}}$ (Fig. 6), the protocol $\Pi_{\text{SSAuth}}^{\text{Alhena}}$ (Fig. 5) is used, taking $[xy]$ as input to produce $\langle xy \rangle$.

We now proceed to describe the preprocessing data generates that helps to process the input, addition and multiplication gates in the online phase.

Input Gates. Let P_d be an input provider, and let v be the corresponding input. The parties $\mathcal{P}$ and HP then jointly generate $[\delta_v]$ where the mask δ_v is revealed to P_d using sub-protocol $\Pi_{\text{RSS}}^{\text{Alhena}}$ (Fig. 4). Subsequently, the parties authenticate $[\delta_v]$ to obtain $\langle \delta_v \rangle$ using sub-protocol $\Pi_{\text{SSAuth}}^{\text{Alhena}}$ (Fig. 5).

Addition Gates. Note that, due to the linearity of $\langle \cdot \rangle$ -sharing, the parties and HP can locally compute the authenticated sharing $\langle \delta_z \rangle_{\mathcal{P}}$ and $\langle \delta_z \rangle_{\text{HP}}$ of the output mask δ_z as the sum of the input masks i.e. from $\langle \delta_x \rangle_{\text{HP}}$ and $\langle \delta_y \rangle_{\text{HP}}$, where δ_x , δ_y and δ_z are the masks corresponding to the input and output wires of the addition gate respectively.

Multiplication gates. Given the $\langle \cdot \rangle$ of a random multiplication triple $(\delta_x, \delta_y, \delta_{xy})$ and a random mask δ_z , the online phase requires to compute the masked value m_z corresponding to the output z . Thus, in the preprocessing phase, parties generate $\langle \delta_{xy} \rangle$ given $\langle \delta_x \rangle$ and $\langle \delta_y \rangle$. $\langle \delta_z \rangle$ is uniformly sampled.

Preprocessing phase. The full protocol $\Pi_{\text{Preproc}}^{\text{Alhena}}$ executed in the preprocessing phase is detailed in (Fig. 7). We note that each of the sub-protocols are in the $\mathcal{F}_{\text{MS-OLE-IA}}$ -hybrid model and have the guarantee that if any of the sub-protocols abort, the honest parties in $\mathcal{P}$ identify a common cheater and the HP aborts.

Protocol $\Pi_{\text{Preproc}}^{\text{Alhena}}$

Input: $\forall i \in [n]$, P_i and HP have the circuit C and common PRF keys $\{k_i, k_{\mathcal{P}}, k_{\text{all}}\}$ generated during the setup phase.

Output: $\forall i \in [1, n]$, P_i outputs preproc_i , and HP outputs $\text{preproc}_{\text{HP}}$.

Protocol: The following steps are executed. If any abort occurs in the sub-protocols, parties simply terminate with the output of the sub-protocol.

- **MAC key generation:** The HP samples a random global MAC key Δ .
- **Input gates:** For each input gate v with input provider P_d , all parties together with HP invoke $\Pi_{\text{RSS}}^{\text{Alhena}}$ (Fig. 4) to generate a sharing $[\delta_v]$, where the masked value δ_v is known to P_d . After the sharing is generated, the parties and the HP execute $\Pi_{\text{SSAuth}}^{\text{Alhena}}$ (Fig. 5) on input $[\delta_v]$. If no abort occurs, parties obtain $\langle \delta_v \rangle$.
- **Addition gates:** For each addition gate with inputs x and y , the parties and HP locally compute the intermediate values.
- **Multiplication gates:** For each multiplication gate $z = x \cdot y$, the parties and HP first generate the authenticated sharing $\langle \delta_z \rangle$ by executing $\Pi_{\text{RandRSS}}^{\text{Alhena}}$ (Fig. 3) and $\Pi_{\text{SSAuth}}^{\text{Alhena}}$ (Fig. 5). Subsequently, parties execute $\Pi_{\text{MultSS}}^{\text{Alhena}}$ (Figure 6) with inputs $[\delta_x]$ and $[\delta_y]$ to obtain a sharing $[\delta_{xy}]$. Finally, the parties and HP authenticate $[\delta_{xy}]$ using $\Pi_{\text{SSAuth}}^{\text{Alhena}}$ (Fig. 5). If no abort occurs, parties obtain $\langle \delta_{xy} \rangle$.

- **Output gates:** For each output gate z , all parties together with HP invoke $\Pi_{\text{RandRSS}}^{\text{Alhena}}$ (Fig. 3) to generate a sharing $[\delta_z]$.

Figure 7: The preprocessing phase of Alhena

3.3. Online Phase

Below, we describe the online phase of our protocol Alhena which achieves GOD. We first present the sub-protocols; the final protocol of the online phase $\Pi_{\text{Online}}^{\text{Alhena}}$ appears in Fig. 10.

Input gates. Let P_d be a party with input v . During the preprocessing phase, P_d has obtained the random mask, δ_v , for v in clear which is also $\langle \cdot \rangle$ -shared among all parties. P_d computes masked value $m_v = v + \delta_v$ and sends it to HP, who then forwards m_v to all parties. All parties thus obtain $\llbracket v \rrbracket_{\mathcal{P}} = (m_v, \langle \delta_v \rangle_{\mathcal{P}})$ and HP obtains $\llbracket v \rrbracket_{\text{HP}} = (m_v, \langle \delta_v \rangle_{\text{HP}})$.

Protocol $\Pi_{\llbracket \cdot \rrbracket\text{-Sh}}^{\text{Alhena}}$

Input:

- Every input provider /dealer P_d , has input v and mask δ_v .
- $\forall i \in [n]$, P_i has the input gate g_{in} and the mask $\langle \delta_v \rangle_{\mathcal{P}}$.
- HP has g_{in} and $\langle \delta_v \rangle_{\text{HP}}$.

Output: $\forall i \in [n]$, P_i outputs $\llbracket v \rrbracket_{\mathcal{P}}$ and HP outputs $\llbracket v \rrbracket_{\text{HP}}$.

Protocol:

- P_d computes $m_v = v + \delta_v$ and sends it to the HP.
- Upon receiving a value m_v , HP forwards it to all parties.
- $\forall i \in [n]$, P_i receives m_v from HP and outputs $\llbracket v \rrbracket_{\mathcal{P}} = (m_v, \langle \delta_v \rangle_{\mathcal{P}})$.
- HP outputs $\llbracket v \rrbracket_{\text{HP}} = (m_v, \langle \delta_v \rangle_{\text{HP}})$.

Figure 8: Online phase of input sharing of Alhena

Protocol $\Pi_{\text{mult}}^{\text{Alhena}}$

Input:

- $\forall i \in [n]$, P_i has $\llbracket x \rrbracket_{\mathcal{P}}$, $\llbracket y \rrbracket_{\mathcal{P}}$ and $\langle \delta_{xy} \rangle_{\mathcal{P}}$, $\langle \delta_z \rangle_{\mathcal{P}}$.
- HP has $\llbracket x \rrbracket_{\text{HP}}$, $\llbracket y \rrbracket_{\text{HP}}$ and $\langle \delta_{xy} \rangle_{\text{HP}}$, $\langle \delta_z \rangle_{\text{HP}}$.

Output:

- $\forall i \in [n]$, P_i has $\llbracket z \rrbracket_{\mathcal{P}} = (m_z, \langle \delta_z \rangle_{\mathcal{P}})$, where $m_z = z + \delta_z$ and $z = x \cdot y$.
- HP has $\llbracket z \rrbracket_{\text{HP}} = (m_z, \langle \delta_z \rangle_{\text{HP}})$.

Protocol:

- If $P = \text{HP}$, then it does the following:
 - computes $[m_z]_{\text{HP}} = m_x \cdot m_y - m_x \cdot [\delta_y]_{\text{HP}} - m_y \cdot [\delta_x]_{\text{HP}} + [\delta_{xy}]_{\text{HP}} + [\delta_z]_{\text{HP}}$
 - send $[m_z]_{\text{HP}}$ to all parties.
 - computes $r_{[m_z]_{\mathcal{P}}} = -m_x \cdot r_{[\delta_y]_{\mathcal{P}}} - m_y \cdot r_{[\delta_x]_{\mathcal{P}}} + r_{[\delta_{xy}]_{\mathcal{P}}} + r_{[\delta_z]_{\mathcal{P}}}$
- $\forall i \in [n]$, HP:
 - receives $[m_z]_{\mathcal{P}}$ and $t_{[m_z]_{\mathcal{P}}}$ from P_i .
 - verifies that $t_{[m_z]_{\mathcal{P}}} \stackrel{?}{=} \Delta \cdot [m_z]_{\mathcal{P}} - r_{[m_z]_{\mathcal{P}}}$
 - If the check verifies, compute $m_z = [m_z]_{\mathcal{P}} + [m_z]_{\text{HP}}$.
- HP outputs $\llbracket z \rrbracket_{\text{HP}} = (m_z, \langle \delta_z \rangle_{\text{HP}})$.
- If $P \neq \text{HP}$, then it does the following:
 - computes $[m_z]_{\mathcal{P}} = -m_x \cdot [\delta_y]_{\mathcal{P}} - m_y \cdot [\delta_x]_{\mathcal{P}} + [\delta_{xy}]_{\mathcal{P}} + [\delta_z]_{\mathcal{P}}$
 - computes $t_{[m_z]_{\mathcal{P}}} = -m_x \cdot t_{[\delta_y]_{\mathcal{P}}} - m_y \cdot t_{[\delta_x]_{\mathcal{P}}} + t_{[\delta_{xy}]_{\mathcal{P}}} + t_{[\delta_z]_{\mathcal{P}}}$.
 - sends $[m_z]_{\mathcal{P}}$ and $t_{[m_z]_{\mathcal{P}}}$ to HP.
 - receives $[m_z]_{\text{HP}}$ from HP. Compute $m_z = [m_z]_{\mathcal{P}} + [m_z]_{\text{HP}}$. Output $\llbracket z \rrbracket_{\mathcal{P}} = (m_z, \langle \delta_z \rangle_{\mathcal{P}})$.

