Asynchronous MPC with Abort

Ananya Appan, David Heath, Ling Ren

University of Illinois Urbana Champaign
{aappan2, daheath, renling}@illinois.edu

Abstract. Most prior works on secure Multi-Party Computation (MPC) in asynchronous networks study Guaranteed Output Delivery (GOD), meaning that all parties learn the function output. Asynchronous MPC protocols with GOD necessarily tolerate only $t < n/3$ corruptions, and they necessarily allow the adversary to exclude the inputs of up to t honest parties from the computation, a phenomenon referred to as *input loss*. Seeking improvements to threshold/input loss, we consider weakening GOD to *security with abort*, a standard notion studied in the context of synchronous networks. Unfortunately we show that, when these standard notions are applied in asynchrony, it is not possible to improve the corruption threshold or the input loss.

We therefore study relaxations of these standard notions under which protocols *can* be improved. In particular, we propose a relaxation of the standard notion of *correctness* for asynchronous MPC protocols. Our relaxed notion requires only that parties obtain the correct output when all parties are honest and when at most a threshold A of the parties are asynchronous. We present several impossibility and feasibility results that completely characterize what is possible in the context of our relaxed correctness. For instance, it is possible to achieve selective abort even when $t < n$ parties are corrupt if (and only if) $A < (n - t)/2$, but it is impossible to achieve unanimous abort unless $t < n/3$, even when $A = 0$. We additionally propose a new notion of identifiable abort for asynchronous networks (aIA), and we show that we can achieve fair MPC with aIA and $\min(A, t)$ input loss.

1 Introduction

A secure multi party computation (MPC) protocol [Yao86,GMW87] allows mutually distrusting parties to jointly compute a function of their private inputs while keeping their inputs hidden, even if a fraction of the parties are *corrupt* and deviate arbitrarily from the protocol. If the number of corrupt parties is below a certain threshold, some MPC protocols ensure that all honest parties learn the function output. This property, called guaranteed output delivery (GOD), is the strongest notion of *output delivery* that MPC protocols achieve.

Many MPC protocols achieve *weaker* notions of output delivery that allow parties to *abort* the protocol without learning the function output when corrupt behavior is detected. Only when all parties are honest and follow the protocol does the protocol ensure *correctness*, i.e., all parties are guaranteed to obtain the function output. Such protocols can, in certain network settings, achieve better corruption tolerance than protocols with GOD, often even tolerating *any* number of corruptions.

Almost all existing works on MPC with abort are in the synchrony model, which assumes that each message that a party sends is delivered within a known bounded time Δ. While synchrony is a simple and clean model, it may not hold in real-world networks. An arguably more realistic communication model is asynchrony, which assumes that each message is delivered eventually but can be arbitrarily delayed by the adversary.

Most prior works on asynchronous MPC only consider GOD. Asynchronous MPC protocols with GOD have two drawbacks. First, they have a lower corruption threshold ($t < n/3$) than synchronous MPC protocols with GOD ($t < n/2$). Second, the function output may exclude the inputs of up to t honest parties. Unfortunately, since we cannot distinguish between a corrupt party and an honest party that is slow, it is not possible to reduce this so-called *input loss* [BOKR94].

In some sense, it is not surprising that prior works on asynchronous MPC did not consider protocols that abort. It is easy to show that relaxing GOD to abort alone cannot help increase the corruption threshold or reduce input loss. In this work, we ask the following natural question:

Under what conditions can the corruption threshold or input loss of asynchronous MPC protocols be improved by allowing parties to abort?

We answer this fundamental question by completely characterizing how computationally-secure asynchronous MPC protocols with abort can and cannot increase the corruption threshold and reduce input loss.

1.1 Our Contributions

Allowing abort alone does not improve the corruption threshold or the input loss in asynchrony, because standard notions of abort require *correctness*, that is, parties must obtain the function output when all parties behave honestly. This is impossible to achieve in asynchrony, because parties cannot differentiate between a corrupt party that remains silent and an honest party that is slow.

In an attempt to find a notion of security that improves the corruption threshold, we relax the notion of correctness. Specifically, our notion of **relaxed correctness** (rC) requires that all parties compute the function output only when no party deviates from the protocol *and less than a certain threshold of parties communicate asynchronously*. This relaxation is in line with the spirit of synchronous MPC with abort, which achieves correctness only in the best case where all parties behave honestly. Analogously, in a potentially asynchronous network, the best case occurs when all parties behave honestly *and most parties communicate synchronously*.

Our main technical contributions are lower and upper bounds that characterize what corruption threshold and input loss can and cannot be achieved under relaxed correctness and variants of abort.

Improving Corruption Threshold. We start by asking if MPC protocols with abort can tolerate a higher corruption threshold.

Abort alone does not help. We first consider MPC with standard correctness and with selective abort (SA), the weakest variant typically considered. SA allows parties to disagree on whether or not to abort, but it ensures that those parties who do not abort compute the same output. Unfortunately, asynchronous MPC with SA is possible if and only if $t < n/3$, which is the same corruption threshold for MPC with GOD. A recent work [LS25] gives a brief proof sketch of this lower bound, and we give a more detailed proof in §4.1.

Relaxing correctness. Suppose we relax correctness to require that all parties compute the function output only when no party deviates from the protocol and when at most a threshold A of parties communicate asynchronously. Can we now increase the corruption threshold?

MPC with selective abort: When we relax correctness, we show that asynchronous MPC with selective abort is possible if and only if $A < (n - t)/2$.

Negative result for MPC with unanimous abort: Encouraged by the above result, we consider a stronger notion of output delivery with abort, Unanimous Abort (UA), which guarantees that all parties *agree* on whether or not to abort the protocol. Unfortunately, we show that even when $A = 0$ (i.e, parties must compute an output only when all parties are honest *and* the network is synchronous), MPC with UA is possible only if $t < n/3$, resulting in no improvement in corruption threshold over asynchronous MPC with GOD.

Reducing Input Loss. An MPC protocol has input loss L if the inputs of at most L honest parties are *not* used to compute the function output. We next ask if, by allowing parties to abort, we can reduce this input loss.

Selective abort alone does not help. Although MPC with abort (with standard correctness) does not help to improve the corruption threshold, can it help reduce input loss? Unfortunately, the answer is also negative. We show that any asynchronous MPC protocol with SA incurs at least t input loss, the same input loss incurred by asynchronous MPC protocols with GOD.

Table 1: The corruption threshold of asynchronous MPC protocols for various output delivery notions. rC denotes our relaxed notion of correctness.

Security	**Threshold**		**Possible?**	**Reference**
	Corruption	**Asynchrony**		
GOD	$t < n/3$	$A < n$	✓	[BOKR94]
SA+rC	$t \geq n/3$	$A \geq (n-t)/2$	✗	§4.1
SA + rC	$t < n$	$A < (n-t)/2$	✓	§4.2
UA + rC	$t \geq n/3$	$A = 0$	✗	§4.3

Table 2: The input loss of asynchronous MPC protocols for various output delivery notions. rC denotes our relaxed notion of correctness; aIA denotes our notion of identifiable abort suited to asynchrony.

Security	**Threshold**	**Input Loss**	**Possible?**	**Reference**
SA	any	$< \min(A, t)$	✗	§5.1
SA + rC	$t < n$	$\min(A, t)$	✓	§4.2
Fair/aIA + rC	$t < n/3$	$\min(A, t)$	✓	§5.2
IA + rC	any	$< t$	✗	§5.3

Positive results by relaxing correctness. We then investigate whether MPC with abort together with our relaxed correctness can achieve better input loss than MPC protocols with GOD, and we obtain positive results. With relaxed correctness and $t < n/3$, we can achieve $\min(A, t)$ input loss for stronger forms of output delivery with abort[1], i.e., fairness (the adversary learns the output only if all parties learn the output) and identifiable abort (IA, explained next).

IA for asynchrony. The traditional notion of IA studied in synchrony is unanimous abort plus the additional requirement that when the parties abort, they identify a common corrupt party. However, IA seems incompatible with asynchrony in general since parties cannot distinguish whether (1) a malicious party remains silent or (2) an honest party is slow. In particular, we prove (in §5.3) that it is not possible to achieve $< t$ input loss by considering the standard notion of IA even when we relax correctness. (In fact, we prove the impossibility of a slightly weaker variant of IA). We introduce a notion of IA suited to asynchrony, which we call *asynchronous Identifiable Abort* (aIA), that allows aborting parties to detect a party that either is corrupt, or that *communicates asynchronously with at least* $t + 1$ *honest parties*. We show that it is possible to achieve MPC with aIA and relaxed correctness with A input loss.

Summary of results. Table 1 summarizes our results for increasing the corruption threshold, and Table 2 summarizes our results for reducing input loss. To design our MPC protocols with SA and aIA, we design protocols for asynchronous broadcast [Bra84] with SA and aIA (with relaxed correctness). We also design a verifiable secret sharing [CGMA85,Fel87] protocol with aIA and relaxed correctness. These protocols may be of independent interest.

1.2 Related Work

Most prior works on MPC with abort assume synchrony, e.g., [GMW87,GL05,IOZ14,CL17,CHOR18]. To our knowledge, only two very recent works study MPC with abort in asynchrony [DPPR25,LS25]. Both works focus on specific cases of MPCs. Damgård et al. show that for *perfectly secure* asynchronous MPC, neither the corruption threshold nor the communication complexity benefits from abort. Li and Song show that for *constant-round* (computational) asynchronous MPC, abort helps achieve a lower communication complexity than GOD [LS25].

[1] Notions of output delivery with abort form a partial order: from weaker ones to stronger ones, we have SA $<$ UA $<$ IA and SA $<$ UA $<$ fairness.

The goal of this work is to initiate a systematic study to fully characterize when and how abort helps (or not) asynchronous MPC protocols in terms of fundamental feasibility in corruption threshold and input loss. We thus consider the more general setting of asynchronous MPC with computational security and any efficiency.

2 Technical Overview

Lower Bounds

Impossibility of SA with less than $\min(A,t)$ *input loss or with* $A \geq (n-t)/2$. These two lower bounds follow standard arguments. When no messages are heard from a set of $\min(A,t)$ parties, the remaining $n-\min(A,t)$ parties *cannot* abort the protocol. This is because correctness requires that no party aborts when all parties are honest and at most A parties are asynchronous, and, in asynchrony, parties cannot tell whether the silent parties are corrupt or are honest but slow. This directly shows we cannot achieve $< \min(A,t)$ input loss. We also cannot tolerate $A \geq (n-t)/2$ asynchronous parties for similar reasons due to the standard "split-brain" attack: an adversary that controls $n - 2\min(A,t)$ (which is $\leq t$) parties can force two sets of honest parties, each of size $\min(A,t)$, to output different values by delaying all messages between them and using one input when interacting with one set and another input with the other.

MPC with UA+rC is impossible when $t \geq n/3$. The proof for the lower bound on the corruption threshold of UA+rC MPC is more intricate and requires new techniques. To our knowledge, no prior work has attempted such a lower bound, since no prior work considers relaxed correctness.

Our proof considers M worlds, where M is an upper bound on the number of *messages* sent in the protocol, and each world corresponds to an adversary strategy. In the i-th world, the first i messages in the protocol are delivered on time, after which some party crashes. We use an inductive argument to show that if the parties do not abort in this world, then they also do not abort in the world in which the first $i-1$ messages are delivered on time, after which some party crashes. When $A = 0$, relaxed correctness enforces that no party aborts when *all* messages are delivered on time. This sequence of induction implies that no party aborts when *no* message is delivered on time, allowing an adversary to make honest parties output inconsistent values. We present details in §4.3.

Our proof can be adapted to show that a weaker variant of Byzantine agreement requires $t < n/3$ in asynchrony. It is well known that Byzantine agreement with the standard validity property (i.e., if all honest parties input x, then the output is x) requires $t < n/3$ in asynchrony. [CGG$^+$23] proved that $t < n/3$ is necessary for a class of relaxed validity properties, including *weak validity* (i.e., if all parties are honest and input x, then the output is x). Our result shows that the even weaker rC-style validity (i.e., if all parties are honest and input x and the network is synchronous, then the output is x) still requires $t < n/3$.

Upper Bounds. We modify existing MPC protocols (with standard correctness) to asynchronous MPC protocols with abort (with relaxed correctness). We modify certain types of asynchronous MPC protocols with GOD that consist of three phases: an *input* phase, where parties exchange secret shares of their inputs, a *computation* phase, where they derive secret-shared outputs, and a *reconstruction* phase, where secret-shares are ultimately reconstructed to complete the protocol. The vast majority of asynchronous MPC protocols that we know of can be split into the above phases. In all cases, we yield an asynchronous protocol with $\min(A,t)$ input loss.

Using a synchronous MPC protocol Π *with SA to obtain an asynchronous MPC protocol* Π' *with SA+rC.* Since Π is designed for a synchronous network, parties may not compute an output when messages are delayed. Π' instead runs the input and computation phases of Π in a way that detects an asynchronous party each time messages are delayed. If an asynchronous party is detected, the input and computation phases of Π are re-run without including the detected party. This is repeated up to A times, after which parties either use the reconstruction phase to compute the output, or abort. §4.2 presents more details.

Our next two construction modifies the *input* phase, and uses the computation and reconstruction phases of any asynchronous MPC protocol with GOD that tolerates $t < n/3$ faults to achieve fairness asynchronous MPC with aIA and relaxed correctness.

