Is It Even Possible?
On the Parallel Composition of Asynchronous MPC Protocols*

Ran Cohen† Pouyan Forghani‡ Juan Garay§ Rutvik Patel¶ Vassilis Zikas‖

November 18, 2026

Abstract

Despite several known idiosyncrasies separating the synchronous and the asynchronous models, asynchronous secure multi-party computation (MPC) protocols demonstrate high-level similarities to synchronous MPC, both in design philosophy and abstract structure. As such, a coveted, albeit elusive, *desideratum* is to devise automatic translators (e.g., protocol compilers) of feasibility and efficiency results from one model to the other.

In this work, we demonstrate new challenges associated with this goal. Specifically, we study the case of *parallel composition* in the asynchronous setting. We provide formal definitions of this composition operation in the UC framework, which, somewhat surprisingly, have been missing from the literature. Using these definitions, we then turn to charting the feasibility landscape of asynchronous parallel composition.

We first prove strong impossibility results for composition operators that do not assume knowledge of the functions and/or the protocols that are being composed. These results draw a grim feasibility picture, which is in sharp contrast with the synchronous model, and highlight the question:

Is asynchronous parallel composition even a realistic goal?

To answer the above (in the affirmative), we provide conditions on the composed protocols that enable a useful form of asynchronous parallel composition, as it turns out to be common in existing constructions.

*A preliminary version of this work appears in *Proc. TCC'25* [CFG+25].

†Efi Arazi School of Computer Science, Reichman University. E-mail: cohenran@runi.ac.il. Research supported in part by NSF grant no. 2055568 and by ISF grant no. 1834/23.

‡School of Cybersecurity and Privacy, Georgia Tech. E-mail: pforghani3@gatech.edu. Work partly done while the author was at Texas A&M University.

§Department of Computer Science and Engineering, Texas A&M University. E-mail: garay@tamu.edu. Research supported in part by NSF grants no. 2001082 and 2055694.

¶rpatel4554@gmail.com. Work partly done while the author was at Texas A&M University.

‖School of Cybersecurity and Privacy, Georgia Tech. E-mail: vzikas@gatech.edu. Research supported in part by NSF grant no. 2448339 and by Sunday Group. Inc.

Contents

1 Introduction **1**
- 1.1 Background on Synchronous Parallel Composition . . . 2
- 1.2 Our Results . . . 3
  - 1.2.1 Modeling Asynchronous Parallel Composition . . . 4
  - 1.2.2 Impossibility Results for Asynchronous Parallel Composition . . . 5
  - 1.2.3 Feasibility Results for Asynchronous Parallel Composition . . . 7

2 Preliminaries and Model **8**
- 2.1 The UC Framework . . . 9
- 2.2 Modeling Eventual Message Delivery in UC . . . 10
- 2.3 Asynchronous Secure Function Evaluation (A-SFE) . . . 11
- 2.4 Ideal Functionalities for Standard Asynchronous Primitives . . . 13

3 Parallel Composition of A-SFE Functionalities **15**
- 3.1 Strong and Weak Asynchronous Parallel Composition . . . 15
- 3.2 Functionally Black-Box (FBB) and Protocol Black-Box (PBB) Composition . . . 17

4 Impossibility of Black-Box Asynchronous Parallel Composition **19**
- 4.1 Impossibility of FBB Weak Parallel Composition . . . 19
- 4.2 Impossibility of Generic PBB Weak Parallel Composition . . . 23
  - 4.2.1 A Contrived A-MPC protocol . . . 23
  - 4.2.2 The Impossibility Result . . . 30

5 Feasibility of *Conditional* PBB Asynchronous Parallel Composition **35**
- 5.1 Feasibility of Conditional PBB Strong Parallel Composition . . . 35
- 5.2 Feasibility of Conditional PBB Weak Parallel Composition . . . 42

Bibliography **48**

1 Introduction

The literature on fault-tolerant distributed computing and secure multi-party computation (MPC) typically focuses on two models of execution with respect to synchrony: *synchronous* and *asynchronous* (with eventual message delivery and guaranteed termination). In the synchronous model, the protocol execution advances in synchronized rounds where parties have a common view of what the current round is, and messages sent in one round are delivered by the beginning of the next one. In the asynchronous model, on the other hand, there is no guarantee of when a message sent from a party will be delivered to its intended recipient; in fact, to obtain worst-case security guarantees, one typically assumes the adversary to effect the worst-case scheduling on message delivery, with the only restriction being that any message sent by an (honest) party has to be *eventually* delivered.

It has long been known that protocols in the two models have several differences in terms of their structure, security and efficiency guarantees, and what types of computation they support. Synchronous protocols tend to be more structured and can be cleanly specified by describing what parties do in each round. Furthermore, tight feasibility results are known, showing general-purpose MPC protocols for any n-ary function: with *perfect* security (assuming only secure communication channels) if and only if at most $t < n/3$ of the parties are corrupted [BGW88, CCD88]; with *statistical* security (assuming also a broadcast channel) if and only if $t < n/2$ [RB89]; and with *computational* security (under standard cryptographic assumptions, and where the adversary may abort the computation) for any $t < n$ [GMW87]. Given this clean structure and strong feasibility results, the synchronous model has attracted most of the attention of the cryptographic community, despite the fact that in reality, guaranteeing the above round structure (which is needed to ensure security) requires pessimistic lower bounds on the network latency, which might incur unnecessary slowdowns.

On the other hand, asynchronous protocols leverage network delays in an optimal manner (and are therefore considered "faster") since they are typically specified in an event-based fashion, which is triggered on message reception. This, however, tends to result in harder-to-understand protocol descriptions, since different parties might receive very different sets of messages and therefore be in far-away parts of the protocol execution. Perhaps more importantly, the asynchronous model is known to severely limit the nature of the computations that can be performed. The underlying reason is that a party P_i that has not received a message it expects from P_j, cannot tell whether the message is simply delayed (e.g., because it is in transit over the asynchronous network or because P_j is still behind) or if P_j is actively cheating (and never sent this message). As such, if P_i waits to receive messages from all parties, the adversary can trivially prevent it from terminating.

This inherent barrier leads to several fundamental differences between the models. For example, as opposed to the synchronous model, there cannot be a deterministic protocol for *asynchronous Byzantine agreement* (A-BA) that is secure against even a single fail-stop corruption [FLP85], and A-BA (and therefore the notion of *asynchronous MPC* (A-MPC) discussed below) cannot tolerate $t \geq n/3$ malicious corruptions (even under computational assumptions) [DLS88, BT85].

The way out of this conundrum, proposed both in the fault-tolerant distributed computing and MPC literature (specifically, by Ben-Or, Canetti, and Goldreich [BCG93], who formally defined the goals of A-MPC), is to allow parties to proceed as soon as they have received messages from $n - t$ parties. The result of this, however, is that the adversary has an inherent capability of forcing the computation to ignore inputs from t of the honest parties (as t corrupted parties can stay silent), and so, the best one can hope for is to compute the target n-ary function only on a "core set" of $n - t$ inputs (where t of them might belong to corrupted parties). An additional side effect is an additional t term penalty in resiliency—for example, *perfect* A-MPC exists if and only if $t < n/4$ [BCG93] and *statistical* (as well as *computational*) A-MPC exists if and only if $t < n/3$ [BKR94].

Notwithstanding, despite the fundamental differences between the two models, there has been a belief in the cryptographic community that there should be a way to build generic bridges to translate synchronous MPC protocols to their asynchronous counterparts—assuming, of course, one is willing to accept the above lack of "input completeness" in the translation. This belief is reinforced by the fact that many of the asynchronous protocols share structural similarities with their synchronous counterpart. For example, the perfect A-MPC protocol of Ben-Or, Canetti and Goldreich [BCG93] can be seen as the asynchronous version of the Ben-Or, Goldwasser and Wigderson protocol [BGW88]; the statistical A-MPC protocol of Ben-Or, Kelmer and Rabin [BKR94] can be seen as the asynchronous version of Rabin and Ben-Or's [RB89]; the computational protocols of Hirt, Nielsen, and Przydatek [HNP05, HNP08] as the analogue of Cramer, Damgård, and Nielsen's [CDN01]; the FHE-based protocol of Cohen [Coh16] as the analogue of Asharov *et al.* [AJL+12]; and the garbled-circuit-based protocol of Coretti *et al.* [CGHZ16] as the asynchronous version of Beaver, Micali and Rogaway's [BMR90]. Indeed, by isolating a procedure called *Agreement on a Core Set* (ACS) for agreeing on $n-t$ of the inputs, and "staring" at these protocols long enough, one can draw parallels in the logic of the above asynchronous protocols and that of their synchronous counterparts.

Thus, the above state of affairs seems to point us in the direction of devising a compilation framework (in the spirit of Ishai *et al.* [IKP+16]) that would abstract the above parallels and allow translating synchronous MPC protocols to the asynchronous realm by systematically addressing the inherent barriers discussed above using well-established techniques, and that should be it, right?

Wrong! As our work indicates, achieving this cornerstone has deeper obstacles than previously believed. In particular, we focus on the *parallel composition* of asynchronous protocols, where each party holds inputs for multiple computations that must be carried out in an independent manner (meaning that corrupted parties cannot choose their inputs to some computation based on the output of another computation) and demonstrate a spectrum of results separating such composition from its synchronous counterpart.

As parallel composition is mostly understood in the synchronous model,[1] assuming an honest majority (as discussed in Section 1.1), such separations hinder the translation of basic synchronous protocols to the asynchronous model and demonstrate that the above cornerstone goal is far more elusive than previously thought. In particular, even if one were able to devise a compiler from synchronous to asynchronous protocols, and an associated translation of the security definition, our result would still imply that this translation does not hold under natural parallel composition operators. On the positive side, we also provide sufficient conditions (on the protocols) and restricted types of parallel composition operations that enable composition, thereby partially restoring the ability to translate the synchronous intuition under such assumptions.

1.1 Background on Synchronous Parallel Composition

Parallel composition has been intensively studied in the synchronous model, assuming an honest majority. Indeed, several tasks are implemented by the parallel composition of a single instance, with the simplest one being the realization of *interactive consistency* from *Byzantine broadcast* [PSL80, LSP82]. Further, many synchronous protocols satisfy basic requirements that enable

[1] We note that in the synchronous setting, parallel composition typically considers protocols that begin at the same round and proceed at the same pace—for example, in the settings of zero-knowledge proofs [GK90, FS90] and Byzantine agreement [BE03, LLR06]. However, when composing in parallel MPC protocols for secure function evaluation (SFE), the resulting protocol is required to realize the "parallel composition" of the functions, ensuring that the inputs to one computation cannot be dependent on the output of another computation; see e.g., [DM00, CCGZ21, CHOR22]. While the former notion has no meaning in the asynchronous setting, in this work we show that the latter does.

executing them together round by round while ensuring input independence across the computations. We begin by presenting some background on parallel composition of synchronous MPC protocols that will provide motivation for the more involved asynchronous case.

For the sake of this discussion, we focus on composing two protocols. Let f_1 and f_2 be two n-ary functions and let π_1 and π_2 be protocols that securely realize them, respectively. Our goal is to devise a protocol for computing the parallel composition of f_1 and f_2, denoted $[f_1 \parallel f_2]$; namely, the functionality which on input a pair of vectors $(\boldsymbol{x}_1, \boldsymbol{x}_2) = ((x_{1,1}, \ldots, x_{1,n}), (x_{2,1}, \ldots, x_{2,n}))$ to the computation (where party P_i inputs $(x_{1,i}, x_{2,i})$), outputs the vector $(f_1(\boldsymbol{x}_1), f_2(\boldsymbol{x}_2))$. Note that this is a strong type of parallel composition with respect to input independence, as, in particular, the adversarial inputs to the different functions do not depend on the honest parties' inputs (to any of the functions).

We shall refer to the above form of parallel composition as *strong composition* and denote it $[f_1 \parallel f_2]_{\text{strong}}$. We remark that the above notion is feasible for synchronous protocols, as long as each protocol is secure under composition. Indeed, parallel composition is simple for protocols that admit a *committal round* (as defined by Micali and Rogaway [MR92]; see also [DM00, CDD+01, LR17, ACS22, ACCI25]): informally, before this round no information is leaked about the outputs, whereas after this round, the output is fixed. As shown by Dodis and Micali [DM00], one can run each protocol in parallel until its committal round (if some protocol reaches this round early, it waits for the other protocols to also reach it before advancing), and then continue with all protocols concurrently; as a result, since all inputs have been committed, the adversary can no longer affect the output distribution.[2]

As shown in [CCGZ21], in the synchronous case, there are also techniques for strong parallel composition *without* requiring a committal round, and by making *black-box* use of the protocols in the spirit of [IKP+16] (as explained in Section 1.2.1): Initially, each party commits to its input for all the functions being composed and proves knowledge of the committed inputs; next, the parties execute the protocols for the different functions in parallel, where in each step and each protocol, parties prove in zero-knowledge (using a protocol black-box zero-knowledge proof, *à la* [IKOS07]) that they are using the committed input for this function. Further, this form of composition can preserve the expected round complexity of the underlying protocol.

We emphasize that the notion of parallel composition of synchronous protocols is different than composable security guarantees that are provided by the UC framework. Indeed, given M protocols $\pi_1, \ldots, \pi_M$ such that each π_i UC realizes the ideal computation of a function f_i, UC security ensures that when running the protocols $\pi_1, \ldots, \pi_M$ in parallel, then each π_i will still realize the ideal computation of f_i. However, this does not guarantee cross-computation properties such as input independence across computations. We consider the notion of synchronous parallel composition that realizes the ideal computation of the "parallel composition" of the functions $f_1, \ldots, f_M$, and as such takes into account cross-computation properties in the ideal world as well. This is the notion used in [DM00, CCGZ21, CHOR22], and also in the literature on bounded concurrent composition [Lin03, PR03, Pas04].

1.2 Our Results

Next, we discuss our contributions and put them in perspective with respect to the cryptographic protocols literature. Our results are proven in Canetti's Universal Composability (UC) framework [Can20], and, although we provide for completeness an overview of the needed components from UC in Section 2.1, for the more technical aspects of the paper we assume the reader has some

[2]The above discussion assumes that the security of *each* of the protocols holds under parallel synchronous composition; it holds, for example, for protocols in the synchronous UC framework [KMTZ13].

basic familiarity with the framework. Nonetheless, in this section, we keep the discussion at a high level.

1.2.1 Modeling Asynchronous Parallel Composition

Our first contribution is to model various notions of asynchronous parallel composition for MPC. To our knowledge, our work is the first to put forth such formalizations, thus already providing an important contribution to the relevant literature.

Strong parallel composition. As mentioned above, translating the notion of synchronous parallel composition to the asynchronous setting is more subtle than one might think, due to the adversarial power of influencing the computations via the core set of input providers. Continuing the example from Section 1.1, recall that the input to (the functionalities corresponding to) the strong parallel composition is a vector $(\boldsymbol{x}_1, \boldsymbol{x}_2) = ((x_{1,1}, \ldots, x_{1,n}), (x_{2,1}, \ldots, x_{2,n}))$ of inputs, where the input of each P_i to the above parallel composition is the vector $(x_{1,i}, x_{2,i})$. Since the parallel composition of asynchronous protocols for f_1 and f_2 will itself be an asynchronous protocol, the same limitations on the inputs that apply to standard A-MPC will need to apply also to the parallel composition of A-MPC protocols: that is, the adversary would be able to choose a core set $\mathcal{CS}$ of $n - t$ parties (t of which may be malicious) whose inputs will be included in the computation of $[f_1 \parallel f_2]_{\text{strong}}$, while the inputs of the remaining t parties will be ignored. In Definition 3.1, we specify the resulting functionality which corresponds to the asynchronous version of strong parallel composition.

Looking ahead, it turns out that unless certain assumptions are made about the underlying protocols, such a notion is **impossible to achieve** in a black-box manner. Furthermore, an asynchronous version of the committal-round assumption discussed earlier appears no longer sufficient for strong parallel composition. In fact, to achieve parallel composition, we have to either further restrict the class of protocols being composed (beyond those that have an "asynchronous committal round"), or to weaken the guarantees provided by their parallel composition.

Weak parallel composition. This leads to the introduction (and formal definition) of *weak parallel composition* (Definition 3.2). Informally, weak parallel composition allows the adversary to choose, for each of the composed functions f_1 and f_2, a (possibly different) core set $\mathcal{CS}_1$ and $\mathcal{CS}_2$, respectively. These sets can be correlated with each other, but cannot depend on the honest parties' inputs. While this notion enables different core sets for different computations, it still ensures that the adversarial choice of the core set for each computation is *independent* of the output values of other computations.

Loose parallel composition. One can further relax the requirements to allow some core set to depend on the outcome of other computations; we call this *loose parallel composition* and note that it is trivially achieved by independently running each computation until completion.

The above situation hints at a complex landscape, making the following question inexorable:

> *What types of parallel composition can be achieved in the asynchronous model, and under what assumptions?*

To address the above question in a holistic manner, in this paper we study the achievability of strong, weak, and (for completeness) loose parallel composition with respect to three types of composition operations/compilers, which we formalize (for the asynchronous model) in the UC framework:

- **Functionally Black-Box (FBB):** This notion, proposed by Rosulek [Ros12] (see also Ishai *et al.* [IKP+16]) and later adapted to address the most general form of synchronous parallel composition by Cohen *et al.* [CCGZ21], mandates that the parallel composition operation/compiler does not have knowledge of the functions being evaluated, besides oracle access to them. Namely, each party has local oracle access to the functions being evaluated, and the parties may use ideal functionalities that, in turn, also have only oracle access to the functions (and in particular, circuit descriptions of the functions are not known); see Definition 3.4.
- **Generic Protocol Black-Box (Generic PBB):** This corresponds to the classical notion of (protocol) black-box constructions, introduced by Ishai *et al.* [IKOS07] and [IKP+16] (which was also tuned to the setting of synchronous parallel composition in [CCGZ21]), and requires that the composition operation/compiler has oracle access to the next-message functions of the composed protocols; see Definition 3.6.
- **Conditional Protocol Black-Box (Conditional PBB):** This notion is similar to generic PBB, but also allows the composition operation/compiler to have knowledge of certain important aspects of the composed protocols: for example, knowledge of points in the protocol execution corresponding to an "asynchronous committal round," or where certain hybrid functionalities are invoked; see the discussion following Definition 3.6.

We now turn to the technical overview of our results.

1.2.2 Impossibility Results for Asynchronous Parallel Composition

Impossibility of FBB Weak Parallel Composition. We begin by considering FBB parallel composition, where the composition operation has only black-box access to the functionalities being computed (and no access at all to protocols computing the functionalities). Our first result (Theorem 4.2) shows that, even very weak adversaries (namely, adversaries that passively corrupt and potentially crash some parties) cannot be tolerated in the asynchronous setting. This holds even under *weak* parallel composition (where each functionality may have a different core set) and even when the adversary corrupts just a single party and does not crash that party during its attack! In contrast, Cohen *et al.* [CCGZ21] showed that in the *synchronous* setting, if the adversary does not crash any party during the attack, then FBB parallel composition is achievable. The source of this impossibility is that in the asynchronous setting, the requirement of security against crashes enables this (de facto) semi-honest adversary to carefully impose delays that induce different core sets for the computations, in a way that yields a privacy violation.

Our impossibility result applies to the computational-security setting, which inherently also excludes information-theoretic security with efficient simulation, i.e., simulation that runs in time polynomial in the adversary's running time [CDD+01, ACS22]. Notably, our proofs remain valid even for inefficient simulators, thereby extending our impossibility to information-theoretic security, even with inefficient simulation. The result also naturally excludes the possibility of *strong* asynchronous parallel composition, since the strong notion implies the weak notion (Proposition 3.3).

The proof of our impossibility result relies on the class of n-ary Boolean functions (or predicates) with inputs of length linear in the security parameter. We prove that any FBB protocol for this class must use the provided ideal functionalities to compute the functions on "admissible" inputs (otherwise, a correctness error will appear), and the final output must be among the output values returned from the ideal functionalities (Lemma 4.3). This allows the adversary to strategically set delays, potentially learning the output for some functions before determining the set of input providers for others. With a carefully chosen function and inputs, we can construct an adversary that violates the security guarantees of the compiled protocol with only a single semi-honest

corruption (without any crashes).

Impossibility of Generic PBB Weak Parallel Composition. The above impossibility of FBB composition raises the question of whether we can circumvent it by assuming the composition operation/compiler has some knowledge about the protocol structure. The weakest type of such knowledge is to assume that the composition is *generic PBB*—recall that this means the compiler only has access to the composed protocols' next-message functions (and thus to the protocol's transcript). In the synchronous setting, generic PBB parallel composition can be achieved against any minority of malicious corruptions (even without assuming a committal round) [CCGZ21].

Our second result (Theorem 4.9) shows that in the asynchronous case, weak parallel composition remains impossible against only slightly stronger adversaries than in the FBB impossibility—specifically, *semi-malicious adversaries*[3] that arbitrarily control the randomness and internal choices of corrupted parties, but must produce messages consistent with the protocol's next-message function [AJL+12]. Although semi-malicious adversaries may crash parties, as before, in our attack no party ever crashes. Instead, the adversary corrupts one party and strategically selects its randomness, which is tolerable by each individual protocol, but leads to issues even under weak parallel composition. Examining our proof, one can see that the security violation against these adversaries emerges again as a side effect of requiring security against crashes.

The above is proven using a delicate argument. We begin by selecting the same class of n-ary predicates as in the FBB case. The first challenge in our impossibility argument is that (in contrast to the FBB setting) oracle access to the protocol's next-message function could enable the compiler to generate full transcripts for arbitrary inputs. This might reveal the function being computed, allowing the compiler to ignore the input (to the compiler) protocols and construct the parallel composition of the functions from scratch. Indeed, the transcript of most A-MPC protocols reveals the circuit used to compute the function. Nonetheless, protocols that hide the function within the transcript do exist, e.g., the protocol of [Coh16], which is based on threshold fully homomorphic encryption (TFHE).

Another challenge is that if the compiler can detect the *core-set selection process*, it could potentially coordinate it and enforce the same core set for all functions, thereby achieving strong parallel composition and avoid our attack (which is based on carefully selecting different core sets to different computations). Additionally, the compiler may not necessarily use the next-message function in the prescribed way; it could generate important messages on its own, re-use messages from one execution in another, or employ other unexpected strategies.

To address these challenges, we introduce a specific class of A-MPC protocols. Our starting point is a simplified version of the A-MPC protocol of [Coh16], secure against semi-malicious adversaries. Similarly to [Coh16], since the depth of the homomorphic evaluation is known ahead of time, it is sufficient to use threshold *leveled* homomorphic encryption (LHE). While threshold LHE alone suffices for the security of our A-MPC, we additionally require *strong homomorphism*, which ensures that ciphertexts produced by homomorphic evaluation are indistinguishable from freshly generated encryptions of the same value; in particular, this ensures *circuit privacy* and addresses our first challenge. We refer the reader to [Hal17] for a detailed definition of LHE with strong homomorphism.

[3] A *semi-malicious adversary* is an interactive Turing machine (ITM) equipped with a special witness tape. Whenever it sends a protocol message m on behalf of a corrupted party P_k, it must also record a witness pair (x, r)—representing an input and randomness—that explains its behavior. Specifically, all messages sent by the adversary on behalf of P_k, including m, must be consistent with an honest execution of the protocol using (x, r). However, the witnesses provided across different messages do not need to be consistent. Additionally, the adversary can choose to make P_k abort at any point during the execution.

To resolve the second challenge, the protocols are designed to potentially iterate twice, but they are intentionally structured so that the semi-malicious adversary can determine in which iteration the output must be generated and from which iteration the set of input providers should be used. This allows the adversary to influence both the core-set selection process and the output generation. Specifically, we embed a backdoor key so that only entities knowing it are able to influence the execution. To conceal these decisions from the compiler until the output is generated, we encrypt critical protocol messages with a hardcoded symmetric-encryption key. This ensures that the compiler cannot determine whether the inputs are fixed until the output becomes available.

To address the third challenge, we embed a MAC key and authenticate critical messages to force the compiler to follow the next-message function correctly. This approach is effective only if the compiler does not have access to the embedded keys (Lemma 4.8). To ensure this, we consider the class of protocols for all possible key combinations and prove that, for any PBB compiler, it fails to work on a uniformly drawn protocol from this class. This suffices because any PBB compiler must be functional for all protocols within the class.

For this class of functions and assuming these underlying protocols, we prove that in any PBB compiler, parties must execute at least one instance of each provided protocol, with each party providing their input until they obtain an output. Moreover, the final output must be one of these obtained outputs (Lemma 4.10). Now the adversary's goal is to learn the output of some functions before determining the set of input providers for others. While the remainder of the proof follows the same overall logic as the FBB impossibility result, it is significantly more intricate. Unlike ideal functionalities, protocols may reach a point where the set of input providers is fixed before the output is generated. As a result, once the adversary learns the output of one function, it may already be too late to influence the inputs to other functions. To overcome this, we construct a probabilistic adversarial strategy that, with noticeable probability, gives the adversary an additional advantage, which is sufficient to violate the protocol's privacy. Notably, the attack only requires a single semi-malicious corruption.

1.2.3 Feasibility Results for Asynchronous Parallel Composition

The above state of affairs paints a grim picture of the potential for asynchronous parallel composition. To someone familiar with the classical A-MPC protocols (e.g., [BCG93, BKR94], and others), this might appear surprising. Indeed, it is not hard to verify that protocols can, in fact, be composed in parallel. Taking an abstract view of the structure of these protocols, they can all be split into three phases:

1. In Phase 1, the parties share their inputs to the player set and/or perform some private computation on these inputs (here, by 'private' we mean that no information leaks about honest parties' inputs).
2. In Phase 2, parties perform a procedure, often referred to as *Agreement on a Core Set* (or ACS [BCG93]), to agree on the core set of at least $n - t$ inputs to be considered in the computation.
3. In Phase 3, the output of the MPC is computed, and (eventually) delivered to all honest parties, on inputs from the core set.

The reason why protocols with the above abstract structure can be strongly parallelly composed is that the core set (in Phase 2) is determined independently of the inputs of honest parties, and the parties included in the core set are chosen by the adversary *before* learning any new information from the protocol. Once the core set is computed, both the inputs and the parties are fixed and cannot be changed. Intuitively, to compose such protocols in parallel, Phase 1 must be executed

for all composed protocols (e.g., π_1 and π_2 in our running example), followed by a single instance of Phase 2 to determine a *single* core set for the parallelly composed computation. This core set is then used in Phase 3 of each protocol, π_1 and π_2, to compute the output.

Readers familiar with the challenges of protocol composition, however, would realize that the above abstract view of the protocols poses several challenges, especially in the asynchronous setting, where the transition between phases might happen at different, adversarially controlled times for different protocols, and even for different parties in the same protocol. So the core question becomes:

> *Can the above intuition be turned into a proof, and if so, how non-black-box in the protocol do we need to be to achieve parallel composition?*

We answer the above question in the affirmative, and identify simple asynchronous protocol properties that enable strong and weak parallel composition, which we now describe.

Feasibility of Conditional PBB Strong Parallel Composition. In Theorem 5.2 we show that it suffices that for each of the composed protocols π_i there exists a protocol ρ_i such that: (1) we can re-write π_i as a ρ_i-hybrid protocol (i.e., it uses ρ_i as a subroutine) which implements the same functionality as π_i; (2) if we replace, in the re-written ρ_i-hybrid protocol, calls to protocol ρ_i with calls to a functionality $\mathcal{F}_{\mathsf{acs}}$ (which we define, and which, intuitively, abstracts the ACS procedure), the resulting $\mathcal{F}_{\mathsf{acs}}$-hybrid protocol also implements the same functionality as π_i; and (3) the simulator guaranteed by the security of π_i chooses the core set consistently with the output of the (simulated) $\mathcal{F}_{\mathsf{acs}}$-hybrid protocol.

We call such protocols *ACS-decomposable* (Definition 5.1). We note that such decomposition is straightforward in most (if not all) existing A-MPC protocols from the literature (e.g., [BCG93, BKR94]), which have an explicitly defined core-set agreement process. Thus, the above implies a very general and strong parallel composability statement for the existing literature. Notwithstanding, our abstraction is far more generic, as it does not even require that ρ_i securely implements $\mathcal{F}_{\mathsf{acs}}$. We believe that this abstraction provides a framework for the design of composable asynchronous protocols—if one's protocol does not fall in this class, then (strong) parallel composability might require white-box access to the protocol's code.