Figure 9: Online phase of multiplication of Alhena

Addition gates. Due to the homomorphic property of the secret sharing scheme, each party locally obtains the sharing of the output wire of addition gates (as elaborated in Figure 10).

Multiplication gates. The protocol $\Pi_{\text{mult}}^{\text{Alhena}}$ (in Figure 9) illustrates how two $[\![\cdot]\!]$ -shared values are multiplied during the online phase. Recall that the sharing $[\![v]\!]$ $= (m_v, \langle \delta_v \rangle)$ of v includes an authenticated sharing $\langle \delta_v \rangle = (\langle \delta_v \rangle_{\mathcal{P}}, \langle \delta_v \rangle_{\text{HP}})$ of the mask δ_v , where each party holds $\langle \delta_v \rangle_{\mathcal{P}} = ([\delta_v]_{\mathcal{P}}, t_{[v]_{\mathcal{P}}})$ and HP holds $\langle \delta_v \rangle_{\text{HP}} = ([\delta_v]_{\text{HP}}, r_{[v]_{\mathcal{P}}})$.

To obtain the sharing $[\![z]\!]$ (where $z = x \cdot y$) from $[\![x]\!]$ and $[\![y]\!]$, the parties would need to generate m_z and $\langle \delta_z \rangle$, where the latter is already obtained in the preprocessing phase. We analyze that m_z can be computed as $m_z = z + \delta_z = x \cdot y + \delta_z = (m_x - \delta_x) \cdot (m_y - \delta_y) = m_x m_y - m_x \delta_y - \delta_x m_y + \delta_{xy} + \delta_z$.

Using its shares, the HP can locally compute its share $[m_z]_{\text{HP}}$, which it can send to all parties which enables them to reconstruct m_z . However, for HP to reconstruct m_z , it must receive the share $[m_z]_{\mathcal{P}}$. Since a potentially malicious party could send an incorrect $[m_z]_{\mathcal{P}}$ to HP, we make the parties send it along with the corresponding tag $t_{[m_z]_{\mathcal{P}}}$. This allows HP to detect and discard the incorrect versions of the share. The presence of at least one honest party ensures that HP would receive at least one share such that the tag verifies and successfully reconstruct m_z . We elaborate the details in Figure 9.

In the implementation, we optimize communication by parties sending the masks $m_z^{(1)}, \dots, m_z^{(m)}$ for m multiplication gates at the same circuit depth d , along with a combined hash over all tags i.e. $H = h(t_{[m_z]_{\mathcal{P}}}^{(1)}, \dots, t_{[m_z]_{\mathcal{P}}}^{(m)})$ to the HP. The HP can compute each of the tags $t_{[m_z]_{\mathcal{P}}}^{(i)}$ itself as $t_{[m_z]_{\mathcal{P}}}^{(i)} = \Delta \cdot [m_z]_{\mathcal{P}}^{(i)} - r_{[m_z]_{\mathcal{P}}}^{(i)}$ and verify whether the combined hash $h(t_{[m_z]_{\mathcal{P}}}^{(1)}, \dots, t_{[m_z]_{\mathcal{P}}}^{(m)})$ is consistent with the received H . The final protocol and theorem appear below.

Protocol $\Pi_{\text{Online}}^{\text{Alhena}}$

Input:

- For every input provided/dealer P_d , it has input v .
- $\forall i \in [1, n]$, P_i and HP have the circuit C , the preprocessing data generated in (Fig. 7) and common PRF keys $\{k_i, k_{\mathcal{P}}, k_{\text{all}}\}$ generated in the setup phase.

Output: $\forall i \in [n]$, P_i outputs $C(x_1, x_2, \dots, x_n)$.

Protocol:

- *Input gates:* For every input gate of C , parties execute $\Pi_{[\![\cdot]\!]-\text{Sh}}^{\text{Alhena}}$ (Fig. 8).
- *Addition gates:* For every addition of C with input x, y and output z , parties and HP locally set $[\![m_z]\!] = [\![m_x]\!] + [\![m_y]\!]$, where $[\![m_x]\!] + [\![m_y]\!] = (m_x + m_y, \langle \delta_x \rangle + \langle \delta_y \rangle)$.
- *Multiplication gates:* For every multiplication gate in C , parties execute $\Pi_{\text{mult}}^{\text{Alhena}}$ (Fig. 9).
- *Output gates:* For every output gate (which is either an addition or multiplication gate), the HP sends $[\delta_z]_{\text{HP}}$ to the parties in $\mathcal{P}$. Parties in $\mathcal{P}$ compute $\delta_z = [\delta_z]_{\text{HP}} + [\delta_z]_{\mathcal{P}}$ and output $z = m_z - \delta_z$.

Figure 10: The online phase of Alhena

Theorem 3.2. *Assuming the existence of PRF and oblivious transfer, the protocol Alhena in the preprocessing-online paradigm (Fig. 7, Fig. 10) achieves non-colluding security (Definition 2.3) with identifiable fairness (Definition 2.2) in the helper-aided MPC model with semi-honest HP.*

We defer the security proof to Appendix B.

4. Asynchronous Protocol Wasat

Network Model. The adversary controls message scheduling and may delay or reorder messages arbitrarily but honest messages are eventually delivered. To handle reordering attacks by the adversary, we assume all messages are enumerated sequentially, so that receiver can process messages in correct order.

Similar to Alhena, our asynchronous protocol Wasat is in the offline-online paradigm and achieves identifiable fairness in the helper-aided setting. As a starting point, we examine how our design of Alhena behaves under asynchrony. We make the following observations and modifications to develop Wasat.

Adapting $\Pi_{\text{MS-OLE-IA}}$. Our building block $\Pi_{\text{MS-OLE-IA}}$ in Alhena, which assumes that HP receives a message from a designated sender P_d , does not work in the asynchronous setting, since HP may end up waiting indefinitely (say, if P_d is corrupt and does not send any message).

To address this, we present a modified protocol $\Pi_{\text{MS-OLE-IA}}^{\text{async}}$ (Figure 12) which realizes $\mathcal{F}_{\text{MS-OLE-IA}}^{\text{async}}$ (Fig. 11) for the asynchronous setting: Each of the regular parties now initiates an OLE instance as a sender (with HP as the receiver). In each step of the OLE, we allow HP to proceed with the first sender from whom it receives a message. The HP then forwards the received message to all parties, along with the sender's identity and the corresponding response. The parties

can then verify whether the sender's message was computed correctly; if not, the sender is identified as cheating. Otherwise, the parties continue to the next step of their OLE instance.

Note that the above approach is guaranteed to terminate due to the presence of at least one honest sender whose message will eventually be received by HP. However, unlike the synchronous setting, HP cannot await confirmation from all parties regarding the computation being correct. Thus, while HP may end up with an incorrect OLE output, however we retain the guarantee that the honest parties abort and reach agreement on the cheater in such a case.

We observe that when honest parties terminate the preprocessing phase after identifying a cheater, the HP remains oblivious of this event and may proceed to the online phase with a set of corrupted parties. As a result, the HP may fail to terminate, since corrupted parties can cause it to wait indefinitely. In practice, this issue can be mitigated by enforcing a timeout for the HP. However, if honest parties reach the online phase, guaranteed output delivery is achieved.

Alternatively, to ensure termination of the HP, we allow honest parties to explicitly signal an abort to the HP, as described in Fig. 12. Though this signal could be delayed by the network adversary, it will eventually be delivered and cause the HP to abort as well. We model this in $\mathcal{F}_{\text{MS-OLE-IA}}^{\text{async}}$ (Fig. 11) via $(\text{lateabort}, i^*, T, d)$, where the adversary specifies the incorrect output d and the amount of delay T . However, this mechanism can be exploited by a malicious adversary, who may force an abort during the online phase by sending an abort signal to the HP. This is modeled in $\mathcal{F}_{\text{MS-OLE-IA}}^{\text{async}}$ via $(\text{deliver}, \text{lateabort}, i^*, T)$. Though this approach does not achieve guaranteed output delivery, identifiable fairness is still preserved as before.