From asynchronous MPC with GOD to fair asynchronous aIA+rC MPC. Recall that we aim to achieve $\min(A, t)$ input loss. We do this by having parties run the computation phase of the GOD protocol only if at least $n - \min(A, t)$ inputs are secret-shared within a fixed time (the time taken in a synchronous network), and allowing them to abort otherwise. To additionally identify parties that are slow or corrupt in the case of abort, we replace the *input* phase with a verifiable secret sharing protocol with the aIA and rC properties. Specifically, any party that does not secret-share its input in time or deviates from the protocol is identified to be slow or corrupt.

These, together with the negative results, fully characterize the feasibility and infeasibility results of MPC when abort with relaxed correctness is considered in asynchrony.

3 Preliminaries

3.1 Notation

- λ denotes the computational security parameter.
- $\mathcal{A}$ is a computationally bounded adversary that controls all (up to t) corrupted parties. $\mathcal{A}$ is malicious, and may arbitrarily deviate from the protocol.
- Unless otherwise specified, $\mathcal{P} = \{P_1, \ldots, P_n\}$ denotes the set of n parties.
- $\mathtt{T}_{\mathtt{N}}^{\mathtt{S},\mathtt{C}}$ denotes a protocol to solve task $\mathtt{T}$ with security $\mathtt{S}$ and correctness $\mathtt{C}$ in network $\mathtt{N}$.
- $\mathtt{T}_{1\mathtt{N}_1}^{\mathtt{S}_1,\mathtt{C}_1} \rightarrow \mathtt{T}_{2\mathtt{N}_2}^{\mathtt{S}_2,\mathtt{C}_2}$ denotes a construction that modifies protocol $\mathtt{T}_{1\mathtt{N}_1}^{\mathtt{S}_1,\mathtt{C}_1}$ to build protocol $\mathtt{T}_{2\mathtt{N}_2}^{\mathtt{S}_2,\mathtt{C}_2}$.
- $\langle m \rangle_i$ denotes message m signed by party P_i.
- $\mathtt{Comm}(\Pi)$ denotes the communication complexity of a protocol Π.
- w.o.p is an abbreviation for "with overwhelming probability". Unless otherwise specified, all our statements hold with overwhelming probability.
- We use "a party aborts" and "a party outputs $\perp$" interchangeably.

3.2 Model and Efficiency Metrics

We assume a public key infrastructure (PKI) setup: a pair of keys $(\mathtt{pk}_i, \mathtt{sk}_i)$ has been generated for each party $P_i \in \mathcal{P}$, where $\mathtt{pk}_i$ is public and $\mathtt{sk}_i$ is held only by P_i. With the help of this PKI setup, each pair of parties communicates using a private and authenticated channel. We assume that parties have a global, synchronized clock. A party is asynchronous if each messages that it sends or receives can be delayed arbitrarily, and is only guaranteed to be delivered eventually. A party is synchronous if each message that it sends will be delivered to the recipient within Δ time, unless the recipient is asynchronous.

3.3 Definitions

We define MPC with abort, where various notions of abort can be plugged in.

Definition 1 (MPC with abort). *An MPC protocol with L input loss guarantees the following when each party P_i participates with an input $x_i \in \mathbb{F}$.*

- *Correctness: When all parties are honest, no party aborts.*
- *Consistency: All parties that do not abort output the same value.*
- *Input Loss: Any party that does not abort outputs $y = F(x'_1, \ldots, x'_n)$ where there is some set $\mathcal{CS} \subseteq \mathcal{P}$ of size at least $n - L$ such that $x'_i = x_i$ for each $P_i \in \mathcal{CS}$ and $x'_i = \perp$ for each $P_i \notin \mathcal{CS}$.*
- *Privacy: Any adversary $\mathcal{A}$ learns nothing from the protocol transcript beyond the outputs of all parties.*
- *Output Delivery: See options later.*

We relax the notion of correctness in asynchronous networks to the following notion of *relaxed correctness* with parameter A.

Definition 2 (Relaxed Correctness (rC)). *When all parties are honest and at most A parties are asynchronous, no party aborts.*

We next define various standard notions of output delivery with abort, which can be used for output delivery with Definition 1.

Definition 3 (Selective Abort (SA)). *Each honest party either outputs or aborts.*

Definition 4 (Unanimous Abort (UA)). *Either all honest parties abort or no honest party aborts.*

Definition 5 (Fairness (F)). *Honest parties abort unanimously (Definition 4). If honest parties abort, then $\mathcal{A}$ outputs $\perp$.*

Definition 6 (Guaranteed Output Delivery (GOD)). *All parties output.*

We define a new notion of asynchronous identifiable abort (aIA) that allows parties to identify a party that is either corrupt or has many asynchronous links.

Definition 7 (Asynchronous Identifiable Abort (aIA)). *Parties abort unanimously (Definition 4). When parties abort, all parties agree on the identity of a party that either is corrupt or has $n-t$ asynchronous communication links.*

Note. We consider MPC protocols that consist of separate input, computation and reconstruction phases. During the input phase, parties output *secret shares* of the inputs of a set $\mathcal{CS} \subseteq \mathcal{P}$ of parties. During the computation phase, parties participate with these secret shares to obtain a secret share of the output y. During the reconstruction phase (which typically requires a single round), parties participate with secret shares of y to output y. No information about y is revealed until the reconstruction phase. We use IC to denote an execution of the input and computation phases of an MPC protocol, and CR to denote an execution of the computation and reconstruction phases.

4 Increasing the Corruption Threshold

4.1 A Lower Bound when $A \geq (n-t)/2$

In this section, we show that no protocol $\mathtt{MPC}^{\mathtt{SA},\mathtt{rC}}_{\mathtt{as}}$ can guarantee output when $A \geq (n-t)/2$ parties are asynchronous. Our proof follows a well known lower bound proof that uses a "split-brain" attack to prove that asynchronous agreement is possible only if $t < n/3$ [DLS88].

Theorem 1. *There exists no MPC protocol with $A \geq \frac{n-t}{2}$ when $t \geq n/3$.*

Proof. For contradiction, assume that there is such an MPC protocol Π. For contradiction, assume that $A \geq t$. Consider three sets S_1, S_2 and S_3 of sizes $\min(A,t)$, $\min(A,t)$ and $n - 2\min(A,t)$ respectively. Note that all three sets have size at most t and could potentially be corrupt. This is clear for sets S_1 and S_2. For S_3, consider two cases:

- **Case 1:** $\min(A,t) = t$**.** Then, $|S_3| \leq t$ since $n - 2t \leq t$.
- **Case 2:** $min(A,t) = A$**.** Since $2A \geq n - t$, $|S_3| = n - 2A \leq t$.

Note further that all parties in sets S_1 and S_2 can potentially be asynchronous. Suppose that y is the output computed using $\mathcal{CS} = S_1 \cup S_3$ and y' is the output computed using $\mathcal{CS}' = S_2 \cup S_3$. Consider the following worlds.

- ***W1* - *All parties in* S_1 *crash at the start of* Π**: The SA and consistency of Π guarantee that, eventually, parties in S_2 and S_3 either output y or abort. Suppose this happens at time T.
- ***W2* - $\mathcal{A}$ *delays all messages from parties in* S_1 *until time* T**: The views of parties in S_2 and S_3 are exactly the same as ***W1***. Since all parties are honest and at most $t \leq A$ parties are asynchronous, the relaxed correctness of Π ensures that all parties output y within time T (no party aborts).
- ***W3* - *All parties in* S_2 *crash at the start of* Π**: Similar to what was argued for ***W1***, parties in S_1 and S_3 eventually output y' or abort, say within time T'.
- ***W4* - $\mathcal{A}$ *delays all messages from parties in* S_2 *until time* T'**: Similar to what was argued for ***W2***, all parties in S_1 and S_3 output y' within time T'.

Attack: $\mathcal{A}$ corrupts all parties in S_3, and delays all messages between S_1 and S_2 beyond time $\max(T,T')$. Parties in S_3 behave towards parties in S_2 as in ***W2***, and towards parties in S_1 as in ***W4***. Parties in S_2 output y and parties in S_1 output y', violating consistency (Definition 1).

4.2 An Upper Bound when $A < (n-t)/2$ for any $t < n$

In this section, we describe how we can design an asynchronous MPC protocol with SA and relaxed correctness ($\mathtt{MPC}_{\mathtt{s}}^{\mathtt{SA,rC}}$) when $A < (n-t)/2$ using any synchronous MPC protocol with SA and standard correctness ($\mathtt{MPC}_{\mathtt{s}}^{\mathtt{SA,C}}$). The key challenge that we face is that even when all parties are honest, $\mathtt{MPC}_{\mathtt{s}}^{\mathtt{SA,C}}$ allows parties to abort when messages are delayed. But $\mathtt{MPC}_{\mathtt{s}}^{\mathtt{SA,rC}}$ should guarantee that parties compute the function output even when messages from up to A parties are delayed.

A Straw-man Attempt. As a straw-man attempt, suppose that parties run $\mathtt{MPC}_{\mathtt{s}}^{\mathtt{SA,C}}$ and, if messages are delayed, repeat $\mathtt{MPC}_{\mathtt{s}}^{\mathtt{SA,C}}$ until no messages are delayed and $\mathtt{MPC}_{\mathtt{s}}^{\mathtt{SA,C}}$ outputs for all parties. While seemingly natural, there are multiple issues with the above straw-man attempt. We next describe each of these issues and how we fix them.

Ensuring Privacy. Running $\mathtt{MPC}_{\mathtt{s}}^{\mathtt{SA,C}}$ repeatedly allows the adversary to possibly learn multiple function outputs, each using a different set of adversarial inputs. Towards a fix to this issue, observe that unless the reconstruction phase of $\mathtt{MPC}_{\mathtt{s}}^{\mathtt{SA,C}}$ starts, the secret-shares in the computation phase reveal nothing about the final output. Thus, we can re-run the *input and computation* phases repeatedly, and only run the reconstruction phase when the input and computation phases successfully complete without message delays.

This plan requires the reconstruction phase to complete even when up to A parties are asynchronous. However, MPC protocols with SA typically use additive secret sharing[2], which requires the participation of *all* parties to complete reconstruction. We fix this by transforming a protocol $\mathtt{MPC}_{\mathtt{s}}^{\mathtt{SA,C}}$ with separate input, computation, and reconstruction phases into a protocol $\mathtt{ThMPC}_{\mathtt{s}}^{\mathtt{SA,C}}$ that guarantees the reconstruction phase completes even if up to A parties are asynchronous. We achieve this by having each party re-share the share that was output during the computation phase via a threshold (Shamir) secret-sharing [Sha79].

Guaranteeing Termination. Repeatedly running the input and computation phases of $\mathtt{ThMPC}_{\mathtt{s}}^{\mathtt{SA,C}}$ alone cannot guarantee that any of these attempts terminate. While an asynchronous network eventually delivers messages between honest parties, a corrupt party can behave as though it is asynchronous forever. However, observe that we only need to guarantee that parties compute a function output when all parties are honest and up to A parties are asynchronous. If an honest party could identify that it is asynchronous or that more than A parties are asynchronous, it could back out of the protocol. The protocol would have to be re-run at most A times, after which parties could decide to abort. To achieve this, we present `Aug`: a procedure that "augments" Π so that parties can compute the output according to a run of Π in a *synchronous* network, or identify that they should back out of the protocol.

Ensuring Consistency. When removing asynchronous parties while re-running `ThMPC`, parties need to *agree* on which parties to remove. To achieve this, we construct and use a broadcast protocol (with SA+rC) that allows a sender to broadcast a message that all parties agree on (even if the sender is corrupt). After running an augmented version of the input and computation phases of $\mathtt{ThMPC}_{\mathtt{s}}^{\mathtt{SA,C}}$, a party that backs out can broadcast a message to indicate this.

We next describe each part of our solution in detail.

Enabling reconstruction when A parties are asynchronous. Figure 1 presents a general transformation from any $\mathtt{MPC}_{\mathtt{s}}^{\mathtt{SA,C}}$ with separate computation and reconstruction phases into an MPC protocol $\mathtt{ThMPC}_{\mathtt{s}}^{\mathtt{SA,C}}$ that guarantees *threshold reconstruction*: the reconstruction phase completes even when only $t+1$ honest parties are synchronous. $\mathtt{ThMPC}_{\mathtt{s}}^{\mathtt{SA,C}}$ builds on top of those of $\mathtt{MPC}_{\mathtt{s}}^{\mathtt{SA,C}}$ by using *Shamir secret-sharing [Sha79];* a scheme that allows a secret to be shared among n parties, but only requires $t+1$ shares to reconstruct the secret [3].

[2] A value x is additively secret shared among parties in $\mathcal{P}$ if each $P_i \in \mathcal{P}$ holds a value x_i such that $\sum_{P_i \in \mathcal{P}} x_i = x$.

[3] A secret s is Shamir secret-shared by a sampling a t-degree polynomial f uniformly at random such that $f(0) = s$. Each share is an evaluation of f at a distinct point. Threshold reconstruction is possible since $t+1$ evaluations of a t-degree polynomial uniquely define it.

During $\texttt{ThMPC}_{\texttt{s}}^{\texttt{SA,C}}$, parties first run the input and computation phase of $\texttt{MPC}_{\texttt{s}}^{\texttt{SA,C}}$ to obtain secret-shares of the function output. To enable threshold reconstruction, each party P_i uses Shamir-secret sharing to re-shares its share y_i. Each party additionally *signs* Shamir secret-shares while distributing them. This ensures that corrupt parties cannot provide incorrect shares during the reconstruction phase. During the reconstruction phase, parties announce the Shamir secret-shares of each y_i. When $t+1$ honest parties are synchronous, this allows all shares to be reconstructed, and the output to be computed.

