An extended abstract of this paper is published in the proceedings of the 10th Theory of Cryptography Conference — TCC 2013. This is the full version.

Universally Composable Synchronous Computation

Jonathan Katz[1], Ueli Maurer[2], Björn Tackmann[2], and Vassilis Zikas[3]*

[1] Dept. of Computer Science, University of Maryland
jkatz@cs.umd.edu
[2] Dept. of Computer Science, ETH Zürich, Switzerland
{maurer,bjoernt}@inf.ethz.ch
[3] Dept. of Computer Science, UCLA
vzikas@cs.ucla.edu

Abstract. In synchronous networks, protocols can achieve security guarantees that are not possible in an asynchronous world: they can simultaneously achieve *input completeness* (all honest parties' inputs are included in the computation) and *guaranteed termination* (honest parties do not "hang" indefinitely). In practice truly synchronous networks rarely exist, but synchrony can be emulated if channels have (known) bounded latency and parties have loosely synchronized clocks.

The widely-used framework of universal composability (UC) is inherently asynchronous, but several approaches for adding synchrony to the framework have been proposed. However, we show that the existing proposals do *not* provide the expected guarantees. Given this, we propose a novel approach to defining synchrony in the UC framework by introducing functionalities exactly meant to model, respectively, bounded-delay networks and loosely synchronized clocks. We show that the expected guarantees of synchronous computation can be achieved given these functionalities, and that previous similar models can all be expressed within our new framework.

1 Introduction

In synchronous networks, protocols can achieve both *input completeness* (all honest parties' inputs are included in the computation) and *guaranteed termination* (honest parties do not "hang" indefinitely). In contrast, these properties cannot simultaneously be ensured in an asynchronous world [7,17].

The traditional model for synchronous computation assumes that protocols proceed in rounds: the current round is known to all parties, and messages sent in some round are delivered by the beginning of the next round. While this is a strong model that rarely corresponds to real-world networks, the model is still useful since it can be *emulated* under the relatively mild assumptions of a known bound on the network latency and loose synchronization of the (honest) parties' clocks. In fact, it is fair to say that these two assumptions are exactly what is meant when speaking of "synchrony" in real-world networks.

The framework of *universal composability* (UC) [12] assumes, by default, completely asynchronous communication, where even eventual message delivery is not guaranteed. Protocol designers working in the UC setting are thus faced with two choices: either work in an asynchronous network and give up on input completeness [7] or guaranteed termination [28,15], or else modify the UC framework so as to incorporate synchronous communication somehow.

Several ideas for adding synchrony to the UC framework have been proposed. Canetti [10] introduced an ideal functionality $\mathcal{F}_{\text{SYN}}$ that was intended exactly to model synchronous communication in a general-purpose fashion. We prove in Section 5.1, however, that $\mathcal{F}_{\text{SYN}}$ does *not* provide the guarantees expected of a synchronous network. Nielsen [33] and Hofheinz and Müller-Quade [25] also propose ways of modeling synchrony with composition guarantees, but their approaches modify the foundations of the UC framework and are not sufficiently general to model, e.g., synchrony in an incomplete network, or the case when synchrony holds only in part of a network (say, because certain links do not have bounded delay while others do). It is fair to say that the proposed modifications to the UC framework are complex, and it is unclear

* Work done while the author was at the University of Maryland.

whether they adequately capture the intuitive real-world notion of synchrony. The timing model considered in [20,23,26] extends the notion of interactive Turing machines by adding a "clock tape." It comes closer to capturing intuition, but (as we show in Section 5.2) this model also does not provide the guarantees expected from a synchronous network. A similar approach is taken in [4], which modifies the reactive-simulatability framework of [6] by adding an explicit "time port" to each automaton. Despite the different underlying framework, this work is most similar to the approach we follow here in that it also captures both guaranteed termination and incomplete networks. Their approach, however, inherently requires changing the underlying model and is based on restricting the class of adversaries (both of which we avoid). Such modifications result in (at least) a reformulation of the composition theorem and proof.

Our approach and results. We aim for an intuitively appealing model that faithfully embeds the actual real-world synchrony assumptions into the standard UC framework. The approach we take is to introduce functionalities specifically intended to (independently) model the two assumptions of bounded network delay and loose clock synchronization. An additional benefit of separating the assumptions in this way is that we can also study the case when only one of the assumptions holds.

We begin by formally defining a functionality corresponding to (authenticated) communication channels with *bounded delay.* Unfortunately, this alone is not sufficient for achieving guaranteed termination. (Throughout, we will always want input completeness to hold.) Intuitively, this is because bounded-delay channels alone—without any global clock—only provide the same "eventual message delivery" guarantee of classical asynchronous networks [7,9]. It thus becomes clear that what is missing when only bounded-delay channels are available is some notion of *time.* To rectify this, we further introduce a functionality $\mathcal{F}_{\text{CLOCK}}$ that directly corresponds to the presence of loosely synchronized clocks among the parties. We then show that $\mathcal{F}_{\text{CLOCK}}$ together with eventual-delivery channels is also not sufficient, but that standard protocols can indeed be used to securely realize any functionality with guaranteed termination in a hybrid world where both $\mathcal{F}_{\text{CLOCK}}$ and bounded-delay (instead of just eventual delivery) channels are available.

Overall, our results show that the two functionalities we propose—meant to model, independently, bounded-delay channels and loosely synchronized clocks—enable us to capture exactly the security guarantees provided by traditional synchronous networks. Moreover, this approach allows us to make use of the original UC framework and composition theorem.

Guaranteed termination. We pursue an approach inspired by constructive cryptography [30,31] to model guaranteed termination. We describe the termination guarantee as a property of functionalities; this bridges the gap between the theoretical model and the realistic scenario where the synchronized clocks of the parties ensure that the adversary *cannot* stall the computation *even if he tries to (time will advance).* Intuitively, such a functionality does not wait for the adversary *indefinitely*; rather, the environment—which represents (amongst others) the parties as well as higher level protocols—can provide the functionality with sufficiently many activations to make it proceed and eventually produce outputs, irrespective of the adversary's strategy. This design principle is applied to both the functionality that shall be realized and to the underlying functionalities formalizing the (bounded-delay) channels and the (loosely synchronized) clocks.

We then require from a protocol to realize a functionality with this guaranteed termination property, given as hybrids functionalities that have the same type of property. In more detail, following the real-world/ideal-world paradigm of the security definition, for any real-world adversary, there must be an ideal-world adversary (or simulator) such that whatever the adversary achieves in the real world can be mimicked by the simulator in the ideal world. As the functionality guarantees to terminate and produce output for any simulator, no (real-world) adversary can stall the execution of a secure protocol indefinitely.

The environment in the UC framework can, at any point in time, provide output and halt the entire protocol execution. Intuitively, however, this corresponds to the environment (which is the distinguisher) *ignoring the remainder of the random experiment*, not the adversary *stalling the protocol execution.* Any environment $\mathcal{Z}$ can be transformed into an environment $\mathcal{Z}'$ that completes the execution and achieves (at least) the same advantage as $\mathcal{Z}$.

A "polling"-based notion of time. The formalization of time we use in this work is different from previous approaches [20,23,26,32]; the necessity for the different approach stems from the inherently asynchronous scheduling scheme of the original UC model. In fact, the order in which protocols are activated in this model is determined by the communication; a party will only be activated during the execution whenever this party receives either an input or a message.

Given this model, we formalize a clock as an ideal functionality that is available to the parties running a protocol and provides a means of synchronization: the clock "waits"until all *honest* parties signal that they are finished with their tasks. This structure is justified by the following observation: the guarantees that are given to parties in synchronous models are that each party will be activated in every time interval, and will be able to perform its local actions fast enough to finish before the deadline (and then it might "sleep" until the next time interval begins). A party's confirmation that it is ready captures exactly this guarantee. As this model differentiates between honest and dishonest parties, we have to carefully design functionalities and protocols such that they do not allow *excessive* capabilities of detecting dishonest behavior. Still, the synchrony guarantee inherently *does* provide *some* form of such detections (e.g., usually by a time-out while waiting for messages, the synchrony of the clocks and the bounded delay of the channels guarantee that messages among honest parties always arrive on time).

Our notion of time allows modeling both composition of protocols that run mutually asynchronously, by assuming that each protocol has its own independent clock, as well as mutually synchronous, e.g. lock-step, composition by assuming that all protocols use the same clock.

Organization of the paper. In Section 2, we include a brief description of the UC model [12] and introduce the necessary notation and terminology. In Section 3, we review the model of completely asynchronous networks, describe its limitations, and introduce a functionality modeling bounded-delay channels. In Section 4, we introduce a functionality $\mathcal{F}_{\text{CLOCK}}$ meant to model loose clock synchronization and explore the guarantees it provides. Further, we define *computation with guaranteed termination* within the UC framework, and show how to achieve it using $\mathcal{F}_{\text{CLOCK}}$ and bounded-delay channels. In Section 5, we revisit previous models for synchronous computation.

2 Preliminaries

Simulation-based security. Most general security frameworks are based on the real-world/ideal-world paradigm: In the real world, the parties execute the protocol using channels as defined by the model. In the ideal world, the parties securely access an ideal functionality $\mathcal{F}$ that obtains inputs from the parties, runs the program that specifies the task to be achieved by the protocol, and returns the resulting outputs to the parties. Intuitively, a protocol *securely realizes* the functionality $\mathcal{F}$ if, for any real-world adversary $\mathcal{A}$ attacking the protocol execution, there is an ideal-world adversary $\mathcal{S}$, also called the *simulator*, that emulates $\mathcal{A}$'s attack. The simulation is good if no distinguisher $\mathcal{Z}$—often called the *environment*—which interacts, in a well defined manner, with the parties and the adversary/simulator, can distinguish between the two worlds.