Functionality $\mathcal{F}_{\text{MS-OLE-IA}}^{\text{async}}$

- **Initialize:** On input $(\text{Init}, S_i, \mathbb{F})$ from S_i (for all $i \in [n]$) and $(\text{Init}, R, \mathbb{F})$ from R , store $\mathbb{F}$.
- **Input:** On input (Input, a_i, c_i) from S_i , for all $i \in [n]$ and (Input, b) from R : store $(a = a_\ell, b, c = c_\ell)$ where ℓ is the lexicographically smallest index in $\mathcal{P} \setminus C$.
- **Output:** On input (Output) from all the parties, send **ready** to the adversary. Wait for an input from the adversary
 - 1) If this is (deliver) , then send $((a \cdot b - c))$ to R .
 - 2) If this is $(\text{deliver}, \text{lateabort}, i^*, T)$, then send $((a \cdot b - c), i^*, T)$ to R , and $(\perp, i^*)$ to all $S_i \in \mathcal{P} \setminus C$.
 - 3) If this is $(\text{lateabort}, i^*, T, d)$, where $d \neq a \cdot b - c$ then send (d, i^*, T) to R , and $(\perp, i^*)$ to all $S_i \in \mathcal{P} \setminus C$.

Figure 11: Ideal functionality for asynchronous Multi-Sender OLE with Identifiable Abort

Preprocessing Phase. The preprocessing phase of Wasat is identical to Alhena, except that OLE invocations are realized using $\Pi_{\text{MS-OLE-IA}}^{\text{async}}$ (Figure 12) instead. We observe that the other steps are non-interactive (like sampling values using shared key setup) which remain unaffected by asynchrony. In the OLE invocations, we exploit the fact that parties in $\mathcal{P}$ can afford to wait for messages from HP even in the asynchronous setting.

Online Phase. The online phase of Wasat is identical to that of Alhena, with the following modifications: First, in the input-gate subprotocol $\Pi_{[\cdot]-\text{Sh}}^{\text{Wasat}}$, HP waits to receive m_v from only $n - t$ parties, since waiting for all parties could result in unbounded delay. For the other parties, a default input is considered. Similar to Castor [46], our construction nonetheless ensures that all honest parties whose messages are delivered in the first round are included in the computation. In practice, this suggests specifying a generous waiting period for the initial step, after which HP waits for only $n - t$ messages in each subsequent step. Second, if the HP receives an abort message from any party (as a delayed message from $\Pi_{\text{MS-OLE-IA}}^{\text{async}}$), the HP terminates after notifying all parties, which then also terminate upon receiving this notification.

Notably, the multiplication sub-protocol is already resilient to asynchrony: HP can simply wait for the first share $[m_z]_{\mathcal{P}}$ whose tag verifies successfully. Termination is guaranteed as such a share will eventually come from an honest party.

Theorem 4.1. *Assuming the existence of PRF and oblivious transfer, the protocol Wasat in the preprocessing-online paradigm achieves non-colluding security (Definition 2.3) with identifiable fairness (Definition 2.2) in the helper-aided asynchronous MPC model with semi-honest HP.*

We defer the security proof to Appendix C.

Protocol $\Pi_{\text{MS-OLE-IA}}^{\text{async}}$

Input:

- $\forall i \in [n]$, P_i has input a, c .
- HP has input b .

Output: HP receives $a \cdot b - c$ (in case of no abort).

Building Block: 2-party Π_{OLE} (instantiated using [62]) and shared key setup.

Protocol:

- Each P_i ($i \in [n]$) and HP participate in an instance of Π_{OLE}^i : P_i participates as sender using its input (a, c) and randomness derived from k_P . HP participates as receiver using input b.
- For simplicity, suppose Π_{OLE} is an R -round alternating-message protocol (i.e. every alternate round is sent by the sender and receiver respectively). The HP does the following:
 - Among all the running instances of $\{\Pi_{\text{OLE}}^i\}_{i \in [n]}$, suppose $(r, \tau_S^{(j,r)})$ was the first round r message that HP received from sender P_j . Then, HP records the message $\tau_S^r = \tau_S^{(j,r)}$, computes the next receiver message τ_R^{r+1} accordingly. HP sends $(r+1, j, \tau_S^{(j,r)}, \tau_R^{r+1})$ to all parties $P_i \in \mathcal{P}$.
 - If all sender messages were received, output the output of Π_{OLE} based on the sender's transcript $\{\tau_S^r\}_{r \in [R]}$.
 - If (r, abort) is received from P_k in $\mathcal{P}$, the HP terminates with output $\perp$ and sends $(r+1, \text{abort}, P_k)$ to all parties.
- Each party $P_i \in \mathcal{P}$ does the following:
 - Consider round $r+1$ of Π_{OLE}^i : When $(r+1, j, \tau_S^{(j,r)}, \tau_R^{r+1})$ is received from the HP, verify if $\tau_S^{(j,r)}$ is computed correctly using input (a, c), randomness derived from k_P and the receiver messages $\tau_R^{<r+1}$ received from HP so far.
 - If the above does not verify, send (r, abort) to HP and terminate with output $(\perp, j)$ identifying P_j as the cheater.
 - Else, update the set of receiver messages $\tau_R = \tau_R || \tau_R^{r+1}$. Compute and send the next sender message $\tau_S^{(i,r+2)}$ in the instance Π_{OLE}^i .
 - If $\tau_S^{(j,r)}$ received from HP was computed correctly, but $(r+1, \text{abort}, P_k)$ was received from HP, terminate with output $(\perp, k)$ identifying P_k as the cheater.

Figure 12: Protocol realizing $\mathcal{F}_{\text{MS-OLE-IA}}$ in the asynchronous setting

5. Experimental Results

In this section, we compare our implementation of Wasat and Alhena to the asynchronous protocols Castor and Pollux and the synchronous protocol Asterisk.

Environment. Benchmarks were performed on a system equipped with a 24-core AMD EPYC-2 (Rome) 7402P CPU and 128GB of RAM. We instantiate parties and the HP in the same manner as in [46], where each party and the HP are executed as a separate process on the same system. Additionally, we reproduce the same network conditions using the Linux `tc` command from the network package `netem` to emulate a WAN network with a bandwidth of 100 Mbps and 50 ms latency.

Implementation of Wasat and Alhena. We implemented our protocols ² in *C++17*, building on top of the codebase of Castor and Pollux [46] and the EMP toolkit [61]. Similar to Castor, our protocol use the same maliciously secure Ferret-COT [62] from the EMP-Toolkit [61] to perform the OLE during the offline phase. For the OT, we set the computational security parameter to $\kappa = 128$ and (providing statistical security of at least 2^{40}). For the arithmetic operations over the prime field $\mathbb{Z}_p$, we use the NTL library [59] used in the codebase of Castor and Pollux, where p is a 64-bit prime.

5.1. Performance benchmarks

All protocols are evaluated on synthetic circuits with 1 million multiplication gates, where the circuit has either depth $d = 10$ or $d = 100$, with 10^5 and 10^4 multiplication gates per layer, respectively. The benchmarks are conducted with varying numbers of parties, ranging from 5 to 50. The communication and runtime overheads incurred during the preprocessing phase are reported in Table 2, whereas the corresponding online phase overheads are shown in Table 3. For each benchmark we report the average communication/runtime overhead of uninterrupted 5 runs.

Offline phase. Unsurprisingly, in the preprocessing phase Asterisk [47] has the lowest communication and runtime overhead, since HP generates the preprocessing data independently of the parties and only sends a one-shot message, 3 elements per multiplication gate, resulting in 24MB for a circuit of size 1 million to P_n (independent of the depth of the circuit). Similarly, the preprocessing phase of Pollux [46] is also very efficient, since the HP can non-interactively generate shares for t parties and only needs to send shares to the remaining $t+1$ parties. On the other hand, Alhena and Wasat exhibit significant performance gains over their closest counterpart Castor with speedups of $5-30\times$ and $\approx 3\times$, respectively. When comparing Wasat to Castor [46], the runtime and communication overhead of Wasat is much lower. While Castor requires $2n$ OLEs per input gate, Wasat requires n OLEs (Alhena requires 1 OLE). For multiplication gates, Wasat requires $4n$ OLEs (Alhena requires 4 OLEs). In contrast, Castor requires $12n$ OLEs per multiplication gate, as the consistency of the preprocessing phase is guaranteed through a triple sacrifice.

2. Our implementation is available at: <https://zenodo.org/records/22792281>. Our code is developed for benchmarking and is not optimized for industry-grade use.