Feasibility of Conditional PBB Weak Parallel Composition. We prove in Theorem 5.8 that the existence of an *asynchronous committal point*, which is the asynchronous analogue of the (synchronous) committal round [MR92, DM00], is sufficient for weak parallel composition. We note that even defining such a variant is far more challenging than in the synchronous setting. We capture this property in Definition 5.5, and prove that any protocol with this property can be weakly parallelly composed in a (conditional) PBB manner.

It is worth noting that it is unclear whether or not such protocols can be strongly parallelly composed (in fact, known techniques fail, and we conjecture that this is not the case). This draws another distinction between synchronous and asynchronous protocols, as in the former case, knowledge of a committal round directly allows for straightforward strong parallel composition (and, as discussed above, a committal round is not even required [CCGZ21]).

We conclude the overview of our results by summarizing them in Table 1.

2 Preliminaries and Model

For $n \in \mathbb{N}$, let $[n]$ denote the set $\{1, \ldots, n\}$. The security parameter is denoted by κ. A function $\nu \colon \mathbb{N} \to [0, 1]$ is said to be *negligible* if for every integer $c > 0$, it holds that $\nu(\kappa) < \kappa^{-c}$ for large

parallel composition	FBB	PBB	conditional PBB
async. strong	✗	✗	✓ **Thm 5.2** (ACS-decomposable)
async. weak	✗ **Thm 4.2**	✗ **Thm 4.9**	✓ **Thm 5.8** (committal point)
async. loose (trivial)	✓	✓	✓
sync.	$\checkmark^{(*)}$ [CCGZ21]	✓ [CCGZ21]	✓ [DM00] (committal round)

Table 1: Summary of our results and comparison to prior work. *Strong* parallel composition requires a single core set for all computations; *weak* parallel composition allows different core sets for different computations, but requires independence of the core sets from the outputs of other computations; *loose* parallel composition permits adversarial correlation of core sets for certain computations with outputs of other computations. *FBB* denotes functionally black-box composition; *PBB* generic protocol black-box composition; and *conditional PBB* protocol black-box composition from a class of protocols satisfying some condition (specified in parentheses). $(*)$ Feasibility holds for protocols resilient to crashes when all corruptions are semi-honest.

enough κ. We use boldface to denote vectors $\boldsymbol{v}$.

We write our protocols using a series of numbered steps. Unless otherwise stated, the interpretation is that a party repeatedly goes through them in sequential order, attempting each instruction. Note that in the asynchronous setting, the necessary precondition for carrying out an instruction might only be met much later in the protocol. We therefore use statements of the form "Wait until <condition>. Then, <instruction>," which is to be executed at most once, or "If <condition>, then <instruction>," which can be executed multiple times.

2.1 The UC Framework

We prove our constructions secure in the UC framework [Can20], which we briefly summarize here. Protocol machines, ideal functionalities, the adversary, and the environment are all modeled as interactive Turing machine (ITM) instances, or ITIs. In some cases we consider machines with oracle access to a function f; this will be denoted in the superscript (e.g., as π^f or $\mathcal{F}^f$). An execution of protocol π consists of a series of activations of ITIs, starting with the environment $\mathcal{Z}$ who provides inputs to and collects outputs from the parties and the adversary $\mathcal{A}$; parties can also give input to and collect output from sub-parties (e.g., hybrid functionalities), and $\mathcal{A}$ can communicate with parties via "backdoor" messages. Although parties can send messages to one another, $\mathcal{A}$ is responsible for delivering them, making this model *completely* asynchronous (see Section 2.2 for the way we capture eventual-delivery channels). Protocol instances are delineated using a unique session identifier (SID). We prefer to work with SIDs explicitly, for additional clarity in the asynchronous setting.

Corruption of parties is modeled by a special `corrupt` message sent from $\mathcal{A}$ to the party; upon receipt of this message, the party sends its entire local state to $\mathcal{A}$, and in all future activations follows the instructions of $\mathcal{A}$ (in case the adversary is malicious). We will also speak of semi-honest and fail-stop adversaries, which behave in the usual way (in particular, a fail-stop adversary is a semi-honest adversary that is also allowed to crash parties). Note that different corruption types in UC are captured via implicit, model-specific "shell" code in the protocol, and not embedded in the framework itself. We emphasize that corruptions can occur at any point during the protocol (i.e., $\mathcal{A}$ is assumed to be adaptive). Denote by $\text{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}$ the probability distribution ensemble, parameterized by $\kappa \in \mathbb{N}$ and $z \in \{0,1\}^*$, corresponding to the (binary) output of $\mathcal{Z}$ on input z at the end of an execution of π with adversary $\mathcal{A}$ and security parameter κ.

The ideal-world process for functionality $\mathcal{F}$ is simply defined as an execution of the ideal protocol $\text{IDEAL}_{\mathcal{F}}$, in which the so-called "dummy" parties just forward inputs from $\mathcal{Z}$ to $\mathcal{F}$ and forward outputs from $\mathcal{F}$ to $\mathcal{Z}$ (in particular, the dummy parties do not communicate with the adversary, but rather the adversary is expected to send backdoor messages directly to $\mathcal{F}$, including corruption requests). Our functionalities implicitly respond to corruption requests in the standard way, according to the corruption type (e.g., all previous inputs of the newly corrupted party are leaked, and a malicious adversary has the option to replace its output); we refer to [Can20] for further details. The adversary in the ideal world is also called the *simulator* and denoted $\mathcal{S}$; the corresponding ensemble is denoted by $\text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$.

We are interested in unconditional (aka information-theoretic) security for our feasibility results. Thus, we say that a protocol Π statistically *UC-realizes* an ideal functionality $\mathcal{F}$ if for any computationally unbounded adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that for any computationally unbounded environment $\mathcal{Z}$, we have $\text{EXEC}_{\pi,\mathcal{A},\mathcal{Z}} \approx \text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ (i.e., the ensembles are statistically close). In case the distribution ensembles are perfectly indistinguishable (denoted $\text{EXEC}_{\pi,\mathcal{A},\mathcal{Z}} \equiv \text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$) we say that Π UC-realizes $\mathcal{F}$ with *perfect* security. Our impossibility results rule out even *computational* security, where $\mathcal{A}$, $\mathcal{S}$, and $\mathcal{Z}$ are all required to be PPT, and the two distribution ensembles are required to be computationally indistinguishable.

When π is a $(\mathcal{G}_1,\ldots,\mathcal{G}_m)$-hybrid protocol (i.e., making subroutine calls to $\text{IDEAL}_{\mathcal{G}_1},\ldots,\text{IDEAL}_{\mathcal{G}_m}$), we say that π UC-realizes $\mathcal{F}$ in the $(\mathcal{G}_1,\ldots,\mathcal{G}_m)$-hybrid model. It is known that (regular) UC-realization is equivalent to UC-realization with respect to a very specific adversary, namely the "dummy" adversary $\mathcal{D}$, which simply follows the instructions of $\mathcal{Z}$ on which messages to send and moreover reports all received messages to $\mathcal{Z}$. We sometimes use this alternate definition of security, as it is simpler to work with and involves one less quantifier.

2.2 Modeling Eventual Message Delivery in UC

As mentioned above, the base communication model in UC is completely unprotected. To capture asynchronous networks with eventual message delivery, we follow [CGHZ16, CFG+23] and work in a hybrid model with access to the multi-use *asynchronous secure message transmission* functionality $\mathcal{F}_{\mathsf{a\text{-}smt}}$, shown below, which is based on the (single-use) *eventual-delivery secure channel* functionality $\mathcal{F}_{\mathsf{ed\text{-}sec}}$ from [KMTZ13].

The functionality $\mathcal{F}_{\mathsf{a\text{-}smt}}$ models a *secure* eventual-delivery channel between a sender P_s and receiver P_r. To reflect the adversary's ability to delay the message by an arbitrary finite duration (even when P_s and P_r are not corrupted), the functionality operates in a "pull" mode, managing a message buffer M and a counter D that represents the current message delay. This counter is decremented every time P_r tries to fetch a message, which is ultimately sent once the counter hits 0. The adversary can at any time provide an additional integer delay T, and if it wishes to immediately release the messages, it needs only to submit a large negative value.[4] It is important to note that T must be encoded in unary; this ensures that the delay, while arbitrary, remains bounded by the adversary's computational resources or running time.[5] Thus, $\mathcal{F}_{\mathsf{a\text{-}smt}}$ guarantees eventual message delivery assuming that the environment gives sufficient resources to the protocol, i.e., activates P_r sufficiently many times. We additionally allow the adversary to permute the messages in the buffer; see [CGHZ16] for a discussion.

[4] Technically, we return the activation to the adversary afterwards. We refrain from explicitly specifying such details in $\mathcal{F}_{\mathsf{a\text{-}smt}}$ and our other functionalities.

[5] We refer to [Can20] for a formal definition of running time in the UC framework.

Functionality $\mathcal{F}_{\text{a-smt}}$

The functionality proceeds as follows. At the first activation, verify that $\mathsf{sid} = (P_s, P_r, \mathsf{sid}')$, where P_s and P_r are player identities. Initialize the current delay $D := 1$ and the message buffer $M := (\texttt{eom})$, where $\texttt{eom}$ is a special "end-of-messages" symbol.

- Upon receiving $(\texttt{send}, \mathsf{sid}, m)$ from P_s, insert (m, mid) at the beginning of M, where mid is a unique message ID sampled by the functionality, and send $(\texttt{leakage}, \mathsf{sid}, \mathsf{mid})$ to the adversary.
- Upon receiving $(\texttt{delay}, \mathsf{sid}, T)$ from the adversary for $T \in \mathbb{Z}$ represented in unary notation, update $D := \max(1, D + T)$.
- Upon receiving $(\texttt{fetch}, \mathsf{sid})$ from P_r, do:
  1. Update $D := \max(0, D - 1)$ and send $(\texttt{fetched}, \mathsf{sid})$ to the adversary.
  2. If $D = 0$ and $M = ((m_1, \mathsf{mid}_1), \ldots, (m_\ell, \mathsf{mid}_\ell), \texttt{eom})$, then update the message buffer as $M := ((m_2, \mathsf{mid}_2), \ldots, (m_\ell, \mathsf{mid}_\ell), \texttt{eom})$ and send $(\texttt{output}, \mathsf{sid}, m_1)$ to P_r.
- Upon receiving $(\texttt{permute}, \mathsf{sid}, \pi)$ from the adversary, if $M = ((m_1, \mathsf{mid}_1), \ldots, (m_\ell, \mathsf{mid}_\ell), \texttt{eom})$ and π is a permutation over $[\ell]$, then update $M := ((m_{\pi(1)}, \mathsf{mid}_{\pi(1)}), \ldots, (m_{\pi(\ell)}, \mathsf{mid}_{\pi(\ell)}), \texttt{eom})$.
- (Adaptive Message Replacement) Upon receiving $(\texttt{replace}, \mathsf{sid}, ((m_1, \mathsf{mid}_1), \ldots, (m_{\ell'}, \mathsf{mid}_{\ell'})), T')$ from the adversary: If P_s is corrupted, $D > 0$, and T' is a valid delay, then update $M := ((m_1, \mathsf{mid}_1), \ldots, (m_{\ell'}, \mathsf{mid}_{\ell'}), \texttt{eom})$ and $D := \max(1, T')$.

Figure 1: The asynchronous secure message transmission functionality.

Since $\mathcal{F}_{\text{a-smt}}$ (and our other asynchronous functionalities) may not provide output immediately, we must clarify what we mean when instructing a party to fetch the output from multiple instances of a hybrid (e.g., one per party), as repeatedly attempting to fetch from the first instance before continuing to the second might not make any progress. Accordingly, we require that parties fetch the output in a "round-robin fashion" across activations (i.e., after attempting to fetch from each instance, the party moves on to try the remaining steps in the protocol). It is implicit that the party stops making fetch requests to any given instance after it receives an output from that instance. Similarly, in our feasibility results for (conditional) PBB parallel composition, the compiler will run each underlying protocol in a round-robin fashion, giving one activation to each protocol in turn.

2.3 Asynchronous Secure Function Evaluation (A-SFE)

In a secure function evaluation, n parties wish to compute a (possibly randomized) function $f = (f_1, \ldots, f_n)$ over their inputs. In case $f_1 = \cdots = f_n$, we say that f has *public output*; otherwise, f is said to have *private output*. The adversary learns only the lengths of the inputs of the honest parties and the outputs of corrupted parties. In the asynchronous version of this task, which we refer to as A-SFE, the inputs of up to t parties might not be considered in the computation, and the remaining $n - t$ parties form a "core set" of input providers to the function, denoted by $\mathcal{CS}$ (which is provided to the parties in addition to the function output so that the computation remains meaningful). Note that this will occur irrespective of the actual number of corrupted parties. Thus, the maximum number t of corruptions is an explicit parameter in most of our asynchronous functionalities, as it fundamentally changes the behavior of the functionality (i.e., the minimum size of the core set).

As noted in [CFG+23], this phenomenon of "input incompleteness" is tricky to properly model in a composable manner while still guaranteeing eventual termination. Early definitional attempts [CGHZ16, Coh16], as well as more recent works, that gave the adversary the power to determine the core set, suffer from a number of issues. For example, such functionalities can stall when used as hybrids in higher-level protocols (in case the adversary does not respond), or even lead to subtle distinguishing attacks by the environment when implemented using known protocols.

To address these issues, [CFG+23] proposed an input-delay mechanism, similar to the output-delay mechanism discussed in Section 2.2, to allow the adversary more fine-grained control over when parties' inputs are considered in the computation (i.e., when parties can be said to have *participated*). We adopt this mechanism in our asynchronous functionalities.

The work of [CFG+23] focused on Byzantine agreement and stopped short of defining an A-SFE functionality. Since asynchronous MPC protocols usually have a specific structure—where inputs are committed, then some form of ACS procedure is run to agree on a set of input providers, and finally the parties collectively generate the output based on the inputs from the core set—it seems necessary to distinguish between "input participation" and "output participation" even at the functionality level. Accordingly, in the A-SFE functionality $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f,t}$ that we present below, we incorporate a third delay type. Breaking from the modeling in [CFG+23], we also reintroduce an explicit `core-set` message enabling the adversary to specify the input providers while including a *default fallback* to prevent abuse when the functionality is used as a hybrid.

Functionality $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f,t}$

The functionality is parameterized by a (randomized) n-ary function $f\colon (\{0,1\}^* \cup \{\bot\})^n \times \{0,1\}^* \to (\{0,1\}^*)^n$ and a resiliency threshold $t \in \mathbb{N}$, and proceeds as follows. At the first activation, verify that $\mathsf{sid} = (\mathcal{P}, \mathsf{sid}')$ where $\mathcal{P}$ is a player set of size n. For each $P_i \in \mathcal{P}$, initialize x_i and y_i to a default value $\bot$, set the participation flags $\mathsf{part}_i^{\text{in}} = \mathsf{part}_i^{\text{out}} := 0$, and set the delay values $D_i^{\text{in-part}} = D_i^{\text{out-part}} = D_i^{\text{out-rel}} := 1$ (where "in-part" stands for input participation, "out-part" for output participation, and "out-rel" for output release). Also initialize the core set $\mathcal{CS}$ to $\bot$.

- Upon receiving $(\texttt{delay}, \mathsf{sid}, P_i, \mathsf{type}, D)$ from the adversary for $P_i \in \mathcal{P}$, with $\mathsf{type} \in \{\text{in-part}, \text{out-part}, \text{out-rel}\}$, and $D \in \mathbb{Z}$ represented in unary, update $D_i^{\mathsf{type}} := \max(1, D_i^{\mathsf{type}} + D)$.
- Upon receiving $(\texttt{input}, \mathsf{sid}, x)$ from $P_i \in \mathcal{P}$ (or the adversary on behalf of corrupted P_i), record $x_i := x$, run the Input Submission Procedure (defined below), and send $(\texttt{leakage}, \mathsf{sid}, P_i, l)$ to the adversary, where $l := (|x_1|, \ldots, |x_n|)$.
- Upon receiving $(\texttt{core-set}, \mathsf{sid}, \mathcal{I})$ from the adversary, if $\mathcal{I} \subseteq \mathcal{P}$ with $|\mathcal{I}| \geq n - t$ and $\mathsf{part}_j^{\text{in}} = 1$ for all $P_j \in \mathcal{I}$, and the function has not been evaluated yet (i.e., $y_1 = \ldots = y_n = \bot$), then record $\mathcal{CS} := \mathcal{I}$.
- Upon receiving $(\texttt{fetch}, \mathsf{sid})$ from $P_i \in \mathcal{P}$ (or the adversary on behalf of corrupted P_i), run the Input Submission, Output Generation, and Output Release Procedures (in that order), and send $(\texttt{fetched}, \mathsf{sid}, P_i)$ to the adversary and any messages set by the Output Release Procedure to P_i.

Input Submission Procedure: If P_i has already provided input then do:

1. Update $D_i^{\text{in-part}} := D_i^{\text{in-part}} - 1$.
2. If $D_i^{\text{in-part}} = 0$, set $\mathsf{part}_i^{\text{in}} := 1$.

Output Generation Procedure: If $\sum_{j=1}^{n} \mathsf{part}_j^{\text{in}} \geq n - t$ then do:

1. Update $D_i^{\text{out-part}} := D_i^{\text{out-part}} - 1$.
2. If $D_i^{\text{out-part}} = 0$, set $\mathsf{part}_i^{\text{out}} := 1$.
3. If $\sum_{j=1}^{n} \mathsf{part}_j^{\text{out}} \geq t + 1$, and $\mathcal{CS} \neq \bot$, and $y_i = \bot$, then choose $r \leftarrow \{0,1\}^*$ and evaluate $(y_1, \ldots, y_n) := f(x'_1, \ldots, x'_n; r)$, where $x'_j := x_j$ for each $P_j \in \mathcal{CS}$ and $x'_j := \bot$ otherwise.
   Additionally, set $(\texttt{adv-output}, \mathsf{sid}, (a_1, \ldots, a_n))$ to be sent to the adversary, where $a_j := y_j$ if P_j is corrupted and $a_j := \bot$ otherwise.

Output Release Procedure: If $\sum_{j=1}^{n} \mathsf{part}_j^{\text{out}} \geq n - t$ then do:

1. Update $D_i^{\text{out-rel}} := D_i^{\text{out-rel}} - 1$.

2. If $D_i^{\text{out-rel}} = 0$, then do the following. First, if $\mathcal{CS} = \bot$, set $\mathcal{CS} := \{P_j \in \mathcal{P} \mid \mathsf{part}_j^{\text{in}} = 1\}$. Now, if $y_i = \bot$, then choose $r \leftarrow \{0,1\}^*$ and evaluate $(y_1, \ldots, y_n) := f(x'_1, \ldots, x'_n; r)$, where $x'_j := x_j$ for each $P_j \in \mathcal{CS}$ and $x'_j := \bot$ otherwise.

   Finally, set $(\texttt{output}, \mathsf{sid}, (\mathcal{CS}, y_i))$ to be sent to P_i (to be delivered as a response to a `fetch` request).

Figure 2: The A-SFE functionality.

2.4 Ideal Functionalities for Standard Asynchronous Primitives

Finally, we present UC functionalities for two asynchronous primitives that are needed by our feasibility results. Such functionalities have appeared in the literature before, but the versions we present incorporate the refined modeling choices described above for A-SFE.

Agreement on a Core Set. Agreement on a Core Set (ACS), coined by Ben-Or, Canetti, and Goldreich [BCG93], is a fundamental primitive used in asynchronous MPC protocols to decide on a common subset of input providers to the computation. The variant we consider processes *dynamic inputs* from parties, which form *accumulative sets*; these inputs represent indices of parties whose (private) inputs have been successfully distributed (e.g., verifiably shared), and it is guaranteed that each honest party's set will eventually accumulate at least $n-t$ indices. Moreover, the accumulated sets of any two honest parties will eventually converge, a property we refer to as the *convergence precondition*. Given these two preconditions, an ACS protocol should terminate with a unique set of at least $n - t$ indices (corresponding to the parties in the core set) that will eventually be contained in the accumulative sets of all honest parties. Note that, in the worst case, the core set will be adversarially chosen and contain the indices of all t corrupted parties.

Several formulations of ACS ideal functionalities (e.g., [GLZS24, AAPP24, AAPS24]) omit eventual-delivery semantics and are not "well-formed" in the UC sense, as they explicitly reference the corruption set when generating outputs. We formulate an ideal functionality for ACS below. As in Section 2.3, $\mathcal{F}_{\mathsf{acs}}^{n,t}$ uses an input-delay mechanism to decide when (dynamic) inputs are considered. The adversary is allowed to choose the core set, as long as it is contained in the accumulative set of at least one honest party (at the time of output generation), which guarantees eventual containment in all honest parties' accumulative sets due to the convergence precondition. We stress that the calling protocol must actually enforce the preconditions in order for the output to be useful. The output-delay mechanism captures eventual termination, as usual.

Functionality $\mathcal{F}_{\mathsf{acs}}^{n,t}$

The functionality is parameterized by n and $t \in \mathbb{N}$. On first activation, verify $\mathsf{sid} = (\mathcal{P}, \mathsf{sid}')$ where $|\mathcal{P}| = n$. For each $P_i \in \mathcal{P}$, initialize $S_i := \emptyset$ (submitted elements), $P_i := \emptyset$ (pending inputs), a priority order $\rho_i := \bot$, and delays $D_i^{\text{in}} = D_i^{\text{out}} := 1$. Also, initialize $A := \bot$ and $Y := \bot$.

- Upon receiving $(\texttt{delay}, \mathsf{sid}, P_i, \mathsf{type}, D)$ from the adversary, where $\mathsf{type} \in \{\text{in}, \text{out}\}$ and $D \in \mathbb{Z}$ (represented in unary), set $D_i^{\mathsf{type}} := \max(1, D_i^{\mathsf{type}} + D)$.
- Upon receiving $(\texttt{input}, \mathsf{sid}, k)$ from $P_i \in \mathcal{P}$ (or the adversary for corrupted P_i), with $k \in [n]$, set $P_i := P_i \cup \{k\}$, run the *Input Submission Procedure*, and send $(\texttt{leakage}, \mathsf{sid}, P_i, k)$ to the adversary.
- Upon receiving $(\texttt{order}, \mathsf{sid}, P_i, \rho)$ from the adversary, where ρ encodes a total order on $[n]$ (e.g., a permutation/ranking), record $\rho_i := \rho$.
- Upon receiving $(\texttt{adv-input}, \mathsf{sid}, X)$ from the adversary, if $X \subseteq [n]$ and $|X| \geq n - t$, set $A := X$.
- Upon receiving $(\texttt{fetch}, \mathsf{sid})$ from $P_i \in \mathcal{P}$ (or the adversary for corrupted P_i), run the *Input Submission Procedure* and the *Output Release Procedure*, then send $(\texttt{fetched}, \mathsf{sid}, P_i)$ to the adversary along with any messages sent to P_i by the Output Release Procedure.

Input Submission Procedure. If P_i has provided at least one input, then:

1. $D_i^{\text{in}} := D_i^{\text{in}} - 1$.
2. If $D_i^{\text{in}} = 0$ and $P_i \neq \emptyset$, choose the next element to submit as follows:
   - If $\rho_i \neq \perp$, let $k^\star$ be the element of P_i with the smallest rank under ρ_i;
   - otherwise, arbitrarily choose $k^\star$ from P_i.

   Update $P_i := P_i \setminus \{k^\star\}$ and $S_i := S_i \cup \{k^\star\}$, then reset $D_i^{\text{in}} := 1$.

Output Release Procedure. For each value $k \in [n]$, count how many parties have already submitted k (i.e., in how many S_j it appears). Call a value *eligible* if it appears in at least $t+1$ parties' submitted sets. If there are at least $n - t$ eligible values, then:

1. $D_i^{\text{out}} := D_i^{\text{out}} - 1$.
2. If $D_i^{\text{out}} = 0$ and $Y = \perp$, then set Y as follows:
   - If $A \neq \perp$ and every element of A is eligible and $|A| \geq n - t$, set $Y := A$;
   - otherwise, set Y to be the set of all eligible values (which is of size at least $n - t$).

   In any case, send $(\texttt{output}, \mathsf{sid}, Y)$ to P_i.

Figure 3: The ACS functionality.

The ACS functionality is realized by many works, starting with the early ACS protocols [BCG93, Can96, BKR94] and continuing with more recent and more efficient constructions [ZD22, CFG+23, AAPS24].

Asynchronous Broadcast (A-Cast). We also require Bracha's asynchronous broadcast (A-Cast) [Bra87], sometimes called *reliable broadcast*. This primitive allows a distinguished sender to distribute its input, such that if the sender is honest, all honest parties will eventually receive the sender's input, but even a corrupted sender can only cause either non-termination or agreement on *some* value. An ideal functionality for A-Cast was given in [CFG+23], which we reproduce below.

Functionality $\mathcal{F}_{\text{a-cast}}^{n,t}$

The functionality is parameterized by the number n of parties and a resiliency threshold $t \in \mathbb{N}$, and it proceeds as follows. Start a session once a message of the form $(\texttt{send}, \mathsf{sid}, \cdot)$ with $\mathsf{sid} = (\mathcal{P}, P_s, \mathsf{sid}')$ is received from (the sender) $P_s \in \mathcal{P}$, where $\mathcal{P}$ is a player set of size n. For each $P_i \in \mathcal{P}$, initialize $\mathsf{part}_i := 0$ and delay values $D_i^{\text{in}} = D_i^{\text{out}} := 1$. Also initialize $y := \perp$.

- Upon receiving $(\texttt{delay}, \mathsf{sid}, P_i, \mathsf{type}, D)$ from the adversary for $P_i \in \mathcal{P}$, with $\mathsf{type} \in \{\text{in}, \text{out}\}$, and $D \in \mathbb{Z}$ represented in unary notation, update $D_i^{\mathsf{type}} := \max(1, D_i^{\mathsf{type}} + D)$.
- Upon receiving $(\texttt{send}, \mathsf{sid}, v)$ from $P_i = P_s$ (or the adversary on behalf of corrupted P_s), run the Input Submission Procedure, record $y := v$, and send $(\texttt{leakage}, \mathsf{sid}, v)$ to the adversary.
- Upon receiving $(\texttt{replace}, \mathsf{sid}, v)$ from the adversary (v can be $\perp$), if P_s is corrupted and no party has received their output yet, then update $y := v$.
- Upon receiving $(\texttt{fetch}, \mathsf{sid})$ from $P_i \in \mathcal{P}$ (or the adversary on behalf of corrupted P_i), run the Input Submission and Output Release Procedures, and send $(\texttt{fetched}, \mathsf{sid}, P_i)$ to the adversary and any messages set by the Output Release Procedure to P_i.

Input Submission Procedure: If $\mathsf{part}_i = 0$ then do:

1. Update $D_i^{\text{in}} := D_i^{\text{in}} - 1$.
2. If $D_i^{\text{in}} = 0$ then set $\mathsf{part}_i := 1$.

Output Release Procedure: If $\sum_{j=1}^{n} \mathsf{part}_j \geq n - t$ and $y \neq \perp$, then do:

1. Update $D_i^{\text{out}} := D_i^{\text{out}} - 1$.
2. If $D_i^{\text{out}} = 0$ then set $(\mathtt{output}, \mathsf{sid}, y)$ to be sent to P_i.

Figure 4: The A-Cast functionality from [CFG+23].

Bracha's original protocol [Bra87] can be used to UC-realize $\mathcal{F}_{\mathsf{a\text{-}cast}}^{n,t}$, provided $t < n/3$.

3 Parallel Composition of A-SFE Functionalities

In this section, we introduce key notions for the parallel composition of asynchronous cryptographic protocols. We start in Section 3.1 by defining two types of parallel composition for A-SFE functionalities, differing only in their core-set mechanism. As we will see later, this choice has implications for the (in)feasibility of black-box composition. Then, in Section 3.2, we pin down the notions of black-box protocols (FBB, generic PBB, and conditional PBB) that will characterize our impossibility results. While our definitions focus on A-SFE, they can be adapted for more general functionalities.

3.1 Strong and Weak Asynchronous Parallel Composition

Let $M, n, t \in \mathbb{N}$ with $t < n$, let $f_1, \ldots, f_M$ be n-ary functions, and let $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t}, \ldots, \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}$ be the corresponding A-SFE functionalities. We wish to define an ideal functionality that represents the parallel composition of $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t}, \ldots, \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}$ (i.e., that securely computes the natural parallel composition $[f_1 \parallel \cdots \parallel f_M]$ of the functions $f_1, \ldots, f_M$). There are various ways to do this, and decisions must be made regarding input provision, output retrieval, and notably, core-set formation. In the composed functionality, we stipulate that the input to each underlying functionality is provided simultaneously, as a vector of M inputs. Similarly, the outputs from the underlying functionalities are returned collectively. Thus, we specifically capture parallel composition rather than general concurrent composition. The handling of the core set is more complex, as we explore next.