The advantage of such security definitions is that they satisfy strong composability properties. Let π_1 be a protocol that securely realizes a functionality $\mathcal{F}_1$. If a protocol π_2, using the functionality $\mathcal{F}_1$ as a subroutine, securely realizes a functionality $\mathcal{F}_2$, then the protocol $\pi_2^{\pi_1/\mathcal{F}_1}$, where the calls to $\mathcal{F}_1$ are replaced by invocations of π_1, securely realizes $\mathcal{F}_2$ (without calls to $\mathcal{F}_1$). Therefore, it suffices to analyze the security of the simpler protocol π_2 in the $\mathcal{F}_1$-*hybrid* model, where the parties run π_2 with access to the ideal functionality $\mathcal{F}_1$. A detailed treatment of protocol composition appears in, e.g., [6,11,12,18,31].

Model of computation. All security models discussed in this work are based on or inspired by the UC framework [12]. The definitions are based on the simulation paradigm, and the entities taking part in the execution (protocol machines, functionalities, adversary, and environment) are described as *interactive Turing machines* (ITMs). The execution is an interaction of *ITM instances* (ITIs) and is initiated by the environment that provides input to and obtains output from the protocol machines, and also communicates with the adversary. The adversary has access to the ideal functionalities in the hybrid models, in some models it also serves as a network among the protocol machines. During the execution, the ITIs are activated one-by-one, where the exact order of the activations depends on the considered model.

Notation, conventions, and specifics of UC '05. We consider protocols that are executed among a certain set of players $\mathcal{P}$, often referred to as the *player set*, where every $p_i \in \mathcal{P}$ formally denotes a unique party ID. A protocol execution involves the following types of ITMs: the environment $\mathcal{Z}$, the adversary $\mathcal{A}$, the protocol machine π, and (possibly) ideal functionalities $\mathcal{F}_1, \ldots, \mathcal{F}_m$. The contents of the environment's output tape after the execution is denoted by the random variable $\textsc{exec}_{\pi,\{\mathcal{F}_1,\ldots,\mathcal{F}_m\},\mathcal{A},\mathcal{Z}}(k, z)$, where $k \in \mathbb{N}$ is the *security parameter* and $z \in \{0, 1\}^*$ is the input to the environment $\mathcal{Z}$.[1] We use the convention that if no protocol π is specified in the above notation, then we take as π the "dummy" protocol which forwards all its inputs to and its outputs from the ideal functionality (see [10]). We say that a protocol π securely realizes $\mathcal{F}$ in the $\mathcal{F}'$-hybrid model if

$$\forall \mathcal{A}\ \exists \mathcal{S}\ \forall \mathcal{Z} : \textsc{exec}_{\pi,\mathcal{F}',\mathcal{A},\mathcal{Z}} \approx \textsc{exec}_{\mathcal{F},\mathcal{S},\mathcal{Z}},$$

where "$\approx$" denotes indistinguishability of the respective distribution ensembles.

As in [10], the statement "the functionality sends a *(private) delayed output* y to party p_i" describes the following process: the functionality requests the adversary's permission to output y to party p_i (without leaking the value y); as soon as the adversary agrees, the output y is delivered. The statement "the functionality sends a *public delayed output* y to party p_i" corresponds to the same process, where the permission request also includes the full message y.

We often prefer to have flexible functionalities that we parametrize by certain functions or values. If such a parameter of a functionality is considered as being fixed in the code of the functionality, we denote it in superscript to the name of the (parametrized) functionality. If such a parameter is supposed to be determined by the session ID of the functionality (such as the IDs of the parties that participate in the protocol), we write it in parentheses behind the name of the functionality. We usually omit the session ID from the description of our functionalities; different instances behave independently.

In the UC framework [12], the potential dishonesty of parties is modeled by the fact that the adversary $\mathcal{A}$ may corrupt protocol machines (adaptively) during the execution by sending them a special (`corrupt`) message. We only consider Byzantine party-corruption, which means that the party sends its entire current local state to $\mathcal{A}$ and, in all future activations, follows $\mathcal{A}$'s instructions. We generally assume that parties do not delete state, and hence the entire execution history can be derived from the party's current state. For more details on party corruption, see [10, Section 4.1].

All our functionalities $\mathcal{F}$ use *standard (adaptive) corruption* as defined in [10]: At any point in time, $\mathcal{F}$ will accept messages of the type (`corrupt`, p_i) for $p_i \in \mathcal{P}$ from the adversary $\mathcal{A}$, in which case $\mathcal{F}$ marks the party p_i as corrupted and replies with (`ok`) and all previous inputs and outputs of p_i. Whenever a corrupted party p_i inputs a message x, the functionality $\mathcal{F}$ sends a message (`input`, p_i, x) to $\mathcal{A}$ and accepts as a reply (`replace`, $\tilde{x}$), before continuing the execution with input $\tilde{x}$. Whenever $\mathcal{F}$ would send an output message y to a corrupted party p_i, it sends a message (`output`, p_i, y) to $\mathcal{A}$ and accepts as a reply (`replace`, $\tilde{y}$). $\mathcal{F}$ then outputs $\tilde{y}$ to p_i. We use the symbol $\mathcal{H}$ to denote (at the current point of the execution) the subset of $\mathcal{P}$ for which no message (`corrupt`, p_i) has been sent.[2] At any point in the execution, we denote by $\mathcal{H}$ the set of "honest" parties that have not (yet) been corrupted. Finally, all of our functionalities use a player set $\mathcal{P}$ that is fixed when the functionality is instantiated, and each functionality has a session ID which is of the form $sid = (\mathcal{P}, sid')$ with $sid' \in \{0, 1\}^*$.

The functionalities in our model and their interpretation are specific to the model of [10] in that they exploit some of the mechanics introduced there, which we recall here. First, the order of activations is strictly defined by the model: whenever an ITI sends a message to some other ITI, the receiving ITI will immediately be activated and the sending ITI will halt. If some ITI halts without sending a message, the "master scheduler," the environment $\mathcal{Z}$, will become active. This scheme allows to model guaranteed termination since the adversary *cannot* prevent the invocation of protocol machines. Second, efficiency is defined as a "reactive" type of polynomial time: the number of steps that an ITI performs is bounded by a polynomial in the security parameter *and* (essentially) the length of the inputs obtained by this ITI. Consequently, the environment can continuously provide "run-time" to protocol machines to make them poll, e.g., at a bounded-delay or eventual-delivery channel. Our modeling of eventual delivery fundamentally relies on this fact.

[1] For the details of the execution, we follow the description in [10]. The slight change of notation has been made to better represent the assumed hybrids with a view towards the reductions.

[2] Corruption in [10] starts by $\mathcal{Z}$ issuing such a message to $\mathcal{A}$. From this point in time, $\mathcal{A}$ is allowed by the control function to send corruption requests (`corrupt`, p_i) to the ideal functionalities or (`corrupt`) to the protocol machine π_i.

3 Synchronous Protocols in an Asynchronous Network

Protocols in asynchronous networks cannot achieve input completeness and guaranteed termination simultaneously [7,17]. Intuitively, the reason is that honest parties cannot distinguish whether a message has been delayed—and to satisfy input completeness they should wait for this message—or whether the sender is corrupted and did not send the message—and for guaranteed termination they should proceed. In fact, there are two main network models for asynchronous protocols: on the one hand, there are fully asynchronous channels that do not at all guarantee delivery [10,15]; on the other hand, there are channels where delivery is guaranteed and the delay might be bounded by a publicly known constant or unknown [7]. In the following, we formalize the channels assumed in each of the two settings as functionalities in the UC framework and discuss how they can be used by round-based, i.e., synchronous, protocols. The results presented here formally confirm—in the UC framework—facts about synchrony assumptions that are known or folklore in the distributed computing literature.

3.1 Fully Asynchronous Network

The communication in a fully asynchronous network where messages are not guaranteed to be delivered is modeled by the functionality $\mathcal{F}_{\text{SMT}}$ from [10], which involves a sender, a receiver, and the adversary. Messages input by the sender p_s are immediately given to the adversary, and delivered to the receiver p_r only after the adversary's approval. Different privacy guarantees are formulated by a so-called leakage function $\ell(\cdot)$ that determines the information leaked during the transmission if both p_s and p_r are honest. In particular, the authenticated channel $\mathcal{F}_{\text{AUTH}}$ is modeled by $\mathcal{F}_{\text{SMT}}$ parametrized by the identity function $\ell(m) = m$, and the secure channel $\mathcal{F}_{\text{SEC}}$ is modeled by $\mathcal{F}_{\text{SMT}}$ with the constant function $\ell(m) = \perp$.[3] An important property of $\mathcal{F}_{\text{SMT}}$ is *adaptive message replacement*: the adversary can, depending on the leaked information, corrupt the sender and replace the sent message. A formal description of the functionality (restated in our conventions) is given below.

Functionality $\mathcal{F}^{\ell}_{\text{SMT}}(p_s, p_r)$ as in [10]

The functionality $\mathcal{F}^{\ell}_{\text{SMT}}$ is parametrized by a leakage function $\ell : \{0,1\}^* \to \{0,1\}^*$.

- Upon receiving $(\texttt{send}, m)$ from p_s, send $(\texttt{sent}, \ell(m))$ to the adversary and provide private delayed output $(\texttt{sent}, m)$ to p_r.
- Upon receiving $(\texttt{replace}, m')$ from the adversary, if p_s is corrupted and no output has been given to p_r, then output $(\texttt{sent}, m')$ to p_r.

Canetti et al. [15] showed that, in this model and assuming a common reference string, any (well-formed) functionality can be realized, without guaranteed termination. Moreover, a combination of the results of Kushilevitz, Lindell, and Rabin [28] and Asharov and Lindell [1] show that appropriate modifications of the protocols from the seminal works of Ben-Or, Goldwasser, and Wigderson [8] and Chaum, Crépeau, and Damgård [16] (for unconditional security) or the work by Goldreich, Micali, and Wigderson [22] (for computational security)—all of which are designed for the synchronous setting—are sufficient to achieve general secure computation without termination in this asynchronous setting, under the same conditions on corruption thresholds as stated in [8,16,22].