TABLE 2: The offline communication and runtime overheads for varying depth ($d \in \{10, 100\}$) and varying numbers of parties ($n \in \{5, 10, 20, 30, 50\}$). $\perp$ indicates a timeout.

d	n	Pollux		Castor		Alhena		Wasat		Asterisk	
		comm. (MB)	runtime (sec)	comm. (MB)	runtime (sec)	comm. (MB)	runtime (sec)	comm. (MB)	runtime (sec)	comm. (MB)	runtime (sec)
10	5	96.13	24.48	60646.45	5113.67	4054.02	1016.59	20251.74	1719.36	24	16.52
	10	192.26	48.91	121197.39	10188.60	4058.61	998.60	40503.49	3408.33	24	25.66
	20	352.47	118.70	242702.93	20406.96	4067.79	977.48	81006.98	6814.20	24	43.51
	30	512.69	254.27	363227.43	30796.97	4076.97	1039.06	121510.47	10194.12	24	59.26
	50	873.60	677.80	$\perp$	$\perp$	4294.27	1110.62	212354.9	17136.88	24	87.11
100	5	92.01	26.62	59752.50	5038.92	3964.91	995.03	19806.22	1684.16	24	16.01
	10	184.02	53.28	119415.50	10045.16	3969.50	982.38	39612.44	3334.78	24	25.22
	20	337.37	116.43	239243.95	20184.48	3978.69	956.49	79224.88	6667.75	24	41.76
	30	490.72	213.59	357961.02	30245.09	3987.86	999.72	118837.32	9966.13	24	57.23
	50	836.16	518.54	$\perp$	$\perp$	4200.83	1070.88	207683.30	16525.61	24	84.31

During the preprocessing phase of Wasat, the parties and HP exchange no field elements beyond those required for the OLE. For both Wasat and Castor, the communication and runtime overheads scale linearly with the number of parties. On the other hand, for Alhena increasing the number of parties only causes a marginal increase in the communication overhead. This is because, regardless of the number of parties, only one designated party performs the OLE with HP, and the other parties verify locally (using messages sent by HP that authenticate the data sent by the designated party) whether the sender party behaved honestly. The marginal increase in communication is therefore only due to the OLE setup and the communication between the verifier parties and HP. Besides the OLEs, parties inform HP whether they found the sender party’s messages to be consistent, and HP in turn informs the verifier parties of each other party’s vote for the verifier party. As communicating these messages only requires single bits or short bit strings, their size is negligible compared to the OLE communication volume (for $d = 10$ and $n = 30$, the remainder is 0.07MB). The runtime of Wasat, on the other hand, fluctuates slightly as the number of parties increases. We expect the runtime to remain relatively constant with an increasing number of parties, since the bulk of the operations—namely, the OLEs between the HP and the single sender party—remains unchanged. However, our experiments show that the runtime of Alhena decreases as the number of parties increases from 5 to 10 and 20, before increasing again for 30 and 50 parties. This behavior can be explained by our benchmarking setup, which uses a single 24-core machine in which computational resources are shared among all parties. When the number of parties is below the number of available cores, increased parallelism improves resource utilization and reduces runtime. Once the number of parties exceeds the number of cores, however, the system becomes oversubscribed, leading to suboptimal resource utilization and increased runtime.

Online phase. In the online phase (common to both Alhena and Wasat), the runtime of our framework outperforms existing protocols, achieving speedups of $3 - 6.5\times$ over Pollux, $2.5 - 5\times$ over Castor, and $4 - 12.5\times$ over Asterisk.

The online phase of our protocols (Alhena and Wasat) are significantly faster than the online phase of Pollux, Castor, and Asterisk. In Table 3, we assume the best-case scenario for Alhena / Wasat, where no party attempts to send inconsistent shares to the HP for multiplication gate reconstruction. In the worst case, up to $n - 1$ parties can send inconsistent shares to HP during multiplication gate reconstruction for a layer and a network adversary may reorder all the messages such that HP receives all the inconsistent shares before it receives the correct share from the single honest party. In this execution, HP is forced to wait for up to $n - 1$ messages before receiving a share that passes the verification. In practice, each dishonest party can send an inconsistent share at most once, since HP checks the consistency of their share immediately and, in case of misbehavior, can ignore their shares for subsequent layers of the circuit. In the next section, we investigate the impact of corrupted parties that send inconsistent shares to the HP in every round, as well as the corresponding worst-case scenario in which $n - 1$ parties are corrupted.

The performance improvement of Alhena and Wasat over previous work stems from the fact that HP verifies MACs for each multiplication gate while it is waiting for a party’s share for the subsequent gate, whereas earlier protocols postpone this check until the end of the protocol. This has the benefit, that in our work the HP can better utilize the waiting time between messages for computation. Moreover, the verification step in the protocols presented in this work is less costly, as the secret is shared using a constant number of two shares, only one of which requires verification.

TABLE 3: The online communication and runtime overheads for varying depth ($d \in \{10, 100\}$) and varying numbers of parties ($n \in \{5, 10, 20, 30, 50\}$).

d	n	Pollux		Castor		Alhena / Wasat		Asterisk	
		comm. (MB)	runtime (sec)	comm. (MB)	runtime (sec)	comm. (MB)	runtime (sec)	comm. (MB)	runtime (sec)
10	5	76.29	45.53	76.29	22.04	76.30	6.96	61.04	28.49
	10	152.59	60.58	152.59	39.85	152.59	12.17	137.33	52.42
	20	305.18	98.67	305.18	85.86	305.18	21.99	289.92	89.41
	30	457.77	138.31	457.77	134.89	457.78	27.56	442.51	128.56
	50	800.23	263.79	800.72	234.43	800.19	46.30	747.43	195.39
100	5	76.30	30.68	76.30	23.84	76.31	9.51	61.04	71.40
	10	152.60	42.68	152.60	38.94	152.63	12.31	137.33	138.87
	20	305.19	79.82	305.19	67.80	305.25	23.35	289.92	266.24
	30	457.79	133.19	457.79	140.05	457.88	31.57	442.51	392.80
	50	800.40	248.09	800.31	239.22	800.20	50.31	783.93	633.48

5.2. Effect of misbehaving parties

To evaluate how the shared online phase of Alhena and Wasat performs when malicious parties attempt to send inconsistent data, we conduct the following experiment.

In Table 4, we report communication and runtime overhead as we gradually increase the number of corrupt parties $t \in \{0, 5, 10, 15, 20, 25, 29\}$. A corrupt party sends inconsistent shares (for which the MAC check fails) to the HP in every round. In each round, HP accepts shares from parties until it obtains a valid verification. We apply the following optimization: once a party has been caught cheating (i.e., it sent a share for which the MAC check failed), HP stops accepting messages from that party in subsequent rounds. Each benchmark is evaluated on a synthetic circuit with 1 million multiplication gates (depth $d = 100$ and 10^4 multiplication gates per layer). We observe that the runtime does not change noticeably as the number of corrupted parties t increases, while the communication decreases linearly in the number of corrupted parties.

Our protocol is designed so that malicious behaviour affects only computational complexity, not round or communication complexity, since all parties send messages to HP simultaneously. The only overhead incurred in the malicious case is that HP is forced to verify multiple messages before finding a correct one. The cost of this local computation is negligible compared to the communication costs. Furthermore, once a party submits a malformed message, it is blacklisted to prevent subsequent wasted computation, explaining the linear decrease in the total communication volume.

5.3. Effect of asynchronous network

We repeat the experiment performed by [46] to compare the performance of asynchronous protocols to that of synchronous protocols. Asynchronous protocols typically require more communication than synchronous ones; however, given the same communication, asynchronous protocols often achieve better performance because they do not wait for all parties' messages.

We reuse the same experimental setup of [46] by introducing an artificial 200ms delay for a subset of parties, assuming the synchronous round delay exceeds this duration. This ensures that recipient parties wait for delayed messages without incorrectly flagging the sender as malicious. We fix the number of parties at $n = 5$ and incrementally introduce delays, where each delayed party waits 200ms before transmitting messages in each round. We measure the average runtime experienced by non-delayed parties. If every party is delayed, we report the mean runtime across all parties.

As illustrated in (Figure 13), we successfully reproduced the results for Asterisk, Pollux, and Castor. Asterisk suffers immediate performance degradation because the reconstructing parties must wait for messages from all participants. In

t	0	5	10	15	20	25	29
Runtime (sec)	31.612	30.233	30.566	29.800	29.735	28.933	31.709
Comm. (MB)	457.88	421.24	384.60	357.12	317.43	276.21	253.31

TABLE 4: The online communication and runtime overheads for depth $d = 100$ and varying numbers of corrupt parties $t \in \{0, 5, 10, 15, 20, 25, 29\}$ sending inconsistent data to HP data in every round.

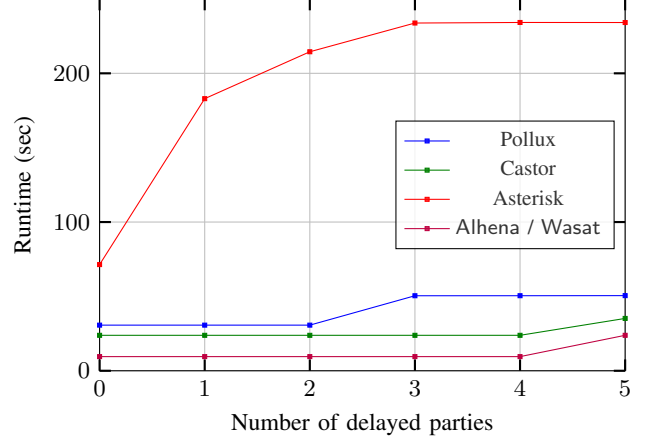


Figure 13: Effect of asynchronous network delays on the online running time of Pollux, Castor, Asterisk and Alhena / Wasat for $n = 5$ parties and for circuits with 10^6 multiplication gates of depth $d = 100$. Delays of 200 ms were incrementally introduced for $0, \dots, 5$ parties.

contrast, Pollux only shows degradation once $n/2$ parties are delayed, and Castor remains resilient until all parties are delayed. As anticipated, Alhena / Wasat exhibit similar behavior to Castor, since for these protocols HP has to wait for only one message. We emphasize once more that we are considering the optimistic best-case scenario for Alhena and Wasat, in which all parties send consistent shares. Although Alhena was not designed for asynchronous networks, in the best-case setting its performance matches that of Wasat even when some parties are slow.