Protocol $\texttt{ThMPC}_{\texttt{s}}^{\texttt{SA,C}}$

Input and Computation Phases:

- (Rounds 1 to R) Run the input and computation phases of $\texttt{MPC}_{\texttt{s}}^{\texttt{SA,C}}$ ($\texttt{IC}_{\texttt{s}}^{\texttt{SA,C}}$) to either compute a share y_i of the output, or abort.
- (Round $R+1$, Shamir-sharing) Do the following:
  - Let f be a t-degree polynomial with $f(0) = y_i$ and every other coefficient sampled uniformly at random. Send $(\texttt{share}, \langle f(j)\rangle_i)$ to each party P_j.
  - If $(\texttt{share}, \langle f_{ji}\rangle_j)$ is received from each $P_j \in \mathcal{P}$, output shares $\{f_{ji}\}_{P_j \in \mathcal{P}}$. Else, abort.

Reconstruction Phase: Perform the following.

- (Round $R+2$) For each $P_j \in \mathcal{P}$, send $(\texttt{recons}, j, \langle f_{ji}\rangle_j)$ to all parties.
- (Round $R+3$) If $t+1$ $(\texttt{recons}, j, \langle f_{kj}\rangle_k)$ messages are received $\forall P_j \in \mathcal{P}$,
  - Interpolate these shares to compute a t-degree polynomial $f_j(\cdot)$ and set $y_j = f_j(0)$.
  - Compute an output y using $\{y_j\}_{P_j \in \mathcal{P}}$ according to the reconstruction phase of $\texttt{MPC}_{\texttt{s}}^{\texttt{SA,C}}$.

  Else, abort.

Fig. 1: Code for party P_i to run a synchronous MPC protocol with SA that allows share reconstruction even when A parties are asynchronous. R denotes the number of rounds used by $\texttt{IC}_{\texttt{s}}^{\texttt{SA,C}}$.

Theorem 2. *If* $\texttt{MPC}_{\texttt{s}}^{\texttt{SA,C}}$ *is an MPC with protocol with SA in synchrony, then*

- $\texttt{ThMPC}_{\texttt{s}}^{\texttt{SA,C}}$ *is also an MPC protocol with SA in synchrony with the same input loss as* $\texttt{MPC}_{\texttt{s}}^{\texttt{SA,C}}$.
- *If* $t+1$ *parties are synchronous at the start of the reconstruction phase of* $\texttt{ThMPC}_{\texttt{s}}^{\texttt{SA,C}}$ *and all parties are honest, then, at the end of the reconstruction phase, synchronous parties compute a function output.*

Proof. If all parties are honest, then by the correctness of $\texttt{IC}_{\texttt{s}}^{\texttt{SA,C}}$, each P_i outputs a valid secret share y_i and Shamir secret-shares y_i to all parties. During the reconstruction phase, even if only $t+1$ parties are synchronous, each synchronous party P_i will receive $t+1$ Shamir secret-shares of y_j for each $P_j \in \mathcal{P}$. P_i will thus be able to reconstruct $\{y_j\}_{P_j \in \mathcal{P}}$ and compute y according to the reconstruction phase of $\texttt{MPC}_{\texttt{s}}^{\texttt{SA,C}}$. Further, note that by round $\mathcal{R}(\texttt{IC}_{\texttt{s}}^{\texttt{SA,C}}) + 3$, each party either computes a function output or aborts. Due to the guarantees of $\texttt{MPC}_{\texttt{s}}^{\texttt{SA,C}}$ in a synchronous network, parties that do not abort compute the same function output, with the same input loss incurred by $\texttt{MPC}_{\texttt{s}}^{\texttt{SA,C}}$.

Communication. Each party sends a signed Shamir share to every other party. $\texttt{ThMPC}_{\texttt{s}}^{\texttt{SA,C}}$ incurs an $O(n^2|\sigma|)$ additive overhead in communication over $\texttt{MPC}_{\texttt{s}}^{\texttt{SA,C}}$.

Augmenting synchronous protocols. `Aug` (presented in Figure 2) is a procedure that "augments" a synchronous protocol Π into an asynchronous protocol in which each party can either compute the output according to a synchronous run of Π or decide to back out. A party backs out only if it convinced that either A parties are asynchronous, or that it is asynchronous.

During $\texttt{Aug}(\Pi)$, parties run Π while additionally *relaying* messages. Specifically, (1) if a party P_i has to send a message to P_j in round r of Π, it instead sends an encryption of this message to *all* parties in round $2r$ of $\texttt{Aug}(\Pi)$ (2) if a party P_i receives an ciphertext that was meant for party P_k during round $2r$, then P_i sends this ciphertext to P_k in round $2r+1$.

While running $\texttt{Aug}(\Pi)$, each party P_i computes a set $\texttt{asyncSet}_i$ of parties that are, in its view, asynchronous. To do this, if P_i did not receive any message from party P_j during any round (relay or otherwise), P_i adds P_j to $\texttt{asyncSet}_i$. If $\texttt{asyncSet}_i$ is of size at most A, then P_i computes its output according to protocol Π. Otherwise, it identifies that it is asynchronous.

Due to relays, parties that have $\texttt{asyncSet}$ of size at most A hear each other's messages on time. This is because each of these parties communicates synchronously with at least $n - A$ parties. Since two sets of size $n - A$ have an intersection of size at least $n - 2A > f$, any two parties that output a message communicate synchronously with a common honest party that can relay messages between them. This allows parties with $\texttt{asyncSet}$ of size at most A to compute an output consistent with a run of Π in a synchronous network.

Protocol $\texttt{Aug}(\Pi)$

- Initialize $\texttt{asyncSet}_i$ to $\{\ \}$.
- For $r = 0, \ldots, R$, do the following.
  - For each message $\texttt{msg}$ to be sent in round r of Π, if the recipient of this message is P_j, send $(\texttt{Enc}_{\texttt{pk}_j}(\texttt{msg}), P_j)$ to all parties. If no message is to be sent, send a $\texttt{ping}$ to all parties.
  - For each message $(\texttt{Enc}_{\texttt{pk}_j}(\texttt{msg}), P_k)$ received from P_j in the previous round, send $\texttt{msg}$ to parties P_j and P_k. If no message is to be sent to party P_k from P_j, send a $\texttt{ping}$ to party P_k instead.
  - Add P_k to $\texttt{asyncSet}_i$ if:
    - $(\texttt{Enc}_{\texttt{pk}_j}(\texttt{msg}), P_j)$ was sent to P_k during round $2r$, but not received from P_k at the end of round $2r + 1$.
    - no $\texttt{ping}$ or $(\texttt{Enc}_{\texttt{pk}_j}(\texttt{msg}), P_i)$ was received from P_k at the end of round $2r + 1$.
- If $|\texttt{asyncSet}_i| > A$, output $\texttt{backout}$. Otherwise, output y, where y is the output computed according to Π, based on messages received.

Fig. 2: Code for party P_i that augments a synchronous protocol Π to additionally guarantee that synchronous parties output when $A < (n - t)/2$. R denotes the number of rounds used by Π

Lemma 1. $\texttt{Aug}(\Pi)$ *guarantees the following.*

- *Relaxed Correctness: When $A < (n - t)/2$ and all parties are honest, each synchronous party computes an output according to Π and does not output* $\texttt{backout}$.
- *Consistency: If a set $\mathcal{H}$ of honest parties compute an output, each $P_i \in \mathcal{H}$ computes an output according to a run of Π in a synchronous network when parties in $\mathcal{H}$ are honest and parties in $\mathcal{P} \setminus \mathcal{H}$ are corrupt.*
- *Termination: Each honest party either computes an output according to Π, or outputs* $\texttt{backout}$.

Proof. Relaxed Correctness. Consider the case when at most A parties are asynchronous and all parties are honest. All synchronous parties will have $\texttt{asyncSet}$ of size at most A, and will not output $\texttt{backout}$.

Consistency. Let $\mathcal{H}$ be the set of honest parties that compute an output according to Π. We will prove that for any two parties $P_i, P_j \in \mathcal{H}$, if P_i sends a message $\texttt{msg}$ to P_j during round r, then P_j receives this by the end of round $r + 1$. If this holds, consistency follows since this emulates a run of Π where parties in $\mathcal{H}$ communicate synchronously with each other, and parties in $\mathcal{P} \setminus \mathcal{H}$ may behave arbitrarily.

Consider sets $S_i = \mathcal{P} \setminus \texttt{asyncSet}_i$ and $S_j = \mathcal{P} \setminus \texttt{asyncSet}_j$. Since $\texttt{asyncSet}_i$ and $\texttt{asyncSet}_j$ are of size $\leq A$, S_i and S_j are of size at least $n - A$. This means that $|S_i \cap S_j| \geq n - 2A > t$ [4]. Hence, S_i and S_j have some honest party, say P_h, in common. Since P_i received $(\texttt{Enc}_{\texttt{pk}_j}(\texttt{msg}), P_j)$ from P_h at the end of round $2r + 1$, P_h received $\texttt{Enc}_{\texttt{pk}_j}(\texttt{msg})$ from P_i at the end of round $2r$, and sends this to P_j during round $2r + 1$. Since $P_h \in S_j$, P_j receives $\texttt{Enc}_{\texttt{pk}_j}(\texttt{msg})$ at the end of round $2r + 1$.

Termination. Holds since, after $2\mathcal{R}(\Pi)$ rounds, each party either computes an output according to Π or outputs $\texttt{backout}$.

Communication. $\texttt{Aug}(\Pi)$ incurs an $O(n)$ multiplicative overhead over Π.

[4] Follows since $|S_i \cap S_j| = |S_i| + |S_j| - |S_i \cup S_j| \leq (n - A) + (n - A) - n = n - 2A$.

Broadcast with SA+rC. Our broadcast protocol is an augmented version of a *synchronous* broadcast protocol $\mathtt{BC}_{\mathsf{s}}^{\mathtt{GOD},\mathtt{C}}$ that provides the following guarantees, even if the network is asynchronous.

Let $\mathcal{H}_{\mathsf{s}}$ be the set of honest parties that communicate synchronously with each other. After running $\mathtt{BC}_{\mathsf{s}}^{\mathtt{GOD},\mathtt{C}}$, each party in $\mathcal{H}_{\mathsf{s}}$ outputs either a message or ϕ, such that:

- *Consistency: All parties in $\mathcal{H}_{\mathsf{s}}$ output the same value.*
- *Correctness: If S is honest, then*
  - *each honest party outputs ϕ or m.*
  - *if $\mathsf{S} \in \mathcal{H}_{\mathsf{s}}$, then all parties in $\mathcal{H}_{\mathsf{s}}$ output m.*

In Appendix A, we prove that the Dolev-Strong [DS83] protocol guarantees the above properties. But such a protocol does not provide any guarantees for honest parties that are *asynchronous*. However, it turns out than an augmented version of the $\mathtt{BC}_{\mathsf{s}}^{\mathtt{GOD},\mathtt{C}}$ protocol suffices.

Lemma 2. *Consider the following protocol: parties run $\mathtt{Aug}(\mathtt{BC}_{\mathsf{s}}^{\mathtt{GOD},\mathtt{C}})$ and abort if they output* `backout` *and otherwise compute the output according to $\mathtt{BC}_{\mathsf{s}}^{\mathtt{GOD},\mathtt{C}}$. This protocol is an asynchronous broadcast protocol with SA+rC ($\mathtt{BC}_{\mathtt{as}}^{\mathtt{SA},\mathtt{rC}}$) that guarantees the following properties when $A < (n-t)/2$:*

- *Relaxed Correctness: If S is honest, then*
  - *each party that does not abort outputs m or ϕ.*
  - *if S does not abort, parties that do not abort output m.*
  - *if at most A parties are asynchronous and all parties are honest, then each synchronous party does not abort.*
- *Selective Abort: See Definition 3.*
- *Consistency: Parties that do not abort output the same value.*

Proof. Let $\mathcal{H}_{\mathtt{out}}$ be the set of honest parties that compute an output according to $\mathtt{BC}_{\mathsf{s}}^{\mathtt{GOD},\mathtt{C}}$ (and do not output `backout`) during $\mathtt{Aug}(\mathtt{BC}_{\mathsf{s}}^{\mathtt{GOD},\mathtt{C}})$. By the consistency of `Aug` (Lemma 1), parties in $\mathcal{H}_{\mathtt{out}}$ compute an output according to a run of $\mathtt{BC}_{\mathsf{s}}^{\mathtt{GOD},\mathtt{C}}$ in a synchronous network in which parties in $\mathcal{P} \setminus \mathcal{H}_{\mathtt{out}}$ are corrupt and parties in $\mathcal{H}_{\mathtt{out}}$ communicate synchronously with each other (like $\mathcal{H}_{\mathsf{s}}$). By the consistency property of $\mathtt{BC}_{\mathsf{s}}^{\mathtt{GOD},\mathtt{C}}$, all parties in $\mathcal{H}_{\mathtt{out}}$ output the same value. By the correctness property of $\mathtt{BC}_{\mathsf{s}}^{\mathtt{GOD},\mathtt{C}}$, when S is honest, each party in $\mathcal{H}_{\mathtt{out}}$ outputs m or ϕ, and outputs m if $\mathsf{S} \in \mathcal{H}_{\mathtt{out}}$. By the relaxed correctness of `Aug`, when at most A parties are asynchronous, each synchronous honest party computes an output according to $\mathtt{BC}_{\mathsf{s}}^{\mathtt{GOD},\mathtt{C}}$. Finally, selective abort holds due to the termination of `Aug`.