The following lemma formalizes the intuition that a fully asynchronous network is insufficient for terminating computation, i.e., computation which cannot be stalled by the adversary. For a functionality $\mathcal{F}$, denote by $[\mathcal{F}]^{\texttt{NT}}$ the *non-terminating relaxation* of $\mathcal{F}$ defined as follows: $[\mathcal{F}]^{\texttt{NT}}$ behaves as $\mathcal{F}$, but whenever $\mathcal{F}$ outputs a value to some honest party, $[\mathcal{F}]^{\texttt{NT}}$ provides this output in a delayed manner (see Section 2). More formally, we show that there are functionalities $\mathcal{F}$ that are not realizable in the $\mathcal{F}_{\text{SMT}}$-hybrid model, but their delayed relaxations $[\mathcal{F}]^{\texttt{NT}}$ are. This statement holds even for stand-alone security, i.e., for environments that do not interact with the adversary during the protocol execution. Additionally, the impossibility applies to all *non-trivial*, i.e., not locally computable, functionalities (see [27]) with guaranteed termination as defined in Section 4. While the lemma is implied by the more general Lemma 5, we describe the proof idea for this simpler case below.

[3] This correspond to *ideally* secure channels as defined in [8,16]. Yet, as such a channel cannot be realized without further assumptions, one typically resorts to the length function $\ell(m) = |m|$, see [14].

Lemma 1. *There is a functionality $\mathcal{F}^\ell$ such that $[\mathcal{F}^\ell]^{\mathtt{NT}}$ can be realized in the $\mathcal{F}^\ell_{\text{SMT}}$-hybrid model, but $\mathcal{F}^\ell$ cannot be realized.*

Proof (sketch). We describe a functionality $\mathcal{F}^\ell$ that satisfies the conditions of the lemma: $\mathcal{F}^\ell$ is a two party functionality that corresponds to a channel between a sender p_s and a receiver p_r. $\mathcal{F}^\ell$ behaves as $\mathcal{F}^\ell_{\text{SMT}}$, but does not necessarily wait for the adversary's approval for delivering a message. Instead, upon receiving a special $(\mathtt{fetch})$-message from the receiver p_r, $\mathcal{F}^\ell$ outputs $(\mathtt{sent}, y)$ to p_r, where $y = m$ if the sender has input the message m, and $y = \perp$ (i.e., a default value), otherwise. The functionality $\mathcal{F}^\ell$ is described in detail in the following:

Functionality $\mathcal{F}^\ell(p_s, p_r)$

The functionality $\mathcal{F}^\ell$ is parametrized by a leakage function $\ell : \{0,1\}^* \to \{0,1\}^*$.

– Upon receiving $(\mathtt{send}, m)$ from p_s, send $(\mathtt{sent}, \ell(m))$ to the adversary and provide a private delayed output $(\mathtt{sent}, m)$ to p_r.
– Upon receiving $(\mathtt{fetch})$ from p_r: if m was received from p_s and was not yet delivered to p_r, then send $(\mathtt{sent}, m)$ to p_r, otherwise send $\perp$ to p_r.
– Upon receiving $(\mathtt{replace}, m')$ from the adversary, if p_s is corrupted and no output has been given to p_r, then output $(\mathtt{sent}, m')$ to p_r.

We show that $\mathcal{F}^\ell$ cannot be realized from $\mathcal{F}^\ell_{\text{SMT}}$. Indeed, consider the following environment $\mathcal{Z}$ with the dummy adversary (which acts as a forwarder between the environment and the functionality): both p_s and p_r are honest, and $\mathcal{Z}$ gives a uniformly chosen input-message $m \in_R \{0,1\}$ to p_s; as soon as $\mathcal{Z}$ receives from the adversary the leaked value $\ell(m)$ from the channel, it activates p_r for fetching a message from the channel (formally, $\mathcal{Z}$ provides input $(\mathtt{fetch})$ to p_r). The (honest) party p_r is expected to answer with a message $(\mathtt{sent}, m')$ as received from the channel.[4] Note that any action other than reporting a message, i.e., producing no output or sending a message to p_s (through the $\mathcal{F}^\ell_{\text{SMT}}$-channel in the other direction) is detected by $\mathcal{Z}$, as in both cases $\mathcal{Z}$ is activated before receiving output from p_r, and can be used for distinguishing. Using the information whether or not m' equals m, the environment can decide whether it witnessed an ideal-world or a real-world execution. In the ideal-world case (where the parties interact via $\mathcal{F}^\ell$) p_r's output will always be $m' = m$. In the real world, as the adversary has not instructed the channel $\mathcal{F}_{\text{SMT}}$ to deliver the message, the (honest) p_r has no information on m. Because m is chosen uniformly from $\{0,1\}$, the probability that $m' = m$ is $1/2$, which provides a noticeable distinguishing advantage to $\mathcal{Z}$.

For the reduction of $[\mathcal{F}^\ell]^{\mathtt{NT}}$ to $\mathcal{F}^\ell_{\text{SMT}}$, we observe that $[\mathcal{F}^\ell]^{\mathtt{NT}}$ can be realized from $\mathcal{F}^\ell_{\text{SMT}}$ using the dummy protocol φ. The simulator $\mathcal{S}$ uses the adversary $\mathcal{A}$ attacking the execution of φ in a black-box manner and behaves as follows: Throughout the simulation, $\mathcal{S}$ forwards all messages sent between $\mathcal{A}$ and $\mathcal{Z}$; whenever $\mathcal{A}$ requests to corrupt a party, $\mathcal{S}$ requests to corrupt this party in the ideal world (simulating the internal state is easy); furthermore, the simulator forwards to $\mathcal{A}$ all the messages that are sent from $[\mathcal{F}^\ell]^{\mathtt{NT}}$, except for the notification about the $(\mathtt{fetch})$-message sent (to $[\mathcal{F}^\ell]^{\mathtt{NT}}$) by p_r. ($\mathcal{S}$ is informed about each $(\mathtt{fetch})$-message, because $[\mathcal{F}^\ell]^{\mathtt{NT}}$ issues a delayed output as a response to each such message. As such a message will make $\mathcal{F}^\ell_{\text{SMT}}$ halt and return the control to $\mathcal{Z}$, $\mathcal{S}$ will react in the same way.) It is straightforward to verify that the above $\mathcal{S}$ is a good simulator for the adversary $\mathcal{A}$. □

3.2 Eventual-Delivery Channels

A stronger variant of asynchronous communication provides the guarantee that messages will be delivered *eventually*, independent of the adversary's strategy [7]. The functionality $\mathcal{F}_{\text{ED-SMT}}$ captures this guarantee, following the principle described in Section 1: The receiver can enforce delivery of the message using "fetch" requests to the channel. The potential delay of the channel is modeled by ignoring a certain number D of such requests before delivering the actual message to p_r; to model the fact that the delay might be arbitrary, we allow the adversary to repeatedly increase the value of D during the computation. Yet, the delay that $\mathcal{A}$ can impose is bounded by $\mathcal{A}$'s running time.[5] The fact that this models eventual delivery

[4] Recall that in UC a party cannot become corrupted without the environments permission, hence the simulator cannot stop p_r from reporting the messages it receives in the ideal world.

[5] This is enforced by specifying the delay in unary notation.

utilizes the "reactive" definition of efficiency in [10]: after the adversary determined the delay D for a certain message, the environment can still provide the protocol machines of the honest parties with sufficiently many activations to retrieve the message from the channel. The eventual delivery channel $\mathcal{F}_{\text{ED-SMT}}$ is, like $\mathcal{F}_{\text{SMT}}$, parametrized by a leakage function $\ell(\cdot)$. We use $\mathcal{F}_{\text{ED-AUTH}}$ and $\mathcal{F}_{\text{ED-SEC}}$ to denote the corresponding authenticated and secure eventual-delivery channel, respectively.

Functionality $\mathcal{F}^{\ell}_{\text{ED-SMT}}(p_s, p_r)$

Initialize $M := \perp$ and $D := 0$.

- Upon receiving a message m from p_s, set $D := 1$ and $M := m$ and send $(\texttt{sent}, \ell(M))$ to the adversary.
- Upon receiving a message $(\texttt{fetch})$ from p_r:
  1. Set $D := D - 1$.
  2. If $D = 0$ then send $(\texttt{sent}, M)$ to p_r (otherwise no message is sent[a]).
- Upon receiving a message $(\texttt{delay}, T)$ from the adversary, if T encodes a natural number in unary notation, then set $D := D + T$ and return $(\texttt{delay-set})$ to the adversary; otherwise ignore the message.
- Upon receiving $(\texttt{replace}, m', T')$ from the adversary, if p_s is corrupted, $D > 0$, and the delay T' is valid, then set $D := T'$ and set $M := m'$.

[a] Following [10], this means that $\mathcal{Z}$ is activated

Channels with eventual delivery are strictly stronger than fully asynchronous communication in the sense of Section 3.1. Indeed, the proof of Lemma 1 extends to the case where $\mathcal{F}^{\ell}$ is the eventual-delivery channel $\mathcal{F}^{\ell}_{\text{ED-SMT}}$: the simulator can delay the delivery of the message only by a polynomial number of steps, and the environment can issue sufficiently many queries at the receiver's interface.

As with fully asynchronous channels, one can use channels with eventual delivery to achieve secure computation without termination. Additionally, however, eventual-delivery channels allow for protocols which are guaranteed to (eventually) terminate, at the cost of violating input completeness. For instance, the protocol of Ben-Or, Canetti, and Goldreich [7] securely realizes any functionality where the inputs of up to $\frac{n}{4}$ parties might be ignored. Yet, the eventual-delivery channels, by themselves, do not allow to compute functionalities with strong termination guarantees, and the result of Lemma 1 holds even if we replace $\mathcal{F}^{\ell}_{\text{SMT}}$ by $\mathcal{F}^{\ell}_{\text{ED-SMT}}$. This is stated in the following lemma, which again translates to both stand-alone security and to arbitrary functionalities that are not locally computable, and is again implied by Lemma 5.