Acknowledgment

Protik Paul received funding from the ERC under the EU’s research and innovation programs Horizon 2020 (PSOTI/850990) and Horizon Europe (PRIVTOOLS/101124778). It was co-funded by the DFG within SFB 1119 CROSSING/236615297. This project was also supported by the European Union and the Estonian Research Council through TEM-TA119 and PRG2531. Divya Ravi was supported by the NWO grant with number VI.Veni.242.295 and by NWO Gravitation Project CiCS (Challenges in Cyber Security) with file number 024.006.037.

References

- [1] Ittai Abraham, Gilad Asharov, Shravani Patil, and Arpita Patra. Perfect asynchronous mpc with linear communication overhead. In *EUROCRYPT*, 2024.
- [2] Gilad Asharov, Tucker Hybinette Balch, Antigoni Polychroniadou, and Manuela Veloso. Privacy-preserving dark pools. In *AAMAS*, 2020.
- [3] Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. *JoC*, 2010.
- [4] Saikrishna Badrinarayanan, Abhishek Jain, Rafail Ostrovsky, and Ivan Visconti. Non-interactive secure computation from one-way functions. In *ASIACRYPT*, 2018.
- [5] Akhil Bandarpalli, Xiaoyu Ji, Aniket Kate, Chen-Da Liu-Zhang, and Yifan Song. Concretely efficient asynchronous mpc from lightweight cryptography. *Cryptology ePrint Archive*, 2024.
- [6] Akhil Bandarpalli, Xiaoyu Ji, Aniket Kate, Chen-Da Liu-Zhang, and Yifan Song. Computationally efficient asynchronous MPC with linear communication and low additive overhead. In *CRYPTO*, 2025.
- [7] Carsten Baum, Nikolas Melissaris, Rahul Rachuri, and Peter Scholl. Cheater identification on a budget: MPC with identifiable abort from pairwise macs. In *CRYPTO*, 2024.
- [8] Carsten Baum, Emmanuela Orsini, and Peter Scholl. Efficient secure multiparty computation with identifiable abort. In *TCC*, 2016.
- [9] Carsten Baum, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez. Efficient constant-round MPC with identifiable abort and public verifiability. In *CRYPTO*, 2020.
- [10] Zuzana Beerliová-Trubníová, Martin Hirt, and Jesper Buus Nielsen. On the theoretical gap between synchronous and asynchronous mpc protocols. In *PODC*, 2010.
- [11] Aner Ben-Efraim, Michael Nielsen, and Eran Omri. Turbospeedz: Double your online SPDZ! improving spdz using function dependent preprocessing. In *ACNS*, 2019.

- [12] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous secure computations with optimal resilience. In *PODC*, 1994.
- [13] Erica Blum, Jonathan Katz, Chen-Da Liu-Zhang, and Julian Loss. Asynchronous byzantine agreement with subquadratic communication. In *TCC*, 2020.
- [14] Ran Canetti. Security and composition of multiparty cryptographic protocols. *J. Cryptol.*, 2000.
- [15] Ran Canetti and Marc Fischlin. Universally composable commitments. In *CRYPTO*, 2001.
- [16] John Carlidge, Nigel P Smart, and Younes Talibi Alaoui. Mpc joins the dark side. In *ASIA-CCS*, 2019.
- [17] Nishanth Chandran, Wutichai Chongchitmate, Rafail Ostrovsky, and Ivan Visconti. Universally composable secure computation with corrupted tokens. In *CRYPTO*, 2019.
- [18] Nishanth Chandran, Vipul Goyal, and Amit Sahai. New constructions for UC secure computation using tamper-proof hardware. In *EUROCRYPT*, 2008.
- [19] David Chaum, Ivan Damgård, and Jeroen van de Graaf. Multiparty computations ensuring privacy of each party’s input and correctness of the result. In *CRYPTO*, 1987.
- [20] 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.
- [21] 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. *JoC*, 2019.
- [22] Annick Chopard, Martin Hirt, and Chen-Da Liu-Zhang. On communication-efficient asynchronous MPC with adaptive security. In Kobbi Nissim and Brent Waters, editors, *TCC*, 2021.
- [23] Annick Chopard, Martin Hirt, and Chen-Da Liu-Zhang. On communication-efficient asynchronous mpc with adaptive security. In *TCC*, 2021.
- [24] Ashish Choudhury and Arpita Patra. Optimally resilient asynchronous mpc with linear communication complexity. In *ICDCN*, 2015.
- [25] Ashish Choudhury and Arpita Patra. On the communication efficiency of statistically secure asynchronous mpc with optimal resilience. *JoC*, 2023.
- [26] Richard Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In *STOC*, 1986.
- [27] Ran Cohen. Asynchronous secure multiparty computation in constant time. In *PKC*. 2016.
- [28] Ran Cohen, Jack Doerner, Yashvanth Kondi, and Abhi Shelat. Secure multiparty computation with identifiable abort from vindicating release. *Cryptology ePrint Archive*, 2023.
- [29] Ran Cohen and Yehuda Lindell. Fairness versus guaranteed output delivery in secure multiparty computation. *JoC*, 2017.
- [30] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. $\text{SPD}_{\mathbb{Z}_2^k}$: Efficient MPC mod 2^k for Dishonest Majority. In *CRYPTO*, 2018.
- [31] Ronald Cramer, Ivan Damgård, and Jesper B Nielsen. Multiparty computation from threshold homomorphic encryption. In *EUROCRYPT*, 2001.
- [32] Dana Dachman-Soled, Tal Malkin, Mariana Raykova, and Muthuramakrishnan Venkatasubramanian. Adaptive and concurrent secure computation from new adaptive, non-malleable commitments. In *ASIACRYPT*, 2013.
- [33] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous multiparty computation: Theory and implementation. In *FOCS*, 2009.
- [34] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In *CRYPTO*, 2007.
- [35] Oded Goldreich. *The Foundations of Cryptography - Volume 2: Basic Applications*. Cambridge University Press, 2004.
- [36] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *STOC*, 1987.
- [37] Vipul Goyal, Chen-Da Liu-Zhang, and Yifan Song. Towards achieving asynchronous mpc with linear communication and optimal resilience. In *CRYPTO*, 2024.
- [38] Carmit Hazay, Antigoni Polychroniadou, and Muthuramakrishnan Venkatasubramanian. Composable security in the tamper-proof hardware model under minimal complexity. In *TCC*, 2016.
- [39] Martin Hirt, Jesper Buus Nielsen, and Bartosz Przydatek. Cryptographic asynchronous multi-party computation with optimal resilience. In *EUROCRYPT*, 2005.
- [40] Martin Hirt, Jesper Buus Nielsen, and Bartosz Przydatek. Asynchronous multi-party computation with quadratic communication. In *ICALP*, 2008.
- [41] Yuval Ishai, Rafail Ostrovsky, and Hakan Seyalioglu. Identifying cheaters without an honest majority. In *TCC*, 2012.
- [42] Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. Secure multi-party computation with identifiable abort. In *CRYPTO*, 2014.
- [43] Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. Secure multi-party computation with identifiable abort. *Cryptology ePrint Archive*, 2015.
- [44] Yuval Ishai, Arpita Patra, Sikhar Patranabis, Divya Ravi, and Akshayaram Srinivasan. Fully-secure mpc with minimal trust. In *TCC*, 2022.
- [45] Xiaoyu Ji, Junru Li, and Yifan Song. Linear-communication asynchronous complete secret sharing with optimal resilience. In *CRYPTO*, 2024.
- [46] Banashri Karmakar, Aniket Kate, Shravani Patil, Arpita Patra, Sikhar Patranabis, Protik Paul, and Divya Ravi. Breaking the barrier for asynchronous mpc with a friend. In *S&P*, 2026.
- [47] Banashri Karmakar, Nishat Koti, Arpita Patra, Sikhar Patranabis, Protik Paul, and Divya Ravi. Asterisk: Super-fast mpc with a friend. In *S&P*, 2024.
- [48] Jonathan Katz. Universally composable multi-party computation using tamper-proof hardware. In *EUROCRYPT*, 2007.
- [49] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Mascot: faster malicious arithmetic secure computation with oblivious transfer. In *CCS*, 2016.