Strong parallel composition of A-SFE functionalities. Perhaps the most natural form of asynchronous parallel composition mandates the same set of input providers to all M functions. This singular core set effectively retains input independence for the composed functionality. Fortunately, we can model this notion (which we term as *strong parallel composition*) by casting it once again as an A-SFE functionality, parameterized with the function $[f_1 \parallel \cdots \parallel f_M]$:

Definition 3.1 (Strong Parallel Composition)**.** *Let $M, n \in \mathbb{N}$ and let $f_1, \ldots, f_M$ be (randomized) n-ary functions. We denote by $[f_1 \parallel \cdots \parallel f_M]$ the (randomized) n-ary function that computes their parallel composition. That is, on input $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$, where each $\boldsymbol{x}_i$ is an M-tuple of inputs $(x_i^1, \ldots, x_i^M)$, the output is $(\boldsymbol{y}_1, \ldots, \boldsymbol{y}_n) = ((y_1^1, \ldots, y_1^M), \ldots, (y_n^1, \ldots, y_n^M))$, where $(y_1^j, \ldots, y_n^j) := f_j(x_1^j, \ldots, x_n^j)$ for each $j \in [M]$.*

For $t \in \mathbb{N}$, the strong parallel composition *of the A-SFE functionalities $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t}, \ldots, \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}$, denoted $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \cdots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{strong}}$, is defined as the A-SFE functionality $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{[f_1 \parallel \cdots \parallel f_M],t}$.* [6]

By the specification of the A-SFE functionality in Section 2.3, each party receives the core set along with the output vector.

[6] With slight abuse of notation, we redefine the leakage as $\boldsymbol{l} = (l^1, \ldots, l^M)$, where $l^j := (|x_1^j|, \ldots, |x_n^j|)$ for each $j \in [M]$.

Weak parallel composition of A-SFE functionalities. We can alternatively permit each underlying functionality to manage its own core set, potentially resulting in M different core sets in total. We refer to this approach as *weak parallel composition*. This notion, while less useful in practice, still has an interesting feasibility landscape for black-box protocols.

Since the A-SFE functionality only considers a single set of input providers when evaluating its function, we model weak parallel composition using a functionality wrapper instead, shown below. The wrapper $\mathcal{W}_{\mathsf{wpc}}$ internally runs copies of the M A-SFE functionalities, and passes individual inputs to them. The adversary is allowed to specify a core set or delay value for an individual functionality, and these are also forwarded by the wrapper. Importantly, a fetch request from a party is translated into a fetch request for each of the underlying functionalities, but the output is only relayed back as a vector, i.e., once each functionality has generated output. Note that this includes the output for corrupted parties; hence, this form of parallel composition at least provides what might be called "core-set independence." That is, while the adversary can select a different set of input providers to each function, it must do so before learning the output of any of those functions.

Wrapper Functionality $\mathcal{W}_{\mathsf{wpc}}(\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t}, \ldots, \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t})$

The wrapper receives $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t}, \ldots, \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}$ as the underlying functionalities and proceeds as follows. At the first activation, verify that $\mathsf{sid} = (\mathcal{P}, \mathsf{sid}')$ where $\mathcal{P}$ is a player set of size n. For each $P_i \in \mathcal{P}$ and $j \in [M]$, initialize $\mathcal{CS}_i^j$ and y_i^j to a default value $\perp$, and define $\mathsf{sid}_j := (\mathsf{sid}, j)$.

- Upon receiving $(\texttt{delay}, \mathsf{sid}, j, P_i, \mathsf{type}, D)$ from the adversary for $j \in [M]$, send $(\texttt{delay}, \mathsf{sid}_j, P_i, \mathsf{type}, D)$ to $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_j,t}$.
- Upon receiving $(\texttt{input}, \mathsf{sid}, \boldsymbol{x})$ from $P_i \in \mathcal{P}$ (or the adversary on behalf of corrupted P_i) where $\boldsymbol{x} = (x^1, \ldots, x^M)$, send $(\texttt{input}, \mathsf{sid}_j, x^j)$ to $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_j,t}$ for each $j \in [M]$.
  After receiving back $(\texttt{leakage}, \mathsf{sid}_j, P_i, l^j)$ from $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_j,t}$ for all $j \in [M]$, let $\boldsymbol{l} := (l^1, \ldots, l^M)$ and send $(\texttt{leakage}, \mathsf{sid}, P_i, \boldsymbol{l})$ to the adversary.
- Upon receiving $(\texttt{core-set}, \mathsf{sid}, j, \mathcal{I})$ from the adversary for $j \in [M]$, send $(\texttt{core-set}, \mathsf{sid}_j, \mathcal{I})$ to $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_j,t}$.
- Upon receiving $(\texttt{fetch}, \mathsf{sid})$ from $P_i \in \mathcal{P}$ (or the adversary on behalf of corrupted P_i), send $(\texttt{fetch}, \mathsf{sid}_j)$ to $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_j,t}$ for each $j \in [M]$.
  After receiving back $(\texttt{output}, \mathsf{sid}_j, (\mathcal{CS}^j, y^j))$ from any $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_j,t}$, record $\mathcal{CS}_i^j := \mathcal{CS}^j$ and $y_i^j := y^j$; also, after receiving back $(\texttt{adv-output}, \mathsf{sid}_j, (a_1^j, \ldots, a_n^j))$, record $y_k^j := a_k^j$ for all corrupted P_k.
  Then, if $y_i^j \neq \perp$ for all $j \in [M]$, let $\boldsymbol{\mathcal{CS}}_i := (\mathcal{CS}_i^1, \ldots, \mathcal{CS}_i^M)$, let $\boldsymbol{y}_i := (y_i^1, \ldots, y_i^M)$, and send $(\texttt{output}, \mathsf{sid}, (\boldsymbol{\mathcal{CS}}_i, \boldsymbol{y}_i))$ to P_i.[a] Similarly, if $y_k^j \neq \perp$ for all corrupted P_k and $j \in [M]$, send $(\texttt{adv-output}, \mathsf{sid}, (\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n))$ to the adversary, where $\boldsymbol{a}_k := (y_k^1, \ldots, y_k^M)$ if P_k is corrupted and $\boldsymbol{a}_k := \perp$ otherwise, if not sent previously.
  Additionally, send $(\texttt{fetched}, \mathsf{sid}, P_i)$ to the adversary.

[a] We stress that even when P_i is corrupted, the adversary does not learn any y_i^j until the entire vector $\boldsymbol{y}_i$ is determined.

Figure 5: Wrapper for the weak parallel composition of A-SFE functionalities.

For clarity in our later statements, we introduce the following notation for the weak parallel composition of A-SFE functionalities, to better distinguish it from strong parallel composition.

Definition 3.2 (Weak Parallel Composition)**.** *Let $M, n, t \in \mathbb{N}$ and let $f_1, \ldots, f_M$ be (randomized) n-ary functions. The* weak parallel composition *of the A-SFE functionalities $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t}, \ldots, \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}$, denoted $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \cdots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{weak}}$, is defined by $\mathcal{W}_{\mathsf{wpc}}(\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t}, \ldots, \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t})$.*

As one would expect, strong parallel composition implies weak parallel composition. We formalize this in the following proposition, whose proof is straightforward. It follows then that feasibility results for the strong notion imply feasibility results for the weak notion, and impossibility results for the weak notion imply impossibility results for the strong notion.

Proposition 3.3. *Let $M, n, t \in \mathbb{N}$ with $t < n$, let $f_1, \ldots, f_M$ be n-ary functions, and let $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t}, \ldots, \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}$ be n-party A-SFE functionalities. The ideal protocol for $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \| \cdots \| \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{strong}}$ UC-realizes $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \| \cdots \| \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{weak}}$ with perfect security in the presence of any t-adversary.*

3.2 Functionally Black-Box (FBB) and Protocol Black-Box (PBB) Composition

Black-box techniques play a prominent role in cryptography. Indeed, constructions that make only black-box use of the underlying primitives are often more efficient than their non-black-box counterparts, and black-box transformations between protocols can lead to more modular implementations of complex functionalities as well as offer ways to generically "lift" from one security notion or model to another. Here we consider asynchronous parallel composition that is black-box in the underlying functionalities or protocols, and give appropriate definitions. While the notions we introduce are not unique to asynchronous protocols (and are in fact based on existing notions for synchronous parallel composition), they must be tuned for the language of A-SFE functionalities.

FBB Protocols. The vast body of work on (general-purpose) MPC typically assumes that an explicit representation of the function to be computed (e.g., as a Boolean or arithmetic circuit) is known and available to the parties running the protocol. The size of such low-level representations therefore presents an efficiency bottleneck for MPC protocols. To investigate the extent to which this limitation is inherent, Rosulek [Ros12] initiated the systematic study of so-called *functionally black-box* (FBB) protocols, where parties have only local oracle access to the function, and in particular do not know its code.[7] This is captured by requiring that the protocol can securely compute any function f in a class $\mathcal{C}$ when instantiated with an oracle for f. Rosulek showed that in the 2-party (synchronous) setting, generic FBB protocols are impossible even for semi-honest security, assuming one-way functions; this assumption was later removed in [IKP+16].

In the following definition, we cast the notion of FBB protocols within our model. As in [CCGZ21], we allow parties to call an (oracle-aided) ideal functionality $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C},t}$ that computes the (hidden) function $f \in \mathcal{C}$.[8] This was used in [CCGZ21] to achieve round-preserving FBB parallel composition against semi-honest adversaries in the synchronous setting, and also to show that even this augmented form of FBB composition has significant barriers when the adversary can be malicious. Our results in Section 4.1 imply that when the network is asynchronous, FBB weak parallel composition against even fail-stop adversaries is impossible, irrespective of whether the protocol must also preserve round complexity. We give both a definition for standalone A-SFE functionalities (which is necessary for some of our arguments) and a definition specifically geared towards realizing the strong/weak parallel composition of multiple A-SFE functionalities. Recall that while the strong parallel composition of (classes of) A-SFE functionalities can be cast as a (class of) A-SFE functionalities, this is not true for the notion of weak parallel composition.

Definition 3.4 (FBB Protocols)**.** *Let $\mathcal{C} = \{f : (\{0,1\}^* \cup \{\bot\})^n \times \{0,1\}^* \to (\{0,1\}^*)^n\}$ be a class of n-ary functions and $t \in \mathbb{N}$. Denote by $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C},t}$ the A-SFE functionality, implemented as an*

[7] We emphasize that this restriction applies only to the protocol, and not to the adversary, simulator, or environment postulated by the security definition.

[8] Technically, we allow access to M such functionalities $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C}_1,t}, \ldots, \mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C}_M,t}$ when realizing their parallel composition.

(uninstantiated) oracle machine, that in order to compute $f(x_1, \ldots, x_n; r)$*, queries the oracle on* $(x_1, \ldots, x_n; r)$ *and stores the response* $(y_1, \ldots, y_n)$*. We say that protocol* $\pi = (\pi_1, \ldots, \pi_n)$ *is an FBB protocol for* $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C},t}$ *if for every* $f \in \mathcal{C}$*, the protocol* $\pi^f = (\pi_1^f, \ldots, \pi_n^f)$ *UC-realizes* $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f,t}$*.*

Now, let $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C}_1,t}, \ldots, \mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C}_M,t}$ *be (classes of) n-party A-SFE functionalities. We say that protocol* ρ *is an* FBB protocol for $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C}_1,t} \| \cdots \| \mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C}_M,t}]_{\text{strong}}$ *(resp.,* $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C}_1,t} \| \cdots \| \mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C}_M,t}]_{\text{weak}}$*) if for every choice of functions* $f_1, \ldots, f_M$ *from* $\mathcal{C}_1, \ldots, \mathcal{C}_M$*, the protocol* $\rho^{f_1,\ldots,f_M}$ *UC-realizes* $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \| \cdots \| \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{strong}}$ *(resp.,* $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \| \cdots \| \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{weak}}$*). Note that* ρ *will typically be specified in a hybrid model with access to the oracle-aided functionalities* $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C}_1,t}, \ldots, \mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C}_M,t}$*, instantiated using the appropriate functions.*

Next, we describe and formally define PBB protocols. To do so, we first define the "next-message" function of a protocol.

Definition 3.5 (Next-Message Functions)**.** *Let* π *be an n-party protocol with input space* $\mathcal{X}$*, randomness space* $\mathcal{R}$*, message space* $\mathcal{M}$*, and output space* $\mathcal{O}$*. For each party* P_i*, the* next-message function *of* P_i *in* π*, denoted by* $f_{\text{nxt-msg}}^{\pi,i}$*, is defined as follows:*

$$f_{\text{nxt-msg}}^{\pi,i} : \mathcal{X} \times \mathcal{R} \times \mathcal{M}^* \to (\mathcal{M}^n \times \mathcal{O}) \cup \{\bot\}.$$

For input $x \in \mathcal{X}$*, randomness* $r \in \mathcal{R}$*, and history* $h \in \mathcal{M}^*$ *(representing the sequence of messages received by* P_i *so far), function* $f_{\text{nxt-msg}}^{\pi,i}(x, r, h)$ *outputs:*

- $(m_1, \ldots, m_n, o) \in \mathcal{M}^n \times \mathcal{O}$*, where* m_j *is the message to be sent to party* P_j *(for* $j \in [n]$*), and* $o \in \mathcal{O}$ *is the potential output of* P_i *at this step, or*
- $\bot$*, if* P_i *sends no messages and produces no output at this step.*

PBB Protocols. We also consider realizing A-SFE functionalities in a *protocol black-box* (PBB) manner (i.e., given access to the next-message function $f_{\text{nxt-msg}}^{\pi} = (f_{\text{nxt-msg}}^{\pi,1}, \ldots, f_{\text{nxt-msg}}^{\pi,n})$ of an arbitrary protocol $\pi = (\pi_1, \ldots, \pi_n)$ that computes the functionality). As with FBB protocols, this only makes sense with respect to an entire *class* of functions, since otherwise the protocol could simply compute the function from scratch. Moreover, if we consider only a single class of A-SFE functionalities, then the protocol could simply emulate a secure execution by having each party use its respective next-message function to determine which messages to send. We therefore define PBB protocols for the strong and weak parallel composition of classes of A-SFE functionalities as follows.

Definition 3.6 (PBB Protocols)**.** *Let* $M, n, t \in \mathbb{N}$ *and let* $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C}_1,t}, \ldots, \mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C}_M,t}$ *be (classes of) n-party A-SFE functionalities. A protocol* π *is a* PBB protocol for $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C}_1,t} \| \cdots \| \mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C}_M,t}]_{\text{strong}}$ *(resp.,* $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C}_1,t} \| \cdots \| \mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C}_M,t}]_{\text{weak}}$*) if for every choice of functions* $f_1, \ldots, f_M$ *from* $\mathcal{C}_1, \ldots, \mathcal{C}_M$*, and any selection of protocols* $\rho_1, \ldots, \rho_M$ *UC-realizing* $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t}, \ldots, \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}$*, the protocol* $\pi^{f_{\text{nxt-msg}}^{\rho_1},\ldots,f_{\text{nxt-msg}}^{\rho_M}}$ *(for convenience, denoted as* $\pi^{\rho_1,\ldots,\rho_M}$*) UC-realizes* $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \| \cdots \| \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{strong}}$ *(resp.,* $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \| \cdots \| \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{weak}}$*).*

The above definition essentially captures *generic* PBB parallel composition, which we show in Section 4.2 is impossible facing semi-malicious adversaries in the asynchronous setting for the weak notion—in stark contrast to the feasibility result in [CCGZ21] for (round-preserving) PBB synchronous parallel composition against even malicious adversaries. By restricting the protocols $\rho_1, \ldots, \rho_M$ to come from a subclass of protocols realizing $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t}, \ldots, \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}$ (e.g., protocols satisfying some "nice" structural properties that are amenable to compilation), we obtain what we call *conditional* PBB parallel composition. In Sections 5.1 and 5.2, we present feasibility results for conditional PBB asynchronous parallel composition in both the strong and the weak sense for different subclasses, which at least include known protocols.

We close this section by noting that FBB parallel composition implies PBB parallel composition, and thus an impossibility for the latter implies an impossibility for the former.

4 Impossibility of Black-Box Asynchronous Parallel Composition

In this section, we demonstrate that for a certain class of functions, it is impossible to compute the parallel composition, of even two instances, in an FBB manner. We then extend this result by proving that for the same class of functions, there exist protocols for which no PBB compiler can achieve weak parallel composition. At first glance, it might seem that the impossibility result for PBB implies the one for FBB. However, due to the different settings in which each result is established, they are in fact incomparable. Specifically, our FBB impossibility result holds against fail-stop adversaries without any additional assumptions, while the PBB result relies on semi-malicious corruptions and the existence of threshold leveled homomorphic encryption with computationally strong homomorphism. Moreover, the two results apply to different ranges of t and n. For these reasons, we present both results here.

Recall that in asynchronous SFE functionalities with n parties, a t-adversary can exclude up to t inputs from the computation and, if acting maliciously, can also arbitrarily modify up to t of the remaining $n-t$ inputs. We refer to this vector of effective inputs, with up to t missing and up to t altered elements, as a *t-admissible input vector*. More formally:

Definition 4.1 (Admissible Inputs)**.** *Let $n, t \in \mathbb{N}$ with $t < n$ and let $\boldsymbol{x} = (x_1, \ldots, x_n) \in (\{0,1\}^*)^n$ be a vector representing n inputs, none of which is $\perp$ (the symbol for exclusion). Let $\boldsymbol{x}' = (x'_1, \ldots, x'_n) \in (\{0,1\}^* \cup \{\perp\})^n$. We say that $\boldsymbol{x}'$ is a* t-admissible input vector *(t-admissible input for short) with respect to $\boldsymbol{x}$ if and only if $\boldsymbol{x}'$ contains at most t instances of $\perp$ and, for any subset $S \subseteq [n]$ of size $|S| > t$, there exists an index $i \in S$ such that either $x'_i = \perp$ or $x'_i = x_i$.*

4.1 Impossibility of FBB Weak Parallel Composition

We begin by proving that there exists a class of functions for which no FBB protocol can achieve the weak notion of parallel composition even against fail-stop adversaries. Recall that FBB protocols also have access to ideal functionalities that compute those functions (see Definition 3.4).

Let $\mathcal{C}_{\mathsf{pred}}(n, \kappa)$ ($\mathcal{C}_{\mathsf{pred}}$ for short) denote the class of all deterministic n-ary polynomial-time computable predicates with inputs from $(\{0,1\}^\kappa \cup \{\perp\})^n$.

Theorem 4.2. *Let $n, t \in \mathbb{N}$ such that $t < n - 2$. There is no FBB protocol for computing $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C}_{\mathsf{pred}},t} \,\|\, \mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C}_{\mathsf{pred}},t}]_{\mathrm{weak}}$ in the $(\mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C}_{\mathsf{pred}},t}, \mathcal{F}_{\mathsf{a\text{-}smt}})$-hybrid model against static fail-stop t-adversaries with computational security.*

The high-level idea is that if the adversary can choose a specific set of input providers for one ideal functionality and receive its corresponding output while delaying the other, it can then set the input providers for the delayed functionality based on the received output. This breaks the input independence between the two functionalities, which are intended to be evaluated in parallel. Using this idea, the adversary can create a certain output distribution that cannot be reproduced in the ideal world. However, several subtle challenges must be addressed. For instance, we must establish that the compiled protocol indeed relies on the ideal functionalities and incorporates their outputs into the final computation. Moreover, from the adversary's perspective, it is not always clear which instance of the ideal functionality is invoked on a t-admissible input and which is merely a dummy call unrelated to the actual inputs. To address these issues, the adversary's strategy (and our argument) requires more nuance than the simplified description above.

Looking ahead, we note that the attack only considers *asynchronous distributed point functions* (A-DPF), but we require the compiler to support the broader class $\mathcal{C}_{\mathsf{pred}}$ to guarantee that it must rely on the ideal functionalities and incorporate their outputs into the final computation (see Part 2 of Lemma 4.3). Further, an interesting aspect of our proof is that the adversary does not crash any party, and a single semi-honest corruption is enough. However, security against crashes is still crucial, as without it, parties could wait for messages from all others at each step, effectively synchronizing the execution.

Proof of Theorem 4.2. For the sake of contradiction, assume that there exists an FBB protocol π that realizes $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C}_{\mathsf{pred}},t} \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C}_{\mathsf{pred}},t}]_{\mathrm{weak}}$ in the $(\mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C}_{\mathsf{pred}},t}, \mathcal{F}_{\mathsf{a\text{-}smt}})$-hybrid model against static fail-stop t-adversaries with computational security.

Initially, we define specific sets of inputs that are essential to our proof. For non-empty sets $S, S' \subseteq \{P_1, \ldots, P_n\}$ with $S \neq S'$ and $v \in \{0,1\}^\kappa$, let $X_{S,S',v}$ represent the set of all inputs $\boldsymbol{x} = (x_1, \ldots, x_n)$ that satisfy $\bigoplus_{P_i \in S} x_i = v$ and $\bigoplus_{P_i \in S'} x_i \neq v$. Additionally, let $S_1 = \{P_1, \ldots, P_{n-t}\}$ and $S_2 = \{P_2, \ldots, P_{n-t+1}\}$.

For any $\beta \in \{0,1\}^\kappa$ and $\boldsymbol{u} \in (\{0,1\}^\kappa \cup \{\perp\})^n$, define the n-ary A-DPF $f_\beta \in \mathcal{C}_{\mathsf{pred}}$ as follows:

$$f_\beta(\boldsymbol{u}) = \begin{cases} 0 & \text{if } \beta \neq \bigoplus_{u_i \neq \perp} u_i \\ 1 & \text{if } \beta = \bigoplus_{u_i \neq \perp} u_i \end{cases}$$

Let $\beta_1, \beta_2 \leftarrow \{0,1\}^\kappa$ be uniformly random and independent strings and denote $f_1 = f_{\beta_1}$ and $f_2 = f_{\beta_2}$. We show that for a carefully designed environment and adversary, the environment can distinguish between the real and ideal worlds with noticeable probability. By the probabilistic method, this implies that there exist fixed values of β_1 and β_2 for which the same distinguishing advantage holds. For any $b \in [2]$ and $i \in [n]$, let $x_{b,i}$ denote the input of party P_i for function f_b, and define $\boldsymbol{x}_b = (x_{b,1}, \ldots, x_{b,n})$.

Consider an environment that activates parties in a round-robin fashion and selects the inputs $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ by randomly choosing between the following two options:

- Sample $\boldsymbol{x}_1 \leftarrow X_{S_1,S_2,\beta_1}$ and $\boldsymbol{x}_2 \leftarrow X_{S_1,S_2,\beta_2}$.
- Sample $\boldsymbol{x}_1 \leftarrow X_{S_2,S_1,\beta_1}$ and $\boldsymbol{x}_2 \leftarrow X_{S_2,S_1,\beta_2}$.

Note that the sampling process is efficient. To sample $\boldsymbol{x}$ from $X_{S,S',\beta}$ for non-empty $S, S' \subseteq \{P_1, \ldots, P_n\}$ with $S' \neq S$ and $\beta \in \{0,1\}^\kappa$, observe that $S' \neq S$ implies either $S' \setminus S \neq \emptyset$ or $S \setminus S' \neq \emptyset$. In the first case, choose an arbitrary coordinate $P_\ell \in S' \setminus S$ and another $P_r \in S$. Then, for all $i \notin \{\ell, r\}$, sample x_i uniformly from $\{0,1\}^\kappa$, set $x_r = \beta \oplus \bigoplus_{i \in S \setminus \{P_r\}} x_i$, and sample x_ℓ uniformly from $\{0,1\}^\kappa \setminus \{\beta \oplus \bigoplus_{i \in S' \setminus \{P_\ell\}} x_i\}$. In the second case (i.e., when $S \setminus S' \neq \emptyset$), choose an arbitrary coordinate $P_r \in S \setminus S'$ and $P_\ell \in S'$, and then proceed in the same way as above.

Consider a static fail-stop 1-adversary $\mathcal{A}$ that uniformly corrupts one party at the start of the execution and behaves as follows. In all instances of $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t}$ and $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_2,t}$ invoked in π, the adversary $\mathcal{A}$ sets no delay in the input-provision phase but keeps delaying the output generation. Once all parties in S_1 have provided their inputs in any instance of $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t}$ and $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_2,t}$, the adversary $\mathcal{A}$ selects S_1 as the set of input providers and removes the delays set for the output generation. The adversary $\mathcal{A}$ continues with this strategy until the first instance of $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t}$ or $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_2,t}$ outputs 1. From now on, in any remaining instances of $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t}$ and $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_2,t}$, the adversary $\mathcal{A}$ delays the output generation until all parties in S_2 have provided their inputs. Then, select S_2 as the set of input providers and remove the delays set for the output generation.

Now we prove a lemma regarding the protocol π, adversary $\mathcal{A}$, and inputs $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ sampled as above. This lemma simplifies the subsequent parts of the proof.

Lemma 4.3. *To compute* $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_2,t}]_{\mathrm{weak}}$ *on inputs* $\boldsymbol{x}_1$ *and* $\boldsymbol{x}_2$ *sampled as described above, using the FBB protocol* π *for computing* $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C}_{\mathsf{pred}},t} \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C}_{\mathsf{pred}},t}]_{\mathrm{weak}}$ *in the* $(\mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C}_{\mathsf{pred}},t}, \mathcal{F}_{\mathsf{a\text{-}smt}})$*-hybrid model, the following properties hold with overwhelming probability for* π *against the above static fail-stop* 1*-adversary* $\mathcal{A}$*:*

1. *All queries on* t*-admissible inputs of* $\boldsymbol{x}_1$ *or* $\boldsymbol{x}_2$ *by* π^{f_1,f_2} *are made by instances of* $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t}$ *and* $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_2,t}$*.*
2. *For any* $i \in [2]$*, the* t*-admissible input of* $\boldsymbol{x}_i$ *used in the final output of* f_i *must be queried from* f_i*.*

Proof. We prove each part separately.

Part 1. For the sake of contradiction, assume that with noticeable probability, for $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ sampled as described above, some party in π^{f_1,f_2} *locally queries* on some t-admissible input of $\boldsymbol{x}_1$ or $\boldsymbol{x}_2$.

As the adversary $\mathcal{A}$ corrupts one party uniformly at random, the local query is made by a corrupt party with noticeable probability. This implies that $\mathcal{A}$ learns a t-admissible input with noticeable probability. In more detail, the adversary can uniformly select one of the queries made by the corrupted party and output it. Since, with noticeable probability, the corrupted party makes a query on a t-admissible input, and since the total number of queries is polynomial, the adversary outputs a t-admissible input with noticeable probability. We now show that this cannot be simulated.

Any t-admissible input contains inputs from at least $n-t$ parties. The simulator learns at most the input of the single corrupted party and potentially the final output on some t-admissible input through the ideal functionality. The final output reveals at most the XOR of the inputs in some t-admissible input. Therefore, the simulator must simulate at least $n - t > 2$ individual inputs, potentially knowing their XOR value and one of them. Since the inputs are uniformly sampled from X_{S_1,S_2,β_i} or X_{S_2,S_1,β_i}, the simulator can only guess the t-admissible input with negligible probability. This leads to a contradiction, which completes the proof of this part.

Part 2. For the sake of contradiction, assume that, with noticeable probability, for $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ sampled as described above, there exists some $i \in [2]$ such that the t-admissible input used in the output of f_i has never been queried from f_i.

By an averaging argument, there must exist a fixed $j \in [2]$ and specific values of $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ such that, with noticeable probability, the t-admissible input used in the output of f_j has never been queried from f_j.

From Part 1, we know that all queries on t-admissible inputs must be made by $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t}$ and $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_2,t}$. According to the description of $\mathcal{A}$, either S_1 or S_2 is selected as the input providers in all instances of $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t}$ and $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_2,t}$.

For the function $f_j \in \{f_1, f_2\}$ and inputs $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$, in each execution of π, let $\mathcal{Q}_j$ denote the set of all the sets of input providers used in the queries with t-admissible inputs of $\boldsymbol{x}_j$ from f_j. Again, by an averaging argument, there must exist at least one fixed set $\mathcal{Q}_j$ such that the event $\mathcal{E}_{\mathsf{inconsistent}}$, in which the following two conditions hold, happens with noticeable probability:

1. All queries with a t-admissible input of $\boldsymbol{x}_j$ from f_j use input providers from $\mathcal{Q}_j$, and
2. the final output for f_j relies on a set of input providers not in $\mathcal{Q}_j$.