Lemma 2. *There are functionalities $\mathcal{F}^{\ell}$ such that $[\mathcal{F}^{\ell}]^{\texttt{NT}}$ can be realized in the $\mathcal{F}^{\ell}_{\text{ED-SMT}}$-hybrid model, but $\mathcal{F}^{\ell}$ cannot be realized.*

Proof (sketch). For the functionality $\mathcal{F}^{\ell}$ introduced in the proof of Lemma 1 (see page 6), we show that $\mathcal{F}^{\ell}$ cannot be realized from $\mathcal{F}^{\ell}_{\text{ED-SMT}}$, but $[\mathcal{F}^{\ell}]^{\texttt{NT}}$ can. Indeed, the dummy protocol φ realizes $[\mathcal{F}^{\ell}]^{\texttt{NT}}$ based on $\mathcal{F}^{\ell}_{\text{ED-SMT}}$: The simulator $\mathcal{S}$ uses the hybrid model adversary $\mathcal{A}$ in a black-box manner and behaves as follows. Throughout the simulation, $\mathcal{S}$ forwards all the messages sent between $\mathcal{A}$ and $\mathcal{Z}$; furthermore, whenever $\mathcal{A}$ requests to corrupt a party, $\mathcal{S}$ requests to corrupt this party in the ideal world (the internal state of the protocol machines can be simulated easily). From the messages $\mathcal{A}$ sends to $\mathcal{F}^{\ell}_{\text{ED-SMT}}$, $\mathcal{S}$ can learn the actual delay T that $\mathcal{A}$ wishes to put on the message transmission. In order to apply the same delay on the $[\mathcal{F}^{\ell}]^{\texttt{NT}}$-hybrid world, $\mathcal{S}$ stalls the output of any message sent through $[\mathcal{F}^{\ell}]^{\texttt{NT}}$ (recall that $[\mathcal{F}^{\ell}]^{\texttt{NT}}$ allows the simulator to deliver output-messages if and when he wishes, as they are issued in a delayed manner) until T $(\texttt{fetch})$-messages have been sent from p_r to $[\mathcal{F}^{\ell}]^{\texttt{NT}}$ ($\mathcal{S}$ is informed about each $(\texttt{fetch})$-message, because $[\mathcal{F}^{\ell}]^{\texttt{NT}}$ issues a delayed output as a response to each such message). The above $\mathcal{S}$ is a good simulator for the adversary $\mathcal{A}$.

We next turn to proving the impossibly of realizing $\mathcal{F}^{\ell}$ from $\mathcal{F}^{\ell}_{\text{ED-SMT}}$. The idea is similar to the impossibility proof of Lemma 1. In particular, consider the dummy adversary $\mathcal{A}$ and the environment $\mathcal{Z}$ that corrupts no party and has the following distinguishing strategy: $\mathcal{Z}$ provides as input to p_s a uniformly random bit m. As soon as it receives the value $(\texttt{sent}, \ell(m))$ from the functionality (forwarded by $\mathcal{A}$), $\mathcal{Z}$ instructs $\mathcal{A}$ to give delay $T = 2$ to $\mathcal{F}^{\ell}_{\text{ED-SMT}}$. Subsequently, $\mathcal{Z}$ activates p_r with input $(\texttt{fetch})$; as a result, if $\mathcal{Z}$ is witnessing the ideal-world execution, he will receive the message $(\texttt{sent}, m)$ from p_r (note that $\mathcal{F}^{\ell}$ does not allow the simulator to stall this reply); hence, in the real world, the protocol of p_r has to also output $(\texttt{sent}, m')$ for some m' to $\mathcal{Z}$. Indeed, the other choices of p_r are to not produce any output, or to send (`fetch`) to $\mathcal{F}^{\ell}_{\text{ED-SMT}}$, or to invoke the $\mathcal{F}^{\ell}_{\text{ED-SMT}}$ in the other direction (i.e., as a sender); in all three cases the environment is activated before receiving output which allows it to detect that it is witnessing the real-world execution. However, as p_r has no information on m, the probability that $m' = m$ is $1/2$, which allows $\mathcal{Z}$ to distinguish with noticeable advantage. □

3.3 Bounded-Delay Channels with a Known Upper Bound

Bounded-delay channels are described by a functionality $\mathcal{F}_{\text{BD-SMT}}$ that is similar to $\mathcal{F}_{\text{ED-SMT}}$ but parametrized by a positive constant δ bounding the delay that the adversary can impose ($\delta = 1$ means immediate delivery). In more detail, the functionality $\mathcal{F}_{\text{BD-SMT}}$ works as $\mathcal{F}_{\text{ED-SMT}}$, but queries of the adversary that lead to an accumulated delay of $T > \delta$ are ignored.[6] A formal specification of $\mathcal{F}_{\text{BD-SMT}}$ is given in the following:

Functionality $\mathcal{F}^{\delta,\ell}_{\text{BD-SMT}}(p_s, p_r)$

Initialize $M := \perp$ and $D := 1$, and $\hat{D} := 1$.

- Upon receiving a message m from p_s, set $D := 1$ and $M := m$ and send $(\texttt{sent}, \ell(M))$ to the adversary.
- Upon receiving a message (`fetch`) from p_r:
  1. Set $D := D - 1$.
  2. If $D = 0$, then send $(\texttt{sent}, M)$ to p_r.
- Upon receiving $(\texttt{delay}, T)$ from the adversary, if $\hat{D} + T \leq \delta$, then set $D := D + T$, $\hat{D} := \hat{D} + T$, and return (`delay-set`) to the adversary; otherwise ignore the message.
- Upon receiving $(\texttt{replace}, m', T')$ from the adversary, if p_s is corrupted, $D > 0$, and the delay T' is valid, then set $D := T'$ and set $M := m'$.

In reality, a channel with latency δ' is at least as useful as one with latency $\delta > \delta'$. Our formulation of bounded-delay channels is consistent with this intuition: for any $0 < \delta' < \delta$, $\mathcal{F}^{\delta,\ell}_{\text{BD-SMT}}$ can be UC-realized in the $\mathcal{F}^{\delta',\ell}_{\text{BD-SMT}}$-hybrid model. Indeed, the simple $\mathcal{F}^{\delta',\ell}_{\text{BD-SMT}}$-hybrid protocol that drops $\delta - \delta'$ (`fetch`)-queries realizes $\mathcal{F}^{\delta,\ell}_{\text{BD-SMT}}$; the simulator also increases the delay appropriately. The converse is not true in general: channels with smaller upper bound on the delay are *strictly* stronger when termination is required. This is formalized in the following lemma, which again extends to both stand-alone security and to not locally computable functionalities with guaranteed termination as in Section 4.

Lemma 3. *For any $0 < \delta' < \delta$, the functionality $[\mathcal{F}^{\delta',\ell}_{\text{BD-SMT}}]^{\texttt{NT}}$ can be realized in the $\mathcal{F}^{\delta,\ell}_{\text{BD-SMT}}$-hybrid model, but $\mathcal{F}^{\delta',\ell}_{\text{BD-SMT}}$ cannot be realized.*

Proof (sketch). We first prove the impossibility result, namely that $\mathcal{F}^{\delta',\ell}_{\text{BD-SMT}}$ cannot be realized in the $\mathcal{F}^{\delta,\ell}_{\text{BD-SMT}}$-hybrid model. The idea is similar to the proof of Lemma 2. In particular, consider the dummy adversary $\mathcal{A}$ and the environment $\mathcal{Z}$ that corrupts no party and provides as input to p_s a uniformly random bit m. After receiving the value $(\texttt{sent}, \ell(m))$ from the functionality (forwarded by $\mathcal{A}$), $\mathcal{Z}$ instructs $\mathcal{A}$ to choose the delay $T = \delta$ at the functionality. In the real ($\mathcal{F}^{\delta,\ell}_{\text{BD-SMT}}$-hybrid) world, this means that the channel will ignore the next δ (`fetch`)-requests from p_r; however, in the ideal world, the functionality $\mathcal{F}^{\delta',\ell}_{\text{BD-SMT}}$ will ignore at most $\delta' < \delta$ (`fetch`)-requests from p_r. Subsequently, $\mathcal{Z}$ activates p_r with input (`fetch`) δ' times; as a result, if $\mathcal{Z}$ is witnessing the ideal-world execution, it will receive the message $(\texttt{sent}, m)$ from p_r ($\mathcal{F}^{\delta',\ell}_{\text{BD-SMT}}$ does not allow the simulator to further delay this reply); hence, in the real world, the protocol of p_r has to also output $(\texttt{sent}, m')$ for some m' to $\mathcal{Z}$. Indeed, the other choices of p_r are to not produce any output, or to send (`fetch`) to $\mathcal{F}^{\delta,\ell}_{\text{BD-SMT}}$, or to use the channel in the opposite direction (i.e., as a sender); in all three cases, $\mathcal{Z}$ is activated before p_r receives the next output, which allows $\mathcal{Z}$ to detect that it is witnessing the real-world execution. However, as p_r has no information on m, the probability that $m' = m$ is $1/2$, which results in a noticeable distinguishing advantage for $\mathcal{Z}$.

[6] Previous versions of this functionality included a query (`LearnBound`) via which a higher-level protocol could learn the guaranteed maximum delay of the channel. This is not needed, as following [10] the higher-level protocol also contains the code of the hybrid functionalities, so the protocol can be assumed to be aware of the maximum delay.