- [50] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. In *EUROCRYPT*, 2018.
- [51] Atsuki Momose. Practical asynchronous MPC from lightweight cryptography. *Cryptology ePrint Archive*, 2024.
- [52] Atsuki Momose. Practical asynchronous mpc from lightweight cryptography. *Cryptology ePrint Archive*, 2024.
- [53] Philipp Muth and Stefan Katzenbeisser. Assisted MPC. *Cryptology ePrint Archive*, 2022.
- [54] Arpita Patra, Ashish Choudhary, and C. Pandu Rangan. Efficient statistical asynchronous verifiable secret sharing with optimal resilience. In *ICITS*, 2009.
- [55] Arpita Patra, Ashish Choudhary, and C Pandu Rangan. Efficient statistical asynchronous verifiable secret sharing with optimal resilience. In *ICITS*, 2009.
- [56] Arpita Patra, Ashish Choudhury, and C Pandu Rangan. Efficient asynchronous multiparty computation with optimal resilience. *Cryptology ePrint Archive*, 2008.
- [57] Antigoni Polychroniadou, Gilad Asharov, Benjamin Diamond, Tucker Balch, Hans Buehler, Richard Hua, Suwen Gu, Greg Gimler, and Manuela Veloso. Prime match: A {Privacy-Preserving} inventory matching system. In *USENIX*, 2023.
- [58] Marc Rivinius. Mpc with publicly identifiable abort from pseudorandomness and homomorphic encryption. In *EUROCRYPT*, 2025.
- [59] Victor Shoup. NTL: A Library for doing Number Theory. <https://libntl.org/>, 2021.
- [60] Yuan Su, Yuan Lu, Jiliang Li, Yuyi Wang, Chengyi Dong, and Qiang Tang. Dumbo-mpc: Efficient fully asynchronous mpc with optimal resilience. *Cryptology ePrint Archive*, 2024.
- [61] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. <https://github.com/emp-toolkit>, 2016.
- [62] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. Ferret: Fast extension for correlated ot with small communication. In *CCS*, 2020.
- [63] Andrew C Yao. Protocols for secure computations. In *SFCS*, 1982.

Appendix A.

Ideal Functionalities.

Definition A.1 (Ideal Computation with Identifiable Abort). *Let $f : (\{0, 1\}^*)^n \rightarrow (\{0, 1\}^*)^n$ be an n -party function and let $C \subset [1, n] \cup \{HP\}$ denote the set of all corrupt parties. Then, the joint ideal execution of f under $(\mathcal{S}, C)$ on input vector $\vec{x} = (x_1, \dots, x_n)$, auxiliary input aux to $\mathcal{S}$ and security parameter κ , denoted $ideal_{f, C, \mathcal{S}(aux)}(\vec{x}, \kappa)$, is defined as the output vector of $P_1, \dots, P_n$ 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 $(y_1, \dots, y_n) = f(x'_1, \dots, x'_n)$. The trusted party sends $\{y_j\}_{j \in C}$ to $\mathcal{S}$.
- 3) Simulator $\mathcal{S}$ responds to trusted party: The simulator can send (abort, i^*) where $i^* \in C$ to the trusted party.
- 4) Trusted party answers parties:
 - a) If the trusted party received (abort, i^*) from the simulator $\mathcal{S}$, it sends $(\perp, i^*)$ to all parties $\mathcal{P} \setminus \{C\}$.
 - b) Otherwise, it sends y_j to every P_j , for all $j \in [n]$.
- 5) Outputs: Honest parties always output the message received from the trusted party, while corrupt parties output nothing. The simulator $\mathcal{S}$ outputs an arbitrary function of the initial inputs $\{x_i\}_{i \in C}$, the messages received by the corrupt parties from the trusted party and its auxiliary input.

Definition A.2 (Ideal Computation with G.O.D.). *Let $f : (\{0, 1\}^*)^n \rightarrow (\{0, 1\}^*)^n$ be an n -party function and let $C \subset [1, n] \cup \{HP\}$ denote the set of all corrupt parties. Then, the joint ideal execution of f under $(\mathcal{S}, C)$ on input vector $\vec{x} = (x_1, \dots, x_n)$, auxiliary input aux to $\mathcal{S}$ and security parameter κ , denoted $ideal_{f, C, \mathcal{S}(aux)}(\vec{x}, \kappa)$, is defined as the output vector of $P_1, \dots, P_n$ 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. The trusted party computes $(y_1, \dots, y_n) = f(x'_1, \dots, x'_n)$ where x'_i is substituted with default input in case the value sent by P_i ($i \in C$) is invalid.
3. Trusted party sends y_j to every P_j , for all $j \in [n]$.
4. Outputs: Honest parties always output the message received from the trusted party, while corrupt parties output nothing. The simulator $\mathcal{S}$ outputs an arbitrary function of the initial inputs $\{x_i\}_{i \in C}$, the messages received by the corrupt parties from the trusted party and its auxiliary input.

Appendix B.

Proof of Security: Alhena

In this section, we show that Alhena achieves non-colluding security with identifiable fairness in the helper-aided MPC model with a semi-honest HP (Theorem 3.2). We design two simulators $\mathcal{S}_A$ and $\mathcal{S}_{HP}$ corresponding to the case of adversary corrupting $n - 1$ regular parties maliciously or the HP in a semi-honest manner respectively.

Indistinguishability. We now argue that the view generated by $\mathcal{S}_A$ is indistinguishable to the real-world view of the malicious adversary corrupting $n - 1$ parties. In the preprocessing phase, the simulation proceeds exactly as per the real-world protocol in the $\mathcal{F}_{MS-OLE-IA}$ -hybrid model.

In the online phase, we make the following observations:

(a) For input gates, if P_d is corrupt, then $\mathcal{S}_A$ does not send any messages. Thus, the view is indistinguishable trivially. If P_d is honest, then $\mathcal{S}_A$ sends uniformly random value m_v , in contrast to the real protocol where m_v is computed as $m_v = v + \delta_v$, where v denotes the honest party's input. Since the mask δ_v is computationally indistinguishable from a uniformly random value by the security of PRF (used to derive $[\delta_v]_{HP}$ with key k_d unknown to $\mathcal{A}$), the indistinguishability holds.

(b) For multiplication gates, we first note that both in the simulation and the real protocol, $[m_z]_{HP}$ received by the $\mathcal{A}$ is a uniformly sampled value. The only difference is that in the real protocol, m_z computed by HP could be incorrect if the HP uses $[m_z]_{\mathcal{P}}' \neq [m_z]_{\mathcal{P}}$ to reconstruct m_v . This is unlike in the simulation, where m_v is always reconstructed using the correct share $[m_z]_{\mathcal{P}}$. It follows from unforgeability of the information-theoretic MAC scheme that the following occurs only with negligible probability: $\mathcal{A}$ was able to compute $t'_{[m_z]_{\mathcal{P}}}$ and $[m_z]_{\mathcal{P}}' \neq [m_z]_{\mathcal{P}}$ such that $t'_{[m_z]_{\mathcal{P}}} = \Delta \cdot [m_z]_{\mathcal{P}}' - r_{[m_z]_{\mathcal{P}}}$ holds.

(c) Lastly, for the output gate, m_z the views is identically distributed to the real protocol as it decodes to the correct output y .

Protocol $\mathcal{S}_A$

Notation: Let $\mathcal{A}$ be the malicious adversary corrupting $C \subset \mathcal{P}$ maliciously where $|C| \leq n - 1$. Let $h \in \mathcal{P}$ denote the index of the honest party. Recall that in this case, HP is honest as well. Let $\mathcal{F}_f^{\text{id-fair}}$ denote the ideal functionality computing f with identifiable fairness (Definition 2.2).

Preprocessing: $\mathcal{S}_A$ computes the preprocessing data honestly as per the protocol specifications. Note that since $\mathcal{S}_A$ acts on behalf of one of the honest regular parties P_h as well as HP, the *entire preprocessing data is known to $\mathcal{S}_A$* . During simulation of $\mathcal{F}_{MS-OLE-IA}$, if $\mathcal{S}_A$ receives (abort, i^*) (where $i^* \in C$) from $\mathcal{A}$, then $\mathcal{S}_A$ invokes the ideal functionality $\mathcal{F}_f^{\text{id-fair}}$ with (abort, i^*) to complete the simulation. Otherwise, the simulation continues.

Online: $\mathcal{S}_A$ does the following for each gate:

- **Input gate:** Let P_d be the input provider. We have two cases based on whether $P_d \in C$ or not.
 - 1) If $P_d \in C$, then $\mathcal{S}_A$ receives m_v from $\mathcal{A}$. $\mathcal{S}_A$ extracts P_d 's input by computing $x_d = m_v - \delta_v$. If P_d does not send m_v , then x_d is set as default input.
 - 2) If $P_d \notin C$, then HP samples a uniformly random masked input m_v on behalf of honest P_d and sends m_v to $\mathcal{A}$ on behalf of HP. $\mathcal{S}_A$ invokes the ideal functionality $\mathcal{F}_f^{\text{id-fair}}$ with $(\{x_d\}_{d \in C})$ and obtains the output y in response.
- **Addition gate:** $\mathcal{S}_A$ does nothing, as there is no communication in this step.
- **Multiplication gate:** $\mathcal{S}_A$ sends uniformly sampled $[m_z]_{HP}$ to $\mathcal{A}$ on behalf of HP. $\mathcal{S}_A$ (on behalf of HP) computes $m_z = [m_z]_{HP} + [m_z]_{\mathcal{P}}$, where $[m_z]_{\mathcal{P}}$ can be computed on behalf of P_h .
- **Output gate:** $\mathcal{S}_A$ computes $\delta_z = y - m_z$, where y denote the output received from the ideal functionality. $\mathcal{S}_A$ (on behalf of HP) sends $[\delta_z]_{HP} = \delta_z - [\delta_z]_{\mathcal{P}}$ to $\mathcal{A}$ (where $[\delta_z]_{\mathcal{P}}$ is known on behalf of P_h).