Now consider an environment that always provides $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$, and define another function f'_j

as follows:

$$f'_j(\boldsymbol{u}) = \begin{cases} \neg f_j(\boldsymbol{u}) & \text{if } \boldsymbol{u} \text{ is a } t\text{-admissible input for } \boldsymbol{x}_j \text{ with respect to some set } S \notin \mathcal{Q}_j \\ f_j(\boldsymbol{u}) & \text{otherwise} \end{cases}$$

Next, consider a coupled experiment where only the function f_j is replaced by f'_j. Note that f'_j still belongs to $\mathcal{C}_{\mathsf{pred}}$ and behaves identically to f_j at all points except for the t-admissible inputs $\boldsymbol{x}_j$ that do not correspond to the input providers in $\mathcal{Q}_j$. Therefore, any randomness that causes event $\mathcal{E}_{\mathsf{inconsistent}}$ to occur in the main experiment will also cause $\mathcal{E}_{\mathsf{inconsistent}}$ to occur in the coupled experiment, as the behavior of f_j and f'_j is identical on all the queried points. Furthermore, in this scenario, both experiments must produce the same output. However, this leads to a contradiction since in event $\mathcal{E}_{\mathsf{inconsistent}}$ the protocol π produces output for a t-admissible input of $\boldsymbol{x}_j$ that is not in $\mathcal{Q}_j$, and for all such t-admissible inputs, f'_j returns the negation of f_j. Since event $\mathcal{E}_{\mathsf{inconsistent}}$ occurs with noticeable probability, this contradiction completes the proof of this part.

This concludes the proof of Lemma 4.3. □

In what follows, we prove that the described adversary $\mathcal{A}$ induces an output distribution that cannot be achieved in the ideal world and is, in fact, distinguishable by a simple polynomial-time environment. Roughly speaking, we show that the adversarial strategy results in two 0's in the output with probability close to $\frac{1}{2}$ while avoiding two 1's in the output with overwhelming probability–an outcome that is not achievable in the ideal world.

We proceed to define the following three events:

- $\mathcal{E}_{\mathsf{none}}$: No instance of $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t}$ or $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_2,t}$ outputs 1.
- $\mathcal{E}_{\mathsf{one}}$: Exactly one instance of $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t}$ or $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_2,t}$ outputs 1.
- $\mathcal{E}_{\mathsf{two}}$: More than one instance of $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t}$ or $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_2,t}$ outputs 1.

We relate these events to the possible outputs of the protocol π and analyze the probability of each. Our analysis relies on the following key observations: From Lemma 4.3, we know that, with overwhelming probability, under event $\mathcal{E}_{\mathsf{none}}$, the output of π contains no 1, under event $\mathcal{E}_{\mathsf{one}}$, the output contains at most one 1, and under event $\mathcal{E}_{\mathsf{two}}$, the output may contain two 1's.

Besides, recall that β_1 and β_2 are sampled uniformly and independently from $\{0,1\}^\kappa$. Therefore, for any $b \in [2]$ and any set of input providers $S \subseteq \{P_1, \ldots, P_n\}$, if $\bigoplus_{P_i \in S} x_{b,i} \neq \beta_b$, the probability that any instance of $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_b,t}$ outputs 1 with input providers S is negligible. Also note that the environment randomly chooses $j \in [2]$ and samples inputs $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ from X_{S_j,S_{3-j},β_1} and X_{S_j,S_{3-j},β_2}, respectively.

Therefore, when $S_1 \neq S_j$, with only negligible probability will any instance of $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t}$ or $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_2,t}$ output 1. On the other hand, when $S_1 = S_j$, then with overwhelming probability exactly one instance will output 1 (because after the first one, the adversary switches the set of input providers to S_2). This implies that $\left|\Pr[\mathcal{E}_{\mathsf{none}}] - \frac{1}{2}\right| \leq \mathsf{negl}(\kappa)$ and $\left|\Pr[\mathcal{E}_{\mathsf{one}}] - \frac{1}{2}\right| \leq \mathsf{negl}(\kappa)$. Consequently, the probability of event $\mathcal{E}_{\mathsf{two}}$ is at most $\mathsf{negl}(\kappa)$.

To summarize, we can say that the final output in the real world:

- has no 1 with probability $\frac{1}{2} \pm \mathsf{negl}(\kappa)$;
- has exactly one 1 with probability $\frac{1}{2} \pm \mathsf{negl}(\kappa)$;
- has two 1's with probability at most $\mathsf{negl}(\kappa)$.

Since the adversary in the real world only uses S_1 and S_2 as the sets of input providers, according to Lemma 4.3, with overwhelming probability the final output and hence the simulator must choose the set of input providers from the set $\{S_1, S_2\}$; otherwise, the two worlds become trivially distinguishable. Besides, in the ideal world, the simulator must choose both sets of input providers

before observing any output. Let $0 \leq p \leq 1$ denote the probability that the simulator selects the same set of input providers for both functions. Consequently, the probability of choosing different sets is $1 - p$. Given this, the output in the ideal world:

- has no 1 with probability $\frac{p}{2} \pm \mathsf{negl}(\kappa)$;
- has exactly one 1 with probability $1 - p \pm \mathsf{negl}(\kappa)$;
- has two 1's with probability $\frac{p}{2} \pm \mathsf{negl}(\kappa)$.

Now we show that the environment can effectively distinguish between the real and ideal worlds using the following strategy: if the output contains two 0's, output 0; if it contains two 1's, output 1; otherwise (i.e., if the output contains exactly one 1), choose uniformly at random and output 0 or 1, each with probability $\frac{1}{2}$.

The above environment outputs 0 with probability at least $\frac{3}{4}$ in the real world and $\frac{1}{2}$ in the ideal world, regardless of the choice of p, which implies that the environment can successfully distinguish between the real and ideal worlds, thereby contradicting the security of π. This concludes the proof of Theorem 4.2. □

4.2 Impossibility of Generic PBB Weak Parallel Composition

We proceed to rule out weak parallel composition of A-MPC protocols in a generic PBB manner in the presence of semi-malicious adversaries. Specifically, we focus on the same class of predicates $\mathcal{C}_{\mathsf{pred}}$ considered in Section 4.1, and prove that, assuming the existence of threshold leveled homomorphic encryption (LHE) schemes with computationally strong homomorphism (see [Hal17] for formal definitions), no PBB compiler can generically compute the weak parallel composition of predicates in $\mathcal{C}_{\mathsf{pred}}$.

Before presenting the formal statement and its proof, we highlight the main differences and challenges compared to the proof of Theorem 4.2 and how we address them. As opposed to the FBB setting, PBB compilers have access to the next-message function of some protocols that compute the functions. This allows them to generate arbitrary transcripts on any inputs, which may reveal the function's code. In fact, the transcripts of many A-MPC protocols, particularly those based on secret sharing, depend on the code of the function. If the compiler learns the function, it could then achieve parallel composition by constructing a protocol to compute the parallel function from scratch.

Even if protocols hide the function's description, the compiler still has access to the protocol's messages. This access may allow the compiler to coordinate different executions for various functions and prevent the adversary from selecting a core set in one execution based on the result of another execution, thus enabling some form of parallel composition. For example, all known A-MPC protocols have an *identifiable* point in their execution where the effective input (i.e., input providers and their inputs) is committed to, even though the output is not yet computed. By exploiting this structure, the compiler can pause the execution of each protocol right after this identifiable point and wait until all protocol executions reach this stage. This is the core idea behind our feasibility result in Section 5.2. Looking ahead, in Section 5, we show that existing protocols have a robust structure that even permits strong parallel composition. As a result, it is impossible to demonstrate the impossibility of weak parallel composition in a PBB manner using existing protocols.

4.2.1 A Contrived A-MPC protocol

To overcome this limitation, we present a simple A-MPC protocol for semi-malicious failures that conceals the function's description and lacks any identifiable point, when used in a PBB manner,

where the input providers are finalized but the output is not computed yet. This protocol has several essential properties: the function being computed is hidden within its transcript; the adversary has the ability to finalize the core-set without the compiled protocol detecting it until the output is generated; and it is ensured that critical messages must be generated by applying the next-message function to relevant histories.

High-level overview of the protocol. The starting point of the protocol is a vanilla version of the protocol from [Coh16] adapted to semi-malicious adversaries. Initially, each party encrypts its input using a threshold LHE scheme, exchanges the ciphertexts with others, forms a core set, homomorphically evaluates the function, and finally performs threshold decryption to recover the output.

A naïve implementation of this approach reveals a specific point in the protocol where inputs are effectively committed before the output has been generated. As discussed above and formalized in Section 5, this structural feature enables PBB parallel composition. To disrupt this structure, we modify the protocol to potentially iterate twice and introduce a backdoor mechanism that allows the adversary to choose both the iteration in which the function is evaluated and the corresponding set of inputs (and core set) to be used. This choice is made by the adversary through the injection of specific randomness along with its input. This is implemented by having each party provide a uniformly random value whenever it supplies its input, and by hard-coding a vector $\boldsymbol{s} = (s^1, s^2) \in (\{0,1\}^\kappa)^2$ into the protocol. In the first iteration, if the random value associated with some input provider matches s^1, the protocol generates the output and terminates. Otherwise, it proceeds to the second iteration. In the second iteration, if the random value associated with some input provider matches s^2, the protocol uses the set of input providers from that iteration, generates the output, and terminates. If no match occurs, it falls back to the set of input providers from the first iteration. Now, a semi-malicious adversary can control the iteration in which the function is evaluated and the corresponding set of inputs (and core set) to be used by strategically choosing the randomness it provides along with its input.

To ensure this choice indeed remains hidden until the output is revealed, we encrypt the adversarial randomness using a symmetric-key encryption scheme. Moreover, to prevent a PBB compiler from overriding the adversary's choice, we bind the randomness to the input by encrypting them together and authenticating the ciphertext using a MAC.

In addition, we leverage threshold LHE to homomorphically compute an encrypted version of the output—again using the same symmetric-key scheme. This design simplifies our argument: a PBB compiler, lacking access to the key, can only learn the output by generating the appropriate message in Step 6 via the next-message function.

We require strong homomorphism to ensure circuit privacy and hide the function being evaluated. This further ensures that other critical aspects of the protocol flow are not prematurely leaked to the compiler. In particular, depending on the adversarial backdoor message, the protocol in its first iteration either homomorphically evaluates the function or produces a fresh LHE encryption of the term '$\mathsf{iterate}$'; strong homomorphism hides this difference from the compiler. This is crucial, as we do not want it to be apparent whether the backdoor has been triggered until the final output is generated.

To avoid unnecessary technical complications in the impossibility proof, we explicitly associate the encryption of inputs in the core sets with protocol messages and authenticate the whole message using a MAC. This ensures that any message generated by a black-box compiler without the secret key is invalid, and that each output is uniquely tied to a well-defined set of inputs.

Finally, to make the symmetric-key encryption secure against PBB compilation, we not only

sample the keys uniformly at random but also ensure that fresh, unpredictable randomness is used for each encryption, which is beyond the control of the compiler. To achieve this, we generate the required randomness using a PRF applied to the current protocol transcript.

To conclude, the cryptographic building blocks used by the protocol are a family of pseudorandom functions (PRF) $\{F_k\}_{k\in\{0,1\}^\kappa}$, a symmetric-key encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$, a message authentication code (MAC) $(\mathsf{MAC.Gen}, \mathsf{MAC}, \mathsf{Ver})$, and a threshold LHE (TLHE) scheme with computationally strong homomorphism ($\mathsf{TLHE.Setup}$, $\mathsf{TLHE.Encrypt}$, $\mathsf{TLHE.Eval}$, $\mathsf{TLHE.PartDec}$, $\mathsf{TLHE.FinDec}$). (We refer to [Hal17] for the formal definition of a threshold LHE scheme with computationally strong homomorphism.) To enable the usage of the "symmetric cryptography" tools, we assume that the key space is $\{0,1\}^\kappa$ and parameterize the protocol with an additional triplet of keys $\boldsymbol{k} = (k_1, k_2, k_3) \leftarrow (\{0,1\}^\kappa)^3$: the first for the PRF, the second for the symmetric encryption, and the third for the MAC scheme.

The protocol $\pi_{\mathrm{TLHE}}^{t,\boldsymbol{k},\boldsymbol{s}}(f)$ is defined in a hybrid model equipped with several ideal functionalities: $\mathcal{F}_{\mathsf{a\text{-}cast}}$, $\mathcal{F}_{\mathsf{acs}}$, $\mathcal{F}_{\mathsf{a\text{-}smt}}$ (defined in Section 2), and $\mathcal{F}_{\mathsf{setup}}$, which is an A-SFE functionality $\mathcal{F}_{\mathsf{a\text{-}sfe}}$ for computing the setup for the threshold LHE scheme. That is, $\mathcal{F}_{\mathsf{setup}}$ evaluates $\mathsf{TLHE.Setup}(1^\kappa, 1^d, (t+1)\text{-}\mathsf{TAS})$, where d is an appropriate depth bound for the function f plus the depth of the symmetric-key encryption scheme Enc used in the protocol, and $(t+1)\text{-}\mathsf{TAS}$ denotes the threshold access structure with threshold $t+1$; the output for each party contains the public parameters, public key, and its share of the secret key. We note that $\mathcal{F}_{\mathsf{setup}}$ can be computed with any A-MPC protocol that allows private output and in the plain model, e.g., [BKR94].

Protocol $\pi_{\mathrm{TLHE}}^{t,\boldsymbol{k},\boldsymbol{s}}(f)$

For every $i \in [n]$, at any point during the execution, Trans_i denotes the transcript of P_i up to that moment. All variables are initialized to $\perp$, all sets to $\emptyset$, and the iteration counter as $\mathsf{itr} := 1$. The protocol is parameterized by a triplet of keys $\boldsymbol{k} = (k_1, k_2, k_3) \in (\{0,1\}^\kappa)^3$ and a pair of strings $\boldsymbol{s} = (s^1, s^2) \in (\{0,1\}^\kappa)^2$. Each party $P_i \in \mathcal{P}$ proceeds as follows:

1. Upon receiving $(\mathtt{input}, \mathsf{sid}, x_i)$ from the environment, send $(\mathtt{input}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{setup}}$.
2. Upon receiving $(\mathtt{output}, \mathsf{sid}, (pp, pk, sk_i))$ from $\mathcal{F}_{\mathsf{setup}}$ or redirection from Step 6, do the following:
   - If $(\mathtt{output}, \mathsf{sid}_{\mathsf{Setup}}, (pp, pk, sk_i))$ is received from $\mathcal{F}_{\mathsf{setup}}$, store (pp, pk, sk_i), sample $r_i^1 \leftarrow \{0,1\}^\kappa$ and compute // it is the first iteration, i.e., $\mathsf{itr} = 1$

$$c_i^1 \leftarrow \mathsf{TLHE.Encrypt}(pk, x_i) \quad \text{and} \quad u_i^1 := \mathsf{Enc}_{k_2}((r_i^1, c_i^1); F_{k_1}(\mathsf{Trans}_i))$$
$$\text{and} \quad t_i^{1,2} := \mathsf{MAC}_{k_3}(1, u_i^1).$$

     Next, send $(\mathtt{input}, \mathsf{sid}^1_{\mathsf{inA},i}, (1, \perp, u_i^1, t_i^{1,2}))$ to $\mathcal{F}_{\mathsf{a\text{-}cast}}$ with $\mathsf{sid}^1_{\mathsf{inA},i} = (\mathsf{sid}, 1, \mathrm{acast}, \mathrm{in}, i)$.
   - If redirected from Step 6, sample $r_i^2 \leftarrow \{0,1\}^\kappa$ and compute // it is the second iteration, i.e., $\mathsf{itr} = 2$, so the set S^1 and corresponding $\{u_j^1\}$ have been defined in Steps 3-4 of the first iteration

$$c_i^2 \leftarrow \mathsf{TLHE.Encrypt}(pk, x_i) \quad \text{and} \quad u_i^2 := \mathsf{Enc}_{k_2}((r_i^2, c_i^2); F_{k_1}(\mathsf{Trans}_i))$$
$$\text{and} \quad t_i^{2,2} := \mathsf{MAC}_{k_3}(2, (u_j^1)_{j\in S^1}, u_i^2).$$

     Next, send $(\mathtt{input}, \mathsf{sid}^2_{\mathsf{inA},i}, (2, (u_j^1)_{j\in S^1}, u_i^2, t_i^{2,2}))$ to $\mathcal{F}_{\mathsf{a\text{-}cast}}$ with $\mathsf{sid}^2_{\mathsf{inA},i} = (\mathsf{sid}, 2, \mathrm{acast}, \mathrm{in}, i)$.
3. Upon receiving $(\mathtt{output}, \mathsf{sid}^{\mathsf{itr}}_{\mathsf{inA},j}, (\mathsf{itr}, \boldsymbol{u}, u, t))$ from $\mathcal{F}_{\mathsf{a\text{-}cast}}$, do the following: // note, $\boldsymbol{u}$ could be $\perp$
   - If $\mathsf{itr} = 1$, verify $\mathsf{Ver}_{k_3}((1, u), t) = 1$ and $\boldsymbol{u} = \perp$,
   - If $\mathsf{itr} = 2$, verify $\mathsf{Ver}_{k_3}((2, \boldsymbol{u}, u), t) = 1$ and $\boldsymbol{u} = (u_j^1)_{j\in S^1}$, // note that for $\mathsf{itr} = 2$, the set S^1 and values $\{u_j^1\}$ have been defined in Steps 3-4 of $\mathsf{itr} = 1$

   set $(r_j^{\mathsf{itr}}, c_j^{\mathsf{itr}}) := \mathsf{Dec}_{k_2}(u_j^{\mathsf{itr}})$ and send $(\mathtt{input}, \mathsf{sid}^{\mathsf{itr}}_{\mathrm{acs}}, P_j)$ to $\mathcal{F}^{n,t}_{\mathsf{acs}}$ with $\mathsf{sid}^{\mathsf{itr}}_{\mathrm{acs}} = (\mathsf{sid}, \mathsf{itr}, \mathrm{acs})$.

4. Upon receiving $(\mathtt{output}, \mathsf{sid}^{\mathsf{itr}}_{\mathsf{acs}}, Y)$ from $\mathcal{F}^{n,t}_{\mathsf{acs}}$, wait until the output of the instance of $\mathcal{F}_{\mathsf{a\text{-}cast}}$ with $\mathsf{sid}^{\mathsf{itr}}_{\mathsf{inA},j}$ is received for all $j \in Y$. Then, set $S^{\mathsf{itr}} := Y$, assign $v^{\mathsf{itr}}_j := c^{\mathsf{itr}}_j$ for each $j \in S^{\mathsf{itr}}$, and $v^{\mathsf{itr}}_j := \mathsf{TLHE.Encrypt}(pk, \perp)$ for each $j \notin S^{\mathsf{itr}}$.
If there exists $P_j \in S^{\mathsf{itr}}$ such that $r^{\mathsf{itr}}_j = s^{\mathsf{itr}}$, set $\mathsf{backdoor\text{-}triggered}^{\mathsf{itr}} := 1$; otherwise, set $\mathsf{backdoor\text{-}triggered}^{\mathsf{itr}} := 0$. // the existential ranges over the core set S^{itr}, so that all honest parties compute the same bit Next, set $v \leftarrow \mathsf{TLHE.Encrypt}(pk, F_{k_1}(\mathsf{Trans}_i))$ and proceed as follows:
   - If $\mathsf{itr} = 1$, do the following:
     - If $\mathsf{backdoor\text{-}triggered}^1 = 1$, set $\mathcal{CS} := S^1$ and compute
     $$z^1_i := \mathsf{TLHE.Eval}(pk, \mathsf{Enc}_{k_2}(f(\cdot);\cdot), (v^1_1, \ldots, v^1_n, v)).$$
     // evaluate $\mathsf{Enc}_{k_2}(y; F_{k_1}(\mathsf{Trans}_i))$ where $y = f((x_j)_{j \in S^1})$
     - If $\mathsf{backdoor\text{-}triggered}^1 = 0$, compute
     $$v^1 = \mathsf{Enc}_{k_2}(\text{`iterate'}; F_{k_1}(\mathsf{Trans}_i)) \quad \text{and} \quad z^1_i \leftarrow \mathsf{TLHE.Encrypt}(pk, v^1)$$
     // encrypt the term 'iterate' to force a second iteration
     - Compute
     $$t^{1,4}_i := \mathsf{MAC}_{k_3}(1, (u^1_j)_{j \in S^1}, z^1_i)$$
     and send $(\mathtt{input}, \mathsf{sid}^1_{\mathsf{outA},i}, (1, (u^1_j)_{j \in S^1}, z^1_i, t^{1,4}_i))$ to $\mathcal{F}_{\mathsf{a\text{-}cast}}$ with $\mathsf{sid}^1_{\mathsf{outA},i} = (\mathsf{sid}, 1, \mathrm{acast}, \mathrm{out}, i)$.
   - If $\mathsf{itr} = 2$, do the following: // note that $\mathsf{itr} > 2$ never happens since in the second iteration we definitely get a correct evaluation of the function, triggering termination
     - If $\mathsf{backdoor\text{-}triggered}^2 = 1$, set $\mathcal{CS} := S^2$ and compute
     $$z^2_i := \mathsf{TLHE.Eval}(pk, \mathsf{Enc}_{k_2}(f(\cdot);\cdot), (v^2_1, \ldots, v^2_n, v)).$$
     // evaluate $\mathsf{Enc}_{k_2}(y; F_{k_1}(\mathsf{Trans}_i))$ where $y = f((x_j)_{j \in S^2})$
     - If $\mathsf{backdoor\text{-}triggered}^2 = 0$, set $\mathcal{CS} := S^1$ and compute
     $$z^2_i := \mathsf{TLHE.Eval}(pk, \mathsf{Enc}_{k_2}(f(\cdot);\cdot), (v^1_1, \ldots, v^1_n, v)).$$
     // evaluate $\mathsf{Enc}_{k_2}(y; F_{k_1}(\mathsf{Trans}_i))$ where $y = f((x_j)_{j \in S^1})$
     - Compute
     $$t^{2,4}_i := \mathsf{MAC}_{k_3}(2, (u^1_j)_{j \in S^1}, (u^2_j)_{j \in S^2}, z^2_i)$$
     and send $(\mathtt{input}, \mathsf{sid}^2_{\mathsf{outA},i}, (2, ((u^1_j)_{j \in S^1}, (u^2_j)_{j \in S^2}), z^2_i, t^{2,4}_i))$ to $\mathcal{F}_{\mathsf{a\text{-}cast}}$ with $\mathsf{sid}^2_{\mathsf{outA},i} = (\mathsf{sid}, 2, \mathrm{acast}, \mathrm{out}, i)$.

5. For any $j \in [n]$, upon receiving $(\mathtt{output}, \mathsf{sid}^{\mathsf{itr}}_{\mathsf{outA},j}, (\mathsf{itr}, \boldsymbol{u}, z, t))$ from $\mathcal{F}_{\mathsf{a\text{-}cast}}$,
   - if $\mathsf{itr} = 1$, and $\boldsymbol{u} = (u^1_j)_{j \in S^1}$, and $\mathsf{Ver}_{k_3}((1, \boldsymbol{u}, z), t) = 1$, compute
   $$w^1_{j,i} := \mathsf{TLHE.PartDec}(pk, z, sk_i) \quad \text{and} \quad t^{1,5}_i := \mathsf{MAC}_{k_3}(1, ((u^1_j)_{j \in S^1}), w^1_{j,i}),$$
   and send $(\mathtt{input}, \mathsf{sid}^1_{i,j}, (1, ((u^1_j)_{j \in S^1}), w^1_{j,i}, t^{1,5}_i))$ to $\mathcal{F}_{\mathsf{a\text{-}smt}}$ with $\mathsf{sid}^1_{i,j} = (\mathsf{sid}, 1, i, j)$.
   - if $\mathsf{itr} = 2$, and $\boldsymbol{u} = ((u^1_j)_{j \in S^1}, (u^2_j)_{j \in S^2})$, and $\mathsf{Ver}_{k_3}((2, \boldsymbol{u}, z), t) = 1$, compute
   $$w^2_{j,i} := \mathsf{TLHE.PartDec}(pk, z, sk_i) \quad \text{and} \quad t^{2,5}_i := \mathsf{MAC}_{k_3}(\mathsf{itr}, (u^1_j)_{j \in S^1}, (u^2_j)_{j \in S^2}, w^2_{j,i}),$$
   and send $(\mathtt{input}, \mathsf{sid}^2_{i,j}, (2, ((u^1_j)_{j \in S^1}, (u^2_j)_{j \in S^2}), w^2_{j,i}, t^{2,5}_i))$ to $\mathcal{F}_{\mathsf{a\text{-}smt}}$ with $\mathsf{sid}^2_{i,j} = (\mathsf{sid}, 2, i, j)$.

6. For any $j \in [n]$, upon receiving $(\mathtt{output}, \mathsf{sid}^{\mathsf{itr}}_{j,i}, (\mathsf{itr}, \boldsymbol{u}, w, t))$ from $\mathcal{F}_{\mathsf{a\text{-}smt}}$, if either
   - $\mathsf{itr} = 1$, $\boldsymbol{u} = (u^1_j)_{j \in S^1}$, and $\mathsf{Ver}_{k_3}((1, \boldsymbol{u}, w), t) = 1$, or
   - $\mathsf{itr} = 2$, $\boldsymbol{u} = ((u^1_j)_{j \in S^1}, (u^2_j)_{j \in S^2})$, and $\mathsf{Ver}_{k_3}((2, \boldsymbol{u}, w), t) = 1$,

set $d^{\mathsf{itr}}_{i,j} := w$ and set $Q^{\mathsf{itr}} := Q^{\mathsf{itr}} \cup \{j\}$. Once $|Q^{\mathsf{itr}}| \geq t+1$, compute

$$y^{\mathsf{itr}}_i := \mathsf{Dec}_{k_2}(\mathsf{TLHE.FinDec}(d^{\mathsf{itr}}_{i,j_1}, \ldots, d^{\mathsf{itr}}_{i,j_{t+1}})),$$

where $j_1, \ldots, j_{t+1}$ are $t+1$ distinct indices in Q^{itr} and do the following:
- If $y^{\mathsf{itr}}_i =$ 'iterate', set $\mathsf{itr} := \mathsf{itr} + 1$ and start over from Step 2.
- If $y^{\mathsf{itr}}_i \in \{0,1\}$, send $(\mathtt{input}, \mathsf{sid}_{\mathsf{termA},i}, y^{\mathsf{itr}}_i)$ to $\mathcal{F}_{\mathsf{a\text{-}cast}}$.

7. Upon receiving $(\mathtt{output}, \mathsf{sid}_{\mathsf{termA},j}, y)$ for a value $y \in \{0,1\}$ from at least $t+1$ indices $j \in [n]$, output $(\mathtt{output}, \mathsf{sid}, (\mathcal{CS}, y))$.

Figure 6: The counter-example protocol used in the proof of Theorem 4.9.

Theorem 4.4. *Let $t < n/2$ and let f be an n-ary computable predicate with public output, and assume that threshold LHE schemes with computationally strong homomorphism exist. Then, for uniformly distributed $\boldsymbol{k} \leftarrow (\{0,1\}^\kappa)^3$ and $\boldsymbol{s} \leftarrow (\{0,1\}^\kappa)^2$, protocol $\pi^{t,\boldsymbol{k},\boldsymbol{s}}_{TLHE}(f)$ UC-realizes $\mathcal{F}^{f,t}_{\mathsf{a\text{-}sfe}}$ in the $(\mathcal{F}_{\mathsf{setup}}, \mathcal{F}_{\mathsf{acs}}, \mathcal{F}_{\mathsf{a\text{-}cast}}, \mathcal{F}_{\mathsf{a\text{-}smt}})$-hybrid model against static semi-malicious t-adversaries.*

Proof. Let $\mathcal{A}$ be static semi-malicious t-adversary; we construct a simulator $\mathcal{S}$ such that no environment $\mathcal{Z}$ can distinguish whether it is interacting with $\pi^{t,\boldsymbol{k},\boldsymbol{s}}_{\mathrm{TLHE}}(f)$ and $\mathcal{A}$, or with $\mathcal{F}^{f,t}_{\mathsf{a\text{-}sfe}}$ and $\mathcal{S}$. We start with the description of the simulator and then prove its validity through a sequence of hybrid games. $\mathcal{S}$ simulates the view of $\mathcal{A}$, including its communication with $\mathcal{Z}$, messages sent by honest parties, and the interactions with the aiding hybrids.