Communication. `DS` incurs a communication of $O(n^3|\sigma|)$ bits. When `DS` is used to instantiate $\mathtt{BC}_{\mathsf{s}}^{\mathtt{GOD},\mathtt{C}}$, $\mathtt{Aug}(\mathtt{DS})$ incurs a communication of $O(n^4|\sigma|)$ bits.

MPC with RSA when $A < (n-t)/2$. We put everything together to obtain our final $\mathtt{MPC}_{\mathtt{as}}^{\mathtt{SA},\mathtt{rC}}$ protocol (presented in Figure 3). Parties run an augmented version of the input and computation phases of $\mathtt{ThMPC}_{\mathsf{s}}^{\mathtt{SA},\mathtt{C}}$. Parties that output `backout` broadcast an `NOK` message indicating this to all parties, and otherwise broadcast `OK`. If a party's broadcast outputs `NOK` of ϕ, that party is considered asynchronous, and $\mathtt{Aug}(\mathtt{ThIC}_{\mathsf{s}}^{\mathtt{SA},\mathtt{C}})$ is run again, without including asynchronous parties. This is repeated at most A times, after which parties that are synchronous and continue to run the protocol (i.e, did not output `backout` and did not abort) reconstruct the output.

On computing an output y, each party sends an $(\mathtt{out}, y)$ message to all parties to inform asynchronous parties about the output. If all parties are honest and at most A parties are asynchronous, parties can expect to receive $t+1$ $(\mathtt{out}, y)$ messages within a fixed time (see step 5 of Figure 3). If they do not receive these, they send `abort` messages indicating their intention to abort. It can be assured that parties either receive $t+1$ $(\mathtt{out}, y)$ or $A+1$ `abort` messages, ensuring that all parties terminate.

Theorem 3. *The protocol presented in Figure 3 is an MPC protocol with SA+rC when $A < (n-t)/2$.*

Protocol $\mathtt{MPC}_{\mathtt{s}}^{\mathtt{SA,C}} \rightarrow \mathtt{MPC}_{\mathtt{as}}^{\mathtt{SA,rC}}$

1. Initialize $\mathtt{asyncSet}$ to be an empty set $\{\ \}$. Initialize $\mathtt{flag}_i$ to 0.
2. For $\mathtt{epoch} = 1, 2, \ldots$:
   - If $|\mathtt{asyncSet}| \geq A$, set $\mathtt{flag}_i$ to 1 and break. Otherwise, let $\mathcal{P}' = \mathcal{P} \setminus \mathtt{asyncSet}$. Run $\mathtt{Aug}(\mathtt{ThIC}_{\mathtt{s}}^{\mathtt{SA,C}})$ among parties in $\mathcal{P}'$.
   - (Round $\mathtt{epoch} \cdot (R_1 + 1)$) If $\mathtt{Aug}(\mathtt{ThIC}_{\mathtt{s}}^{\mathtt{SA,C}})$ outputs $\mathtt{backout}$, use $\mathtt{BC}_{\mathtt{as}}^{\mathtt{SA,rC}}(i)$ to broadcast $\mathtt{NOK}$. Otherwise, use $\mathtt{BC}_{\mathtt{as}}^{\mathtt{SA,rC}}(i)$ to broadcast $\mathtt{OK}$.
   - (Round $\mathtt{epoch} \cdot (R_1 + R_2)$) Perform the following.
     - If $\mathtt{BC}_{\mathtt{as}}^{\mathtt{SA,rC}}(j)$ aborts for any $P_j \in \mathcal{P}'$, set $\mathtt{flag}_i$ to 1 and break.
     - Else, for each $P_j \in \mathcal{P}'$, if $\mathtt{NOK}$ or ϕ is output during $\mathtt{BC}_{\mathtt{as}}^{\mathtt{SA,rC}}(j)$, add P_j to $\mathtt{asyncSet}$. If the size of $\mathtt{asyncSet}$ increases, continue to the next epoch. Otherwise, break.
3. (Round $\mathtt{epoch} \cdot (R_1 + R_2) + 1$) If $\mathtt{flag}_i = 0$, run $\mathtt{ThRec}_{\mathtt{s}}^{\mathtt{SA,C}}$ among parties in $\mathcal{P}'$.
4. (Round $\mathtt{epoch} \cdot (R_1 + R_2) + R_3$) If $\mathtt{flag}_i = 0$ and an output y is obtained at the end of $\mathtt{ThRec}_{\mathtt{s}}^{\mathtt{SA,C}}$, send $\langle \mathtt{out}, y \rangle_i$ to all parties.
5. (Round $\mathtt{epoch} \cdot (R_1 + R_2) + R_3 + 1$) If $C_{t+1}(\mathtt{out}, y)$ is not received, send $\mathtt{abort}$ to all parties. Else, forward $C_{t+1}(\mathtt{out}, y)$ to all parties.
6. At any point of time:
   - Abort on receiving $A + 1$ $\mathtt{abort}$ messages.
   - Output y on receiving $C_{t+1}(\mathtt{out}, y)$.

Fig. 3: Code for party P_i to run an asynchronous MPC protocol with SA+rC when $A < (n - t)/2$. $C_{t+1}(x)$ is a certificate of $t + 1$ different signatures on the same value x. R_1, R_2 and R_3 refer to the number of rounds used by $\mathtt{Aug}(\mathtt{ThIC}_{\mathtt{s}}^{\mathtt{SA,C}})$, $\mathtt{BC}_{\mathtt{as}}^{\mathtt{SA,rC}}$ and $\mathtt{ThRec}_{\mathtt{s}}^{\mathtt{SA,C}}$, respectively.

Proof. Relaxed Correctness. We prove that each synchronous party computes a common output y during step 4. This would mean that at least $n - A > t$ parties send $(\mathtt{out}, y)$ to all parties, and that all parties eventually output y. Since at most A asynchronous parties send $\mathtt{abort}$, no party aborts.

During each epoch, by the relaxed correctness of $\mathtt{Aug}$, no synchronous party outputs $\mathtt{backout}$ during $\mathtt{Aug}(\mathtt{ThIC}_{\mathtt{s}}^{\mathtt{SA,C}})$ and only asynchronous parties broadcast $\mathtt{NOK}$ messages. By the relaxed correctness of $\mathtt{BC}_{\mathtt{as}}^{\mathtt{SA,rC}}$, each synchronous party obtains an output during all instances of $\mathtt{BC}_{\mathtt{as}}^{\mathtt{SA,rC}}$, and no synchronous party P_i outputs ϕ during $\mathtt{BC}_{\mathtt{as}}^{\mathtt{SA,rC}}(j)$ for any synchronous P_j. Hence, each synchronous party only adds asynchronous parties to $\mathtt{asyncSet}$. By the consistency of $\mathtt{BC}_{\mathtt{as}}^{\mathtt{SA,rC}}$, all synchronous parties compute the same $\mathtt{asyncSet}$.

Since at most A parties are asynchronous, the size of $\mathtt{asyncSet}$ grows at most A times, and parties eventually execute an epoch e where the size of $\mathtt{asyncSet}$ does not increase. During this epoch, $\mathtt{BC}_{\mathtt{as}}^{\mathtt{SA,rC}}(j)$ outputs $\mathtt{OK}$ for each $P_j \in \mathcal{P}'$. By the consistency of $\mathtt{Aug}$, all parties in $\mathcal{P}'$ compute an output according to $\mathtt{ThIC}_{\mathtt{s}}^{\mathtt{SA,C}}$. Since all parties are honest, by the correctness of $\mathtt{ThIC}_{\mathtt{s}}^{\mathtt{SA,C}}$, all parties output shares of f computed using the inputs of parties in $\mathcal{P}'$.

All parties in $\mathcal{P}'$ start the reconstruction phase. Since $n - A > t$ parties are synchronous, the reconstruction phase completes for all synchronous parties while they execute step 4, and the $n - A > t$ synchronous parties compute an output. Due to the guarantees of $\mathtt{ThRec}_{\mathtt{s}}^{\mathtt{SA,C}}$ in a synchronous network, all synchronous parties output the same value y.

Consistency. Consider two parties P_i and P_j that output y and y', respectively. P_i and P_j must have received $(\mathtt{out}, y)$ and $(\mathtt{out}, y')$ from some honest parties P_h and $P_{h'}$. Since $\mathtt{flag}_h$ and $\mathtt{flag}_{h'}$ are 0, P_h and $P_{h'}$ do not abort during any instance of $\mathtt{BC}_{\mathtt{as}}^{\mathtt{SA,rC}}$. By the consistency of $\mathtt{BC}_{\mathtt{as}}^{\mathtt{SA,rC}}$, P_h and $P_{h'}$ agree on $\mathcal{P}'$ for all epochs, and both parties compute their output during the same epoch, say e using the same $\mathcal{P}'$. Since the size of $\mathtt{asyncSet}$ does not increase during epoch e, both P_h and $P_{h'}$ compute an output according to $\mathtt{Aug}(\mathtt{ThIC}_{\mathtt{s}}^{\mathtt{SA,C}})$ run during epoch e and use $\mathtt{ThRec}_{\mathtt{s}}^{\mathtt{SA,C}}$ to reconstruct the corresponding shares. By the consistency of $\mathtt{Aug}$, these shares are computed according to a run of $\mathtt{ThMPC}_{\mathtt{s}}^{\mathtt{SA,C}}$ in which P_i and P_j are honest. By the consistency of $\mathtt{ThMPC}_{\mathtt{s}}^{\mathtt{SA,C}}$ in a synchronous network, $y = y'$ must hold.

Selective Abort. Since parties run at most A epochs, all honest parties execute step 5 of Figure 3, meaning that all honest parties either send an $\mathtt{abort}$ message, or send $t + 1$ $(\mathtt{out}, y)$ messages for the same value y. Consider two cases:

- **Case 1 - At least one honest party P_h receives $C_{t+1}(\texttt{out}, y)$:** P_h forwards these to all parties. All honest parties eventually receive these and output y.
- **Case 2 - No honest party receives $C_{t+1}(\texttt{out}, y)$:** All $n - t > A$ honest parties send `abort` messages. All honest parties eventually receive more than A `abort` messages and abort.

Input Loss. Any party that computes an output does so using $\mathcal{P}'$ of size at least $n - A$. By using [DPSZ12] to instantiate $\texttt{ThMPC}_{\texttt{s}}^{\texttt{SA,C}}$, we can obtain $\texttt{ThMPC}_{\texttt{s}}^{\texttt{SA,C}}$ with no input loss, following which $\texttt{MPC}_{\texttt{as}}^{\texttt{SA,rC}}$ incurs at most A input loss.

Privacy. Follows from the privacy of $\texttt{MPC}_{\texttt{s}}^{\texttt{SA,C}}$, and the fact that parties compute the output corresponding to an instance of $\texttt{MPC}_{\texttt{as}}^{\texttt{SA,rC}}$ for a single set $\mathcal{P}'$.

Communication. $\texttt{MPC}_{\texttt{as}}^{\texttt{SA,rC}}$ incurs a communication of $O(A) \cdot \texttt{Comm}(\texttt{Aug}(\texttt{ThIC}_{\texttt{s}}^{\texttt{SA,C}})) + n \cdot \texttt{Comm}(\texttt{BC}_{\texttt{s}}^{\texttt{SA,C}}) + \texttt{Comm}(\texttt{ThRec})$. Substituting relevant quantities, this is $O(nA) \cdot \texttt{Comm}(\texttt{IC}_{\texttt{s}}^{\texttt{SA,C}}) + O(n^5|\sigma|)$.

4.3 Impossibility of MPC with UA+rC when $t \geq n/3$ and $A = 0$

Even by relaxing correctness and assuming $A = 0$, we cannot improve the corruption threshold of asynchronous MPC with UA beyond $t \geq n/3$. We prove the impossibility result for three parties $\mathcal{P} = \{P_1, P_2, P_3\}$ among which one is corrupt; player partitioning can be used to extend the impossibility proof to the case when $t \geq n/3$.

Since $A = 0$, when any UA+rC protocol Π is executed in a synchronous network when all parties are honest, all parties must complete the protocol with overwhelming probability. We consider the output of Π in executions where the first i messages are delivered on time, and then some party P_j crashes, and the remaining two parties behave honestly. Let O_i^j denote this output (which is a random variable). At a high level, our proof shows that, with overwhelming probability, for any number of steps i and regardless of which party P_j crashes, $O_i^j \neq \perp$, i.e., the parties do not abort. This means that when any party crashes at the *start* of the protocol, the remaining two parties need to compute a function output rather than abort. This allows the adversary to perform a split-brain attack to violate UA.

Notation. We assume Π to be any PPT protocol, that is, the round complexity of Π is upper bounded by some value T that is polynomial in the security parameter λ. Without loss of generality, we assume that *all* executions of Π proceed for T rounds, and that each party sends a message to every other party during each round. For any protocol Π, this can be arranged by extending Π to run for T rounds, and making parties send "dummy" messages whenever they have no messages to send. Thus, the protocol sends $M = 6T$ messages in total ($\binom{3}{2} = 6$ per round). We can order the messages by round (the 6 messages within a round are ordered arbitrarily). Let P_{r_i} (where $1 \leq i \leq M$) denote the party that receives the i-th message in an execution of Π. Note that P_{r_i} is fixed in advance for all i.

We first show that whether $O_i^j \neq \perp$ does not depend on which party P_j crashes.