The reduction of $[\mathcal{F}^{\delta',\ell}_{\text{BD-SMT}}]^{\texttt{NT}}$ to $\mathcal{F}^{\delta,\ell}_{\text{BD-SMT}}$ is done along the lines of the proof of Lemma 2, but we have to take care of some details occurring due to the different bounds δ and δ'. We show that $[\mathcal{F}^{\delta',\ell}_{\text{BD-SMT}}]^{\texttt{NT}}$ can be realized from $\mathcal{F}^{\delta,\ell}_{\text{BD-SMT}}$ using the simple protocol $\pi_{\delta'}$ that, on every ($\texttt{fetch}$)-query, activates the adversary. On the other hand, upon an activation from the adversary, the protocol issues a ($\texttt{fetch}$)-query to $\mathcal{F}^{\delta,\ell}_{\text{BD-SMT}}$ and outputs the result. The simulator $\mathcal{S}$ works as follows: It simulates copies of $\mathcal{A}$, $\mathcal{F}^{\delta,\ell}_{\text{BD-SMT}}$, and the protocol machines internally and instructs the (ideal) channel to proceed without additional delay ($[\mathcal{F}^{\delta',\ell}_{\text{BD-SMT}}]^{\texttt{NT}}$ is defined to make only delayed outputs). $\mathcal{S}$ analyzes the communication among the simulated $\mathcal{A}$ and $\mathcal{F}^{\delta,\ell}_{\text{BD-SMT}}$ to determine the chosen delay, and examines the interaction between the protocol machines, $\mathcal{A}$, and the simulated channel. From this communication, $\mathcal{S}$ can easily deduce when the receiving protocol machine would provide output and can instruct the ideal channel to deliver the message. □

The proof technique already suggests that bounded-delay channels, without additional assumptions such as synchronized clocks, are not sufficient for terminating computation. While Lemma 3 only handles the case where the assumed channel has a strictly positive upper bound on the delay, the (more general) impossibility in Lemma 5 holds even for *instant-delivery* channels, i.e., bounded-delay channels which become ready to deliver as soon as they get input from the sender.

In the remainder of this paper, we use instant-delivery channels, i.e., $\mathcal{F}^{\delta,\ell}_{\text{BD-SMT}}$ with $\delta = 1$; however, our results easily extend to arbitrary values of δ. To simplify notation, we completely omit the delay parameter, i.e., we write $\mathcal{F}^{\ell}_{\text{BD-SMT}}$ instead of $\mathcal{F}^{1,\ell}_{\text{BD-SMT}}$. Furthermore, we use $\mathcal{F}_{\text{BD-AUTH}}$ and $\mathcal{F}_{\text{BD-SEC}}$ to denote the corresponding authenticated and secure bounded-delay channel with $\delta = 1$, respectively.

4 Computation with Guaranteed Termination

Assuming bounded-delay channels is not, by itself, sufficient for achieving both input completeness and termination. In this section, we introduce the functionality $\mathcal{F}_{\text{CLOCK}}$ that, together with the bounded-delay channels $\mathcal{F}^{\ell}_{\text{BD-SMT}}$, allows synchronous protocols to satisfy both properties simultaneously. In particular, we define what it means for a protocol to UC-realize a given multi-party function *with guaranteed termination*, and show how $\{\mathcal{F}_{\text{CLOCK}}, \mathcal{F}^{\ell}_{\text{BD-SMT}}\}$-protocols can satisfy this definition.

4.1 The Synchronization Functionality

To motivate the functionality $\mathcal{F}_{\text{CLOCK}}$, we examine how synchronous protocols in reality use the assumptions of bounded-delay (with a known upper bound) channels and synchronized clocks to satisfy the input-completeness and the termination properties simultaneously: they assign to each round a time slot that is long enough to incorporate the time for computing and sending all next-round messages, plus the network delay. The fact that their clocks are (loosely) synchronized allows the parties to decide (without explicit communication) whether or not all honest parties have finished all their operations for some round. Note that it is sufficient, at the cost of having longer rounds, to assume that the clocks are not advancing in a fully synchronized manner but there is an known upper bound on the maximum clock drift [23,26,32].

The purpose of $\mathcal{F}_{\text{CLOCK}}$ is to provide the described functionality to (UC) protocols. But as $\mathcal{F}_{\text{CLOCK}}$ is an ordinary (UC) functionality, it has no means of knowing whether or not a party has finished its intended operations for a certain round. This problem is resolved by having the parties signal their round status (i.e, whether or not they are "done" with the current round) to $\mathcal{F}_{\text{CLOCK}}$. In particular, $\mathcal{F}_{\text{CLOCK}}$ keeps track of the parties' status in a vector $(d_1, \ldots, d_n)$ of indicator bits, where $d_i = 1$ if p_i has signaled that it has finished all its actions for the current round and $d_i = 0$, otherwise. As soon as $d_i = 1$ for all $p_i \in \mathcal{H}$, $\mathcal{F}_{\text{CLOCK}}$ resets $d_i = 0$ for all $p_i \in \mathcal{P}$. In addition to the notifications, any party p_i can send a synchronization request to $\mathcal{F}_{\text{CLOCK}}$, which is answered with d_i. A party p_i that observes that d_i has switched can conclude that all honest parties have completed their respective duties.[7] As $\mathcal{F}_{\text{CLOCK}}$ does *not* wait for signals from corrupted parties, $\mathcal{F}_{\text{CLOCK}}$ cannot be realized based on well-formed functionalities. Nevertheless, as discussed above, in reality time *does* offer this functionality to synchronous protocols.

[7] For arbitrary protocols, the functionality offers too strong guarantees. Hence, we restrict ourselves to considering protocols that are either of the type described here or do not use the clock at all.

Functionality $\mathcal{F}_{\text{CLOCK}}(\mathcal{P})$

Initialize for each $p_i \in \mathcal{P}$ a bit $d_i := 0$.

- Upon receiving message (`RoundOK`) from party p_i set $d_i := 1$. If for all $p_j \in \mathcal{H} : d_j = 1$, then reset $d_j := 0$ for all $p_j \in \mathcal{P}$. In any case, send $(\texttt{switch}, p_i)$ to $\mathcal{A}$.[a]
- Upon receiving (`RequestRound`) from p_i, send d_i to p_i.
- Upon receiving $(\texttt{corrupt}, p_i)$ from $\mathcal{A}$, set $\mathcal{H} := \mathcal{H} \cup \{p_i\}$.

[a] The adversary is notified in each such call to allow attacks at any point in time.

Synchronous protocols as $\{\mathcal{F}_{\text{CLOCK}}, \mathcal{F}^{\ell}_{\text{BD-SMT}}\}$-hybrid protocols. The code of every party is a sequence of "send," "receive," and "compute" operations, where each operation is annotated by the index of the round in which it is to be executed. In each round r, each party first receives its messages from round $r-1$, then computes and sends its messages for round r. The functionalities $\mathcal{F}_{\text{CLOCK}}$ and $\mathcal{F}^{\ell}_{\text{BD-SMT}}$ are used in the straightforward manner: At the onset of the protocol execution, each p_i sets its local round index to 1; whenever p_i receives a message from some entity other than $\mathcal{F}_{\text{CLOCK}}$ (i.e., from $\mathcal{Z}$, $\mathcal{A}$, or some other functionality), if a (`RoundOK`) messages has not yet been sent for the current round (i.e., the computation for the current round is not finished) the party proceeds with the computation of the current round (the last action of each round is sending (`RoundOK`) to $\mathcal{F}_{\text{CLOCK}}$); otherwise (i.e., if (`RoundOK`) has been sent for the current round), the party sends a (`RequestRound`) message to $\mathcal{F}_{\text{CLOCK}}$, which replies with the indicator bit d_i. The party p_i uses this bit d_i to detect whether or not every party is done with the current round and proceeds to the next round or waits for further activations accordingly.

In an immediate application of the above described protocol template, the resulting protocol would not necessarily be secure. Indeed, some party might start sending its round $r+1$ messages before some other party has even received its round r messages, potentially sacrificing security. (Some models in the literature, e.g. [33], allow such an ordering, while others, e.g. [25], don't.) The slackness can be overcome by introducing a "re-synchronization" round between every two rounds, where all parties send empty messages.

The definition of corruption for synchronous protocols differs slightly from the standard definition in [10], because the functionality $\mathcal{F}_{\text{CLOCK}}$ needs to be aware of the exact set of parties that are corrupted. Technically, we require the adversary to send the message $(\texttt{corrupt}, p_i)$ to the clock functionality $\mathcal{F}_{\text{CLOCK}}$ *immediately* after receiving a corresponding request (`corrupt`) for party $p_i \in \mathcal{P}$ or $(\texttt{corrupt}, p_i)$ for a functionality other than $\mathcal{F}_{\text{CLOCK}}$ from the environment.[8]

Perfect vs. imperfect clock synchronization. $\mathcal{F}_{\text{CLOCK}}$ models that once a single party observes that a round is completed, every party will immediately (upon activation) agree with this view. As a "real world" assumption, this means that all parties perform the round switch at exactly the same time, which means that the parties' clocks must be in perfect synchronization. A "relaxed" functionality that models more realistic synchrony assumptions, i.e., imperfectly synchronized clocks, can be obtained by incorporating "delays" as for the bounded-delay channel $\mathcal{F}^{\ell}_{\text{BD-SMT}}$. The high-level idea for this "relaxed" clock $\mathcal{F}^{\delta}_{\text{CLOCK}}$, for some integer δ, is the following: for each party p_i, $\mathcal{F}^{\delta}_{\text{CLOCK}}$ maintains a value t_i that corresponds to the number of queries needed by p_i before learning that the round has switched. The adversary is allowed to choose (at the beginning of each round), for each party p_i a delay t_i up to the upper bound $\delta > 0$. A detailed description of the functionality $\mathcal{F}^{\delta}_{\text{CLOCK}}$ is given in the following:

[8] The requirement to immediately notify $\mathcal{F}_{\text{CLOCK}}$ of corruptions is achieved by a simple change to the UC control function.