Simulator $\mathcal{S}$

$\mathcal{S}$ sends messages received from $\mathcal{Z}$ directly to $\mathcal{A}$ and forwards messages $\mathcal{A}$ sends to $\mathcal{Z}$.

To simulate $\mathcal{F}_{\mathsf{setup}}$, the simulator $\mathcal{S}$ honestly generates the TLHE keys; i.e., $(pp, pk, sk_1, \ldots, sk_n)$. Then, for each corrupted party P_j, it sends (pp, pk, sk_j) to $\mathcal{A}$. Additionally, whenever an honest party P_i needs to encrypt its input, $\mathcal{S}$ encrypts 0 instead; i.e., $c^{\mathsf{itr}}_i \leftarrow \mathsf{TLHE.Encrypt}(pk, 0)$. Then, in each iteration, $\mathcal{S}$ simulates $\mathcal{F}_{\mathsf{a\text{-}cast}}$ in Steps 2 and 3 using the generated c^{itr}_i for each honest party P_i. The simulator $\mathcal{S}$ also honestly simulates $\mathcal{F}_{\mathsf{acs}}$ in Steps 3 and 4. In each iteration, the simulator can determine the bit $\mathsf{backdoor\text{-}triggered}^{\mathsf{itr}}$ by following the instructions of the protocol. Also note that the simulator can extract the input of a corrupted party P_j by obtaining the first available value u^1_j or u^2_j and using the known secret keys, and then can provide this input to $\mathcal{F}^{f,t}_{\mathsf{a\text{-}sfe}}$.

- If $\mathsf{itr} = 1$ and $\mathsf{backdoor\text{-}triggered}^1 = 0$, $\mathcal{S}$ follows the protocol description for the simulation; i.e., each honest party computes the LHE encryption of the symmetric-key encryption of the value 'iterate' in Step 4 and partially decrypts the ciphertext provided by the corrupted parties in Step 5 and sends it back to them. Finally, after recovering the value 'iterate' for the output, the process starts over from Step 2.
- Otherwise, $\mathcal{S}$ sets the core set in the ideal functionality $\mathcal{F}^{f,t}_{\mathsf{a\text{-}sfe}}$ as follows:
  - If $\mathsf{backdoor\text{-}triggered}^{\mathsf{itr}} = 1$, use the output of $\mathcal{F}_{\mathsf{acs}}$ in iteration itr, i.e., S^{itr} (the current iteration's set).
  - If $\mathsf{itr} = 2$ and $\mathsf{backdoor\text{-}triggered}^2 = 0$, use the output of $\mathcal{F}_{\mathsf{acs}}$ in iteration 1, i.e., S^1.

  Next, for each corrupted party P_j, the simulator receives the output y_j. Instead of computing $w^{\mathsf{itr}}_{j,i}$ per protocol instruction, for each corrupted party P_j, $\mathcal{S}$ uses P_j's output received from the ideal functionality y_j to generate $z^{\mathsf{itr}}_j := \mathsf{TLHE.Encrypt}(pk, y_j)$ and subsequently its partial decryption by each honest P_i as $w^{\mathsf{itr}}_{j,i} := \mathsf{TLHE.PartDec}(pk, z^{\mathsf{itr}}_j, sk_i)$. Finally, the simulator follows the termination procedure.

Figure 7: The simulator for $\pi^{t,\boldsymbol{k},\boldsymbol{s}}_{\mathrm{TLHE}}(f)$.

Towards the proof of indistinguishability of real- and ideal-world executions, we now define two series of hybrid games. The output of each game is the output of the environment.

The games $\text{HYB}^{\ell}_{\text{out}}$, for $0 \leq \ell \leq n$. The game interacts with the ideal functionality $\mathcal{F}^{f,t}_{\mathsf{a\text{-}sfe}}$ and executes the protocol $\pi^{t,\boldsymbol{k},\boldsymbol{s}}_{\text{TLHE}}(f)$ with the following modifications. When $\mathsf{itr} = 1$ and $\mathsf{backdoor\text{-}triggered}^1 = 0$ it follows the protocol; otherwise,

- when $\mathsf{backdoor\text{-}triggered}^{\mathsf{itr}} = 1$, the game sets the core set in the ideal functionality $\mathcal{F}^{f,t}_{\mathsf{a\text{-}sfe}}$ to S^{itr} (the output of $\mathcal{F}_{\mathsf{acs}}$ in itr).
- when $\mathsf{itr} = 2$ and $\mathsf{backdoor\text{-}triggered}^2 = 0$, the game sets the core set in the ideal functionality $\mathcal{F}^{f,t}_{\mathsf{a\text{-}sfe}}$ to S^1 (the output of $\mathcal{F}_{\mathsf{acs}}$ in $\mathsf{itr} = 1$).

Also, for each corrupted party P_j, it extracts the input by obtaining the first available value u^1_j or u^2_j and provides it to $\mathcal{F}^{f,t}_{\mathsf{a\text{-}sfe}}$. Then, for each $i > \ell$, if party P_i is corrupted, the game receives the output y_i from $\mathcal{F}^{f,t}_{\mathsf{a\text{-}sfe}}$ and instead of the encryption provided by party P_i in Step 4, it uses the encryption of y_i to create partial decryptions. Else, if party P_i is honest, it only sends y_i to $\mathcal{Z}$ instead of P_i's output generated by execution.

It is straightforward to observe that $\text{REAL}_{\pi^{t,\boldsymbol{k},\boldsymbol{s}}_{\text{TLHE}}(f),\mathcal{A},\mathcal{Z}} = \text{HYB}^{n}_{\text{out}}$.

Claim 4.5. *For every $0 \leq \ell \leq n-1$, it holds that $\text{HYB}^{\ell}_{\text{out}} \approx_c \text{HYB}^{\ell+1}_{\text{out}}$.*

Proof. The only difference between $\text{HYB}^{\ell}_{\text{out}}$ and $\text{HYB}^{\ell+1}_{\text{out}}$ is in the behavior of $P_{\ell+1}$ when $\mathsf{backdoor\text{-}triggered}^{\mathsf{itr}} = 1$ or $\mathsf{itr} = 2$ and $\mathsf{backdoor\text{-}triggered}^2 = 0$. Following the protocol description, in either case, with overwhelming probability in Step 4, any party, including $P_{\ell+1}$, evaluates f under TLHE on inputs from the set that the game uses as the core set for $\mathcal{F}^{f,t}_{\mathsf{a\text{-}sfe}}$. This is because all honest parties receive A-Cast at least from all honest parties in Step 3. Hence, there are always $t+1$ parties suggesting/inputting the same $n-t$ indices to $\mathcal{F}_{\mathsf{acs}}$ in Step 3. This, with overwhelming probability, enables $\mathcal{F}_{\mathsf{acs}}$ to output a set with size at least $n-t$ such that all its indices are suggested by at least $t+1$ parties. Since any $t+1$ parties contain at least one honest party, at least one honest party has received the A-Cast from all the indices in the output of $\mathcal{F}_{\mathsf{acs}}$. Then, according to $\mathcal{F}_{\mathsf{a\text{-}cast}}$ guarantees, all honest parties must eventually receive the A-Cast from all the indices in the output of $\mathcal{F}_{\mathsf{acs}}$ in Step 4. Thus, no honest party gets stuck in Step 4, and all will eventually continue with received A-Cast values for exactly the same set as the output of $\mathcal{F}_{\mathsf{acs}}$. Note that the output of $\mathcal{F}_{\mathsf{a\text{-}cast}}$ is the same for all honest parties, and if the sender is honest, it is equal to its input. Besides, the corrupted parties' input to $\mathcal{F}^{f,t}_{\mathsf{a\text{-}sfe}}$ is also extracted from the A-Casted value received in Step 3. This means for any party P_i, the computed encryption in Step 4 is for the same value y_i received from $\mathcal{F}^{f,t}_{\mathsf{a\text{-}sfe}}$. Since $\mathcal{F}^{f,t}_{\mathsf{a\text{-}sfe}}$ outputs the same value to all honest parties, party $P_{\ell+1}$ will eventually receive $t+1$ A-Cast for the same value as the output of $\mathcal{F}_{\mathsf{a\text{-}sfe}}$ in Step 7. Therefore, in the case of a corrupted party $P_{\ell+1}$, using the encryption of $y_{\ell+1}$ in Step 4 ensures that $\text{HYB}^{\ell}_{\text{out}}$ and $\text{HYB}^{\ell+1}_{\text{out}}$ are computationally indistinguishable, which relies on the computational strong homomorphism property of TLHE. In contrast, for an honest party $P_{\ell+1}$, simply outputting $y_{\ell+1}$ at the end does not rely on this property, yet still preserves the indistinguishability of the hybrids. □

The games $\text{HYB}^{\ell}_{\text{in}}$, for $0 \leq \ell \leq n$. The game is the same as $\text{HYB}^{0}_{\text{out}}$ with the following modification. For each $i > \ell$, the game ignores P_i's actual input x_i and instead uses 0; that is, it computes $c^{\mathsf{itr}}_i \leftarrow \mathsf{TLHE.Encrypt}(pk, 0)$. All other modifications and interactions with $\mathcal{F}^{f,t}_{\mathsf{a\text{-}sfe}}$ remain as in $\text{HYB}^{0}_{\text{out}}$.

Claim 4.6. *For every $0 \leq \ell \leq n-1$ it holds that $\text{HYB}^{\ell}_{\text{in}} \approx_c \text{HYB}^{\ell+1}_{\text{in}}$.*

Proof. This follows from the semantic security of the TLHE scheme, since the only difference between $\text{HYB}^{\ell}_{\text{in}}$ and $\text{HYB}^{\ell+1}_{\text{in}}$ is that $P_{\ell+1}$'s input is replaced with the value 0 under TLHE encryption and subsequent evaluation. Note also that these encryptions, and the evaluations derived from them, are never decrypted during the hybrids. In fact, the only ciphertexts ever decrypted are those corresponding to the output provided by the ideal functionality. □

Using the above two claims and since $\text{REAL}_{\pi^{t,\boldsymbol{k},\boldsymbol{s}}_{\text{TLHE}}(f),\mathcal{A},\mathcal{Z}} = \text{HYB}^{n}_{\text{out}}$, $\text{IDEAL}_{\mathcal{F}^{f,t}_{\text{a-sfe}},\mathcal{S},\mathcal{Z}} = \text{HYB}^{0}_{\text{in}}$, and $\text{HYB}^{0}_{\text{out}} = \text{HYB}^{n}_{\text{in}}$, we deduce that $\text{REAL}_{\pi^{t,\boldsymbol{k},\boldsymbol{s}}_{\text{TLHE}}(f),\mathcal{A},\mathcal{Z}} \approx_c \text{IDEAL}_{\mathcal{F}^{f,t}_{\text{a-sfe}},\mathcal{S},\mathcal{Z}}$. This concludes the proof of the theorem. □

Before stating our main theorem, we present a definition for the "valid history" of a party in an execution of a protocol, and a lemma about valid histories for the above protocol.

Definition 4.7 (Valid History)**.** *Let π be an n-party protocol with input space $\mathcal{X}$, randomness space $\mathcal{R}$, message space $\mathcal{M}$, and next-message function $f^{\pi}_{\text{nxt-msg}}(\cdot,\cdot,\cdot,\cdot)$. For each party P_i, let $x_i \in \mathcal{X}$ and $r_i \in \mathcal{R}$ be its input and randomness, respectively. We call h_i a* valid history *for P_i with input x_i and randomness r_i (or simply a valid history) if and only if $f^{\pi,i}_{\text{nxt-msg}}(i, x_i, r_i, h_i) \neq \mathsf{abort}$.*

Lemma 4.8. *For any n-ary computable predicate with public output f, and uniformly sampled and unknown values $\boldsymbol{k} \in (\{0,1\}^{\kappa})^3$ and $\boldsymbol{s} \in (\{0,1\}^{\kappa})^2$, all messages from Steps 2, 4, and 5 in any valid history h of $\pi^{t,\boldsymbol{k},\boldsymbol{s}}_{TLHE}(f)$ are, with overwhelming probability, generated by the next-message function $f^{\pi^{t,\boldsymbol{k},\boldsymbol{s}}_{TLHE}(f)}_{\text{nxt-msg}}$ on valid histories where:*

- *The sets S^1 and S^2, defined in Step 4 of the protocol, remain the same as in h.*
- *The values $(u^1_j)_{j\in S^1}$ and $(u^2_j)_{j\in S^2}$, received in Step 3 of the protocol, remain the same as in h.*

Proof. This directly follows from the security of the MAC scheme (as long as the MAC key is kept hidden), since all the mentioned messages contain $(u^1_j)_{j\in S^1}$ and $(u^2_j)_{j\in S^2}$ and are authenticated using a uniformly sampled unknown key. □

For any party P_i, the generation of the message for Step 6 (which includes y^{itr}_i) via the next-message function requires a valid history that contains sufficient messages from Steps 2, 4, and 5. Note that without knowing k_2, one cannot avoid using the next-message function for Step 6, except for negligible probability. Therefore, according to Lemma 4.8, each party P_i knows the state (including the random values r^1_i and r^2_i it provided in Step 2 of the corresponding valid history) used to generate any Step 6 message it produces. As a result, in the proof below, we may refer to expressions like "y^{itr}_i for a valid history with r^1_i and r^2_i" for clarity and brevity.

Note that in an honest execution, with overwhelming probability the protocol completes two iterations, taking inputs and forming a core set in each, and generates the output in the second iteration using the inputs from the first. However, a semi-malicious party P_c that knows $\boldsymbol{s} \in (\{0,1\}^{\kappa})^2$ can deviate by either forcing the protocol to generate an output in the first iteration (using inputs from that iteration) or by generating the output in the second iteration using inputs from the second iteration. These can be done by choosing $r^1_c = s^1$ and $r^2_c = s^2$, respectively. When the protocol is used in a PBB setting without knowledge of k_2, these adversarial strategies remain undetectable until the Step 6 message has been generated.

Using Lemma 4.8, we can associate any Step 6 message generated by the next-message function with the sets S^1 and S^2 (defined in Step 4), along with the values $(u^1_j)_{j\in S^1}$ and $(u^2_j)_{j\in S^2}$ received in Step 3 of the protocol. This information fully determines the t-admissible input used to generate the message in Step 6. In what follows, we refer to this input as the "effective t-admissible input for a Step 6 message." For brevity, we may also say "a Step 6 message for a certain t-admissible input."

4.2.2 The Impossibility Result

We now proceed to formally state our main impossibility result. We consider the case of computing $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C}_{\mathsf{pred}},t} \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C}_{\mathsf{pred}},t}]_{\text{weak}}$ with a PBB protocol. Recall that, by Definition 3.6, a PBB protocol for such a functionality is a protocol that, given black-box access to any pair of protocols UC-realizing $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C}_{\mathsf{pred}},t}$, composes them in parallel and itself UC-realizes the weak composition $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C}_{\mathsf{pred}},t} \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C}_{\mathsf{pred}},t}]_{\text{weak}}$.

Theorem 4.9. *Let $n, t \in \mathbb{N}$ with $1 < t < n/2$. Assuming the existence of threshold LHE schemes with computationally strong homomorphism, there is no PBB protocol for computing $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C}_{\mathsf{pred}},t} \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C}_{\mathsf{pred}},t}]_{\text{weak}}$ against static semi-malicious t-adversaries with computational security for the function class $\mathcal{C}_{\mathsf{pred}}$ defined earlier.*

The structure of the proof of this theorem is similar to that of Theorem 4.2, but considering PBB compilers introduces additional challenges. As previously explained, we address many of these challenges by introducing the simple A-MPC protocol π_{TLHE}.

Following a similar approach as in Theorem 4.2, we first establish that the compiled protocol indeed relies on the underlying protocols and incorporates their outputs into the final computation. Then, through a probabilistic attack, we demonstrate that the adversary can choose the set of input providers for one function, obtain the corresponding output, and subsequently determine the set of input providers for the other function. This again disrupts the input independence required for both strong and weak parallel composition.

Similarly to the FBB case, the attack here considers only asynchronous DPF. However, we require the compiler to support the broader class $\mathcal{C}_{\mathsf{pred}}$ to ensure that it relies on the underlying protocols and incorporates their outputs into the final computation (see Part 2 of Lemma 4.10). Also, as in the FBB case, the adversary does not crash any party, but as opposed to the FBB case, it may modify the parties' randomness. Nonetheless, security against crashes remains crucial to the proof, as without it, at each step, parties could wait to receive messages from all others before proceeding, effectively synchronizing the execution.

Proof. Towards a contradiction, assume that there exists a PBB protocol π that securely realizes $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C}_{\mathsf{pred}},t} \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C}_{\mathsf{pred}},t}]_{\text{weak}}$ against static semi-malicious t-adversaries with computational security.

In this proof, we use the same inputs, functions, and environment as in the proof of Theorem 4.2. However, for completeness, we briefly restate the necessary definitions here. We begin by defining several sets of inputs. For non-empty $S, S' \subseteq \{P_1, \ldots, P_n\}$ where $S \neq S'$ and $v \in \{0,1\}^\kappa$, let $X_{S,S',v}$ denote the set of all inputs $\boldsymbol{x} = (x_1, \ldots, x_n)$ satisfying $\bigoplus_{P_i \in S} x_i = v$ and $\bigoplus_{P_i \in S'} x_i \neq v$. Additionally, let $S_1 = \{P_1, \ldots, P_{n-t}\}$ and $S_2 = \{P_2, \ldots, P_{n-t+1}\}$.

For any $\beta \in \{0,1\}^\kappa$, define the n-ary A-DPF $f_\beta \in \mathcal{C}_{\mathsf{pred}}$ as follows:

$$f_\beta(\boldsymbol{u}) = \begin{cases} 0 & \text{if } \bigoplus_{u_i \neq \perp} u_i \neq \beta \\ 1 & \text{if } \bigoplus_{u_i \neq \perp} u_i = \beta \end{cases}$$

Let $\beta_1, \beta_2 \leftarrow \{0,1\}^\kappa$ and denote $f_1 = f_{\beta_1}$ and $f_2 = f_{\beta_2}$. We show that for a carefully designed environment and adversary, the environment can distinguish between the real and ideal worlds with noticeable probability. By the probabilistic method, this implies that there exist fixed values of β_1 and β_2 for which the same distinguishing advantage holds. We use the following notation to refer to the different inputs provided by each of the n parties to the two functions. For any $b \in [2]$ and $i \in [n]$, let $x_{b,i}$ denote the input of party P_i for function f_b, and define $\boldsymbol{x}_b = (x_{b,1}, \ldots, x_{b,n})$.

Consider an environment that activates parties in a round-robin fashion and selects the inputs $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ by choosing between the following two options with equal probability:

1. Sample $\boldsymbol{x}_1 \leftarrow X_{S_1,S_2,\beta_1}$ and $\boldsymbol{x}_2 \leftarrow X_{S_1,S_2,\beta_2}$.
2. Sample $\boldsymbol{x}_1 \leftarrow X_{S_2,S_1,\beta_1}$ and $\boldsymbol{x}_2 \leftarrow X_{S_2,S_1,\beta_2}$.

Note that, as argued in the proof of Theorem 4.2, the above sampling procedure is efficient.

Next, let $\boldsymbol{k}_1, \boldsymbol{k}_2 \leftarrow (\{0,1\}^\kappa)^3$, and $\boldsymbol{s}_1, \boldsymbol{s}_2 \leftarrow (\{0,1\}^\kappa)^2$ and consider $\pi_{\text{TLHE}}^{t,\boldsymbol{k}_1,\boldsymbol{s}_1}(f_1)$ and $\pi_{\text{TLHE}}^{t,\boldsymbol{k}_2,\boldsymbol{s}_2}(f_2)$ as the underlying protocols computing $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t}$ and $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_2,t}$, respectively. Here again, using the probabilistic method, one can deduce that there exist fixed values for $\boldsymbol{k}_1$, $\boldsymbol{k}_2$, $\boldsymbol{s}_1$, and $\boldsymbol{s}_2$ such that the following attack holds.

We now describe the adversary used in this proof. Consider a static semi-malicious 1-adversary $\mathcal{A}$ that arbitrarily corrupts a party P_c from the set $S_1 \cap S_2$ at the start of the execution and proceeds as follows: $\mathcal{A}$ uniformly samples $\alpha \in [2]$ and only delivers messages sent by parties in S_1 while delaying the rest.

Let $N(\kappa)$ denote the expected number of times P_c invokes the next-message function of $\pi_{\text{TLHE}}^{t,\boldsymbol{k}_\alpha,\boldsymbol{s}_\alpha}(f_\alpha)$ to provide its input $x_{(\alpha,c)}$ for the first iteration (i.e., in Step 2 when $\mathsf{itr} = 1$), given the honest execution, the specified delay pattern, and the chosen environment. Later, we argue that $N(\kappa)$ is a well-defined positive integer with overwhelming probability and is bounded by the running time of π.

At the j^{th} such invocation of the next-message function (for each $j \in [2N(\kappa)]$), with probability $\frac{1}{2N(\kappa)-j+1}$, $\mathcal{A}$ sets $r_{\alpha,c}^1 := s_\alpha^1$ (this constitutes the semi-malicious deviation) at the moment P_c provides its input (provided $\mathcal{A}$ has not already done so in a previous invocation).

This stopping rule is equivalent to uniformly selecting exactly one index from the range $[2N(\kappa)]$: intuitively, the chance of not stopping before position j telescopes to $\frac{2N(\kappa)-j+1}{2N(\kappa)}$, so when multiplied by the stopping probability $\frac{1}{2N(\kappa)-j+1}$ at step j, the result is $\frac{1}{2N(\kappa)}$ for every j. If the chosen invocation corresponds to a step in which P_c provides its input, then the party deviates by setting $r_c^1 := s_\alpha^1$. The adversary then observes the messages generated by P_c through the next-message function of $\pi_{\text{TLHE}}^{t,\boldsymbol{k}_\alpha,\boldsymbol{s}_\alpha}(f_\alpha)$ and proceeds as follows:

- If ever $y_c^1 = 1$, from that point onward, $\mathcal{A}$ delivers messages only from parties in S_2, delays all others, and sets $r_{3-\alpha,c}^2 := s_{3-\alpha}^2$ (this constitutes the semi-malicious deviation) each time P_c provides its input $x_{3-\alpha,c}$ to the next-message function of $\pi_{\text{TLHE}}^{t,\boldsymbol{k}_{3-\alpha},\boldsymbol{s}_{3-\alpha}}(f_{3-\alpha})$ (i.e., in Step 2).
- Otherwise, i.e., never $y_c^1 = 1$, it continues with the current strategy, which is only delivering messages from S_1.

Lemma 4.10. *To compute $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_2,t}]_{\text{weak}}$ on inputs $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ sampled as described above, using any PBB protocol π for computing $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C}_{\mathsf{pred}},t} \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{\mathcal{C}_{\mathsf{pred}},t}]_{\text{weak}}$ with the underlying protocols $\pi_{TLHE}^{t,\boldsymbol{k}_1,\boldsymbol{s}_1}(f_1)$ and $\pi_{TLHE}^{t,\boldsymbol{k}_2,\boldsymbol{s}_2}(f_2)$, the following properties hold with overwhelming probability for π against the above static semi-malicious 1-adversary $\mathcal{A}$:*

1. *For each $i \in [2]$, no party will invoke the next-message function of $\pi_{TLHE}^{t,\boldsymbol{k}_i,\boldsymbol{s}_i}(f_i)$ with another party's input.*
2. *For any $i \in [2]$, the t-admissible input of $\boldsymbol{x}_i$ used in the final output of f_i must have already served as the effective t-admissible input for a Step 6 message generated by the next-message function of $\pi_{TLHE}^{t,\boldsymbol{k}_i,\boldsymbol{s}_i}(f_i)$.*

Proof. We prove each part separately.

Part 1. For the sake of contradiction assume that, with noticeable probability, for some $i \in [2]$ and $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ sampled as described, some party in π invokes the next-message function of $\pi_{\text{TLHE}}^{t,\boldsymbol{k}_i,\boldsymbol{s}_i}(f_i)$ with the input of another party.

Now consider another adversary $\mathcal{A}'$ that behaves identically to $\mathcal{A}$ except that instead of choosing $P_c \in S_1 \cap S_2$, it selects it uniformly at random among all parties. $\mathcal{A}'$ is still a static semi-malicious t-adversary tolerated by the protocols. $\mathcal{A}$ and $\mathcal{A}'$ are indistinguishable to all honest parties, so the above event also holds for $\mathcal{A}'$ with noticeable probability. Since $\mathcal{A}'$ corrupts one party uniformly at random, the probability of the above event occurring and $\mathcal{A}'$ corrupting the party that invokes the next-message function of $\pi_{\mathrm{TLHE}}^{t,\boldsymbol{k}_i,\boldsymbol{s}_i}(f_i)$ on the input of another party remains noticeable. The adversary $\mathcal{A}'$ can uniformly select one of the invocations made by the corrupted party and output the input on whose behalf it was made. Since, with noticeable probability, the corrupted party invokes the next-message function on another (honest) party's input, and since the total number of invocations is polynomial, $\mathcal{A}'$ outputs an honest party's input with noticeable probability.

The simulator knows only the input of the corrupted party and may potentially learn the final output on some t-admissible input via the ideal functionality. Since inputs are uniformly sampled from either X_{S_1,S_2,β_i} or X_{S_2,S_1,β_i}, if the honest input observed by the adversary is not part of the t-admissible input used in the final output, the simulator has only a negligible chance of guessing the correct t-admissible input.

We now show that even when the honest input known to the adversary is included in the t-admissible input used in the final output, the simulator still cannot succeed with more than negligible probability. Indeed, the final output reveals at most the XOR of the inputs in that t-admissible set. Since $t > 1$ and $n > 2t$, every t-admissible input includes inputs from at least three parties. Knowing at most one of these values and the XOR of the entire set does not help the simulator reconstruct the remaining inputs with better than negligible probability, again due to the uniform sampling from X_{S_1,S_2,β_i} or X_{S_2,S_1,β_i}.

Part 2. Assume towards a contradiction that, with noticeable probability, there exists some $i \in [2]$ such that the t-admissible input of $\boldsymbol{x}_i$ used in the final output of f_i has not served as the effective t-admissible input for a Step 6 message generated by the next-message function of $\pi_{\mathrm{TLHE}}^{t,\boldsymbol{k}_i,\boldsymbol{s}_i}(f_i)$.

By an averaging argument, there must exist a fixed $j \in [2]$ and specific values of $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ where the above event occurs with noticeable probability.

We note that, prior to invoking the next-message function, no party P_r that does not know the PRF key k_1 and the encryption key k_2 can learn any nontrivial information about the Step 6 plaintext y_r^{itr}. The reason is a two-hybrid argument for the specific ciphertext that contributes to y_r^{itr}.

Hybrid H_1 (PRF security). Modify the execution only by replacing the coins that Step 4 derives from the transcript prefix (party P_r's transcript up to Step 4) with fresh, unpredictable coins of the same format. By PRF security, this change is computationally indistinguishable to any PPT observer who doesn't know k_1.

Hybrid H_2 (Semantic security). Starting from H_1, replace the Step 4 ciphertext itself by a dummy encryption produced exactly as in the protocol (same public parameters, same transcript prefix, same formatting/length), except that its plaintext is set to an arbitrary element $m^\star \in \{0, 1, \text{`}\mathsf{iterate}\text{'}\}$ (e.g., $m^\star = \text{`}\mathsf{iterate}\text{'}$). By semantic security (or equivalently IND-CPA) of the symmetric encryption, this substitution is computationally indistinguishable to any PPT observer who doesn't know k_2.

Hence the view before the next-message call is indistinguishable from one where the Step 6 contribution is simply an encryption of a fixed dummy string under an unknown key with fresh coins, so nothing about y_r^{itr} can be learned beyond negligible. In particular, computing y_r^{itr} without the next-message function has only negligible advantage.

Therefore, with an overwhelming probability, Step 6 messages must be generated using the

next-message function on a valid history with sufficient messages. Note that from Part 1, with overwhelming probability, each input in the t-admissible inputs used for any Step 6 message is provided by the corresponding party itself.