Claim 1 *If $O_i^j \neq \perp$ w.o.p, then $O_i^k \neq \perp$ w.o.p.*

Proof. We first prove that $\Pr\left[O_i^k \neq \perp \mid O_i^j \neq \perp\right]$ is overwhelming. For contradiction, suppose that $O_i^k = \perp$ with some non negligible probability ϵ. An adversary $\mathcal{A}$ that corrupts P_ℓ ($P_\ell \neq P_i$ and $P_\ell \neq P_j$) can perform the following attack that succeeds with probability ϵ:

- $\mathcal{A}$ allows the first i messages to be delivered synchronously, after which it delays all messages between P_j and P_k.
- After the first i messages are delivered synchronouly, $\mathcal{A}$ makes P_ℓ interact with P_j as though P_k crashed, and with P_k as though P_j crashed.

Since $O_i^j \neq \perp$, P_k does not abort. However, if $O_i^k = \perp$ with probability ϵ, P_j aborts with non-negligible probability, violating unanimous abort. Now,

$$\begin{aligned}\Pr\left[O_i^k \neq \perp\right] &\geq \Pr\left[O_i^k \neq \perp \wedge O_i^j \neq \perp\right] \\ &= \Pr\left[O_i^k \neq \perp \mid O_i^j \neq \perp\right] \times \Pr\left[O_i^j \neq \perp\right]\end{aligned}$$

This is overwhelming since the product of two overwhelming probabilities is overwhelming.

We next show that $O_i^{r_i}$ cannot differ from $O_{i-1}^{r_i}$, where r_i is the receiver of message i.

Claim 2 $O_i^{r_i} = O_{i-1}^{r_i}$.

Proof. Any party $P_j \in \mathcal{P} \setminus \{P_{r_i}\}$ cannot distinguish between an execution in which P_{r_i} crashes *before* receiving m_i and one in which P_{r_i} crashes just *after* receiving m_i. Hence, the random variables $O_i^{r_i}$ and $O_{i-1}^{r_i}$ are *exactly* the same.

Theorem 4. *There exists no protocol* $\texttt{MPC}_{\texttt{as}}^{\texttt{UA,rC}}$ *for any* $t \geq n/3$ *when* $A = 0$.

Proof. For contradiction, suppose that there exists an asynchronous MPC protocol Π with UA+rC when $A = 0$. The relaxed correctness of Π guarantees that, w.o.p, $O_M^j \neq \perp$. By Claim 2, w.o.p, parties do not abort even when r_M crashes after the first $M-1$ messages are delivered synchronously:

$$\begin{aligned} O_M^{r_M} &\neq \perp && \text{by relaxed correctness} \\ \implies O_{M-1}^{r_M} &\neq \perp && \text{(Claim 2)}\end{aligned}$$

By Claim 1, w.o.p, parties also do not abort when r_{M-1} crashes after the first $M-1$ messages are delivered synchronously:

$$O_{M-1}^{r_M} \neq \perp \implies O_{M-1}^{r_{M-1}} \neq \perp \qquad \text{(Claim 1)}$$

By repeatedly applying Claims 1 and 2 M times, we show that when a party crashes at the start of the protocol, the remaining two parties do not abort w.o.p. We do this by considering $O_i^{r_i}$ for each message $0 \leq i \leq M$:

$$\begin{aligned} O_M^{r_M} &\neq \perp && \text{by relaxed correctness} \\ \implies O_{M-1}^{r_M} &\neq \perp && \text{(Claim 2)} \\ \implies O_{M-1}^{r_{M-1}} &\neq \perp && \text{(Claim 1)} \\ \implies O_{M-2}^{r_{M-1}} &\neq \perp && \text{(Claim 2)} \\ \implies O_{M-2}^{r_{M-2}} &\neq \perp && \text{(Claim 1)} \\ &\cdots \\ \implies O_0^{r_1} &\neq \perp && \text{(Claim 1)} \\ \implies O_0^j &\neq \perp \ \forall P_j \in \mathcal{P} && \text{(Claim 2)}\end{aligned}$$

Consider two executions in which parties P_j and P_k crash at the start of the protocol. The outputs in these executions must be computed using subsets [5] of $\mathcal{CS} = \{P_\ell, P_k\}$ and $\mathcal{CS}' = \{P_\ell, P_j\}$ respectively, where $P_\ell \in \mathcal{P} \setminus \{P_j, P_k\}$; let these outputs be y and y' . This allows a corrupt P_i to cause a split-brain attack by delaying all messages between P_j and P_k, and forcing P_j to output y' and P_k to output y.

[5] We remark that, by using the player partitioning technique, considering such subsets allows us to rule out any input loss $t \leq L \leq n$. When a party P_j crashes at the start of the protocol, the remaining two parties in $\mathcal{P} \setminus \{P_j\}$ must obtain a function output, and this rules out any protocol $\texttt{MPC}_{\texttt{as}}^{\texttt{UA,rC}}$ with input loss less than t.

5 Reducing the Input Loss

In this section, we characterize how we can reduce the input loss of asynchronous MPC protocols by allowing parties to abort.

5.1 Impossibility of MPC with SA with $< \min(A, t)$ input loss

It is impossible for an asynchronous MPC protocol to achieve less than $\min(A, t)$ even by considering SA with relaxed correctness. The below proof extends a known impossibility proof for GOD ([BLZL20], Theorem 3) to the case of SA.

Theorem 5. *For any $t < n$, there exists no protocol $\mathtt{MPC}^{\mathtt{SA,C}}_{\mathtt{as}}$ with input loss less than $\min(A, t)$.*

Proof. Partition the n parties into two sets: S_1 of size $\min(A, t)$ and S_2 of size $n - \min(A, t)$. Suppose, for contradiction, that a protocol with less than $\min(A, t)$ input loss exists. Consider the following worlds.

- ***W1 - All parties in S_1 crash at the start of the protocol***: By SA, each party in S_2 eventually outputs some value or aborts, say by time T.
- ***W2 - $\mathcal{A}$ delays all messages from parties in S_1 until time T***: The views of parties in S_2 are exactly the same as in ***W1***. Due to relaxed correctness, parties must compute a function output at time T and not abort.

In ***W2***, only the inputs of parties in S_2 can be considered to compute the output, resulting in at least $\min(A, t)$ input loss.

5.2 MPC with Asynchronous Identifiable Abort and Fairness

An aIA+rC protocol provides relaxed correctness while enabling parties to learn, upon abort, the identity of a party that is either corrupt or that has asynchronous communication links with $t + 1$ honest parties. Henceforth, we call such a party a *seemingly corrupt* party.

Here, we present how we can modify existing asynchronous MPC protocols with GOD into a protocol that outputs an asynchronous *fair* aIA+rC MPC protocol with $\min(A, t)$ input loss when $t < n/3$. Our modification is performed at the *input* phase; we do not modify the computation and reconstruction phases.

Our modification requires new building blocks that achieve aIA. In particular, parties share their inputs using a verifiable secret sharing (VSS) protocol with the aIA property ($\mathtt{VSS}^{\mathtt{aIA,rC}}_{\mathtt{as}}$). Parties proceed to the computation and reconstruction phases only if secret-shares are obtained from at least $n - \min(A, t)$ VSS instances. Otherwise, parties abort and identify the seemingly corrupt dealers.

Prior asynchronous VSS protocols do not achieve aIA, and cannot be easily modified to achieve aIA. Asynchronous VSS traditionally requires that parties output shares at the end of the sharing phase, without aborting. Since parties cannot tell whether D is corrupt or slow, the sharing phase need not terminate. While these protocols can be modified to abort the sharing phase after a timeout, they do not give a way to *identify* a seemingly corrupt party.

We build a new VSS protocol with aIA ($\mathtt{VSS}^{\mathtt{aIA,rC}}_{\mathtt{as}}$) that aborts when the *dealer* is seemingly corrupt. $\mathtt{VSS}^{\mathtt{aIA,rC}}_{\mathtt{as}}$ is built using a broadcast protocol with aIA ($\mathtt{BC}^{\mathtt{aIA,rC}}_{\mathtt{as}}$). We present our new protocols in a bottom-up manner: $\mathtt{BC}^{\mathtt{aIA,rC}}_{\mathtt{as}}$, followed by $\mathtt{VSS}^{\mathtt{aIA,rC}}_{\mathtt{as}}$, and finally, our $\mathtt{MPC}^{\mathtt{aIA,rC}}_{\mathtt{as}}$ protocol.

Broadcast with Asynchronous Identifiable Abort. A broadcast protocol with aIA+rC enables parties to either agree on the message sent by a sender S, or abort by agreeing that S is seemingly corrupt.

Definition 8 (Broadcast (BC) with aIA+rC). *A broadcast protocol with aIA+rC allows a sender S to send a message m to all parties, and achieves the following properties:*

- *Relaxed Correctness: If S is honest, each party that does not abort outputs m. If S is honest and has synchronous communication links with $> t$ honest parties, then honest parties do not abort.*
- *Consistency: If parties do not abort, they output the same value $m^\star$.*
- *Asynchronous Identifiable Abort: See Definition 7.*

To be able to identify a "slow" S with $t+1$ asynchronous communication links, we require that parties *start* $\mathtt{BC}_{\mathtt{as}}^{\mathtt{aIA,rC}}$ at similar times. We call this requirement *timely participation*.

Definition 9 (Timely Participation for protocol Π). *Let T_i denote the (global) time at which party P_i starts protocol Π. Protocol Π has timely participation if, for each honest party P_i, each honest P_j with synchronous links to P_i starts Π at time $T_j \leq T_i + \Delta$.*

We describe $\mathtt{BC}_{\mathtt{as}}^{\mathtt{aIA,rC}}$ (Figure 4) assuming that it has timely participation. $\mathtt{BC}_{\mathtt{as}}^{\mathtt{aIA,rC}}$ relies on a byzantine agreement ($\mathtt{BA}$) protocol that allows parties to agree on a common bit that is input by at least one honest party.

Definition 10 (Byzantine Agreement). *Each party P_i participates with an input b_i and outputs a bit b, with the following properties:*

- *Validity: If all honest parties input the same bit b, then all honest parties output b.*
- *Agreement: If any honest party outputs b, then all honest parties eventually output b.*
- *Termination: All honest parties eventually output.*

$\mathtt{BA}$ is used by parties to agree on whether to output S's message, or abort. We next describe $\mathtt{BC}_{\mathtt{as}}^{\mathtt{aIA,rC}}$ in more detail.

Message propagation. S sends m to all parties. Due to timely participation, parties can expect to receive m from S within time 2Δ since the start of the protocol. If a party does not receive a message within this time, it indicates this by sending a `blame` message to all parties. Each party sends a `vote` for the *first* message it receives that is signed by S (regardless of when it is received and even after sending a `blame` message).

The first instance of $\mathtt{BA}_{\mathtt{as}}$. The first BA instance is used to agree on whether at least one honest party has seen a message from S, and abort otherwise. If a party receives a message from S, it indicates this by participating in BA with input 1. If a party receives $n-t$ `blame` messages, it indicates that it would like to abort by participating with input 0. With some careful reasoning, we can show that each honest party eventually participates in this BA with some input (see proof of Theorem 6), and hence, that this instance outputs.

If the output is 0, then some honest party has received $n-t$ `blame` messages, and the parties abort the protocol by identifying S to be seemingly corrupt. If the output is 1, parties continue to run a second instance of BA.

The second instance of $\mathtt{BA}_{\mathtt{as}}$. This instance of BA is used to agree on a *common* message sent by S, or identify S to be corrupt. If a party receives a certificate of $n-t$ `vote`s for the *same* message, it forwards this to all parties, and participates in BA with input 1 indicating that it is ready to output this message. If a party receives equivocating (different) messages signed by S, it participates in BA with input 0, indicating that it is ready to abort with evidence that S is corrupt. Once again, we can prove that all honest parties eventually participate with some input, ensuring that the BA protocol terminates with an output.

If the second instance of BA outputs 0, then all parties abort. If the second BA outputs 1, then parties output a message on receiving a certificate of $n-t$ `vote`s for it. Honest parties output consistent messages since, by the standard quorum intersection argument, two certificates for *different* messages cannot form.

Theorem 6. *If protocol $\mathtt{BC}_{\mathtt{as}}^{\mathtt{aIA,rC}}$ has timely participation, then $\mathtt{BC}_{\mathtt{as}}^{\mathtt{aIA,rC}}$ is an asynchronous broadcast protocol with aIA+rC.*

Proof. We show that $\mathtt{BC}_{\mathtt{as}}^{\mathtt{aIA,rC}}$ has the following properties.

Protocol $\mathtt{BC}^{\mathtt{aIA,rC}}_{\mathtt{as}}$

- S sends $\langle m \rangle_{\mathsf{S}}$ to all parties.
- If no message is received after time 2Δ since the start of the protocol, each P_i sends $\langle \mathtt{blame} \rangle_i$ to all parties.
- Each P_i, on receiving $\langle m \rangle_{\mathsf{S}}$ for the first time (even after time 2Δ since the start of the protocol), sends $(\langle m \rangle_{\mathsf{S}}, \langle \mathtt{vote}, m \rangle_i)$ to all parties.
- Each P_i participates in $\mathtt{BA}_{\mathtt{as}}(1)$ with input 1 on obtaining $\langle m \rangle_{\mathsf{S}}$ and 0 on obtaining $C_{n-t}(\mathtt{blame})$.
- If the output of $\mathtt{BA}_{\mathtt{as}}(1)$ is 0, all parties output $\perp$ and identify S to be either slow or corrupt.
- If the output of $\mathtt{BA}_{\mathtt{as}}(1)$ is 1, each P_i participates in $\mathtt{BA}_{\mathtt{as}}(2)$ with input:
  - 1 on receiving $C_{n-t}(\mathtt{vote}, m)$. It sends $C_{n-t}(\mathtt{vote}, m)$ to all parties.
  - 0 on seeing equivocating messages $\langle m \rangle_{\mathsf{S}}$ and $\langle m' \rangle_{\mathsf{S}}$.
- If the output of $\Pi^{\mathtt{GOD,C,as}}_{\mathtt{EVBA}}(2)$ is 0, parties output $\perp$ and identify S to be corrupt. Otherwise, on receiving $C_{n-t}(\mathtt{vote}, m)$, parties output m.