Functionality $\mathcal{F}^{\delta}_{\text{CLOCK}}(\mathcal{P})$

For each $p_i \in \mathcal{P}$, initialize a bit $d_i := 0$, a current delay $D_i := 1$, and a round-total delay $\hat{D}_i := 1$.

- Upon receiving message ($\texttt{RoundOK}$) from p_i set $d_i := 1$. If, for all $p_j \in \mathcal{H} : d_j = 1$, then reset $d_j := 0$, $D_j := 1$, and $\hat{D}_j := 1$ for all $p_j \in \mathcal{P}$. Send $(\texttt{switch}, p_i)$ to $\mathcal{A}$.[a]
- Upon receiving ($\texttt{RequestRound}$) from p_i:
  - If $d_i = 1$ or $D_i > 1$, then set $D_i := D_i - 1$ and return 1 to p_i.
  - If $d_i = 0$ and $D_i = 1$, then return 0 to p_i.
- Upon receiving a message $(\texttt{delay}, T, p_i)$ from the adversary, if $\hat{D}_i + T \leq \delta$ then set $D_i := D + T$, $\hat{D} := \hat{D} + T$, and return $(\texttt{delay-set}, p_i)$ to the adversary; otherwise ignore the message.
- Upon receiving $(\texttt{corrupt}, p_i)$ from $\mathcal{A}$, set $\mathcal{H} := \mathcal{H} \cup \{p_i\}$.

[a] The adversary is notified in each such call to facilitate attacks at any point in time.

4.2 Defining Guaranteed Termination

In formalizing what it means to UC-securely compute some specification *with guaranteed termination*, we follow the principle described in Section 1. For simplicity, we restrict ourselves to non-reactive functionalities (secure function evaluation, or SFE), but our treatment can be easily extended to reactive multi-party computation. We refer to Appendix A.2 for details on this extension.

Let $f : (\{0,1\}^*)^n \times R \longrightarrow (\{0,1\}^*)^n$ denote an n-party (randomized) function, where the i-th component of f's input (or output) corresponds to the input (or output) of p_i, and the $(n+1)$-th input $r \in R$ corresponds to the randomness used by f. In simulation-based frameworks like [10], the secure evaluation of such a function f is generally captured by an ideal functionality parametrized by f. For instance, the functionality $\mathcal{F}^f_{\text{SFE}}$ described in [10] works as follows: Any honest party can either submit input to $\mathcal{F}^f_{\text{SFE}}$ or request output. Upon input x_i from some party p_i, $\mathcal{F}^f_{\text{SFE}}$ records x_i and notifies $\mathcal{A}$. When some party requests its output, $\mathcal{F}^f_{\text{SFE}}$ checks if all honest parties have submitted inputs; if so, $\mathcal{F}^f_{\text{SFE}}$ evaluates f on the received inputs (missing inputs of corrupted parties are replaced by default values), stops accepting further inputs, and outputs to p_i its output of the evaluation. For completeness we have included a detailed description of $\mathcal{F}^f_{\text{SFE}}$ in Appendix A.1.

As described in Section 1, an ideal functionality for evaluating a function f captures *guaranteed termination* if the honest parties (or higher-level protocols, which are all encompassed by the environment in the UC framework) are able to make the functionality proceed and (eventually) produce outputs, irrespective of the adversary's strategy. (Technically, we allow the respective parties to "poll" for their outputs.) The functionality $\mathcal{F}^f_{\text{SFE}}$ has this "terminating" property; yet, for most choices of the function f, there exists no synchronous protocol realizing $\mathcal{F}^f_{\text{SFE}}$ from any "reasonable" network functionality. More precisely, we say that a network functionality $\mathcal{F}_{\text{NET}}$ provides *separable rounds* if for any synchronous $\mathcal{F}_{\text{NET}}$-hybrid protocol which communicates exclusively through $\mathcal{F}_{\text{NET}}$, $\mathcal{F}_{\text{NET}}$ activates the adversary at least once in every round.[9] The following lemma then shows that for any function f which requires more than one synchronous round to be evaluated, $\mathcal{F}^f_{\text{SFE}}$ cannot be securely realized by any synchronous protocol in the $\mathcal{F}_{\text{NET}}$-hybrid model. Note that this includes many interesting functionalities such as broadcast, coin tossing, etc.

Lemma 4. *For any function f and any network functionality $\mathcal{F}_{\text{NET}}$ with separable rounds, every $\mathcal{F}_{\text{NET}}$-hybrid protocol π that securely realizes $\mathcal{F}^f_{\text{SFE}}$ computes its output in a single round.*

Proof (sketch). Assume, towards a contradiction, that π is a two-round protocol securely computing $\mathcal{F}^f_{\text{SFE}}$. Consider the environment $\mathcal{Z}$ that provides input to all parties and immediately requests the output from some honest party. As $\mathcal{F}_{\text{NET}}$ provides separable rounds, after all inputs have been submitted, the adversary will be activated at least twice before the protocols first generate outputs. This is not the case for the simulator in the ideal evaluation of $\mathcal{F}^f_{\text{SFE}}$. Hence, the dummy adversary cannot be simulated, which contradicts the security of π. □

[9] Functionalities in [10] are not necessarily of that form. A priori, if some ITI sends a message to some other ITI, the receiving ITI will be activated next. Only if an ITI halts without sending a message, the "master scheduler"—the environment—will be activated.

To obtain an SFE functionality that matches the intuition of guaranteed termination, we need to circumvent the above impossibility by making the functionality activate the simulator during the computation. We parametrize $\mathcal{F}_{\text{SFE}}$ with a function $Rnd(k)$ of the security parameter which corresponds to the number of rounds required for evaluating f; one can easily verify that for any (polynomial) round function $Rnd(\cdot)$ the functionality $\mathcal{F}_{\text{SFE}}^{f,Rnd}$ will terminate (if there are sufficiently many queries at the honest parties' interfaces) independently of the simulator's strategy. In each round, the functionality gives the simulator $|\mathcal{P}|$ activations which will allow him to simulate the activations that the parties need for exchanging their protocol messages and notifying the clock $\mathcal{F}_{\text{CLOCK}}$.

Functionality $\mathcal{F}_{\text{SFE}}^{f,Rnd}(\mathcal{P})$

$\mathcal{F}_{\text{SFE}}^{f,Rnd}$ proceeds as follows, given a function $f : (\{0,1\}^* \cup \{\perp\})^n \times R \to (\{0,1\}^*)^n$, a round function Rnd, and a player set $\mathcal{P}$. For each $p_i \in \mathcal{P}$, initialize variables x_i and y_i to a default value $\perp$ and a current delay $t_i := |\mathcal{P}|$. Moreover, initialize a global round counter $l := 1$.

- Upon receiving input $(\texttt{input}, v)$ from some party $p_i \in \mathcal{P}$, set $x_i := v$ and send a message $(\texttt{input}, p_i)$ to the adversary.
- Upon receiving input $(\texttt{output})$ from some party $p_i \in \mathcal{P}$, if $p_i \in \mathcal{H}$ and x_i has not yet been set then ignore p_i's message, else do:
  - If $t_i > 1$, then set $t_i := t_i - 1$. If (now) $t_j = 1$ for all $p_j \in \mathcal{H}$, then set $l := l + 1$ and $t_j := |\mathcal{P}|$ for all $p_j \in \mathcal{P}$. Send $(\texttt{activated}, p_i)$ to the adversary.
  - Else, if $t_i = 1$ but $l < Rnd$, then send $(\texttt{early})$ to p_i.
  - Else,
    - if x_j has been set for all $p_j \in \mathcal{H}$, and $y_1, \ldots, y_n$ have not yet been set, then choose $r \overset{R}{\leftarrow} R$ and set $(y_1, \ldots, y_n) := f(x_1, \ldots, x_n, r)$.
    - Output y_i to p_i.

Definition 1 (Guaranteed Termination). *A protocol π UC-securely evaluates a function f* with guaranteed termination *if it UC-realizes a functionality $\mathcal{F}_{\text{SFE}}^{f,Rnd}$ for some polynomial $Rnd(\cdot)$.*

Remark 1 (Lower Bounds). The above formulation offers a language for making UC statements about (lower bounds on) the round complexity of certain problems in the synchronous setting. In particular, the question whether $\mathcal{F}_{\text{SFE}}^{f,Rnd}$ can be realized by a synchronous protocol corresponds to the question: "Does there exist a synchronous protocol π which securely evaluates f in $Rnd(k)$ rounds?", where k is the security parameter. As an example, the statement: "A function f needs at least r rounds to be evaluated." is (informally) translated to "There exists no synchronous protocol which UC-realizes the functionality $\mathcal{F}_{\text{SFE}}^{f,r'}$, where $r' < r$."

Known results on feasibility of secure computation, e.g. [8,16,22,34], extend to our setting of UC with termination. (This follows from the below theorem and the fact that these protocols are secure with respect to an efficient straight-line black-box simulator.) The only modification is that the protocols start with a void synchronization round where no honest party sends or receives any message. For a synchronous protocol ρ, we denote by $\hat{\rho}$ the protocol which is obtained by extending ρ with such a start-synchronization round. The proof is based on ideas from [28].

Theorem 1. *Let f be a function and let ρ be a protocol that, according to the notion of [11], realizes f with computational (or statistical or perfect) security in the stand-alone model, with an efficient straight-line black-box simulator. Then $\hat{\rho}$ UC-realizes f with computational (or statistical or perfect) security and guaranteed termination in the $\{\mathcal{F}_{\text{CLOCK}}, \mathcal{F}_{\text{BD-SEC}}\}$-hybrid model with a static adversary.*

Before proving the above theorem, we make the following observations that are relevant for the proof: In this section, we translate a result from Kushilevitz et al. [28] into our setting to obtain UC-secure protocols for secure function evaluation with guaranteed termination. The structure of the proof of Theorem 1 is similar to the proof of [28, Theorem 7.1] with some adaptations.