For the function $f_j \in \{f_1, f_2\}$ and inputs $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$, in each execution of π, let Q_j denote the set of all the sets of input providers in all the t-admissible inputs used to generate Step 6 messages for $\pi_{\text{TLHE}}^{t,\boldsymbol{k}_j,\boldsymbol{s}_j}(f_j)$. By an averaging argument, there must exist at least one fixed set Q_j such that the event $\mathcal{E}_{\mathsf{inconsistent}}$, in which the following two conditions hold, happens with noticeable probability:

1. All the generated Step 6 messages for $\pi_{\text{TLHE}}^{t,\boldsymbol{k}_j,\boldsymbol{s}_j}(f_j)$ use t-admissible inputs with input providers from Q_j, and
2. the final output for f_j relies on a set of input providers not in Q_j.

Now consider an environment that always provides $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$, and define another function f'_j as follows:

$$f'_j(\boldsymbol{u}) = \begin{cases} \neg f_j(\boldsymbol{u}) & \text{if } \boldsymbol{u} \text{ is an } t\text{-admissible input for } \boldsymbol{x}_j \text{ with respect to some set } S \notin Q_j \\ f_j(\boldsymbol{u}) & \text{otherwise} \end{cases}$$

Next, consider a coupled experiment where only the function f_j is replaced by f'_j. Note that f'_j still belongs to $\mathcal{C}_{\mathsf{pred}}$ and behaves identically to f_j at all points except for the t-admissible inputs $\boldsymbol{x}_j$ that do not correspond to the input providers in Q_j. Therefore, event $\mathcal{E}_{\mathsf{inconsistent}}$ occurs in the coupled experiment with probability at least $\Pr[\mathcal{E}_{\mathsf{inconsistent}}] - \mathsf{negl}(\kappa)$, which is still noticeable. This is because executions of π in both the main and the coupled experiments are computationally indistinguishable to all honest parties as long as Step 6 message for $\pi_{\text{TLHE}}^{t,\boldsymbol{k}_j,\boldsymbol{s}_j}(f_j)$ is generated with a t-admissible input of $\boldsymbol{x}_j$ using a set of input providers in Q_j. This indistinguishability follows from three key observations: first, the messages in Steps 1–3 are independent of the specific function; second, the messages in Step 4 do not leak information about the function due to strong homomorphism; and third, the messages in Step 5 consist of partial decryption shares of the symmetric-key encryption of an evaluation of f_j under the uniformly sampled, unknown key k_j^2, ensuring that the evaluation itself does not leak.

Additionally, conditioned on event $\mathcal{E}_{\mathsf{inconsistent}}$, both experiments produce the same output up to a negligible difference, because f_j and f'_j behave identically on all t-admissible inputs of $\boldsymbol{x}_j$ with input providers in Q_j. However, this leads to a contradiction since in event $\mathcal{E}_{\mathsf{inconsistent}}$, π produces output for a t-admissible input of $\boldsymbol{x}_j$ that is not in Q_j, and for all such t-admissible inputs, f'_j returns the negation of f_j. Given that event $\mathcal{E}_{\mathsf{inconsistent}}$ occurs with noticeable probability, this contradiction completes the proof of this second part of the lemma. □

In what follows, we prove that the described adversary $\mathcal{A}$ induces an output distribution that cannot be replicated in the ideal world and is distinguishable by a simple polynomial-time environment. Roughly speaking, we show that the adversary's strategy leads to an output containing two 0's with probability close to $\frac{1}{2}$, while simultaneously having an output with two 1's with probability noticeably less than $\frac{1}{2}$—an outcome that is not achievable in the ideal world.

Now we define three events that simplify the argument:

- $\mathcal{E}_{\mathsf{none}}$: No Step 6 messages contain value 1 for $\pi_{\text{TLHE}}^{t,\boldsymbol{k}_1,\boldsymbol{s}_1}(f_1)$ or $\pi_{\text{TLHE}}^{t,\boldsymbol{k}_2,\boldsymbol{s}_2}(f_2)$.
- $\mathcal{E}_{\mathsf{one}}$: Step 6 messages contain value 1 for exactly one of $\pi_{\text{TLHE}}^{t,\boldsymbol{k}_1,\boldsymbol{s}_1}(f_1)$ and $\pi_{\text{TLHE}}^{t,\boldsymbol{k}_2,\boldsymbol{s}_2}(f_2)$ (but not both).
- $\mathcal{E}_{\mathsf{two}}$: Step 6 messages contain value 1 at least once for both $\pi_{\text{TLHE}}^{t,\boldsymbol{k}_1,\boldsymbol{s}_1}(f_1)$ and $\pi_{\text{TLHE}}^{t,\boldsymbol{k}_2,\boldsymbol{s}_2}(f_2)$.

We now relate the events to the possible outputs of the protocol π and analyze the probability of each. Our analysis relies on the following key observations.

From Lemma 4.10, we know that, with overwhelming probability, under event $\mathcal{E}_{\mathsf{none}}$, the output of π contains no 1, under event $\mathcal{E}_{\mathsf{one}}$, the output contains at most one 1, and under event $\mathcal{E}_{\mathsf{two}}$, the output may contain two 1's.

Besides, recall that β_1 and β_2 are sampled uniformly and independently from $\{0,1\}^\kappa$. Therefore, for any $b \in [2]$, any party P_r, and any set of input providers $S \subseteq \{P_1, \ldots, P_n\}$, if $\bigoplus_{P_i \in S} x_{b,i} \neq \beta_b$, then the probability of $y_r^1 = 1$ in any history of $\pi_{\text{TLHE}}^{t,\boldsymbol{k}_b,\boldsymbol{s}_b}(f_b)$ with input providers S in the first iteration is negligible. Also note that the environment randomly chooses $j \in [2]$ and samples inputs $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ from X_{S_j,S_{3-j},β_1} and X_{S_j,S_{3-j},β_2}, respectively.

Therefore, when $S_1 \neq S_j$, with only negligible probability will any valid history of $\pi_{\text{TLHE}}^{t,\boldsymbol{k}_1,\boldsymbol{s}_1}(f_1)$ or $\pi_{\text{TLHE}}^{t,\boldsymbol{k}_2,\boldsymbol{s}_2}(f_2)$ have value 1 as its Step 6 message. This implies that $\left|\Pr[\mathcal{E}_{\mathsf{none}}] - \frac{1}{2}\right| \leq \mathsf{negl}(\kappa)$.

Now consider the case where $S_j = S_1$. Note that executions against $\mathcal{A}$ and honest executions where only messages from S_1 are delivered are indistinguishable to all honest parties until the history with $r_{\alpha,c}^1 = s_\alpha^1$ generates the message for Step 6. This is because the adversary's use of the backdoor in the first iteration is not detectable until the output is revealed. The only difference is that the protocol either homomorphically evaluates the function or produces a fresh encryption of the value '$\mathsf{iterate}$', and this distinction remains hidden due to the strong homomorphism property. In honest executions where only messages from S_1 are delivered, π must terminate; otherwise, π would fail to terminate against an adversary crashing $\mathcal{P} \setminus S_1$. According to Lemma 4.10, with overwhelming probability, π must generate Step 6 messages for both $\pi_{\text{TLHE}}^{t,\boldsymbol{k}_1,\boldsymbol{s}_1}(f_1)$ and $\pi_{\text{TLHE}}^{t,\boldsymbol{k}_2,\boldsymbol{s}_2}(f_2)$ on t-admissible input of $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$, respectively, before termination. Since only messages from S_1 are delivered, any t-admissible input corresponds to S_1, which contains P_c. It means P_c must provide its input at least once to the next-message function of both protocols. Thus, $N(\kappa)$ is a well-defined positive integer with overwhelming probability. Moreover, $N(\kappa)$ is bounded by the expected running time of π, which is a polynomial in κ.

Again, according to Lemma 4.10, with overwhelming probability, for each $i \in [2]$, the t-admissible input of $\boldsymbol{x}_i$ used in the final output of f_i must have already served as the effective t-admissible input for a Step 6 message generated by the next-message function of $\pi_{\text{TLHE}}^{t,\boldsymbol{k}_i,\boldsymbol{s}_i}(f_i)$. In addition, in honest executions of π with any delay strategy, with overwhelming probability, no Step 6 message is generated for the first iteration, and every Step 6 message generated for the second iteration must use the set of input providers from the first iteration. This follows from the fact that $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$ are sampled uniformly and independently from $(\{0,1\}^\kappa)^2$. Therefore, in honest executions of π with any delay strategy, with overwhelming probability $p(\kappa)$, for each $i \in [2]$, at least one Step 6 message for the second iteration must be generated on valid history that has a t-admissible input for the first iteration. Therefore, with probability $\frac{p(\kappa)}{2}$, $\pi_{\text{TLHE}}^{t,\boldsymbol{k}_\alpha,\boldsymbol{s}_\alpha}(f_\alpha)$ is the protocol for which P_c first generates a message for its second iteration using a valid history with a t-admissible input in the first iteration.

By Markov's inequality, in the honest execution where only messages from S_1 are delivered, the probability that the number of times P_c invokes the next-message function of $\pi_{\text{TLHE}}^{t,\boldsymbol{k}_\alpha,\boldsymbol{s}_\alpha}(f_\alpha)$ to provide its input $x_{(\alpha,c)}$ in the first iteration (i.e., when $\mathsf{itr} = 1$) does not exceed $2N(\kappa)$ is at least $\frac{1}{2}$. Moreover, for any $\ell \in [2N(\kappa)]$, the probability that $\mathcal{A}$ sets $r_{\alpha,c}^1 = s_\alpha^1$ the ℓ-th time P_c invokes the next-message function of $\pi_{\text{TLHE}}^{t,\boldsymbol{k}_\alpha,\boldsymbol{s}_\alpha}(f_\alpha)$ on its input $x_{(\alpha,c)}$ for the first iteration is $\frac{1}{2N(\kappa)}$. As long as $\mathcal{A}$ delivers messages only from S_1, any valid history with a t-admissible input must contain an input from P_c since it is in S_1. Therefore, with probability at least $\frac{p(\kappa)}{2 \cdot 2 \cdot 2N(\kappa)}$, P_c generates $y_c^1 = 1$ before generating any message for the second iteration of $\pi_{\text{TLHE}}^{t,\boldsymbol{k}_{3-\alpha},\boldsymbol{s}_{3-\alpha}}(f_{3-\alpha})$. In this case $\mathcal{A}$ learns $S_1 = S_j$, starts delivering messages from S_2, and, by setting $r_{3-\alpha,c}^2 = s_{3-\alpha}^2$ in all inputs for the second iteration of $\pi_{\text{TLHE}}^{t,\boldsymbol{k}_{3-\alpha},\boldsymbol{s}_{3-\alpha}}(f_{3-\alpha})$, enforces set of input providers S_2 for all Step 6 messages of

$\pi_{\text{TLHE}}^{t,\boldsymbol{k}_{3-\alpha},\boldsymbol{s}_{3-\alpha}}(f_{3-\alpha})$. Therefore, with probability $\frac{p(\kappa)}{2\cdot 2\cdot 2N(\kappa)}$, all Step 6 messages of $\pi_{\text{TLHE}}^{t,\boldsymbol{k}_{3-\alpha},\boldsymbol{s}_{3-\alpha}}(f_{3-\alpha})$ will be 0.

The above argument implies that when $S_j = S_1$, event $\mathcal{E}_{\mathsf{one}}$ occurs with noticeable probability, which consequently limits the probability of event $\mathcal{E}_{\mathsf{two}}$ to at most $\frac{1}{2} - \mathsf{notice}(\kappa)$.

Therefore, to summarize, we can say that the final output in the real world:

- Has no 1 with probability $\frac{1}{2} \pm \mathsf{negl}(\kappa)$.
- Has exactly one 1 with probability at least $\mathsf{notice}(\kappa)$.
- May have two 1's with probability $\frac{1}{2} - \mathsf{notice}(\kappa)$.

Since the adversary in the real world only uses S_1 and S_2 as the sets of input providers, according to Lemma 4.10, with overwhelming probability the final output and hence the simulator must use the set of input providers from either S_1 or S_2; otherwise, the two worlds become trivially distinguishable. Besides, in the ideal world, the simulator must choose both sets of input providers before observing any output. Let $0 \leq q \leq 1$ denote the probability that the simulator selects the same set of input providers for both functions. Consequently, the probability of choosing different sets is $1 - q$. Given this, the output in the ideal world:

- Has no 1 with probability $\frac{q}{2} \pm \mathsf{negl}(\kappa)$.
- Has exactly one 1 with probability $1 - q \pm \mathsf{negl}(\kappa)$.
- Has two 1's with probability $\frac{q}{2} \pm \mathsf{negl}(\kappa)$.

Now we show that the environment can effectively distinguish between the real and ideal worlds using the following strategy: if the output contains two 0's, output 0; if it contains two 1's, output 1; otherwise (i.e., if the output contains exactly one 1), choose uniformly at random and output 0 or 1, each with probability $\frac{1}{2}$.

The above environment outputs 0 with probability at least $\frac{1}{2} + \mathsf{notice}(\kappa)$ in the real world and $\frac{1}{2}$ in the ideal world, regardless of the choice of q. This implies that the environment can successfully distinguish between the real and ideal worlds, thereby contradicting the security of π. This concludes the proof of Theorem 4.9. □

From Proposition 3.3, we know that the PBB impossibility result in Theorem 4.9 directly extends to the FBB setting. However, this does not render the FBB impossibility result in Theorem 4.2 redundant. In fact, these two theorems are incomparable, as Theorem 4.2 does not rely on the existence of any cryptographic assumptions and applies even to fail-stop adversaries.

5 Feasibility of *Conditional* PBB Parallel Composition

In this section, we investigate the feasibility of PBB composition for different notions of parallel composition and protocol classes. We begin in Section 5.1 with the results for strong parallel composition and then, in Section 5.2, demonstrate how relaxing to the weak notion allows for a broader class of protocols.

5.1 Feasibility of Conditional PBB Strong Parallel Composition

We begin by defining the class of protocols that can achieve strong parallel composition with conditional PBB compilers.

Definition 5.1 (ACS-Decomposable Protocol Class)**.** *Let $n, t \in \mathbb{N}$ where $t < n$. A protocol class Π is said to be (n,t)-ACS-decomposable if all protocols in Π have n participants, and every protocol $\pi \in \Pi$ manifests the following properties:*

1. *The protocol π UC-realizes $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f,t}$ against adaptive malicious t-adversaries with perfect security, where f is an n-ary function.*

2. *The protocol π makes an identifiable subroutine call to a subprotocol π_{acs} (i.e., π is in the π_{acs}-hybrid model) such that protocol $\hat{\pi}$, which is derived from π by replacing the call to π_{acs} with an invocation of $\mathcal{F}_{\mathsf{acs}}^{n,t}$, UC-realizes the same $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f,t}$ with the same level of security.*

   This UC-realization is achieved using a simulator $\mathcal{S}$ that given any environment $\mathcal{Z}$ with input $z \in \{0,1\}^*$ chooses the core set to be the output Y of the invocation of $\mathcal{F}_{\mathsf{acs}}^{n,t}$ in the simulated execution of $\hat{\pi}$ with respect to $\mathcal{Z}$ and z.

The above structure is essential for our compiler, but its applicability can be expanded by allowing more flexibility. Specifically, we can define a class of protocols, each having an equivalent protocol with the required structure, where equivalence is defined by bi-directional UC-emulation.

Also, the second property dictates that the simulator directly assigns the set of input providers to the output of $\mathcal{F}_{\mathsf{acs}}^{n,t}$. While this property is broad enough to encompass all existing A-MPC protocols, we can further enhance its generality by considering simulators that determine the set of input providers through a mapping applied to the output of $\mathcal{F}_{\mathsf{acs}}^{n,t}$. This refined version is less limiting but still facilitates strong parallel composition using our compiler.

Note that (n,t)-ACS-decomposable classes are defined with perfect security against adaptive malicious adversaries. One can relax this definition to include other natural security notions such as statistical or computational security against static and/or passive adversaries.

The conditional PBB compiler. Next, we present our PBB compiler $\mathsf{Comp}_{\mathrm{strong}}$ that achieves strong parallel composition for (n,t)-ACS-decomposable protocols. The core idea is to aggregate all inputs to the ACS-related parts of the underlying protocols, replace these ACS-related parts with a single instance of the ACS functionality, provide the aggregated ACS inputs to this instance, and use its output as the ACS-related output for each underlying protocol. By aggregating, we mean that a party includes another party in its ACS input only if that party is included in its ACS input across all underlying protocols. This ensures a unified set of input providers across all protocols. Implementing this approach in the asynchronous setting presents additional challenges. For instance, we must prove that if the ACS inputs in all protocols satisfy the convergence precondition (see Section 2.4), then the aggregated inputs from all protocols also meet it.

Compiler $\mathsf{Comp}_{\mathrm{strong}}(\pi_1, \ldots, \pi_M)$

The compiler is given oracle access to the next-message functions $f_{\text{nxt-msg}}^{\pi_1}, \ldots, f_{\text{nxt-msg}}^{\pi_M}$ of protocols $\pi_1 \ldots, \pi_M$ in an (n,t)-ACS-decomposable class. Let $\mathsf{sid}_{\mathrm{acs}}^j$ be the SID for the identifiable subroutine call to π_{acs}^j in π_j for each $j \in [M]$. At the first activation, verify that $\mathsf{sid} = (\mathcal{P}, \mathsf{sid}')$ where $\mathcal{P}$ is the participant set.

Party $P_i \in \mathcal{P}$ proceeds as follows: For each $j \in [M]$, initialize a set $\mathcal{X}^j := \emptyset$ and an output value $y_i^j := \perp$, and define $\mathsf{sid}^j := (\mathsf{sid}, j)$. Also, initialize the "ready" set $\mathcal{R} := \emptyset$.

- Upon activation with input $(\mathtt{input}, \mathsf{sid}, \boldsymbol{x})$ where $\boldsymbol{x} = (x^1, \ldots, x^M)$, pass $(\mathtt{input}, \mathsf{sid}^j, x^j)$ to protocol π_j, for every $j \in [M]$.
- Upon activation with input $(\mathtt{fetch}, \mathsf{sid})$, do:

  1. For each $j \in [M]$, send the message generated by protocol π_j, except in the following cases:
     - For a message $(\mathtt{input}, \mathsf{sid}_{\mathrm{acs}}^j, x)$, update $\mathcal{X}^j := \mathcal{X}^j \cup \{x\}$.
       If $x \in \mathcal{X}^k$ for every $k \in [M]$, then send $(\mathtt{input}, \mathsf{sid}_{\mathrm{acs}}, x)$ to $\mathcal{F}_{\mathsf{acs}}^{n,t}$ with SID $\mathsf{sid}_{\mathrm{acs}} := (\mathsf{sid}, \mathtt{acs})$.
     - For a message $(\mathtt{fetch}, \mathsf{sid}_{\mathrm{acs}}^j)$, send $(\mathtt{fetch}, \mathsf{sid}_{\mathrm{acs}})$ to $\mathcal{F}_{\mathsf{acs}}^{n,t}$.
       Upon receiving back $(\mathtt{output}, \mathsf{sid}_{\mathrm{acs}}, \mathcal{Y})$, pass $(\mathtt{output}, \mathsf{sid}_{\mathrm{acs}}^j, \mathcal{Y})$ to π_j.

- For a message $(\mathtt{output}, \mathsf{sid}^j, (\mathcal{Y}, y^j))$, record $y_i^j := y^j$.

2. Wait until $y_i^j \neq \bot$ for every $j \in [M]$.
   Then, send $(\mathtt{input}, \mathsf{sid}_{\mathrm{acast},i}, \mathtt{ready})$ to $\mathcal{F}_{\mathsf{a\text{-}cast}}$ with SID $\mathsf{sid}_{\mathrm{acast},i} := (\mathcal{P}, P_i, \mathsf{sid}', \mathtt{a\text{-}cast})$.
3. For each $k \in [n]$, send $(\mathtt{fetch}, \mathsf{sid}_{\mathrm{acast},k})$ to $\mathcal{F}_{\mathsf{a\text{-}cast}}$.
   Upon receiving back $(\mathtt{output}, \mathsf{sid}_{\mathrm{acast},k}, \mathtt{ready})$, update $\mathcal{R} := \mathcal{R} \cup \{P_k\}$.
4. Wait until $y_i^j \neq \bot$ for every $j \in [M]$ and $|\mathcal{R}| \geq n - t$.
   Then, let $\boldsymbol{y}_i := (y_i^1, \ldots, y_i^M)$ and $\mathcal{CS} := \mathcal{Y}$, and output $(\mathtt{output}, \mathsf{sid}, (\mathcal{CS}, \boldsymbol{y}_i))$.

Figure 8: Conditional PBB compiler for strong parallel composition.

Before stating the security of $\mathsf{Comp}_{\mathrm{strong}}$, we start with the high-level ideas of the simulation. We consider the dummy adversary $\mathcal{D}$.[9] As the protocols π_j are (n,t)-ACS-decomposable, for every π_j there exists another protocol $\hat{\pi}_j$ in the ACS-hybrid model that realizes the same functionality. At a high level, the compiler essentially removes calls to the ACS protocols in each π_j and replaces them with calls to an ACS functionality. This modification makes each underlying protocol in the compiler resemble $\hat{\pi}_j$. For each $\hat{\pi}_j$, the compiler's simulator, $\mathcal{S}_{\mathrm{Comp}}$, utilizes the simulator guaranteed to exist for $\hat{\pi}_j$ (as per Definition 5.1) to simulate the modified protocol. Recall that $\hat{\pi}_j$ UC-realizes $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_j,t}$ against adaptive malicious t-adversaries (including $\mathcal{D}$) with a simulator that selects the set of input providers as the output of the single $\mathcal{F}_{\mathsf{acs}}^{n,t}$ invocation. Let $\mathcal{S}_j$ denote such a simulator for $\mathcal{D}$.

This approach works because all protocols use the same instance of $\mathcal{F}_{\mathsf{acs}}^{n,t}$. In fact, in this situation, the second property in Definition 5.1 implies that for all $j \in [M]$, the simulator $\mathcal{S}_j$ will fix the same set of input providers in its ideal functionality, which allows our simulator $\mathcal{S}_{\mathrm{strong}}$ to use a single set of input providers for all functions in the parallel ideal functionality. There are some details about how $\mathcal{S}_{\mathrm{strong}}$ needs to aggregate messages from different simulators and how it needs to split messages from the ideal functionality among these simulators that we now turn to. The details of this approach, along with the sequence of hybrid experiments used to establish its validity, are provided in the formal proof below.

Theorem 5.2. *Let $M, n, t \in \mathbb{N}$, let Π be a non-empty (n,t)-ACS-decomposable class, and let $f_1, \ldots, f_M$ be n-ary computable functions with respective protocols $\pi_1, \ldots, \pi_M \in \Pi$ realizing them against adaptive malicious t-adversaries with perfect security.*

The compiled protocol $\pi_{\mathsf{strong}} := \mathsf{Comp}_{\mathrm{strong}}(\pi_1, \ldots, \pi_M)$ UC-realizes $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\mathrm{strong}}$ in the $(\mathcal{F}_{\mathsf{acs}}^{n,t}, \mathcal{F}_{\mathsf{a\text{-}cast}}^{n,t}, \mathcal{F}_{\mathsf{a\text{-}smt}})$-hybrid model, against adaptive malicious t-adversaries with perfect security.

Proof. We consider the equivalent notion of UC-realization against the dummy adversary $\mathcal{D}$. That is, we construct a simulator $\mathcal{S}_{\mathrm{strong}}$ such that no environment $\mathcal{Z}$ can distinguish whether it is interacting with π_{strong} and $\mathcal{D}$, or with $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\mathrm{strong}}$ and $\mathcal{S}_{\mathrm{strong}}$.

As previously mentioned, to simulate π_j for each $j \in [M]$, the simulator $\mathcal{S}_{\mathrm{strong}}$ uses the simulator $\mathcal{S}_j$ for $\hat{\pi}_j$ and $\mathcal{D}$ that is guaranteed to exist by Definition 5.1. Below is a detailed description of $\mathcal{S}_{\mathrm{strong}}$. For clarity, we briefly recall the meaning of the counters and flags used in the simulator:

- For each party P_i and each instance $j \in [M]$:
  - $D_i^{\text{in-part},j}$ is the *input-participation delay counter* for instance j; that is, how many delay steps remain before P_i's input is treated as provided to $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_j,t}$.

[9] Recall that security against the dummy adversary, who passes every message to the environment and follows its instructions, implies security against arbitrary adversaries, see [Can20].

- $D_i^{\text{out-part},j}$ is the *output-participation delay counter* for instance j; that is, how long until P_i becomes eligible to receive output from $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_j,t}$.
- $D_i^{\text{out-rel},j}$ is the *output-release delay counter* for instance j; when it hits 0, P_i's output in instance j is actually released (assuming eligibility).
- $\mathsf{ready}_i^{\text{in},j}$ is the Boolean flag set once P_i has finished all required input steps in instance j.
- $\mathsf{part}_i^{\text{in},j}$ is the flag set once P_i is counted as having participated in the input phase of instance j.
- $\mathsf{part}_i^{\text{out},j}$ is the flag set once P_i is counted as having participated in the output phase of instance j.

- We also use global (aggregated) counters for each party P_i:
  - $D_i^{\text{in-part}}$ is the *aggregated input-participation delay* over all M instances, used by the joint functionality $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \cdots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{strong}}$.
  - $D_i^{\text{out-part}}$ and $D_i^{\text{out-rel}}$ are analogous global counters for output participation and output release, respectively, in the joint functionality.
- For each P_i, the following counters are for the ACS and A-Cast functionalities:
  - $D_i^{\text{in,acs}}$ is the ACS *input* delay counter; how many delay steps remain before the unified $\mathcal{F}_{\mathsf{acs}}^{n,t}$ accepts P_i's (compiled) input.
  - $D_i^{\text{out,acs}}$ is the ACS *output* delay counter; how many delay steps remain before P_i can receive the ACS output from the unified $\mathcal{F}_{\mathsf{acs}}^{n,t}$.
  - $D_i^{\text{out,acast},k}$ is the A-Cast readiness delay for messages from P_k; how many delay steps remain before P_i receives P_k's `ready` via $\mathcal{F}_{\mathsf{a\text{-}cast}}$.
- Finally, $\boldsymbol{y}_i$ denotes the vector of outputs $(y_i^1, \ldots, y_i^M)$ gathered across all instances for party P_i.

Simulator $\mathcal{S}_{\text{strong}}$

For each $j \in [M]$, to simulate the execution segment pertaining to π_j, the simulator $\mathcal{S}_{\text{strong}}$ invokes $\mathcal{S}_j$ and utilizes its responses to interact with $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{strong}}$ and $\mathcal{Z}$. The following parts detail how the simulator plays the role of the environment and the ideal functionality $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_j,t}$ for each $\mathcal{S}_j$, and outlines how messages from each simulator instance are used to engage with $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{strong}}$ and $\mathcal{Z}$.