Figure 14: Simulator for the case of $(n - 1)$ malicious corruptions

Guaranteed output delivery in online phase.. Finally, we note that since $\mathcal{S}_A$ invokes $\mathcal{F}_f^{\text{id-fair}}$ with abort only during the preprocessing phase, the protocol Alhena clearly achieves guaranteed output delivery in the online phase whenever preprocessing succeeds.

Indistinguishability.. We note that the only difference between the view generated by $\mathcal{S}_{HP}$ and the real-world view of the adversary corrupting HP is in the input stage (as all other simulation steps are identical to the real protocol). $\mathcal{S}_{HP}$ sends uniformly random value m_v to HP, in contrast to the real protocol where m_v is computed as $m_v = v + \delta_v$, where v denotes the honest party's input. Since the mask δ_v is computationally indistinguishable from a uniformly random value by the security of PRF (used to derive $[\delta_v]_{\mathcal{P}}$ with key $k_{\mathcal{P}}$ unknown to HP), the indistinguishability claim holds.

Protocol $\mathcal{S}_{\text{HP}}$

Adversary: Let $\mathcal{A}$ be semi-honest adversary corrupting HP. In this case, all parties in $\mathcal{P}$ are honest. The $\mathcal{S}_{\text{HP}}$ has no input or output.

Preprocessing: $\mathcal{S}_{\text{HP}}$ acts honestly on behalf of parties in $\mathcal{P}$.

Online: $\mathcal{S}_{\text{HP}}$ does the following for each gate:

- **Input gate:** $\mathcal{S}_{\text{HP}}$ sends a random value m_v^i to HP on behalf of each honest $P_i \in \mathcal{P}$. $\mathcal{S}_{\text{HP}}$ would then receive the same value from $\mathcal{A}$ (on behalf of HP).
- **Addition gate:** $\mathcal{S}_{\text{HP}}$ does nothing, as there is no communication in this step.
- **Multiplication gate:** $\mathcal{S}_{\text{HP}}$ computes $[m_z]_{\mathcal{P}}, t_{[m_z]_{\mathcal{P}}}$ honestly as per protocol specifications and sends it to HP on behalf of each $P_i \in \mathcal{P}$.
- **Output gate:** $\mathcal{S}_{\text{HP}}$ does nothing, as there is no communication in this step.

Figure 15: Simulator for the case of semi-honest HP

Appendix C.

Proof of Security: Wasat

Lemma C.1. $\Pi_{\text{MS-OLE-IA}}^{\text{async}}$ realizes $\mathcal{F}_{\text{MS-OLE-IA}}^{\text{async}}$ (Figure 12).

Proof. To prove this, we show that if HP outputs $d \neq a \cdot b - c$ or aborts, then all the honest parties agree on a common cheater. Else, the output obtained by the HP must be correct. We have the following two cases.

- 1) Suppose all the received messages are correctly formed. Then, by the correctness of Π_{OLE} , the output received by a non-aborting HP must be correct, that is, $a \cdot b - c$.
- 2) Even when all the received messages were correctly formed, the adversary can enforce abort by sending an abort signal to the HP.

The simulation depends on when the network adversary delivers the abort message from a corrupt party, say P_i to HP.

- *Fast delivery:* HP does not output anything. The simulator sends $(\text{lateabort}, i, T, d)$ with $d = \perp$ to the functionality.
- *Late delivery (after HP has computed the correct output):* The simulator sends $(\text{deliver}, \text{lateabort}, i, T)$ to the functionality, in which case HP, acting as a receiver, receives the correct output.

Since this is forwarded by the HP along with the identity of the sender (of the abort signal), this leads to all honest parties agreeing on a common cheater.

- 3) If all the received messages are not correctly formed, then there exists a round r in Π_{OLE} , corresponding to which the first sender message received by the HP was malformed. Let this message received by HP be denoted as τ_S^{r, i^*} from the party P_{i^*} . Since HP sends (i^*, τ_S^{r, i^*}) to all the parties, and each honest party locally recomputes the correct sender message τ_S^r using prior round messages from HP. Thus, all honest parties identify P_{i^*} as a cheater since $\tau_S^{r, i^*} \neq \tau_S^r$ and abort the protocol.

For protocol $\Pi_{\text{MS-OLE-IA}}^{\text{async}}$ we construct the simulator $\mathcal{S}_{\mathcal{A}-\Pi_{\text{MS-OLE-IA}}^{\text{async}}}$ (Figure 16) for the case where $\mathcal{A}$ maliciously corrupts $n - 1$ parties in $\mathcal{P}$ and $\mathcal{S}_{\text{HP}-\Pi_{\text{MS-OLE-IA}}^{\text{async}}}$ (Figure 17) where $\mathcal{A}$ corrupts HP in a semi-honest manner. Intuitively, we rely on the security of Π_{OLE} to argue security for $\Pi_{\text{MS-OLE-IA}}^{\text{async}}$. Let $\mathcal{S}_{\Pi_{\text{OLE}}}$ be the simulator for Π_{OLE} .

Protocol $\mathcal{S}_{\mathcal{A}-\Pi_{\text{MS-OLE-IA}}^{\text{async}}}$

Notation: Let $\mathcal{A}$ be the malicious adversary corrupting $C \subset \mathcal{P}$ maliciously where $|C| \leq n - 1$. Let $h \in \mathcal{P}$ denote the index of the honest party. Recall that in this case, HP is honest as well.

Simulator:

- Consider a round r of Π_{OLE} in $\Pi_{\text{MS-OLE-IA}}$:
- $\mathcal{S}_{\mathcal{A}}$ generates message $\tau_S^{(h, r)}$ for honest sender P_h by following the steps in Π_{OLE} honestly, using (a, c) and randomness derived from $k_{\mathcal{P}}$ (known to $\mathcal{S}_{\mathcal{A}}$ emulating the setup and acting on behalf of HP). $\mathcal{S}_{\mathcal{A}}$ sends $\tau_S^{(h, r)}$ to $\mathcal{A}$.
- $\mathcal{S}_{\mathcal{A}}$ waits to receive $(r, \tau_S^{(j, r)})$ from $\mathcal{A}$ on behalf of HP.
- Invoke $\mathcal{S}_{\Pi_{\text{OLE}}}$ with $\tau_S^{(j, r)}$ to obtain the receiver message τ_R^{r+1} and send $(r + 1, j, \tau_S^{(j, r)}, \tau_R^{r+1})$ to $\mathcal{A}$ on behalf of HP.
- If $\tau_S^{(j, r)} \neq \tau_S^{(h, r)}$, send abort to HP on behalf of P_h ; P_h terminates with output $(\perp, j)$. Depending on when the abort message is received on behalf of HP, $\mathcal{S}_{\mathcal{A}}$ invokes $\mathcal{F}_{\text{MS-OLE-IA}}^{\text{async}}$ with $(\text{lateabort}, j, T, d)$, where T is the delay specified by the network adversary and d is either $\perp$ or the output of Π_{OLE} (which can be computed when $\mathcal{S}_{\Pi_{\text{OLE}}}$ extracts the corrupt sender's input (a^*, c^*)).

- If in any round r , an abort is received on behalf of HP from P_k but the $\tau_S^{(j,r)}$ messages were consistent, $\mathcal{S}_A$ sends (abort, P_k) to $\mathcal{P}$. Depending on when the abort message is received on behalf of HP, $\mathcal{S}_A$ invokes $\mathcal{F}_{\text{MS-OLE-IA}}^{\text{async}}$ with (deliver, lateabort, k, T) if the transcript was complete or with (lateabort, $k, T, \perp$) if the transcript was incomplete; where T is the delay specified by the network adversary.
- If $\tau_S^{(j,r)}$ verified for each r and no abort was received on behalf of HP, then $\mathcal{S}_A$ invokes $\mathcal{F}_{\text{MS-OLE-IA}}^{\text{async}}$ with (deliver).

Figure 16: Simulator for $\Pi_{\text{MS-OLE-IA}}^{\text{async}}$ in the case of $(n-1)$ malicious corruptions

Protocol $\mathcal{S}_{\text{HP}} - \Pi_{\text{MS-OLE-IA}}^{\text{async}}$

Adversary: Let $\mathcal{A}$ be semi-honest adversary corrupting HP. In this case, all parties in $\mathcal{P}$ are honest.