Because of the considered model of communication channels, it is important that the simulator can faithfully emulate the "communication pattern" of the protocol. In contrast, most of the literature on secure function evaluation assumes that communication channels are perfectly secure: they do not even leak the information that a message transmission takes place. This is in contrast to our channel $\mathcal{F}_{\text{BD-SMT}}$, which activates the adversary upon message transmission and, hence, leaks this information. But as our proof considers "standard synchronous" protocols that proceed in rounds and, within each such round, send exactly one message to each one of the other parties, this assumption can be seen to be fulfilled. To allow for a simulation of the protocol behavior, the realized functionality $\mathcal{F}_{\text{SFE}}^{f,Rnd}$ requires $|\mathcal{P}|$ activations from each honest party in every round, and for each such activation issues a message $(\texttt{activated}, p_i)$ to the simulator. The simulator then uses these activations to simulate the message transmissions, reproducing the communication pattern of the protocol.

As, in our model, the parties have access to the clock functionality $\mathcal{F}_{\text{CLOCK}}$, we redefine the notion of start synchronization from [28] in a simpler way—we say that a protocol ρ has *start synchronization* if it begins with the following steps for *all* parties:

1. Send $(\texttt{RoundOK})$ to $\mathcal{F}_{\text{CLOCK}}$,
2. In each further activation, query $(\texttt{RequestRound})$ to $\mathcal{F}_{\text{CLOCK}}$. Once $\mathcal{F}_{\text{CLOCK}}$ answers with 0, proceed with the execution of ρ.

The goal of start synchronization is that the first message of the protocol that actually depends on the input of an honest party is sent only after the inputs to all honest parties are fixed. Technically, this is needed to "fix" the honest users' inputs from the environment $\mathcal{Z}$ in the simulated UC-execution. Recall that for a synchronous protocol ρ, we denote by $\hat{\rho}$ the protocol which is obtained by extending ρ with such a start-synchronization round.

We do not use the fully non-adaptive version of [11] to start from, but the "initially adaptive" version where the adversary is allowed to adaptively choose the parties to be corrupted until the first protocol message is sent. This is equivalent to the non-adaptive version by [11, Section 4.1, Remark 5]. The same argument is used implicitly in the proof of [28]. As in [28], we prove the statement only for the computational setting, the extension to the statistical case is straightforward.

Furthermore, the network implied by [11] provides strong synchrony guarantees: the messages that *all* honest parties receive in a round are determined when the *first* honest party starts the computation in that round. (In reality, it would be conceivable that the first party p_i has finished computation and sending for round r before another party p_j's clock tells p_j to start computing. In this case, the adversary could inject round $r-1$ messages to p_j that actually depend on the round r messages of p_i.) This strong synchrony assumption becomes explicit in our model by doubling the round number of ρ and implementing alternating "receive" and "compute/send" rounds.

The synchronous protocol ρ has to be detailed slightly to implement the interface of $\mathcal{F}_{\text{SFE}}^{f,Rnd}$. For each pair of parties and round of the protocol, an independent channel $\mathcal{F}_{\text{BD-SEC}}$ is used. In each round r, the protocol first computes the messages for that round and then uses the following $|\mathcal{P}|$ activations to send the messages $m_{i,j,r}$ via the channels to the protocol machines ρ_j and to send $(\texttt{RoundOK})$ to $\mathcal{F}_{\text{CLOCK}}$. Upon each further activation, the protocol sends $(\texttt{RequestRound})$ to $\mathcal{F}_{\text{CLOCK}}$ and outputs $(\texttt{early})$ locally while $d = 1$, and proceeds with setting $r := r + 1$ and sending $(\texttt{fetch})$ to the channels $\mathcal{F}_{\text{BD-SEC}}$ to obtain all messages once $d = 0$.

Proof (of Theorem 1). By the assumption on the security of ρ, we know that there is a (stand-alone) efficient straight-line black-box simulator $\mathcal{S}_\rho$. Without loss of generality, we assume that ρ has start synchronization, i.e., $\rho = \hat{\rho}$ (if this is not the case, then we can trivially extend ρ with the two "start-synchronizing" steps sketched above). We describe a UC-simulator $\mathcal{S}$ for the protocol ρ executed in the $\{\mathcal{F}_{\text{CLOCK}}, \mathcal{F}_{\text{BD-SEC}}\}$-hybrid model with the dummy adversary $\mathcal{D}$, which makes black-box use of $\mathcal{S}_\rho$.

UC-simulator $\mathcal{S}$

Create instances of the assumed stand-alone simulator $\mathcal{S}_\rho$, the ideal functionalities $\mathcal{F}_{\text{CLOCK}}$ and $\mathcal{F}_{\text{BD-SEC}}$, and the dummy adversary $\mathcal{D}$.

- Forward all messages between the environment $\mathcal{Z}$ and the emulated adversary $\mathcal{D}$ (via input and subroutine output tapes), as well as between $\mathcal{D}$ and the simulated ideal functionalities.
- Forward the corruption-related communication between $\mathcal{S}_\rho$ and $\mathcal{F}_{\text{SFE}}^{f,Rnd}$ (via the respective communication tapes).
- Receiving $(\texttt{activated}, p_i)$ from $\mathcal{F}_{\text{SFE}}^{f,Rnd}$:
  - If this is the first such message in a round r, provide the messages sent via $\mathcal{F}_{\text{BD-SEC}}(p_j, p_i)$ with $p_j \in \mathcal{P} \setminus \mathcal{H}$ for round $r-1$ to $\mathcal{S}_\rho$ and obtain the round r messages $m_{i,j,r}$ from ρ_i to ρ_j with $p_j \in \mathcal{P} \setminus \mathcal{H}$ in response.
  - Using the communication pattern of the protocol, simulate either the transmission of a message via $\mathcal{F}_{\text{BD-SEC}}(p_i, p_j)$ if $p_j \in \mathcal{H}$, the message $m_{i,j,r}$ if $p_j \in \mathcal{P} \setminus \mathcal{H}$, or the notification $(\texttt{switch}, p_i)$ to $\mathcal{D}$.

In the following, we have to show that if there exists an environment $\mathcal{Z}$ that distinguishes between the real execution of protocol ρ and the ideal execution of $\mathcal{F}_{\text{SFE}}^{f,Rnd}$ with simulator $\mathcal{S}$ with non-negligible advantage, then there is a stand-alone adversary $\mathcal{A}_{\mathcal{Z}}$ that attacks ρ in the stand-alone model with non-negligible probability.

Stand-alone adversary $\mathcal{A}_{\mathcal{Z}}$

Split the auxiliary input into a random tape and an auxiliary input for $\mathcal{Z}$. Create and run an instance of the environment $\mathcal{Z}$ with the random tape and input as above, as well as the (dummy) adversary $\mathcal{D}$, the clock $\mathcal{F}_{\text{CLOCK}}$ and the channels $\mathcal{F}_{\text{BD-SEC}}(p_i, p_j)$ for $p_i, p_j \in \mathcal{P}$, $p_i \neq p_j$. In this simulated computation, forward the communication between the simulated parties $\mathcal{D}$, $\mathcal{Z}$, $\mathcal{F}_{\text{CLOCK}}$, and $\mathcal{F}_{\text{BD-SEC}}$.

- Run the simulation until $\mathcal{Z}$ provides the first input to a protocol, answer corruption requests from $\mathcal{D}$ with simulated "vanilla" ρ-states, and corrupt the corresponding parties in the execution of ρ.
- The UC-interaction is simulated almost entirely, with the exception for the computation of the honest parties ρ_i. Instead, on every other round switch at $\mathcal{F}_{\text{CLOCK}}$, provide all messages injected by $\mathcal{D}$ into $\mathcal{F}_{\text{BD-SEC}}(p_j, p_i)$ in the name of the corrupted parties $p_j \in \mathcal{P} \setminus \mathcal{H}$ to the real ρ_i.
- Being activated with the messages $m_{i,j,r}$ with $p_i \in \mathcal{H}$ and $p_j \in \mathcal{P} \setminus \mathcal{H}$, use these values in the further simulation until the next but one round switch of $\mathcal{F}_{\text{CLOCK}}$ signals that the round r messages of the corrupted parties are also fixed.
- When a protocol machine ρ_i in the UC-execution terminates, output the transcript of the execution (including the auxiliary input) and halt. This transcript is complete, as by the time that ρ_i *does* produce output, it is guaranteed that all messages of the final round must be in the channels.

The main idea of the proof is the following. Since the protocol ρ has start synchronization, the input and the random tape of $\mathcal{Z}$ determine the inputs that $\mathcal{Z}$ gives to the honest protocol machines (because the rest of the execution within $\mathcal{A}_{\mathcal{Z}}$ is deterministic). This allows us to restrict our attention to this particular set of inputs in the stand-alone execution. If the environment distinguishes the real and the ideal UC-executions, then the distribution of the environment's "view" of the execution of ρ (as a random variable consisting of all messages sent and received by the environment) together with the outputs of the honest ρ_i must be distinguishable. But this corresponds to the fact that the transcripts in the stand-alone executions of ρ with $\mathcal{A}_{\mathcal{Z}}$ and $\mathcal{S}_\rho$ (can be extended to transcripts which) are distinguishable.