Fig. 4: A broadcast protocol with aIA+rC that allows a sender S to broadcast message m. $C_{n-t}(v)$ denotes a certificate that consists of the same value v signed by $n-t$ different parties.

Relaxed Correctness. When S is honest, any party that does not abort outputs m, due to the unforgeability of the digital signature. When S has synchronous communication links with $t+1$ honest parties, due to timely participation, at most $n-t-1$ parties send $\mathtt{blame}$ messages. Each honest party eventually receives $\langle m \rangle_{\mathsf{S}}$ from S and participates in $\mathtt{BA}_{\mathtt{as}}(1)$ with input 1. By the validity and termination of $\mathtt{BA}_{\mathtt{as}}(1)$, all parties output 1 and proceed to participate in $\mathtt{BA}_{\mathtt{as}}(2)$.

No honest party inputs 0 to $\mathtt{BA}_{\mathtt{as}}(2)$ since an honest S does not equivocate. Since all $(n-t)$ honest parties eventually vote for m, a certificate $C(\mathtt{vote}, m)$ eventually forms, and each honest party participates in $\mathtt{BA}_{\mathtt{as}}(2)$ with input 1. Due to validity and termination, $\mathtt{BA}_{\mathtt{as}}(2)$ eventually outputs 1 and each party eventually outputs m.

Asynchronous Identifiable Abort. We first prove that UA holds. It is easy to see that, by the agreement property of $\mathtt{BA}_{\mathtt{as}}$, parties agree on whether to abort or not. We next prove that all parties *terminate* the protocol. Note that each honest party eventually participates in $\mathtt{BA}_{\mathtt{as}}(1)$ with some input by considering two cases.

- *Case 1: No honest party receives a message from* S *within time* 2Δ *since the start of the protocol:* All $n-t$ honest parties send $\mathtt{blame}$ messages. Each honest party eventually receives these and participates with input 0 in $\mathtt{BA}_{\mathtt{as}}(1)$.
- *Case 2: Some honest party* P_h *receives a message from* S *within time* 2Δ *since the start of the protocol:* P_h forwards this message to all parties. Each honest party eventually receives this and can participate with input 1 in $\mathtt{BA}_{\mathtt{as}}(1)$.

The termination property of $\mathtt{BA}_{\mathtt{as}}(1)$ ensures that all parties obtain an output from $\mathtt{BA}_{\mathtt{as}}(1)$. If this output is 0, all parties abort and identify S to be either slow or corrupt. When the output is 1, there are two cases:

- *Case 1: All honest parties vote for the same message* m*:* A certificate $C(\langle \mathtt{vote}, m \rangle)$ eventually forms. All honest parties can participate in $\mathtt{BA}_{\mathtt{as}}(2)$ with input 1.
- *Case 2: Two honest parties* P_i *and* P_j *vote for different messages* m *and* m'*:* Since P_i and P_j forward $\langle m \rangle_{\mathsf{S}}$ and $\langle m' \rangle_{\mathsf{S}}$ to all parties, each party eventually receives equivocation evidence against S and can participate in $\mathtt{BA}_{\mathtt{as}}(2)$ with input 0.

All parties eventually obtain an output from $\mathtt{BA}_{\mathtt{as}}(2)$ as well. If this output is 0 all parties abort. If the output is 1, then (by the validity of $\mathtt{BA}_{\mathtt{as}}(2)$) some honest party receives sends $C(\mathtt{vote}, m)$ to all parties, who eventually receive it and output m.

We next prove that S, when identified to be seemingly corrupt, is correctly identified. Consider the following two cases when parties abort.

- *Case 1 - All parties receive output* 0 *from* $\mathtt{BA}_{\mathtt{as}}(1)$*:* At least one honest party sees $C(\mathtt{blame})$, meaning that $n-t$ parties did not receive messages from S within 2Δ time since the start of the protocol. This means that $n-2t > t$ honest parties have asynchronous communication links with S.

– *Case 2 - All parties receive output* 0 *from* $\texttt{BA}_{\texttt{as}}(2)$*:* At least one honest party sees equivocating messages signed by S, indicating that S is corrupt.

Consistency. For contradiction, suppose that two honest parties P_i and P_j output different messages m and m' respectively. This means that P_i has seen a certificate C of $n-t$ votes for m, and P_j a certificate C' of $n-t$ votes for m'. Since $|C \cap C'| \geq n - 2t > t$, an honest party votes for both m and m', leading to a contradiction.

Communication Cost. Each party sends a certificate of `vote` and/or a `blame` message to every other party, costing $O(n^3(\lambda + |m|))$ bits of communication. Using threshold signatures, this can be reduced to $O(n^2(\lambda + |m|))$ bits. This is the final asymptotic cost when [CKS00] is used as $\texttt{BA}_{\texttt{as}}$.

VSS with Asynchronous Identifiable Abort. An asynchronous VSS protocol with aIA ($\texttt{VSS}_{\texttt{as}}^{\texttt{aIA,rC}}$) enables parties to either receive shares of a secret s held by a dealer D, or abort while agreeing that D is seemingly corrupt.

Definition 11 (Verifiable Secret Sharing with aIA+rC). *A VSS protocol consists of two phases, sharing and reconstruction, and allows a dealer* D *to share a secret* s *to all parties. Each party* P_i *outputs a share* s_i *such that:*

– *Relaxed Correctness: If* D *is honest, each* P_i *that does not abort outputs* s_i*. If* D *is honest and has synchronous communication links with* $> t$ *honest parties, then no honest party aborts.*
– *Privacy: If* D *is honest, then the view of an* $\mathcal{A}$ *that learns up to* t *shares can be simulated.*
– *Consistency: If parties do not abort the sharing phase, then each honest party* P_i *outputs* s_i *that is the share of some* $s^\star$*.*
– *Asynchronous Identifiable Abort: Definition 7 holds for the sharing phase.*
– *Reconstruction: If all honest parties start the reconstruction phase with a share of* s*, then all honest parties output* s *at the end of the reconstruction phase.*

Our VSS protocol performs Shamir secret-sharing [Sha79]. Asynchronous VSS protocols with GOD use *bivariate* polynomials (polynomials over two variables) to share the secret.

Definition 12 (Polynomials over a Field). *A* d*-degree univariate polynomial over* $\mathbb{F}$ *is of the form* $f(x) = \sum_{i=0}^{i=d} c_i x^i$*, where* $c_i \in \mathbb{F}$*. A* (d,d)*-bivariate polynomial over* $\mathbb{F}$ *is of the form* $F(x,y) = \sum_{i=0,j=0}^{i=d,j=d} c_{ij} x^i y^j$*, where* $c_{ij} \in \mathbb{F}$*.*

For simplicity, we will assume that the bivariate polynomial is *symmetric* i.e $F(i,j) = F(j,i)$. Asynchronous VSS protocols embed the secret s in the *constant* term of a bivariate polynomial F ($F(0,0) = s$).

Given $F(x,y)$, by fixing $y = \alpha$, we obtain a d-degree univariate polynomial $f_i(x) = F(x,i)$. $d+1$ different univariate polynomials $f_i(x)$ uniquely determine, and can be used to reconstruct, a (d,d)-bivariate polynomial $F(x,y)$, and thereby s. This is formalized in the below claim.

Lemma 3 (Claim 5.2 [AL17]). *Let* $\alpha_1, \ldots, \alpha_{d+1}$ *be* $d+1$ *unique points in* $\mathbb{F}$ *and let* $f_1(x), \ldots f_{d+1}(x)$ *be* $d+1$ d*-degree univariate polynomials. Then there exists a* (d,d)*-bivariate polynomial* $F(x,y)$ *such that* $F(x,i) = f_i(x)$ *for* $i = 1, \ldots, d+1$*.*

Asynchronous VSS protocols use (t,t)-bivariate (instead of univariate) polynomials to *guarantee output* during the sharing phase. Since we cannot distinguish between slow and corrupt parties in asynchrony, parties must output their shares on receiving messages from $n-t$ parties. This means that when D is corrupt, up to t honest parties may not have received any shares.

When VSS is performed using a bivariate polynomial, each party P_i receives a *univariate* polynomial $f_i(x) = F(x,i)$, with the constant term being s_i ($f_i(0) = s_i$). By the symmetric nature of F, each pair of parties has a point in common since $f_i(j) = f_j(i)$. In this way, $t+1$ honest parties can help the remaining honest parties reconstruct their univariate polynomials, and thereby, their shares.

A corrupt D may also send arbitrary polynomials to each party. To ensure that each party receives shares that lie on a bivariate polynomial, we use a polynomial commitment scheme.

Definition 13. *A **bivariate-polynomial commitment scheme** $\mathtt{PC}$ consists of the following algorithms:*

- $\mathtt{PC.Setup}(1^\kappa, d) \to \mathtt{pp}$*: Generates public parameters to commit to a polynomial over* $\mathbb{F}$ *of degree* d*. These parameters are input to all other functions provided by* $\mathtt{PC}$*, and are omitted for clarity.*
- $\mathtt{PC.Commit}(F(x,y), n) \to (\mathtt{Com}, \mathtt{wit})$*: For a bivariate polynomial* $F(x,y)$ *and the number of evaluations* n*, outputs a witness* $\mathtt{wit}$ *and a commitment* $\mathtt{Com}$*.*
- $\mathtt{PC.Eval}(\mathtt{wit}, F(x,y), \alpha, \beta) \to (v, \pi)$*: Evaluates* $F(\alpha, \beta)$ *to generate a proof* π *that can be used to check that the evaluation is correct i.e that* $F(\alpha, \beta) = v$*.*
- $\mathtt{PC.Verify}(\mathtt{Com}, \alpha, \beta, v, \pi) \to \{0, 1\}$*: Given a bivariate polynomial commitment* $\mathtt{Com}$*, outputs* 1 *if* $v = F(\alpha, \beta)$*; it outputs* 0 *otherwise.*

Prior works (§3 of [AJM+23], §2.2 of [BCV24]) describe how such a commitment scheme can be constructed by committing to $O(d^2)$ values.

We require the bivariate polynomial commitment scheme to satisfy the standard properties of *correctness*, *hiding*, and *binding*. It is *hiding* if, except for negligible probability, an adversary that is provided with the polynomial commitment and up to d^2 evaluations of the polynomial *cannot* learn an evaluation of the polynomial that is not provided to it. It is *binding* if, except for negligible probability, the adversary cannot generate a valid proof for an incorrect evaluation. It is *correct* if every proof of evaluation that is generated honestly is successfully verified. [KZG10] provides a formal definition.

We present protocol $\mathtt{VSS}_{\mathtt{as}}^{\mathtt{aIA,rC}}$ in Figure 5. We describe just the sharing phase; any asynchronous protocol to reconstruct Shamir secret shared values can be used to perform the reconstruction phase.

Share Distribution. D picks a uniformly random (t,t)-bivariate polynomial $F(x,y)$ with s as the constant term and computes a univariate polynomial $f_i(x) = F(x,i)$ to be sent to each party P_i. D uses a first instance of broadcast ($\mathtt{BC}_{\mathtt{as}}^{\mathtt{GOD,C}}(1)$) to broadcast the shares of *all* parties, along with a polynomial commitment $\mathtt{Com}$ that the parties use to check the validity of their received polynomial. For privacy, each party's polynomial is encrypted using the party's public key. While it is not typical for D to broadcast the polynomials of *all* parties, this step is crucial for our construction since it ensures that either all parties obtain their polynomial or all parties abort.

The remaining steps resemble those in existing asynchronous VSS protocols.

Voting. On receiving $f_i(x)$, each P_i uses $\mathtt{Com}$ to check that $f_i(x)$ is consistent with the committed bivariate polynomial. Each party sends a $\mathtt{vote}$ message along with either $\mathtt{OK}$ or $\mathtt{NOK}$ to inform other parties about whether this check passes. Upon receiving a set V of $n-t$ $\mathtt{vote}$ messages, D uses a second instance of broadcast ($\mathtt{BC}_{\mathtt{as}}^{\mathtt{GOD,C}}(2)$) to broadcast V along with the polynomial of each party that sent a vote with an $\mathtt{NOK}$ message. Each party then either aborts this instance, or receives an output V.

Share Computation. If V is not a set of $n-t$ votes along with the univariate polynomials of parties that voted $\mathtt{NOK}$, then parties abort the protocol. Otherwise, since $|V| = n-t$, the univariate polynomials of $(>t)$ honest parties with votes in V uniquely determine the bivariate (Lemma 3), and each party outputs its share by interpolating the points received from parties in V.

Before we prove that $\mathtt{VSS}_{\mathtt{as}}^{\mathtt{aIA,rC}}$ achieves the required properties, we show that the second instance of broadcast ($\mathtt{BC}_{\mathtt{as}}^{\mathtt{aIA,rC}}(2)$) has timely participation.

Lemma 4. *Protocol* $\mathtt{BC}_{\mathtt{as}}^{\mathtt{aIA,rC}}(2)$ *has timely participation.*

Proof. Each party forwards the first $n-t$ $\mathtt{vote}$ messages that it sees. Since this is the event that triggers each party to start $\mathtt{BC}_{\mathtt{as}}^{\mathtt{aIA,rC}}(2)$, timely participation holds.