- Initialization: For every $i \in [n]$,
  - Initialize $D_i^{\text{out-part}}$, $D_i^{\text{out-rel}}$, $D_i^{\text{in,acs}}$, and $D_i^{\text{out,acs}}$ to 1.
  - Initialize $\boldsymbol{y}_i$ to $\perp$.
  - Set $D_i^{\text{in-part}} := \sum_{j=1}^{M} D_i^{\text{in-part},j}$ in $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{strong}}$.
  - For every $k \in [n]$, initialize $D_i^{\text{out,acast},k}$ to 1.
  - For every $j \in [M]$,
    - Initialize $D_i^{\text{in-part},j}$, $D_i^{\text{out-part},j}$, and $D_i^{\text{out-rel},j}$ to 1.
    - Initialize $\mathsf{ready}_i^{\text{in},j}$, $\mathsf{part}_i^{\text{in},j}$, and $\mathsf{part}_i^{\text{out},j}$ to 0.
- Handling corruption requests:
  - Upon a direct corruption request from $\mathcal{Z}$ for some party P_i, send the request to $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{strong}}$ and to every $\mathcal{S}_j$ ($j \in [M]$). Also corrupt the relevant party in all simulated $\mathcal{F}_{\mathsf{a\text{-}cast}}$'s and $\mathcal{F}_{\mathsf{acs}}$ functionalities. Use the leakage from $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{strong}}$ to accordingly leak to $\mathcal{S}_j$ for all $j \in [M]$ and from all simulated $\mathcal{F}_{\mathsf{a\text{-}cast}}$'s and $\mathcal{F}_{\mathsf{acs}}$.
  - For all $j \in [M]$, send corruption requests from $\mathcal{S}_j$ to $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{strong}}$ and relay back relevant leakages.
- Communication with $\mathcal{Z}$ (includes simulating $\mathcal{F}_{\mathsf{acs}}^{n,t}$ and $\mathcal{F}_{\mathsf{a\text{-}cast}}$ towards $\mathcal{D}$):

- For each $j \in [M]$, route all messages from $\mathcal{Z}$ that target the segment associated with π_j (excluding messages with SID $\mathsf{sid}_{\text{acs}}$ targeting $\mathcal{F}_{\mathsf{acs}}^{n,t}$, which are treated differently as described below) directly to $\mathcal{S}_j$, ensuring they are treated as if coming directly from the environment.
- For each $j \in [M]$, forward messages from $\mathcal{S}_j$ to the environment directly to $\mathcal{Z}$, excluding messages that contain the SID $\mathsf{sid}_{\text{acs}}^j$, which are treated differently as described below.
- Upon receiving $(\texttt{delay}, \mathsf{sid}_{\text{acs}}, P_i, \mathsf{type}, D)$ from $\mathcal{Z}$, do the following:
  1. If $P_i \in \mathcal{P}$, $\mathsf{type} \in \{\text{in}, \text{out}\}$, and $D \in \mathbb{Z}$ (represented in unary notation), update
  $$D_i^{\mathsf{type},\text{acs}} := \max(1, D_i^{\mathsf{type},\text{acs}} + D).$$
  2. Send $(\texttt{delay}, \mathsf{sid}_{\text{acs}}^j, P_i, \mathsf{type}, D)$ to $\mathcal{S}_j$ for all $j \in [M]$.
- Upon receiving $(\texttt{leakage}, \mathsf{sid}_{\text{acs}}^j, P_i, x)$ from $\mathcal{S}_j$ for all $j \in [M]$, send $(\texttt{leakage}, \mathsf{sid}_{\text{acs}}, P_i, x)$ to $\mathcal{Z}$.
- Upon receiving $(\texttt{adv-input}, \mathsf{sid}_{\text{acs}}, X)$ from $\mathcal{Z}$, send $(\texttt{adv-input}, \mathsf{sid}_{\text{acs}}^j, X)$ to $\mathcal{S}_j$ for all $j \in [M]$.
- Upon receiving $(\texttt{fetched}, \mathsf{sid}_{\text{acs}}^j, P_i)$ from $\mathcal{S}_j$ for any $j \in [M]$, do the following:
  1. For each $\mathsf{type} \in \{\text{in}, \text{out}\}$, update the global $D_i^{\mathsf{type},\text{acs}}$ according to the description of $\mathcal{F}_{\mathsf{acs}}^{n,t}$ (ignore if already 0).
  2. Notify other simulators of this ACS activation by sending a $(\texttt{fetched}, \mathsf{sid}_{\text{acs}}^k, P_i)$ to each $\mathcal{S}_k$ for $k \in [M] \setminus \{j\}$ so they also update their local delay counters.
  3. Send $(\texttt{fetched}, \mathsf{sid}_{\text{acs}}, P_i)$ to $\mathcal{Z}$.
- Upon receiving $(\texttt{output}, \mathsf{sid}_{\text{acs}}^j, \mathcal{Y})$ (directed towards a corrupted party) from $\mathcal{S}_j$ for some $j \in [M]$, send $(\texttt{output}, \mathsf{sid}_{\text{acs}}, \mathcal{Y})$ to $\mathcal{Z}$ if no such message was sent before.
- When each simulated $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_j,t}$ is ready to release the output to some party P_i according to its tracked delays, send $(\texttt{leakage}, \mathsf{sid}_{\text{acast},i}, \texttt{ready})$ to $\mathcal{Z}$.
- For each $k \in [n]$, upon receiving $(\texttt{delay}, \mathsf{sid}_{\text{acast},k}, \cdot)$ and $(\texttt{replace}, \mathsf{sid}_{\text{acast},k}, \cdot)$, act as instructed by $\mathcal{F}_{\mathsf{a\text{-}cast}}$.

- Influencing $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{strong}}$ and for each $j \in [M]$, playing the role of $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_j,t}$ for $\mathcal{S}_j$:
  - Upon receiving $(\texttt{leakage}, \mathsf{sid}, P_i, \boldsymbol{l})$ from $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{strong}}$ where $\boldsymbol{l} = (l^1, \ldots, l^M)$, send $(\texttt{leakage}, \mathsf{sid}^j, P_i, l^j)$ to $\mathcal{S}_j$, where $j \in [M]$ is the index of the protocol to which this activation of (the simulated) P_i was given, ensuring it is treated as if coming directly from $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_j,t}$.
  - Upon receiving $(\texttt{core-set}, \mathsf{sid}^j, \mathcal{I})$ from $\mathcal{S}_j$ for any $j \in [M]$, if $\mathcal{I} \subseteq \mathcal{P}$ with $|\mathcal{I}| \geq n - t$ and $\mathsf{part}_i^{\text{in},j} = 1$ for all $P_i \in \mathcal{I}$, send $(\texttt{core-set}, \mathsf{sid}, \mathcal{I})$ to $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{strong}}$.
  - Upon receiving $(\texttt{output}, \mathsf{sid}, (\mathcal{CS}, \boldsymbol{y}))$ from $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{strong}}$ with $\boldsymbol{y} = (y^1, \ldots, y^M)$ (directed towards a corrupted party P_i), update $\boldsymbol{y}_i := \boldsymbol{y}$. Send $(\texttt{output}, \mathsf{sid}^j, (\mathcal{CS}, y^j))$ to $\mathcal{S}_j$ according to the description of $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_j,t}$ and its tracked delay counters.
  - Upon receiving $(\texttt{adv-output}, \mathsf{sid}, (\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n))$ from $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{strong}}$, where $\boldsymbol{a}_i = (a_i^1, \ldots, a_i^M)$ for each $i \in [n]$, update $\boldsymbol{y}_i := \boldsymbol{a}_i$ if P_i is corrupted. Send $(\texttt{adv-output}, \mathsf{sid}^j, a_i^j)$ to $\mathcal{S}_j$ according to the description of $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_j,t}$ and its tracked delay counters.
  - Upon receiving $(\texttt{fetched}, \mathsf{sid}, P_i)$ from $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{strong}}$, do the following where $j \in [M]$ is the index of the protocol to which this activation of (the simulated) P_i was given:
    - If $\mathsf{part}_i^{\text{in},j} = 0$, $D_i^{\text{in-part},j} > 1$, and P_i already provided input (i.e., $\mathcal{S}_{\text{strong}}$ has received $(\texttt{leakage}, \mathsf{sid}^j, P_i, \cdot)$):
      1. Update $D_i^{\text{in-part},j} := D_i^{\text{in-part},j} - 1$.
      2. Send $(\texttt{fetched}, \mathsf{sid}^j, P_i)$ to $\mathcal{S}_j$.
    - If $\mathsf{part}_i^{\text{in},j} = 0$, $D_i^{\text{in-part},j} = 1$, and P_i already provided input:
      1. Set $\mathsf{ready}_i^{\text{in},j} := 1$.
      2. If $\bigwedge_{k=1}^{M} \mathsf{ready}_i^{\text{in},k} = 1$, do the following:
        1. Update $D_i^{\text{in-part},j} := 0$.

2. Set $\mathsf{part}_i^{\text{in},j} := 1$.

3. Send $(\texttt{fetched}, \mathsf{sid}^j, P_i)$ to $\mathcal{S}_j$.

– If $\mathsf{part}_i^{\text{out},j} = 0$ and $\sum_{h=1}^{n}(\bigwedge_{k=1}^{M} \mathsf{part}_h^{\text{in},k}) \geq n - t$, do the following:

1. Update $D_i^{\text{out-part},j} := D_i^{\text{out-part},j} - 1$.

2. If $D_i^{\text{out-part},j} = 0$, set $\mathsf{part}_i^{\text{out},j} := 1$.

3. Send $(\texttt{fetched}, \mathsf{sid}^j, P_i)$ to $\mathcal{S}_j$.
Additionally, if P_i is honest and $(\texttt{core-set}, \mathsf{sid}, \mathcal{I})$ is sent to $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{strong}}$ in response of receiving $(\texttt{core-set}, \mathsf{sid}^j, \mathcal{I})$ from $\mathcal{S}_j$, for all corrupted parties such as P_k send $(\texttt{output}, \mathsf{sid}^j, (\mathcal{CS}, y_k^j))$ (y_k^j is the j^{th} component of $\boldsymbol{y}_k$) to $\mathcal{S}_j$.

- Handling delays:
  - Upon receiving $(\texttt{delay}, \mathsf{sid}^j, P_i, \text{in-part}, D)$ from $\mathcal{S}_j$, if $P_i \in \mathcal{P}$ and $D \in \mathbb{Z}$ (represented in unary notation), update $D_i^{\text{in-part},j} := \max(1, D_i^{\text{in-part},j} + D)$.
  - Keep track of each $D_i^{\text{out-part},j}$ and $D_i^{\text{out-rel},j}$ based on delay inputs received from $\mathcal{S}_j$ and parties activation through fetch request and protocol instructions.
  - Set $D_i^{\text{in-part}} := \sum_{j=1}^{M} D_i^{\text{in-part},j}$ in $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{strong}}$.
  - Set $D_i^{\text{out-part}}$ such that once each $\mathcal{S}_j$ lets $D_i^{\text{out-rel},j}$ hit 0 (i.e., P_i is activated and sends $(\texttt{input}, \mathsf{sid}_{\text{acast},i}, \text{ready})$), $D_i^{\text{out-part}}$ becomes 0, and set the flag $\mathsf{part}_i^{\text{out}} := 1$.
  - Set $D_i^{\text{out-rel}}$ such that once $D_i^{\text{out,acast},k} = 0$ for $n - t$ values of $k \in [n]$, $D_i^{\text{out-rel}}$ becomes 0.

Figure 9: The simulator for $\mathsf{Comp}_{\text{strong}}(\pi_1, \ldots, \pi_M)$.

Scheduling note (buffering is benign). The compiler temporarily buffers per-instance ACS inputs/fetches and forwards them to the unified $\mathcal{F}_{\mathsf{acs}}^{n,t}$ only once the same value has appeared across all instances. This does not risk deadlock or add distinguishing power: it is equivalent to an admissible delay pattern that an adversary could impose inside the ACS hybrid anyway. Since each $\hat{\pi}_j$ eventually supplies all honest inputs (by ACS-decomposability), the forwarding condition is met for at least $n - t$ common ACS inputs across parties and instances, so the unified $\mathcal{F}_{\mathsf{acs}}^{n,t}$ outputs. Thus buffering merely re-schedules deliveries and preserves both liveness and indistinguishability.

Now we prove that no environment $\mathcal{Z}$ can distinguish whether it is interacting with the dummy adversary $\mathcal{D}$ and the honest parties running π_{strong} or with the above simulator $\mathcal{S}_{\text{strong}}$ and the dummy parties interacting with $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{strong}}$. We prove this through a series of hybrid games. The output in each hybrid game is the output of the environment in that game.

The games HYB^{j^*}, for $0 \leq j^* \leq M$. For $j \leq j^*$, in the execution of π_{strong} with the dummy adversary $\mathcal{D}$ and the environment $\mathcal{Z}$, the segment corresponding to π_j is replaced with the simulated transcript generated by $\mathcal{S}_j$.

In more detail, the experiment proceeds as follows.

– **Setup and invocation.**
  – The experiment interacts with the joint ideal functionality $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{strong}}$.
  – For every $j \leq j^*$, it invokes $\mathcal{S}_j$ (instead of running π_j). If $0 < j$, the output of the joint functionality for honest parties is sent to $\mathcal{Z}$; if $j = 0$, the output of π_{strong} is sent to $\mathcal{Z}$.
  – For every $j^* < j \leq M$, it also runs the simulator for π_j to obtain the (simulated) delay values needed for bookkeeping.

– **Routing leakage to per-instance simulators.**
  – Upon receiving leakage $(\texttt{leakage}, \mathsf{sid}, P_i, (l^1, \ldots, l^M))$ from the joint functionality, forward $(\texttt{leakage}, \mathsf{sid}^j, P_i, l^j)$ to *each* $\mathcal{S}_j$ for all $j \in [M]$.

- **Activation (fetch) scheduling.**
  - The experiment tracks activations via $(\texttt{fetched}, \mathsf{sid}, P_i)$ messages from the joint functionality.
  - When an activation is directed (per π_{strong}'s instruction) to the segment of π_j for some $j \in [M]$ and party P_i, deliver $(\texttt{fetched}, \mathsf{sid}^j, P_i)$ to $\mathcal{S}_j$.
- **Environment inputs to ACS.**
  - Upon receiving delay/adversarial inputs from $\mathcal{Z}$ that target $\mathcal{F}_{\mathsf{acs}}$ with SID $\mathsf{sid}_{\text{acs}}$, record them centrally and, for every $j \in [M]$, forward the corresponding input to $\mathcal{S}_j$ as targeting its local ACS with SID $\mathsf{sid}^j_{\text{acs}}$.
- **ACS fetch effects: centralized accounting + mirrored events.**
  - Whenever $\mathcal{D}$ reports a fetch from $\mathcal{F}_{\mathsf{acs}}$ with SID $\mathsf{sid}_{\text{acs}}$, or some $\mathcal{S}_j$ reports a fetch from its ACS with SID $\mathsf{sid}^j_{\text{acs}}$ for party P_i, update the *global* ACS counters $D_i^{\text{in,acs}}/D_i^{\text{out,acs}}$ according to $\mathcal{F}^{n,t}_{\mathsf{acs}}$.
  - To keep per-instance transcripts aligned, deliver the same ACS event as a notification $(\texttt{fetched}, \mathsf{sid}^k_{\text{acs}}, P_i)$ to *every* $\mathcal{S}_k$ for all $k \in [M]$.
- **Core-set forwarding.**
  - Whenever any $\mathcal{S}_j$ sends $(\texttt{core-set}, \mathsf{sid}^j, \mathcal{I})$, forward $(\texttt{core-set}, \mathsf{sid}, \mathcal{I})$ to the joint functionality.
- **Global delays for the joint functionality.**
  - Maintain $D_i^{\text{in-part}} = \sum_{j=1}^{M} D_i^{\text{in-part},j}$.
  - Set $D_i^{\text{out-part}}$ so that once P_i A-Casts `ready` (i.e., when each per-instance output-participation is satisfied), its global output-participation delay becomes 0.
  - Set $D_i^{\text{out-rel}}$ so that once P_i has received `ready` via $\mathcal{F}_{\mathsf{a\text{-}cast}}$ from at least $n - t$ parties, its global output-release delay becomes 0.
- **Outputs for corrupted parties.**
  - Upon receiving outputs for corrupted parties from the joint functionality—either $(\texttt{adv-output}, \mathsf{sid}, \boldsymbol{a})$ or $(\texttt{output}, \mathsf{sid}, (\mathcal{CS}, \boldsymbol{y}))$— record them; for each $j \in [M]$, deliver the corresponding per-instance slice (either a_i^j or y^j) to $\mathcal{S}_j$ *only when* $\mathcal{S}_j$'s per-instance output-release condition for P_i is met, and otherwise continue to deliver $(\texttt{fetched}, \mathsf{sid}^j, P_i)$ activations as needed.
- **Other communications.**
  - For all $j \in [j^*]$, relay all other communications between $\mathcal{S}_j$ and $\mathcal{Z}$ verbatim.

Claim 5.3. $\text{HYB}^0 \equiv \text{HYB}^M$.

Proof. We start by proving that for any $j \in [M]$, $\text{HYB}^{j-1} \equiv \text{HYB}^j$. For the sake of contradiction, assume that for some $j \in [M]$, $\text{HYB}^{j-1} \not\equiv \text{HYB}^j$.

First, we prove that for all $0 \leq j \leq M$, in HYB^j, the instance of $\mathcal{F}_{\mathsf{acs}}$ with SID $\mathsf{sid}_{\text{acs}}$ produces output and that output is the same in all. To do so, we show that in HYB^0, the instance of $\mathcal{F}_{\mathsf{acs}}$ with SID $\mathsf{sid}_{\text{acs}}$ generates output and then we prove that for any $0 < j \leq M$, the outputs of $\mathcal{F}_{\mathsf{acs}}$ with SID $\mathsf{sid}_{\text{acs}}$ in both HYB^{j-1} and HYB^j are the same.

In HYB^0, a party is provided as input to the instance of $\mathcal{F}_{\mathsf{acs}}$ with SID $\mathsf{sid}_{\text{acs}}$ once all $\pi_1, \ldots, \pi_M$ provide input for that party. Each π_k eventually provides input for all honest parties; otherwise, $\hat{\pi}_k$—the protocol derived from π_k by replacing the call to π_{acs} with an invocation of $\mathcal{F}^{n,t}_{\mathsf{acs}}$—may never receive output from its instance of $\mathcal{F}_{\mathsf{acs}}$. According to the second property of the ACS-decomposable class (see Definition 5.1), $\hat{\pi}_k$ cannot generate output, which would contradict its security. Therefore, each π_k eventually provides input for all honest parties, ensuring that in

HYB^0, the instance of $\mathcal{F}_{\mathsf{acs}}$ with SID $\mathsf{sid}_{\text{acs}}$ is eventually provided with inputs from at least $n - t$ parties, thereby guaranteeing output generation. By the scheduling note, buffering these inputs is indistinguishable from admissible adversarial delays in $\mathcal{F}_{\mathsf{acs}}^{n,t}$ and does not affect liveness or outputs.

For all $0 < j \leq M$, since the only part that changes in the inputs provided to $\mathcal{F}_{\mathsf{acs}}$ with SID $\mathsf{sid}_{\text{acs}}$ in HYB^{j-1} and HYB^j is the set of messages simulated by $\mathcal{S}_j$, any discrepancy in the output of $\mathcal{F}_{\mathsf{acs}}$ with SID $\mathsf{sid}_{\text{acs}}$ between HYB^{j-1} and HYB^j implies an environment that violates the simulation of $\mathcal{S}_j$. By the scheduling note, any timing differences due to buffering are also simulatable as delays and cannot aid $\mathcal{Z}$ in distinguishing the hybrids. Therefore, for all $0 \leq j \leq M$, in HYB^j, the instance of $\mathcal{F}_{\mathsf{acs}}$ with SID $\mathsf{sid}_{\text{acs}}$ produces output and that output is the same in all.

We also verify the claim made in Section 5.1 that the aggregated ACS inputs inherit the convergence precondition. Suppose that index k belongs to the aggregated ACS input of some honest P_i, i.e., that $k \in \mathcal{X}^j$ in P_i's view for every $j \in [M]$. Fix any $j \in [M]$. Since π_j satisfies the convergence precondition for its own ACS call, every honest party eventually includes k in its ACS input for π_j. As this holds for every $j \in [M]$, every honest party eventually includes k in its aggregated ACS input as well. Hence the aggregated inputs converge, and the output of the unified $\mathcal{F}_{\mathsf{acs}}^{n,t}$ is eventually contained in the aggregated set of every honest party, as each $\hat{\pi}_j$ requires.

Since for all $1 \leq j \leq M$, HYB^j takes the output of honest parties from $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{strong}}$ and they all set the same set of input providers, their outputs would be the same. Therefore, $\text{HYB}^{j-1} \not\equiv \text{HYB}^j$ directly implies an environment that violates the simulation of $\mathcal{S}_j$.

The same thing also holds for HYB^0 and HYB^1 but requires slightly different reasoning because HYB^0 and HYB^1 also differ in how they generate the outputs for the honest parties.

As we discussed, the output of $\mathcal{F}_{\mathsf{acs}}$ with SID $\mathsf{sid}_{\text{acs}}$ is the same in both HYB^0 and HYB^1 which defines the set of input providers in each π_i, for $i \in [M]$ and $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{strong}}$. This follows from the second property of Definition 5.1 and the description of HYB^1. We also expect the protocols $\pi_1, \ldots, \pi_M$ and the functionality $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{strong}}$ to compute the same set of functions. Therefore, any inconsistency in the output of some function f_k, contradicts the correctness of π_k (implies an environment that violates the simulation of $\mathcal{S}_k$).

Therefore, for any $j \in [M]$, $\text{HYB}^{j-1} \equiv \text{HYB}^j$. Thus, the claim follows from a standard hybrid argument. □

This completes the proof since HYB^0 exactly represents the execution of π_{strong} with $\mathcal{D}$ and $\mathcal{Z}$ and HYB^M exactly represents the execution of $\text{IDEAL}_{[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{strong}}}$. □

Recall that (n, t)-ACS-decomposable classes are defined with perfect security against adaptive malicious adversaries. For similar classes that offer only more relaxed security guarantees the above compiler would provide a matching security.

Remark 5.4. *Under the conditions of Theorem 5.2, if the underlying protocols are secure against other adversarial models—whether static or adaptive, including malicious, fail-stop, or semi-malicious adversaries—and provide statistical or computational security, then the compiled protocol π_{strong} preserves these guarantees.*

5.2 Feasibility of Conditional PBB Weak Parallel Composition

We proceed to prove the feasibility of the weak parallel composition. Our result applies to a specific class of protocols that allow the parties to carry out each execution until a certain point which guarantees that the output value of the computation is *fixed* (regardless of the adversary's future actions), but not known to any subset of t parties. We capture this notion in the spirit of a *committal round* [MR92] that was put forth for synchronous protocols. A committal round ensures

that the simulator for the protocol can be split into two phases: first, the simulation is carried out till the committal round; next, the simulator sends its inputs to the functionality and receives the output; finally, the second phase of the simulation is carried out.

We begin by adjusting the notion of a committal round to the asynchronous setting, where parties proceed out of sync and honest parties might be at different stages of execution. Here, it is sufficient for a single honest party to reach the committal point to ensure the simulator's input is committed. However, a global point might not be recognizable by all honest parties, especially those stuck in earlier stages. Thus, we define a *locally identifiable committal point* for each party, which is an identifiable point in its local view past the global committal point. We require that even if any honest party reaching that point halts, at least $n - 2t$ honest parties will reach it too.

In the existing protocols, the point immediately after completing the ACS can serve as a locally identifiable committal point. This point is locally identifiable, and by then, the effective input that determines the output is fixed.

Definition 5.5 (Asynchronous Committal Point)**.** *Let $n, t \in \mathbb{N}$ such that $t < n/3$, let f be an n-ary function, and let π be a protocol that securely realizes $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f,t}$. We say that π has a* locally identifiable asynchronous committal point *if for every party there is a committal point (i.e., a specific protocol instruction) such that the following hold:*

1. *The committal point is locally identified by the party (formally, the next-message function outputs an additional bit, which is set to 1 only for this instruction).*
2. *Even if each honest party that reaches its locally identifiable committal point gets corrupted, at least $n - 2t$ honest parties will still reach their locally identifiable committal point.*
3. *For every adversary, the simulator (interacting with an environment and with $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f,t}$) that simulates this adversary consists of two sub-simulators $\mathcal{S}^{(1)}$ and $\mathcal{S}^{(2)}$. In the beginning, the simulation is carried out using $\mathcal{S}^{(1)}$ (in particular, $\mathcal{S}^{(1)}$ never obtains the output value from $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f,t}$); once an honest party reaches its committal point in the simulated execution, then the simulator obtains the output value from $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f,t}$, and invokes $\mathcal{S}^{(2)}$ on the state of $\mathcal{S}^{(1)}$ and the output value until the completion of the simulation.*

We define the class of protocols with an identifiable asynchronous committal point (ACP), which we argue is broader than the ACS-decomposable class. We also propose a PBB compiler for this class that achieves weak parallel composition.

Definition 5.6 (ACP Protocol Class)**.** *Let $n, t \in \mathbb{N}$ with $t < n/3$. Protocol class Π is said to be (n, t)-ACP if every protocol $\pi \in \Pi$ has n participants, UC-realizes $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f,t}$ against adaptive malicious t-adversaries with perfect security where f is an n-ary function, and has a locally identifiable asynchronous committal point.*

Proposition 5.7. *Any (n, t)-ACS-decomposable protocol class is also (n, t)-ACP.*

Proof. Let Π be an arbitrary (n, t)-ACS-decomposable class. For any protocol $\pi \in \Pi$, the instruction immediately following the termination of π_{acs} serves as a locally identifiable asynchronous committal point. □

As before, we note that while the current definition of (n, t)-ACP classes offers perfect security against adaptive malicious adversaries, they can also be defined with more relaxed security guarantees in a natural way.

Compiler $\mathsf{Comp}_{\mathrm{weak}}(\pi_1, \ldots, \pi_M)$

The compiler is given oracle access to the next-message functions $f^{\pi_1}_{\text{nxt-msg}}, \ldots, f^{\pi_M}_{\text{nxt-msg}}$ of protocols $\pi_1, \ldots, \pi_M$ in an (n,t)-ACP class. At the first activation, verify that $\mathsf{sid} = (\mathcal{P}, \mathsf{sid}')$ where $\mathcal{P}$ is the participant set.

Party $P_i \in \mathcal{P}$ proceeds as follows: For each $j \in [M]$, initialize $\mathcal{CS}_i^j := \emptyset$ and $y_i^j := \bot$, set $\mathsf{ready}_i^j := 0$ and $\mathsf{cntr}_i^j := 0$, and define $\mathsf{sid}^j := (\mathsf{sid}, j)$ and $\mathsf{sid}_i^j := (P_i, j, \mathsf{sid})$.

- Upon activation with input $(\mathtt{input}, \mathsf{sid}, \boldsymbol{x})$ where $\boldsymbol{x} = (x^1, \ldots, x^M)$, pass $(\mathtt{input}, \mathsf{sid}^j, x^j)$ to protocol π_j for all $j \in [M]$.
- Upon activation with input $(\mathtt{fetch}, \mathsf{sid})$, do:
  1. For each $j \in [M]$ such that $\mathsf{ready}_i^j = 0$, send the message generated by π_j.
  2. For each $j \in [M]$ such that P_i reaches its identifiable committal point in protocol π_j (as indicated by $f^{\pi_j, i}_{\text{nxt-msg}}$), set $\mathsf{ready}_i^j := 1$ and send the message $(\mathtt{input}, \mathsf{sid}_i^j, \mathsf{ready})$ to $\mathcal{F}_{\mathsf{a\text{-}cast}}$.
  3. For each $j \in [M]$ and $\ell \in [n]$, in a round-robin fashion, fetch the output from $\mathcal{F}_{\mathsf{a\text{-}cast}}$ with SID sid_ℓ^j. Upon receiving back $(\mathtt{output}, \mathsf{sid}_\ell^j, \mathsf{ready})$ update $\mathsf{cntr}_i^j := \mathsf{cntr}_i^j + 1$.
  4. If for all $j \in [M]$ it holds that $\mathsf{ready}_i^j = 1$ or $\mathsf{cntr}_i^j \geq t+1$, continue running each π_j in a round-robin fashion.
     Upon generating the output $(\mathtt{output}, \mathsf{sid}^j, (\mathcal{CS}^j, y^j))$, record $\mathcal{CS}_i^j := \mathcal{CS}^j$ and $y_i^j := y^j$.
  5. Wait until $y_i^j \neq \bot$ for all $j \in [M]$.
     Then, let $\boldsymbol{\mathcal{CS}}_i := (\mathcal{CS}_i^1, \ldots, \mathcal{CS}_i^M)$ and $\boldsymbol{y}_i := (y_i^1, \ldots, y_i^M)$, and output $(\mathtt{output}, \mathsf{sid}, (\boldsymbol{\mathcal{CS}}_i, \boldsymbol{y}_i))$.

Figure 10: Conditional PBB compiler for weak parallel composition.

Before stating the security of $\mathsf{Comp}_{\mathrm{weak}}$, we start with the high-level ideas of the simulation. To simulate π_j for each $j \in [M]$, the simulator $\mathcal{S}_{\mathrm{Comp}}$ (against dummy adversary $\mathcal{D}$) uses the simulator respecting the identifiable ACP (see Definition 5.5) guaranteed to exist for π_j by Definition 5.6. Recall that the simulator provides the set of input providers and the inputs for corrupted parties in that set before receiving any leakages about the output. Let $\mathcal{S}_j$ be such a simulator for $\mathcal{D}$.

This approach is effective because in the compiler, once the first honest party passes the identifiable ACP, it is guaranteed that the party has already reached to the identifiable ACP in all other protocols. According to the definition of the identifiable asynchronous committal point, it means that all simulators $\mathcal{S}_j$ must have selected the core set already without requiring any leakage about the output so $\mathcal{S}_{\mathrm{weak}}$ can also set the core sets in $[\mathcal{F}^{f_1,t}_{\mathsf{a\text{-}sfe}} \parallel \ldots \parallel \mathcal{F}^{f_M,t}_{\mathsf{a\text{-}sfe}}]_{\mathrm{weak}}$ for all functions without requiring any leakage about any of the outputs. There are some details about how $\mathcal{S}_{\mathrm{weak}}$ needs to aggregate messages from different simulators and how it needs to split messages from the ideal functionality among these simulators that we discuss in detail in the formal proof of the theorem.