Simulator:

- In the first round $r = 1$, $\mathcal{S}_{\text{HP}}$ obtains the messages $\tau_S^{(j,1)}$ for P_j from $\mathcal{S}_{\Pi_{\text{OLE}}}$. $\mathcal{S}_{\text{HP}}$ sends $\tau_S^{(j,1)}$ to $\mathcal{A}$ on behalf of each party in $\mathcal{P}$ (since all parties in $\mathcal{P}$ compute the same message using the same inputs and randomness).
- Consider a round $r > 1$ of Π_{OLE} in $\Pi_{\text{MS-OLE-IA}}$:
- $\mathcal{S}_{\text{HP}}$ waits to receive a message $(j, \tau_S^{(j,r-1)}, \tau_R^r)$ from $\mathcal{A}$ on behalf of each party in $\mathcal{P}$.
- It computes $\tau_S^{(j,r+1)}$ by invoking $\mathcal{S}_{\Pi_{\text{OLE}}}$ with τ_R^r . It sends $\tau_S^{(j,r+1)}$ to $\mathcal{A}$ on behalf of each party in $\mathcal{P}$.

Figure 17: Simulator for $\Pi_{\text{MS-OLE-IA}}^{\text{async}}$ in the case of semi-honest HP

In this section, we show that Wasat achieves non-colluding security with identifiable fairness in the helper-aided asynchronous MPC model with a semi-honest HP (Theorem 4.1). We design two simulators $\mathcal{S}_A$ and $\mathcal{S}_{\text{HP}}$ corresponding to the case of adversary corrupting $n-1$ regular parties maliciously or the HP in a semi-honest manner respectively.

Indistinguishability. We now argue that the view generated by $\mathcal{S}_A$ is indistinguishable to the real-world view of the malicious adversary corrupting $n-1$ parties. In the preprocessing phase, the simulation proceeds exactly as per the real-world protocol in the $\mathcal{F}_{\text{MS-OLE-IA}}^{\text{async}}$ -hybrid model.

In the online phase, we make the following observations:

(a) For input gates, if P_d is corrupt, then $\mathcal{S}_A$ does not send any messages. Thus, the view is indistinguishable trivially. If P_d is honest, then $\mathcal{S}_A$ sends uniformly random value m_v , in contrast to the real protocol where m_v is computed as $m_v = v + \delta_v$, where v denotes the honest party's input. Since the mask δ_v is computationally indistinguishable from a uniformly random value by the security of PRF (used to derive $[\delta_v]_{\text{HP}}$ with key k_d unknown to $\mathcal{A}$), the indistinguishability claim holds.

(b) For multiplication gates, we first note that both in the simulation and the real protocol, $[m_z]_{\text{HP}}$ received by the $\mathcal{A}$ is a uniformly sampled value. The only difference is that in the real protocol, m_z computed by HP could be incorrect if the HP uses $[m_z]_{\mathcal{P}}' \neq [m_z]_{\mathcal{P}}$ to reconstruct m_v . This is unlike in the simulation, where m_v is always reconstructed using the correct share $[m_z]_{\mathcal{P}}$. This occurs only if $\mathcal{A}$ was able to compute $t'_{[m_z]_{\mathcal{P}}}$ and $[m_z]_{\mathcal{P}}' \neq [m_z]_{\mathcal{P}}$ such that $t'_{[m_z]_{\mathcal{P}}} = \Delta \cdot [m_z]_{\mathcal{P}} - r_{[m_z]_{\mathcal{P}}}$ holds. It follows from unforgeability of the information-theoretic MAC scheme that this occurs only with negligible probability.

(c) Lastly, for the output gate, m_z the views is identically distributed to the real protocol as it decodes to the correct output y .

Indistinguishability. We note that the only difference between the view generated by $\mathcal{S}_{\text{HP}}$ and the real-world view of the adversary corrupting HP is in the input stage (as all other simulation steps are identical to the real protocol). $\mathcal{S}_{\text{HP}}$ sends uniformly random value m_v to HP, in contrast to the real protocol where m_v is computed as $m_v = v + \delta_v$, where v denotes the honest party's input. Since the mask δ_v is computationally indistinguishable from a uniformly random value by the security of PRF (used to derive $[\delta_v]_{\mathcal{P}}$ with key $k_{\mathcal{P}}$ unknown to HP), the indistinguishability claim holds.

Protocol $\mathcal{S}_A$

Notation: Let $\mathcal{A}$ be the malicious adversary corrupting $C \subset \mathcal{P}$ maliciously where $|C| \leq n-1$. Let $h \in \mathcal{P}$ denote the index of the honest party. Recall that in this case, HP is honest as well. Let $\mathcal{F}_f^{\text{id-fair}}$ denote the ideal functionality computing f with identifiable fairness (Definition 2.2).

Preprocessing: $\mathcal{S}_A$ computes the preprocessing data honestly as per the protocol specifications. Note that since $\mathcal{S}_A$ acts on behalf of one of the honest regular parties P_h as well as HP, the *entire preprocessing data is known to $\mathcal{S}_A$* . We have the following cases:

- 1) If $\mathcal{S}_A$ sends (deliver) to $\mathcal{F}_{\text{MS-OLE-IA}}^{\text{async}}$, the simulation continues as below.
- 2) If $\mathcal{S}_A$ sends (deliver, lateabort, i^*, T) to $\mathcal{F}_{\text{MS-OLE-IA}}^{\text{async}}$, then simulation continues as below up until T messages are received from $\mathcal{A}$. Then, the simulation terminates.
- 3) If $\mathcal{S}_A$ sends (lateabort, i^*, T, d) to $\mathcal{F}_{\text{MS-OLE-IA}}^{\text{async}}$, $\mathcal{S}_A$ invokes the ideal functionality $\mathcal{F}_f^{\text{id-fair}}$ with (abort, i^*). $\mathcal{S}_A$ executes the

protocol (on behalf of HP) with corrupt parties as per protocol specifications using default inputs for honest parties. The simulation terminates after receiving T messages from the $\mathcal{A}$.

Online: $\mathcal{S}_{\mathcal{A}}$ does the following for each gate:

- **Input gate:** Let P_d be the input provider. We have two cases based on whether $P_d \in C$ or not.
 - 1) If $P_d \in C$, then $\mathcal{S}_{\mathcal{A}}$ receives m_v from $\mathcal{A}$. $\mathcal{S}_{\mathcal{A}}$ extracts P_d 's input by computing $x_d = m_v - \delta_v$. If P_d does not send m_v , then x_d is set as default input.
 - 2) If $P_d \notin C$, then HP samples a uniformly random masked input m_v on behalf of honest P_d and sends m_v to $\mathcal{A}$ on behalf of HP.
- $\mathcal{S}_{\mathcal{A}}$ invokes the ideal functionality $\mathcal{F}_f^{\text{id-fair}}$ with $(\{x_d\}_{d \in C})$ and obtains the output y in response.
- **Addition gate:** $\mathcal{S}_{\mathcal{A}}$ does nothing, as there is no communication in this step.
- **Multiplication gate:** $\mathcal{S}_{\mathcal{A}}$ sends uniformly sampled $[m_z]_{\text{HP}}$ to $\mathcal{A}$ on behalf of HP. $\mathcal{S}_{\mathcal{A}}$ (on behalf of HP) computes $m_z = [m_z]_{\text{HP}} + [m_z]_{\mathcal{P}}$, where $[m_z]_{\mathcal{P}}$ can be computed on behalf of P_h .
- **Output gate:** $\mathcal{S}_{\mathcal{A}}$ computes $\delta_z = y - m_z$. $\mathcal{S}_{\mathcal{A}}$ (on behalf of HP) sends $[\delta_z]_{\text{HP}} = \delta_z - [\delta_z]_{\mathcal{P}}$ to $\mathcal{A}$ (where $[\delta_z]_{\mathcal{P}}$ is known on behalf of P_h).

Figure 18: Simulator for the case of $(n - 1)$ malicious corruptions

Protocol $\mathcal{S}_{\text{HP}}$

Adversary: Let $\mathcal{A}$ be semi-honest adversary corrupting HP. In this case, all parties in $\mathcal{P}$ are honest. The $\mathcal{S}_{\text{HP}}$ has no input or output.

Preprocessing: $\mathcal{S}_{\text{HP}}$ acts honestly on behalf of parties in $\mathcal{P}$.

Online: $\mathcal{S}_{\text{HP}}$ does the following for each gate:

- **Input gate:** $\mathcal{S}_{\text{HP}}$ sends a random value m_v^i to HP on behalf of each honest $P_i \in \mathcal{P}$. $\mathcal{S}_{\text{HP}}$ would then receive the same value from $\mathcal{A}$ (on behalf of HP).
- **Addition gate:** $\mathcal{S}_{\text{HP}}$ does nothing, as there is no communication in this step.
- **Multiplication gate:** $\mathcal{S}_{\text{HP}}$ computes $[m_z]_{\mathcal{P}}, t_{[m_z]_{\mathcal{P}}}$ honestly as per protocol specifications and sends it to HP on behalf of each $P_i \in \mathcal{P}$.
- **Output gate:** $\mathcal{S}_{\text{HP}}$ does nothing, as there is no communication in this step.

Figure 19: Simulator for the case of semi-honest HP