Theorem 7. *If protocol* $\mathtt{VSS}_{\mathtt{as}}^{\mathtt{aIA,rC}}$ *has timely participation, then* $\mathtt{VSS}_{\mathtt{as}}^{\mathtt{aIA,rC}}$ *(Figure 5) is an asynchronous VSS protocol with aIA.*

Proof. We show that $\mathtt{VSS}_{\mathtt{as}}^{\mathtt{aIA,rC}}$ has the following properties.

Protocol $\mathtt{VSS}_{\mathtt{as}}^{\mathtt{aIA,rC}}$ (Sharing Phase)

- D samples a random (t,t)-degree bivariate polynomial $F(x,y)$ such that $F(0,0)=s$. D uses $\mathtt{BC}_{\mathtt{as}}^{\mathtt{aIA,rC}}(1)$ to broadcast the following.
  - $(\mathtt{Com}, \mathtt{wit})$, where $\mathtt{PC.Commit}(F(x,y), n) \to (\mathtt{Com}, \mathtt{wit})$
  - $\mathtt{Enc}_{pk_i}\{(f_{ij}, \pi_{ij})\}_{j\in[n]}$, where $\mathtt{PC.Eval}(\mathtt{wit}, F(x,y), i, j) \to (f_{ij}, \pi_{ij})$.
- Each party P_i starts participating in $\mathtt{BC}_{\mathtt{as}}^{\mathtt{aIA,rC}}(1)$.
- If $\mathtt{BC}_{\mathtt{as}}^{\mathtt{aIA,rC}}(1)$ outputs $\perp$, each party outputs $\perp$ and identifies D to be either slow or corrupt. Otherwise, each P_i does the following.
  - If $\mathtt{PC.Verify}(\mathtt{Com}, i, j, f_{ij}, \pi_{ij}) \to 1$ for all $j \in [n]$, send $\langle \mathtt{vote}, \mathtt{OK}\rangle_i$ and (π_{ij}, f_{ij}) to each P_j.
  - Otherwise, send $\langle \mathtt{vote}, \mathtt{NOK}\rangle_i$ to each P_j.
- D, on receiving $\mathtt{vote}$ messages from $n-t$ parties, denotes this set to be V. D uses $\mathtt{BC}_{\mathtt{as}}^{\mathtt{aIA,rC}}(2)$ to broadcast $(V, \{f_i(x)\}_{\langle \mathtt{vote,NOK}\rangle_i \in V})$.
- Each P_i, on receiving $\mathtt{vote}$ messages from $n-t$ parties, forwards this to all parties, and starts participating in protocol $\mathtt{BC}_{\mathtt{as}}^{\mathtt{aIA,rC}}(2)$.
- If $\mathtt{BC}_{\mathtt{as}}^{\mathtt{aIA,rC}}(2)$ outputs $\perp$, each party outputs $\perp$ and identifies D to be either slow or corrupt. Otherwise, each P_i computes its share as follows.
  - If the output of $\mathtt{BC}_{\mathtt{as}}^{\mathtt{aIA,rC}}(2)$ is not of the form $(V, \{f_i(x)\}_{\langle \mathtt{vote,NOK}\rangle_i \in V})$ with V consisting of $n-t$ distinct $\langle \mathtt{vote}, \mathtt{OK}\rangle_i$ messages, then output $\perp$.
  - Else, compute s_i as follows.
    - If P_i sent $\langle \mathtt{vote}, \mathtt{OK}\rangle_i$, P_i outputs $s_i = f_i(0)$.
    - Otherwise, on receiving $t+1$ values (π_{ji}, f_{ji}) such that $\mathtt{PC.Verify}(\mathtt{Com}, j, i, f_{ji}, \pi_{ji}) \to 1$ (either directly from P_j or from D's broadcast), P_i interpolates these values to obtain $f_i'(x)$ and outputs $f_i'(0)$.

Fig. 5: Sharing Phase of a VSS protocol with aIA+rC that allows a dealer D to Shamir secret share a secret s on a t-degree polynomial with relaxed identifiable abort.

Relaxed Correctness. When D is honest, each P_i that outputs a share verifies the share against the committed polynomial. By the correctness of the polynomial commitment scheme, this share is s_i.

Next, consider that D has synchronous links with at least $t+1$ honest parties. Due to the timely participation of $\mathtt{VSS}_{\mathtt{as}}^{\mathtt{aIA,rC}}$ and rC of $\mathtt{BC}_{\mathtt{as}}^{\mathtt{aIA,rC}}(1)$, each party obtains an output from $\mathtt{BC}_{\mathtt{as}}^{\mathtt{aIA,rC}}(1)$ and does not abort. Each honest party P_i checks that each f_{ij} lies on the broadcasted polynomial commitment $\mathtt{Com}$ [6] and sends $\langle \mathtt{vote}, \mathtt{OK}, f_{ij}\rangle_i$ to each P_j. Each party, including D, eventually receives $n-t$ $\mathtt{vote}$ messages and starts protocol $\mathtt{BC}_{\mathtt{as}}^{\mathtt{aIA,rC}}(2)$. By the relaxed correctness of $\mathtt{BC}_{\mathtt{as}}^{\mathtt{aIA,rC}}(2)$, each party obtains an output from $\mathtt{BC}_{\mathtt{as}}^{\mathtt{aIA,rC}}(1)$ and does not abort. Each honest party checks that the output is correct, and outputs $s_i = f_i(0)$.

Privacy. When D is honest, all honest parties verify that their received points lie on $\mathtt{Com}$, and no honest party votes $\mathtt{NOK}$. D *does not* broadcast the points of any honest party during $\mathtt{BC}_{\mathtt{as}}^{\mathtt{aIA,rC}}(2)$. By the *hiding* property of $\mathtt{PC}$, $\mathcal{A}$ learns at most t^2 evaluations of F sent to the corrupt parties. Since $\mathcal{A}$'s view remains independent of s and since F is chosen uniformly at random.

Asynchronous Identifiable Abort. We first prove that unanimous abort holds. Parties agree on whether to abort due to the consistency of $\mathtt{BC}_{\mathtt{as}}$. We next prove that all parties terminate. All parties obtain an output from $\mathtt{BC}_{\mathtt{as}}^{\mathtt{aIA,rC}}(1)$ due to its termination property. If $\mathtt{BC}_{\mathtt{as}}^{\mathtt{aIA,rC}}(1)$ aborts, then all parties abort and terminate. If $\mathtt{BC}_{\mathtt{as}}^{\mathtt{aIA,rC}}(1)$ does not abort, then each honest party checks if each point it receives lies on $\mathtt{Com}$ or not, and accordingly sends a $\mathtt{vote}$ to all other parties.

Thus, each party eventually receives $n-t$ $\mathtt{vote}$ messages and starts executing $\mathtt{BC}_{\mathtt{as}}^{\mathtt{aIA,rC}}(2)$. By the termination property of $\mathtt{BC}_{\mathtt{as}}^{\mathtt{aIA,rC}}(2)$, all parties either abort, or receive an output from $\mathtt{BC}_{\mathtt{as}}^{\mathtt{aIA,rC}}(2)$. In the latter case, parties abort if the output $\mathtt{BC}_{\mathtt{as}}^{\mathtt{aIA,rC}}(2)$ is not of the correct form. Otherwise, since $|V| \geq n-t$, V contains the $\mathtt{vote}$ messages of at least $n-2t > t$ honest parties. Each honest P_i receives f_{ji} from each honest $P_j \in V$ either directly from P_j, or as part of D's broadcast. Each P_i can interpolate these points to obtain $f_i'(x)$ and output $f_i'(0)$.

[6] We use "v lies on $\mathtt{Com}$" to mean that P_i checks that $\mathtt{PC.Verify}(\mathtt{Com}, i, j, v, \pi) \to 1$ holds.

We next prove that when parties abort, D is seemingly corrupt. When either instance of $\mathtt{BC}_{\mathtt{as}}^{\mathtt{aIA,rC}}$ aborts, by the aIA of $\mathtt{BC}_{\mathtt{as}}^{\mathtt{aIA,rC}}$, D is seemingly corrupt. Parties also abort when the output of $\mathtt{BC}_{\mathtt{as}}^{\mathtt{aIA,rC}}(2)$ is not of the correct form. In this case, D is corrupt.

Consistency. Due to the binding property of $\mathtt{Com}$, each P_i outputs $f_i(x)$ by interpolating points that lie on a commitment $\mathtt{Com}$ of some bivariate polynomial $F^\star(x,y)$, that is, $F^\star(x,i) = f_i(x)$. Hence, each $f_i(0)$ lies on the univariate polynomial $F^\star(0,y)$.

Reconstruction. Follows from the reconstruction property of $\mathtt{Rec}_{\mathtt{as}}^{\mathtt{GOD,C}}$.

Communication Cost. The communication cost is dominated by the cost of broadcasting $\{f_{ij}\}_{j\in[n]}$ to each party P_i. This incurs a cost of $O(n^4\lambda)$ bits, where λ is the size of a commitment / ciphertext / signature.

MPC with Asynchronous Identifiable Abort. We construct $\mathtt{MPC}_{\mathtt{as}}^{\mathtt{aIA,rC}}$ (Figure 6) by modifying the input phase of $\mathtt{MPC}_{\mathtt{as}}^{\mathtt{GOD,C}}$ into one that allows parties to agree on a set of $n - \min(A,t)$ secret-shared inputs, or to identify a common seemingly corrupt party. In our input phase, parties use $\mathtt{VSS}_{\mathtt{as}}^{\mathtt{aIA,rC}}$ to secret-share their inputs, and wait until *all* instances of $\mathtt{VSS}_{\mathtt{as}}^{\mathtt{aIA,rC}}$ terminate. If at least $\min(A,t)$ instances do not abort, parties use the shares obtained in these instances to execute $\mathtt{CR}_{\mathtt{as}}^{\mathtt{GOD,C}}$. Otherwise, parties abort and identify the seemingly corrupt dealer of an instance of $\mathtt{VSS}_{\mathtt{as}}^{\mathtt{aIA,rC}}$ that aborts.

Theorem 8. *Protocol* $\mathtt{MPC}_{\mathtt{as}}^{\mathtt{aIA,rC}}$ *output by* $\mathtt{CR}_{\mathtt{as}}^{\mathtt{GOD,C}} \to \mathtt{MPC}_{\mathtt{as}}^{\mathtt{Fair+aIA,rC}}$ *is an asynchronous MPC protocol with relaxed correctness, fairness and aIA when* $t < n/3$.

Proof. By the aIA property, all instances of $\mathtt{VSS}_{\mathtt{as}}^{\mathtt{aIA,rC}}$ terminate. If at most $\min(t,A)$ instances of $\mathtt{VSS}$ abort, then by the correctness of $\mathtt{CR}_{\mathtt{as}}^{\mathtt{GOD,C}}$, all parties compute a consistent output y. Otherwise, by the aIA property of $\mathtt{VSS}_{\mathtt{as}}^{\mathtt{aIA,rC}}$, parties agree on the identity of a seemingly corrupt party. Fairness and privacy hold due to the privacy of $\mathtt{VSS}_{\mathtt{as}}^{\mathtt{aIA,rC}}$ and the privacy of the computation phase.

Communication Overhead. We invoke n instances of $\mathtt{VSS}_{\mathtt{as}}^{\mathtt{aIA,rC}}$. This results in a communication overhead of $O(n^5\lambda)$ bits.

Protocol $\mathtt{CR}_{\mathtt{as}}^{\mathtt{GOD,C}} \to \mathtt{MPC}_{\mathtt{as}}^{\mathtt{Fair+aIA,rC}}$

- Run an instance of $\mathtt{VSS}_{\mathtt{as}}^{\mathtt{aIA,rC}}(i)$ to share x_i. $\forall P_j \in \mathcal{P}$, participate in $\mathtt{VSS}_{\mathtt{as}}^{\mathtt{aIA,rC}}(j)$.
- Wait for $\mathtt{VSS}_{\mathtt{as}}^{\mathtt{aIA,rC}}(j)$ to terminate for all $P_j \in \mathcal{P}$. Compute the output as follows.
  - Let $\mathcal{P}'$ be the set of parties P_j whose $\mathtt{VSS}_{\mathtt{as}}^{\mathtt{aIA,rC}}(j)$ instances do not output $\perp$. If $|\mathcal{P}'| \geq n - \min(A,t)$, participate in $\mathtt{CR}_{\mathtt{as}}^{\mathtt{GOD,C}}$ with input $\{x_{ij}\}_{P_j\in\mathcal{P}'}$ to obtain output y. Output y.
  - Else, identify the smallest (ID) party in $\mathcal{P} \setminus \mathcal{P}'$ to be seemingly corrupt.

Fig. 6: Code for party P_i to run a fair asynchronous MPC protocol with aIA+rC

5.3 IA+rC does not help reduce input loss

It is impossible to achieve MPC with $< t$ input loss with the standard notion of IA (parties identify a common corrupt party on abort, even when the network is asynchronous) when $t < n/3$, even when we relax correctness and assume $A = 0$. By the lower bound in §4.3, this is the best corruption threshold we can achieve. Our result in fact holds for a *weaker* notion of IA, which we call weak IA (wIA), that requires a corrupt party to be identified only when the network behaves synchronously.