Theorem 5.8. *Let $M, n, t \in \mathbb{N}$, let Π be an (n,t)-ACP class, and let $f_1, \ldots, f_M$ be n-ary computable functions with respective protocols $\pi_1, \ldots, \pi_M \in \Pi$ realizing them against adaptive malicious t-adversaries with perfect security.*

The compiled protocol $\pi_{\mathsf{weak}} := \mathsf{Comp}_{\mathrm{weak}}(\pi_1, \ldots, \pi_M)$ UC-realizes $[\mathcal{F}^{f_1,t}_{\mathsf{a\text{-}sfe}} \parallel \ldots \parallel \mathcal{F}^{f_M,t}_{\mathsf{a\text{-}sfe}}]_{\mathrm{weak}}$ in the $(\mathcal{F}^{n,t}_{\mathsf{a\text{-}cast}}, \mathcal{F}_{\mathsf{a\text{-}smt}})$-hybrid model, against adaptive malicious t-adversaries with perfect security.

Proof. We consider the equivalent notion of UC-realization against the dummy adversary $\mathcal{D}$. That is, we construct a simulator $\mathcal{S}_{\mathrm{weak}}$ such that no environment $\mathcal{Z}$ can distinguish whether it is interacting with π_{weak} and $\mathcal{D}$, or with $[\mathcal{F}^{f_1,t}_{\mathsf{a\text{-}sfe}} \parallel \ldots \parallel \mathcal{F}^{f_M,t}_{\mathsf{a\text{-}sfe}}]_{\mathrm{weak}}$ and $\mathcal{S}_{\mathrm{weak}}$.

As previously mentioned, to simulate π_j for each $j \in [M]$, the simulator $\mathcal{S}_{\mathrm{weak}}$ uses the simulator $\mathcal{S}_j$ for $\mathcal{D}$, adhering to the identifiable ACP (see Definition 5.5) that is guaranteed to exist for π_j by Definition 5.6. Below is a detailed description of $\mathcal{S}_{\mathrm{weak}}$. For clarity, we briefly recall the meaning

of the counters and flags used in the simulator.

- For each party P_i and each instance $j \in [M]$:
  - $D_i^{\text{in-part},j}$ is the *input-participation delay counter* for instance j; that is, how many delay steps remain before P_i's input is treated as provided to $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_j,t}$.
  - $D_i^{\text{out-part},j}$ is the *output-participation delay counter* for instance j; that is, how long until P_i becomes eligible to receive output from $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_j,t}$.
  - $D_i^{\text{out-rel},j}$ is the *output-release delay counter* for instance j; when it hits 0, P_i's output in instance j is actually released (assuming eligibility).
  - $\mathsf{part}_i^{\text{in},j}$ is a flag set once P_i is counted as having participated in the input phase of instance j.
  - $\mathsf{part}_i^{\text{out},j}$ is a flag set once P_i is counted as having participated in the output phase of instance j.
- A-Cast–specific readiness bookkeeping:
  - $\mathsf{ready}_{i,j}$ is a flag indicating whether P_i has reached the identifiable committal point in π_j and (conceptually) A-Casted `ready` for protocol j.
  - $\mathsf{cntr}_{i,j} \in \mathbb{N}$ is a counter for how many `ready` A-Casts for protocol j party P_i has received (used to enforce the $t+1$ threshold before proceeding).
- Core sets / outputs and SIDs:
  - $\mathcal{CS}_i^j$ and y_i^j are the core set and output value recorded for P_i in instance j; and the aggregates $\boldsymbol{\mathcal{CS}}_i = (\mathcal{CS}_i^1, \ldots, \mathcal{CS}_i^M)$, $\boldsymbol{y}_i = (y_i^1, \ldots, y_i^M)$.
  - $\mathsf{sid}^j := (\mathsf{sid}, j)$ is the SID for the π_j segment (and its $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_j,t}$ interface) within the joint sid.
  - $\mathsf{sid}_{i,j} := (P_i, j, \mathsf{sid})$ is the SID for the $\mathcal{F}_{\mathsf{a\text{-}cast}}$ instance carrying P_i's `ready` for protocol j.

Simulator $\mathcal{S}_{\text{weak}}$

For each $j \in [M]$, to simulate the execution segment pertaining to π_j, the simulator $\mathcal{S}_{\text{weak}}$ invokes $\mathcal{S}_j$ and utilizes its responses to interact with $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{weak}}$ and $\mathcal{Z}$. The following parts detail how the simulator plays the role of the environment and the ideal functionality $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_j,t}$ for each $\mathcal{S}_j$, and outlines how messages from each simulator instance are used to engage with $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{weak}}$ and $\mathcal{Z}$.

- Initialization:
  - For every $i \in [n]$ and $j \in [M]$, initialize $D_i^{\text{in-part},j} = D_i^{\text{out-part},j} = D_i^{\text{out-rel},j} := 1$.
  - For every $i \in [n]$ and $j \in [M]$, initialize $\mathsf{part}_i^{\text{in},j} = \mathsf{part}_i^{\text{out},j} := 0$.
- Handling corruption requests:
  - Upon a direct corruption request from $\mathcal{Z}$ for some party P_i, send the request to $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{weak}}$ and to every $\mathcal{S}_j$ ($j \in [M]$). Also corrupt the relevant party in all simulated $\mathcal{F}_{\mathsf{a\text{-}cast}}$'s functionalities. Use the leakage from $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{weak}}$ to accordingly leak to $\mathcal{S}_j$ for all $j \in [M]$ and from all simulated $\mathcal{F}_{\mathsf{a\text{-}cast}}$'s.
  - For all $j \in [M]$, send corruption requests from $\mathcal{S}_j$ to $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{weak}}$ and relay back relevant leakages.
- Communication with $\mathcal{Z}$:
  - For each $j \in [M]$, route all messages from $\mathcal{Z}$ that target the segment associated with π_j directly to $\mathcal{S}_j$, ensuring they are treated as if coming directly from the environment.
  - For each $j \in [M]$, forward messages from $\mathcal{S}_j$ to the environment directly to $\mathcal{Z}$.
- Influencing $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{weak}}$ and for each $j \in [M]$, playing the role of $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_j,t}$ for $\mathcal{S}_j$:

- Upon receiving $(\texttt{leakage}, \mathsf{sid}, P_i, \boldsymbol{l})$ from $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{weak}}$ where $\boldsymbol{l} = (l^1, \ldots, l^M)$, send $(\texttt{leakage}, \mathsf{sid}^j, P_i, l^j)$ to $\mathcal{S}_j$, where $j \in [M]$ is the index of the protocol to which this activation of (the simulated) P_i was given, ensuring it is treated as if coming directly from $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_j,t}$.
- Upon receiving $(\texttt{core-set}, \mathsf{sid}^j, \mathcal{I})$ from $\mathcal{S}_j$ for any $j \in [M]$, send $(\texttt{core-set}, \mathsf{sid}, j, \mathcal{I})$ to $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{weak}}$. // Note that this must happen before any honest party enters the identifiable asynchronous committal point.
- Upon receiving $(\texttt{output}, \mathsf{sid}, (\boldsymbol{\mathcal{CS}}, \boldsymbol{y}))$ from $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{weak}}$ with $\boldsymbol{\mathcal{CS}} = (\mathcal{CS}^1, \ldots, \mathcal{CS}^M)$ and $\boldsymbol{y} = (y^1, \ldots, y^M)$ (directed towards a corrupted party P_i), update $\boldsymbol{\mathcal{CS}}_i := \boldsymbol{\mathcal{CS}}$ and $\boldsymbol{y}_i := \boldsymbol{y}$. Send $(\texttt{output}, \mathsf{sid}^j, (\mathcal{CS}^j, y^j))$ to $\mathcal{S}_j$ according to the description of $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_j,t}$ and its tracked delay counters.
- Upon receiving $(\texttt{adv-output}, \mathsf{sid}, (\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n))$ from $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{weak}}$, where $\boldsymbol{a}_i = (a_i^1, \ldots, a_i^M)$ for each $i \in [n]$, update $\boldsymbol{y}_i := \boldsymbol{a}_i$ if P_i is corrupted. Send $(\texttt{adv-output}, \mathsf{sid}^j, a_i^j)$ to $\mathcal{S}_j$ according to the description of $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_j,t}$ and its tracked delay counters.
- Upon receiving $(\texttt{fetched}, \mathsf{sid}, P_i)$ from $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{weak}}$, do the following where $j \in [M]$ is the index of the protocol to which this activation of (the simulated) P_i was given:
  - If $\mathsf{part}_i^{\text{in},j} = 0$ and P_i already provided input:
    1. Update $D_i^{\text{in-part},j} := 0$.
    2. Set $\mathsf{part}_i^{\text{in},j} := 1$.
    3. Send $(\texttt{fetched}, \mathsf{sid}^j, P_i)$ to $\mathcal{S}_j$.
  - If $\mathsf{part}_i^{\text{out},j} = 0$ and $\sum_{h=1}^n \mathsf{part}_h^{\text{in},j} \geq n - t$, do the following:
    1. Update $D_i^{\text{out-part},j} := D_i^{\text{out-part},j} - 1$.
    2. If $D_i^{\text{out-part},j} = 0$, set $\mathsf{part}_i^{\text{out},j} := 1$.
    3. Send $(\texttt{fetched}, \mathsf{sid}^j, P_i)$ to $\mathcal{S}_j$. Additionally, if P_i is honest and $(\texttt{core-set}, \mathsf{sid}, \mathcal{I})$ is sent to $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{weak}}$ in response of receiving $(\texttt{core-set}, \mathsf{sid}^j, \mathcal{I})$ from $\mathcal{S}_j$, for all corrupted parties such as P_k send $(\texttt{output}, \mathsf{sid}^j, (\mathcal{CS}_k^j, y_k^j))$ to $\mathcal{S}_j$ (where $\mathcal{CS}_k^j$ is the j^{th} component of $\boldsymbol{\mathcal{CS}}_k$ and y_k^j is the j^{th} component of $\boldsymbol{y}_k$). // Note than this leakage does not happen until all $\mathcal{S}_k$'s select their core sets.
- Handling delays:
  - Upon receiving a delay message $(\texttt{delay}, \mathsf{sid}^j, P_i, \mathsf{type}, D)$ from $\mathcal{S}_j$ for any $\mathsf{type} \in \{\text{in-part}, \text{out-part}, \text{out-rel}\}$, if $P_i \in \mathcal{P}$ and $D \in \mathbb{Z}$ (represented in unary notation), update $D_i^{\mathsf{type},j} := \max(1, D_i^{\mathsf{type},j} + D)$.
    Additionally, send $(\texttt{delay}, \mathsf{sid}, j, P_i, \mathsf{type}, D)$ to $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{weak}}$.

Figure 11: The simulator for $\mathsf{Comp}_{\text{weak}}(\pi_1, \ldots, \pi_M)$.

Now we prove that no environment $\mathcal{Z}$ can distinguish whether it is interacting with the dummy adversary $\mathcal{D}$ and the honest parties running π_{weak} or with the above simulator $\mathcal{S}_{\text{weak}}$ and the dummy parties interacting with $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{weak}}$. We prove this through a series of hybrid games. The output in each hybrid game is the output of the environment in that game.

The games HYB^{j^*}, for $0 \leq j^* \leq M$. For $j \leq j^*$, in the execution of the protocol π_{weak} with the dummy adversary $\mathcal{D}$ and the environment $\mathcal{Z}$, the segment corresponding to π_j is replaced with the simulated transcript generated by $\mathcal{S}_j$.

In more detail, the experiment proceeds as follows.

- **Setup and invocation.**
  - The experiment interacts with the joint ideal functionality $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{weak}}$.
  - For every $j \leq j^*$, it invokes $\mathcal{S}_j$ (instead of running π_j). If $0 < j$, the output of the joint functionality for honest parties is sent to $\mathcal{Z}$; if $j = 0$, the output of π_{weak} is sent to $\mathcal{Z}$.

  - For every $j^* < j \le M$, it also runs the simulator for π_j to obtain the (simulated) delay values needed for bookkeeping.
- **Routing leakage to per-instance simulators.**
  - Upon receiving leakage $(\texttt{leakage}, \mathsf{sid}, P_i, (l^1, \ldots, l^M))$ from the joint functionality, forward $(\texttt{leakage}, \mathsf{sid}^j, P_i, l^j)$ to *each* $\mathcal{S}_j$ for all $j \in [M]$.
- **Committal/A-Cast readiness.**
  - Whenever some $\mathcal{S}_j$ (with $j \le j^*$) indicates that an honest P_ℓ reached its identifiable committal point in π_j, send $(\texttt{input}, \mathsf{sid}_{\ell,j}, \texttt{ready})$ to the corresponding $\mathcal{F}_{\mathsf{a\text{-}cast}}$ instance (SID $\mathsf{sid}_{\ell,j}$) and update the auxiliary tallies $\mathsf{ready}_{\ell,j}$ and $\mathsf{cntr}_{\cdot,j}$ as in the compiler.
  - For each $i \in [n]$ and $j \in [M]$, when P_i fetches from $\mathcal{F}_{\mathsf{a\text{-}cast}}$ and receives $(\texttt{output}, \mathsf{sid}_{k,j}, \texttt{ready})$, update $\mathsf{cntr}_{i,j} := \mathsf{cntr}_{i,j} + 1$.
- **Activation (fetch) scheduling.**
  - Deliver activations $(\texttt{fetched}, \mathsf{sid}^j, P_i)$ to $\mathcal{S}_j$ *only when the compiler's condition for proceeding in instance j holds*, namely when $\mathsf{ready}_i^j = 1$ or $\mathsf{cntr}_i^j \ge t+1$. (There is no ACS here; activations remain local to each instance.)
- **Core-set forwarding.**
  - Whenever any $\mathcal{S}_j$ sends $(\texttt{core-set}, \mathsf{sid}^j, \mathcal{I})$, forward $(\texttt{core-set}, \mathsf{sid}, j, \mathcal{I})$ to the joint functionality.
- **Delays for the joint functionality.**
  - Upon $(\texttt{delay}, \mathsf{sid}^j, P_i, \mathsf{type}, D)$ from $\mathcal{S}_j$ with $\mathsf{type} \in \{\text{in-part}, \text{out-part}, \text{out-rel}\}$, update $D_i^{\mathsf{type},j} := \max(1, D_i^{\mathsf{type},j} + D)$ and relay $(\texttt{delay}, \mathsf{sid}, j, P_i, \mathsf{type}, D)$ to the joint functionality.
- **Outputs for corrupted parties.**
  - Upon receiving outputs for corrupted parties from the joint functionality—either $(\texttt{adv-output}, \mathsf{sid}, \boldsymbol{a})$ or $(\texttt{output}, \mathsf{sid}, (\boldsymbol{\mathcal{CS}}, \boldsymbol{y}))$—record them; for each $j \in [M]$, deliver the corresponding per-instance slice (either a_i^j or $(\mathcal{CS}^j, y^j)$) to $\mathcal{S}_j$ *only when* $\mathcal{S}_j$'s per-instance output-release condition for P_i is met; otherwise continue to deliver $(\texttt{fetched}, \mathsf{sid}^j, P_i)$ activations as needed.
- **Other communications.**
  - For all $j \in [j^*]$, relay all other communications between $\mathcal{S}_j$ and $\mathcal{Z}$ verbatim.

Claim 5.9. $\text{HYB}^0 \equiv \text{HYB}^M$.

Proof. We start by proving that there is no deadlock in HYB^0 and then continue by showing that for any $j \in [M]$, indeed $\text{HYB}^{j-1} \equiv \text{HYB}^j$.

Note that, as per Definition 5.5, even if every party that reaches its identifiable committal point halts, at least $n - 2t$ honest parties will eventually reach their committal points. Consequently, in our compiler, for each protocol, at least $n - 2t \ge t + 1$ honest parties will eventually A-Cast "ready." This prevents potential deadlock, as per the description of the compiler, a party continues past its identifiable committal point in a protocol upon receiving $t + 1$ "ready" messages for that protocol.

Now for the sake of contradiction, assume that for some $j \in [M]$, $\text{HYB}^{j-1} \not\equiv \text{HYB}^j$. For any $1 < j \le M$, $\text{HYB}^{j-1} \not\equiv \text{HYB}^j$ implies an environment that violates the simulation of $\mathcal{S}_j$. The reason is that the only difference between those two hybrid games is the replacement of π_j with its simulated transcript and the simulation of $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_j,t}$ toward $\mathcal{S}_j$ is flawless. This is the case because, due to the pause in execution of protocol π_j before the identifiable committal point, the experiment has enough time to learn the required leakages from $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{weak}}$ to provide them to $\mathcal{S}_j$ as needed. In more detail, $\mathcal{S}_j$ will not invoke $\mathcal{S}_j^{(2)}$ until some honest party in π_j has passed its

committal point and $\mathcal{S}_j^{(1)}$ never requires the output. Furthermore, by the description of $\mathsf{Comp}_{\text{weak}}$, an honest party only passes its committal point in π_j once, for every other protocol π_k, either it has itself reached its committal point in π_k or it has received $t+1$ `ready` A-Casts for π_k; in the latter case at least one of these comes from an honest party, so in either case some honest party has already reached its committal point in π_k, and hence $\mathcal{S}_k$ has already fixed the core set for $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_k,t}$. Therefore, by the time $\mathcal{S}_j$ invokes $\mathcal{S}_j^{(2)}$, all inputs to $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{weak}}$ are fixed, and the output is leaked which can then be provided to $\mathcal{S}_j^{(2)}$.

Also, $\text{HYB}^0 \not\equiv \text{HYB}^1$ results in a contradiction but it requires slightly more argument. In fact, in addition to the replacement of π_1 with its simulated transcript, those two hybrid games generate the output of honest parties in different ways. HYB^0 uses π_{weak} and HYB^1 uses $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{weak}}$. We first argue that both cases generate the same outputs for the honest parties. This is the case because, for each protocol π_j, the simulator $\mathcal{S}_j$ (more accurately $\mathcal{S}_j^{(1)}$) must set the core set to the set of input providers in the protocol; otherwise, there exists an environment that violates the simulation of $\mathcal{S}_j$. Moreover, the experiment also uses the core sets reported by $\mathcal{S}_j$ for f_j in $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{weak}}$. Therefore, both $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{weak}}$ and π_{weak} use the same set of input providers for each function. Now if the output of any function f_k differs in $[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \parallel \ldots \parallel \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{weak}}$ and π_{weak}, it implies an environment that violates the simulation by $\mathcal{S}_k$. This is due to the same reasoning discussed above, namely the simulation of $\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_j,t}$ toward $\mathcal{S}_j$ for all $j \in [M]$ is flawless.

Therefore, for any $j \in [M]$, $\text{HYB}^{j-1} \equiv \text{HYB}^j$. Thus, the claim follows from a standard hybrid argument. □

This completes the proof of Theorem 5.8 since HYB^0 exactly represents the execution of π_{weak} with $\mathcal{D}$ and $\mathcal{Z}$ and HYB^M exactly represents the execution of $\text{IDEAL}_{[\mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_1,t} \| \ldots \| \mathcal{F}_{\mathsf{a\text{-}sfe}}^{f_M,t}]_{\text{weak}}}$. □

Similarly to the case of strong parallel composition, for similar protocol classes that offer only more relaxed security guarantees, the above compiler would provide a matching security.

Remark 5.10. *Under the conditions of Theorem 5.8, if the underlying protocols are secure against other adversarial models—whether static or adaptive, including malicious, fail-stop, or semi-malicious adversaries—and provide statistical or computational security, then the compiled protocol π_{weak} preserves these security guarantees.*

Bibliography

[AAPP24] Ittai Abraham, Gilad Asharov, Shravani Patil, and Arpita Patra. Perfect asynchronous MPC with linear communication overhead. In Marc Joye and Gregor Leander, editors, *EUROCRYPT 2024, Part V*, volume 14655 of *LNCS*, pages 280–309. Springer, Cham, 2024.

[AAPS24] Ittai Abraham, Gilad Asharov, Arpita Patra, and Gilad Stern. Asynchronous agreement on a core set in constant expected time and more efficient asynchronous VSS and MPC. In Elette Boyle and Mohammad Mahmoody, editors, *TCC 2024, Part IV*, volume 15367 of *LNCS*, pages 451–482. Springer, Cham, 2024.

[ACCI25] Gilad Asharov, Anirudh Chandramouli, Ran Cohen, and Yuval Ishai. Peeking into the future: MPC resilient to super-rushing adversaries. In *EUROCRYPT 2025, Part V*, volume 15605 of *LNCS*, pages 390–420. Springer, Cham, 2025.

[ACS22] Gilad Asharov, Ran Cohen, and Oren Shochat. Static vs. adaptive security in perfect MPC: A separation and the adaptive security of BGW. In *3rd Conference on Information-Theoretic*

Cryptography, ITC 2022, volume 230 of *LIPIcs*, pages 15:1–15:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[AJL+12] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. Multiparty computation with low communication, computation and interaction via threshold FHE. In *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 483–501. Springer, Berlin, Heidelberg, 2012.

[BCG93] Michael Ben-Or, Ran Canetti, and Oded Goldreich. Asynchronous secure computation. In *25th ACM STOC*, pages 52–61. ACM Press, 1993.

[BE03] Michael Ben-Or and Ran El-Yaniv. Resilient-optimal interactive consistency in constant time. *Distributed Computing*, 16(4):249–262, 2003.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th ACM STOC*, pages 1–10. ACM Press, 1988.

[BKR94] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous secure computations with optimal resilience (extended abstract). In *13th ACM PODC*, pages 183–192. ACM, 1994.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *22nd ACM STOC*, pages 503–513. ACM Press, 1990.

[Bra87] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, 1987.

[BT85] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824–840, 1985.

[Can96] Ran Canetti. *Studies in secure multiparty computation and applications*. PhD thesis, Weizmann Institute of Science, 1996.

[Can20] Ran Canetti. Universally composable security. *Journal of the ACM*, 67(5):1–94, 2020.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *20th ACM STOC*, pages 11–19. ACM Press, 1988.

[CCGZ21] Ran Cohen, Sandro Coretti, Juan A. Garay, and Vassilis Zikas. Round-preserving parallel composition of probabilistic-termination cryptographic protocols. *Journal of Cryptology*, 34(2):12, 2021.

[CDD+01] Ran Canetti, Ivan Damgård, Stefan Dziembowski, Yuval Ishai, and Tal Malkin. On adaptive vs. non-adaptive security of multiparty protocols. In Birgit Pfitzmann, editor, *EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 262–279. Springer, Berlin, Heidelberg, 2001.

[CDN01] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. Multiparty computation from threshold homomorphic encryption. In Birgit Pfitzmann, editor, *EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 280–299. Springer, Berlin, Heidelberg, 2001.

[CFG+23] Ran Cohen, Pouyan Forghani, Juan A. Garay, Rutvik Patel, and Vassilis Zikas. Concurrent asynchronous byzantine agreement in expected-constant rounds, revisited. In Guy N. Rothblum and Hoeteck Wee, editors, *TCC 2023, Part IV*, volume 14372 of *LNCS*, pages 422–451. Springer, Cham, 2023.

[CFG+25] Ran Cohen, Pouyan Forghani, Juan Garay, Rutvik Patel, and Vassilis Zikas. Is it even possible? On the parallel composition of asynchronous MPC protocols. In Benny Applebaum and Huijia (Rachel) Lin, editors, *TCC 2025, Part I*, volume 16268 of *LNCS*, pages 33–63. Springer, Cham, December 2025.

[CGHZ16] Sandro Coretti, Juan Garay, Martin Hirt, and Vassilis Zikas. Constant-round asynchronous multi-party computation based on one-way functions. In *ASIACRYPT 2016, Part II*, volume 10032 of *LNCS*, pages 998–1021. Springer, Berlin, Heidelberg, 2016.

[CHOR22] Ran Cohen, Iftach Haitner, Eran Omri, and Lior Rotem. From fairness to full security in multiparty computation. *Journal of Cryptology*, 35(1):4, January 2022.

[Coh16] Ran Cohen. Asynchronous secure multiparty computation in constant time. In *PKC 2016, Part II*, volume 9615 of *LNCS*, pages 183–207. Springer, Berlin, Heidelberg, 2016.

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

[DM00] Yevgeniy Dodis and Silvio Micali. Parallel reducibility for information-theoretically secure computation. In Mihir Bellare, editor, *CRYPTO 2000*, volume 1880 of *LNCS*, pages 74–92. Springer, Berlin, Heidelberg, 2000.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[FS90] Uriel Feige and Adi Shamir. Witness indistinguishable and witness hiding protocols. In *22nd ACM STOC*, pages 416–426. ACM Press, 1990.

[GK90] Oded Goldreich and Hugo Krawczyk. On the composition of zero-knowledge proof systems. In Mike Paterson, editor, *Automata, Languages and Programming, 17th International Colloquium, ICALP90, Warwick University, England, UK, July 16-20, 1990, Proceedings*, volume 443 of *Lecture Notes in Computer Science*, pages 268–282. Springer, 1990.

[GLZS24] Vipul Goyal, Chen-Da Liu-Zhang, and Yifan Song. Towards achieving asynchronous MPC with linear communication and optimal resilience. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part VIII*, volume 14927 of *LNCS*, pages 170–206. Springer, Cham, 2024.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, 1987.

[Hal17] Shai Halevi. Homomorphic encryption. In *Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich*, pages 219–276. Springer, 2017.

[HNP05] Martin Hirt, Jesper Buus Nielsen, and Bartosz Przydatek. Cryptographic asynchronous multiparty computation with optimal resilience (extended abstract). In *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 322–340. Springer, Berlin, Heidelberg, 2005.

[HNP08] Martin Hirt, Jesper Buus Nielsen, and Bartosz Przydatek. Asynchronous multi-party computation with quadratic communication. In *ICALP 2008, Part II*, volume 5126 of *LNCS*, pages 473–485. Springer, Berlin, Heidelberg, 2008.

[IKOS07] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In David S. Johnson and Uriel Feige, editors, *39th ACM STOC*, pages 21–30. ACM Press, 2007.

[IKP+16] Yuval Ishai, Eyal Kushilevitz, Manoj Prabhakaran, Amit Sahai, and Ching-Hua Yu. Secure protocol transformations. In *CRYPTO 2016, Part II*, volume 9815 of *LNCS*, pages 430–458. Springer, Berlin, Heidelberg, 2016.

[KMTZ13] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Universally composable synchronous computation. In *TCC 2013*, volume 7785 of *LNCS*, pages 477–498. Springer, Berlin, Heidelberg, 2013.

[Lin03] Yehuda Lindell. Bounded-concurrent secure two-party computation without setup assumptions. In *35th ACM STOC*, pages 683–692. ACM Press, June 2003.

[LLR06] Yehuda Lindell, Anna Lysyanskaya, and Tal Rabin. On the composition of authenticated byzantine agreement. *Journal of the ACM*, 53(6):881–917, 2006.

[LR17] Yehuda Lindell and Tal Rabin. Secure two-party computation with fairness - A necessary design principle. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017, Part I*, volume 10677 of *LNCS*, pages 565–580. Springer, Cham, 2017.

[LSP82] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[MR91] Silvio Micali and Phillip Rogaway. Secure computation (abstract). In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 392–404. Springer, Berlin, Heidelberg, August 1991.

[MR92] Silvio Micali and Phillip Rogaway. Secure computation (unpublished manuscript), 1992. Preliminary version in [MR91]. (All references within refer to the unpublished manuscript.).

[Pas04] Rafael Pass. Bounded-concurrent secure multi-party computation with a dishonest majority. In László Babai, editor, *36th ACM STOC*, pages 232–241. ACM Press, June 2004.

[PR03] Rafael Pass and Alon Rosen. Bounded-concurrent secure two-party computation in a constant number of rounds. In *44th FOCS*, pages 404–415. IEEE Computer Society Press, October 2003.

[PSL80] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

[RB89] Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority (extended abstract). In *21st ACM STOC*, pages 73–85. ACM Press, 1989.

[Ros12] Mike Rosulek. Must you know the code of f to securely compute f? In *CRYPTO 2012*, volume 7417 of *LNCS*, pages 87–104. Springer, Berlin, Heidelberg, 2012.

[ZD22] Haibin Zhang and Sisi Duan. PACE: fully parallelizable BFT from reproposable byzantine agreement. In *ACM CCS 2022*, pages 3151–3164. ACM-CCS, 2022.