Theorem 9. *When* $t < n/3$*, there exists no asynchronous MPC protocol that achieves the following properties (wIA) while guaranteeing* $< t$ *input loss.*

- *it achieves aIA (Definition 7).*

Protocol $\mathtt{MPC}_{\mathsf{as}}^{\mathsf{Fair+wIA,rC}} \to \mathtt{MPC}_{\mathsf{ps}}^{\mathsf{GOD,C}}$

Initialize $\mathtt{Out}$ to $\perp$. Perform the following until $\mathtt{Out} \neq \perp$.

- Participate in $\mathtt{MPC}_{\mathsf{as}}^{\mathsf{Fair+wIA,rC}}$ to obtain output (y, b_i). Set $\mathtt{Out} = y$.
- If $\mathtt{Out} = \perp$, then if some party P_j is identified and $b_i = 1$, set $\mathcal{P} = \mathcal{P} \setminus \{P_j\}$.

Fig. 7: Code for party P_i that uses a fair MPC protocol with wIA and L input loss to run a partially-synchronous MPC protocol with GOD with the same input loss.

Protocol $\mathtt{MPC}_{\mathsf{as}}^{\mathsf{wIA,rC}} \to \mathtt{MPC}_{\mathsf{as}}^{\mathsf{Fair+wIA,rC}}$

- Run $\mathtt{RandShare}_{\mathsf{as}}^{\mathsf{GOD,C}}$ to obtain share r_i.
- Participate in $\mathtt{MPC}_{\mathsf{as}}^{\mathsf{wIA,rC}}$ to compute the function $F(x_1, \ldots, x_n) + \sum_{P_j \in \mathcal{P}} r_j$.
- If $\mathtt{MPC}_{\mathsf{as}}^{\mathsf{wIA,rC}}$ aborts, identify the party identified during $\mathtt{MPC}_{\mathsf{as}}^{\mathsf{wIA,rC}}$. Otherwise, participate in $\mathtt{Rec}_{\mathsf{as}}^{\mathsf{GOD,C}}(r_j)$ for each $P_j \in \mathcal{P}$.
- Let y be the output of $\mathtt{MPC}_{\mathsf{as}}^{\mathsf{wIA,rC}}$. Output $y - \sum_{P_j \in \mathcal{P}} r_j$.

Fig. 8: Code for party P_i for a protocol that uses any protocol with wIA ($\mathtt{MPC}_{\mathsf{as}}^{\mathsf{wIA,rC}}$) to additionally achieve fairness.

- *Each honest P_i outputs a bit b_i that is 1 if the identified party is corrupt.*
- *For all honest P_i and P_j, $b_i = b_j$.*
- *In a synchronous network, when parties abort, each honest P_i outputs $b_i = 1$.*

Proof. We prove that if there exists a protocol with wIA and $L < t$ input loss, then there also exists a protocol with L input loss in a *partially synchronous* network: a network that is asynchronous until an unknown global stabilization time, after which it is synchronous. This is a contradiction since the impossibility proof of achieving MPC with GOD and $< t$ input loss in asynchrony [BLZL20] also extends to a partially synchronous network.

Our proof is inspired by a compiler from prior work [CL17], designed for a synchronous network, that compiles a *fair* protocol with IA (Π) to a protocol with GOD. Their compiler makes sequential invocations to Π up to $t+1$ times. In each invocation of Π, parties either compute the function output or abort while eliminating a corrupt party. After t invocations, only honest parties remain, and the correctness of Π ensures that all parties compute the function output. It is crucial that Π is fair, as otherwise, $\mathcal{A}$ may learn *multiple* outputs of Π by changing the inputs of corrupt parties during each invocation.

Achieving partially synchronous GOD using fair MPC with aIA. We present a similar compiler that works in a *partially synchronous* network in Figure 7. Such a compiler uses a *fair* asynchronous protocol Π with wIA. If the identified party is seemingly corrupt, the parties simply re-run Π. If the identified party is *corrupt*, then this party is *removed* from $\mathcal{P}$. Since a partially synchronous network eventually becomes synchronous, parties eventually start removing corrupt parties, and complete the protocol. Each invocation of Π incurs L input loss, and so, the compiled protocol with GOD also incurs L input loss.

Upgrading any protocol with wIA to achieve fairness. This rules out the possibility of any *fair* protocol with wIA and $< t$ input loss. To rule out the possibility of *any* protocol with wIA and $< t$ input loss, we show (in Figure 8, with proof in Appendix B) that when $t < n/3$, any such protocol Π can be modified into a protocol that is also fair. To securely compute a function F, parties use Π to instead securely compute $F(x_1, \ldots, x_n) + r$, where r is a secret mask that is unknown to all parties.

To obtain r, we make parties run a protocol $\mathtt{RandShare}$ after which each party outputs (with GOD) a share r_i of a random unknown value r. Any asynchronous MPC protocol with GOD can be used to implement $\mathtt{RandShare}$. On successfully completing Π, parties reconstruct r to learn the final output.

References

[AJM+23] Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, and Gilad Stern. Bingo: Adaptivity and asynchrony in verifiable secret sharing and distributed key generation. In *Annual International Cryptology Conference*, pages 39–70. Springer, 2023.

[AL17] Gilad Asharov and Yehuda Lindell. A full proof of the bgw protocol for perfectly secure multiparty computation. *Journal of Cryptology*, 30(1):58–151, 2017.

[BCV24] Nidhish Bhimrajka, Ashish Choudhury, and Supreeth Varadarajan. Network-agnostic multi-party computation revisited. In *IACR International Conference on Public-Key Cryptography*, pages 171–204. Springer, 2024.

[BLZL20] Erica Blum, Chen-Da Liu-Zhang, and Julian Loss. Always have a backup plan: fully secure synchronous mpc with asynchronous fallback. In *Annual International Cryptology Conference*, pages 707–731. Springer, 2020.

[BOKR94] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous secure computations with optimal resilience. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, pages 183–192, 1994.

[Bra84] Gabriel Bracha. An asynchronous [(n-1)/3]-resilient consensus protocol. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 154–162, 1984.

[CGG+23] Pierre Civit, Seth Gilbert, Rachid Guerraoui, Jovan Komatovic, and Manuel Vidigueira. On the validity of consensus. In *Proceedings of the 2023 ACM symposium on principles of distributed computing*, pages 332–343, 2023.

[CGMA85] Benny Choc, Shafi Goldwasser, Silvio Micali, and Baruch Awerbuch. Verifiable secret sharing and achieving simultaneity in the presence of faults. In *Annual Symposium on Foundations of Computer Science (Proceedings)*, pages 383–395, 1985.

[CHOR18] Ran Cohen, Iftach Haitner, Eran Omri, and Lior Rotem. Characterization of secure multiparty computation without broadcast. *Journal of Cryptology*, 31:587–609, 2018.

[CKS00] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantipole: practical asynchronous byzantine agreement using cryptography. In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 123–132, 2000.

[CL17] Ran Cohen and Yehuda Lindell. Fairness versus guaranteed output delivery in secure multiparty computation. *Journal of Cryptology*, 30(4):1157–1186, 2017.

[DLS88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[DPPR25] Ivan Damgård, Shravani Patil, Arpita Patra, and Lawrence Roy. New upper and lower bounds for perfectly secure mpc. *Cryptology ePrint Archive*, 2025.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Annual cryptology conference*, pages 643–662. Springer, 2012.

[DS83] Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

[Fel87] Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, pages 427–438. IEEE, 1987.

[GL05] Shafi Goldwasser and Yehuda Lindell. Secure multi-party computation without agreement. *Journal of Cryptology*, 18:247–287, 2005.

[GMW87] 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.

[IOZ14] Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. Secure multi-party computation with identifiable abort. In *Annual Cryptology Conference*, pages 369–386. Springer, 2014.

[KZG10] Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *Advances in Cryptology-ASIACRYPT 2010: 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings 16*, pages 177–194. Springer, 2010.

[LS25] Junru Li and Yifan Song. Constant-round asynchronous mpc with optimal resilience and linear communication. In *Annual International Cryptology Conference*, pages 36–69. Springer, 2025.

[Sha79] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th annual symposium on foundations of computer science (Sfcs 1986)*, pages 162–167. IEEE, 1986.

A The Dolev Strong Protocol

In this section, we prove that the Dolev Strong (DS) protocol provides the properties required by $\mathtt{BC}_{\mathsf{S}}^{\mathtt{GOD,C}}$ (see Broadcast with SA+rC, §4.2). During the DS protocol (presented in Figure 9), S signs and sends its message all parties during an initial round. This is followed by t rounds during which parties forward the message sent by S, along with a certificate of valid signatures. A party forwards the message (with a certificate) in round r if it receives a certificate that contains r distinct signatures (including the sender's) during round $r-1$. The DS protocol ensures consistency since it can be shown that each message that is received by a honest party P_i will also be received by all honest parties P_j. If P_i receives this during a round $r < t$, then P_i will forward the message to P_j. Otherwise, if P_i receives $C_{t+1}(m)$ this at the end of round t, then some honest party has seen $C_t(m)$ during round t, and will send $C_{t+1}(m)$ to P_j.

Protocol DS

- Initialize $\mathtt{cnt}_i$ to 0 and V_i to { }.
- **Round** 0**:** S sends $\langle m \rangle_{\mathsf{S}}$ to all parties.
- **For rounds** $r = 1, \ldots, t$**:** If $C_r(m)$ is received during round $r-1$ that includes S's signature, then add m to V and increment $\mathtt{cnt}_i$. Add $\langle m \rangle_i$ to $C_r(m)$ and send the resulting certificate $C_{r+1}(m)$ to all parties if the following hold:
  - m is seen for the first time.
  - $\mathtt{cnt}_i \leq 2$.
- If a $C_{t+1}(m)$ is received at the end of round t that includes S's signature, add m to V_i.
- Compute the output as follows.
  - If $|V_i| = 0$, output ϕ.
  - If $|V_i| = 1$, output the message m contained in V_i.
  - Else, output a default message.

Fig. 9: Code for party P_i to run the Dolev-Strong synchronous broadcast protocol. $C_n(m)$ a message m signed using n distinct signatures.

Lemma 5. *During a run of the Dolev-Strong protocol [DS83], let $\mathcal{H}$ be the set of honest parties that communicate synchronously with each other. Each party in $\mathcal{H}$ outputs either a message or ϕ, such that:*

- *Consistency: All parties in $\mathcal{H}$ output the same value.*
- *Correctness: If* S *is honest, then*
  - *each party in $\mathcal{H}$ outputs ϕ or m.*
  - *if* $\mathsf{S} \in \mathcal{H}$*, then all parties in $\mathcal{H}$ output m.*

Proof. Consistency holds since DS is designed to work in a synchronous network for any number of corruptions (i.e $t < n$). We next show that correctness is guaranteed. When S is honest, since digital signatures are unforgeable, V_i computed by each honest P_i either contains m, or is empty. Hence, honest P_i either outputs m or ϕ. If S is synchronous, then each $P_i \in \mathcal{H}$ receives m from S during round 0, and adds m to V_i. In this case, each $P_i \in \mathcal{H}$ outputs m.

B Upgrading wIA to Fairness

In Figure 8 of §5.3, we presented a protocol that upgrades any protocol $\mathtt{MPC}_{\mathtt{as}}^{\mathtt{wIA,rC}}$ with wIA into one that is additionally fair. Here, we prove that this construction works.

Lemma 6. *If $\mathtt{MPC}_{\mathtt{as}}^{\mathtt{wIA,rC}}$ is an MPC protocol with wIA, then $\mathtt{MPC}_{\mathtt{as}}^{\mathtt{wIA,rC}} \rightarrow \mathtt{MPC}_{\mathtt{as}}^{\mathtt{Fair+wIA,rC}}$ (Figure 8) outputs a protocol that also allows for this, while additionally being fair.*

Proof. The protocol $\mathtt{MPC}_{\mathtt{as}}^{\mathtt{wIA,rC}} \rightarrow \mathtt{MPC}_{\mathtt{as}}^{\mathtt{Fair+wIA,rC}}$ satisfies the following.

Relaxed Correctness. $\texttt{RandShare}_{\texttt{as}}^{\texttt{GOD},\texttt{C}}$ terminates by its GOD property. All parties participate in $\texttt{MPC}_{\texttt{as}}^{\texttt{wIA},\texttt{rC}}$ and, by relaxed correctness, all parties complete the protocol to output a value.

Weak Identifiable Abort. Assume that the network is synchronous. By GOD, each party P_i outputs r_i after running $\texttt{RandShare}_{\texttt{as}}^{\texttt{GOD},\texttt{C}}$, and participates in $\texttt{MPC}_{\texttt{as}}^{\texttt{wIA},\texttt{rC}}$. All parties run $\texttt{MPC}_{\texttt{as}}^{\texttt{wIA},\texttt{rC}}$, which either aborts or completes due to its wIA property. If it aborts, each P_i outputs $b_i = 1$. Otherwise, by the GOD of $\texttt{Rec}_{\texttt{as}}^{\texttt{GOD},\texttt{C}}(r_j)$, all parties reconstruct r and learn the output.

Input Loss. When parties compute a function output, the input loss incurred is the same input loss incurred by $\texttt{MPC}_{\texttt{as}}^{\texttt{wIA},\texttt{rC}}$.

Privacy and Fairness. When $\texttt{MPC}_{\texttt{as}}^{\texttt{wIA},\texttt{rC}}$ aborts, the adversary learns the function output masked by a random value r. Since honest parties do not start reconstructing this mask, the output remains uniformly random to the adversary, and its view can be simulated.

Consistency. Follows from the consistency of $\texttt{MPC}_{\texttt{as}}^{\texttt{wIA},\texttt{rC}}$.