The protocol execution begins with a corruption phase: according to the definition of static corruption in [10], only parties p_i for which the environment sent a $(\texttt{corrupt})$ message prior to the first input to a protocol machine may be corrupted during the protocol execution (by the definition of the control function). This is compatible with the "initially adaptive" corruption allowed for $\mathcal{A}_{\mathcal{Z}}$ by the description in [11]. In the real stand-alone execution of ρ with $\mathcal{A}_{\mathcal{Z}}$, the inputs provided by the environment $\mathcal{Z}$ to the honest parties p_i are determined by the auxiliary input and the random tape of the (simulated) environment $\mathcal{Z}$: By the start synchronization, the first message of the protocol ρ is sent only after *all* parties p_i have sent their synchronization messages, acknowledging their input. All computations that $\mathcal{A}_{\mathcal{Z}}$, $\mathcal{D}$, the honest ρ_i, and the ideal functionalities have performed up to this point are deterministic. In the following description, we restrict our attention to executions in the stand-alone model where the inputs to p_i and ρ_i indeed coincide (both in the real execution with $\mathcal{A}_{\mathcal{Z}}$ and the black-box ideal execution with $\mathcal{A}_{\mathcal{Z}}$ and $\mathcal{S}_\rho$).

The environment's "view" in the real UC-execution is the same as the simulated "view" in the stand-alone execution within $\mathcal{A}_{\mathcal{Z}}$ restricted to the cases where the inputs and outputs of the honest ρ_i indeed coincide. For all inputs except for those that provoke a message from the channel $\mathcal{F}_{\text{BD-SEC}}(p_i, p_j)$ to $\mathcal{D}$, this is straightforward: $\mathcal{A}_{\mathcal{Z}}$ only performs a straightforward simulation of the other ITMs. But for the messages from $\mathcal{F}_{\text{BD-SEC}}(p_i, p_j)$, the inputs and messages sent to the protocol machines ρ_i are equal (resp. have the same distribution) by assumption, and the computation of the stand-alone and the UC-version of ρ are exactly the same.

The environment's "view" in the ideal UC-execution is the same as the simulated "view" in the stand-alone execution in case the inputs and outputs of the honest ρ_i coincide. Again, responses that do not contain (simulated) messages from the honest ρ_i are the same by the fact that both $\mathcal{A}_{\mathcal{Z}}$ and $\mathcal{S}$ perform a straightforward simulation of the real setting. But responses that involve messages are also the same in both cases: the simulator $\mathcal{S}$ in the UC-execution uses the messages supplied by $\mathcal{S}_\rho$ as message intended from an honest ρ_i to a corrupted ρ_j, and the same messages are used by $\mathcal{A}_{\mathcal{Z}}$ in the ideal (black-box straight-line) execution with $\mathcal{S}$.

Our goal is to conclude that, if an environment $\mathcal{Z}$ distinguishes with non-negligible advantage between the real and ideal UC-executions, the stand-alone transcripts of the real execution with $\mathcal{A}_{\mathcal{Z}}$ and the ideal execution with $\mathcal{A}_{\mathcal{Z}}$ and $\mathcal{S}_\rho$ can also be distinguished with non-negligible advantage. We construct a distinguisher D as follows: By the above construction, the output of $\mathcal{A}_{\mathcal{Z}}$ contains the auxiliary input to $\mathcal{A}_{\mathcal{Z}}$, which determines the input and random tape to $\mathcal{Z}$. Hence, D can replay the complete execution and obtain the state of $\mathcal{Z}$ at the time the execution was aborted. D continues the simulation of the real model and provides $\mathcal{Z}$ with the output of the honest protocol machines ρ_i as obtained from the transcript. Once the environment $\mathcal{Z}$ halts with output bit b, D uses the same bit b as its output.

The auxiliary input to $\mathcal{A}_{\mathcal{Z}}$ is interpreted as a tuple $x = (r, z)$ where $r \in \{0,1\}^*$ is used as a random tape and $z \in \{0,1\}^*$ as an auxiliary output to $\mathcal{Z}$. Hence, if we consider the ensembles corresponding to the stand-alone settings where the variable R corresponding to $\mathcal{Z}$'s random tape is chosen according to the correct distribution, we obtain

$$\left\{D\big(\text{REAL}^{\mathsf{SA}}_{\rho,\mathcal{A}_{\mathcal{Z}}}(k,(R,z),\mathbf{x}(R,z))\big)\right\}_{k\in\mathbb{N},z\in\{0,1\}^*} \equiv \left\{\text{REAL}^{\mathsf{UC}}_{\rho,\mathcal{D},\{\mathcal{F}_{\text{CLOCK}},\mathcal{F}_{\text{BD-SEC}}\},\mathcal{Z}}(k,z)\right\}_{k\in\mathbb{N},z\in\{0,1\}^*}$$

and

$$\left\{\text{IDEAL}^{\mathsf{UC}}_{\mathcal{F}^{f,c}_{\text{SFE}},\mathcal{S},\mathcal{Z}}(k,z)\right\}_{k\in\mathbb{N},z\in\{0,1\}^*} \equiv \left\{D\big(\text{IDEAL}^{\mathsf{SA}}_{f,\mathcal{S}^{\mathcal{A}}}(k,(R,z),\mathbf{x}(R,z))\big)\right\}_{k\in\mathbb{N},z\in\{0,1\}^*},$$

where $\mathbf{x}(R,z)$ denotes the inputs that $\mathcal{Z}$ computes for the honest parties p_i, given random tape R and auxiliary input z. By the above arguments, these are the only parameters that determine the inputs. The statement holds, as all values in the simulated UC-executions are determined according to exactly the same computations as in the corresponding "real" UC-executions; note that the distribution ensembles in the stand-alone case are *not* the same distributions that appear in the corresponding security definition, as part of the auxiliary input is also chosen according to a specific distribution. Still, by the assumption that $\mathcal{Z}$ is a successful distinguisher, we obtain that

$$\left\{D\big(\text{REAL}^{\mathsf{SA}}_{\rho,\mathcal{A}_{\mathcal{Z}}}(k,x)\big)\right\}_{k\in\mathbb{N},x\in\{0,1\}^*} \not\approx \left\{D\big(\text{IDEAL}^{\mathsf{SA}}_{f,\mathcal{S}^{\mathcal{A}}}(k,x)\big)\right\}_{k\in\mathbb{N},x\in\{0,1\}^*},$$

which concludes the main argument.

As the functionality $\mathcal{F}^{f,c}_{\text{SFE}}$ models guaranteed termination, this proof implies that the protocol ρ achieves guaranteed termination if it is executed using channels of the type $\mathcal{F}_{\text{BD-SEC}}$. The guaranteed-delivery property of the channels is used in the proof to conclude that the messages provided to the honest parties ρ_i are indeed available in the beginning of each "receive" sub-round by the fact that all honest parties inject them into the channels before they agree to switch rounds, and the channel $\mathcal{F}_{\text{BD-SEC}}$ immediately makes the messages available to the intended receiver. $\square$

4.3 The Need for Both Synchronization and Bounded Delay

In this section, we formalize the intuition that each one of the two "standard" synchrony assumptions, i.e., bounded-delay channels and synchronized clocks, is *alone* not sufficient for computation with guaranteed termination. We first show in Lemma 5 that bounded-delay channels (even with instant delivery) are, by themselves, not sufficient; subsequently, we show in Lemma 6 that (even perfectly) synchronized clocks are also not sufficient, even in combination with eventual-delivery channels (with no known bound on the delay).

The proof of the insufficiency of (solely) bounded-delay channels for terminating computation is based on the following idea: Consider the two-party function f which, on input a bit $x_1 \in \{0,1\}$ from party p_1 (and nothing from p_2), outputs x_1 to p_2 (and nothing to p_1). The functionality $\mathcal{F}_{\text{SFE}}^{f,Rnd}$ guarantees that an honest p_1 will be able to provide input, independently of the adversary's behavior. On the other hand, a corrupted p_1 will not keep p_2 from advancing (potentially with a default input for p_1).[10] However, in the real world, the behavior of the bounded-delay channel in the above two cases is identical. The functionality $[\mathcal{F}_{\text{SFE}}^{f,Rnd}]^{\texttt{NT}}$ can be realized from $\mathcal{F}_{\text{BD-SEC}}$: p_1 simply has to send the input to p_2 via the $\mathcal{F}_{\text{BD-SEC}}$-channel. The simulator makes sure that the output in the ideal model is delivered to p_2 only after $\mathcal{Z}$ acknowledges the delivery.

Lemma 5. *There is a function f such that for any polynomial Rnd and any $\delta > 0$: $[\mathcal{F}_{\text{SFE}}^{f,Rnd}]^{\texttt{NT}}$ can be realized in the $\mathcal{F}_{\text{BD-SEC}}^{\delta}$-hybrid model, but $\mathcal{F}_{\text{SFE}}^{f,Rnd}$ cannot.*

Proof (sketch). Consider the deterministic two-party function f that on input a bit $x_1 \in \{0,1\}$ from p_1 (and no input from p_2) outputs x_1 to p_2 (f outputs $y_2 := 0$ if $x_1 = \perp$) and nothing to p_2. For the impossibility of realizing $\mathcal{F}_{\text{SFE}}^{f,Rnd}$, it is sufficient to argue for the case $\delta = 1$, as the result extends to the case $\delta > 1$ since channels with larger δ can be realized from channels with smaller δ (see Section 3.3).

Assume, towards a contradiction, that there exists a $\mathcal{F}_{\text{BD-SEC}}^{1}$-hybrid protocol π which securely realizes $\mathcal{F}_{\text{SFE}}^{f,Rnd}$. Consider the following two scenarios which involve a dummy adversary and an environment that never corrupts p_2:

- scenario 1: p_1 is corrupted and the adversary is instructed to force p_1 to never send any message. As a result, in protocol π, the (`fetch`)-queries issued by p_2 will result in an activation of $\mathcal{Z}$.
- scenario 2: p_1 is honest, but never activated by the environment. Again, in protocol π, the (`fetch`)-queries issued by p_2 will result in an activation of $\mathcal{Z}$, and from the perspective of p_2 the scenarios are identical.

In scenario 1, after receiving $|\mathcal{P}| \cdot Rnd$ queries of the type (`output`) from $\mathcal{Z}$, the (real world) program of p_2 should output 0 (this is the default input of p_1) to $\mathcal{Z}$, as this is the result in the ideal world. However, in scenario 2, p_2 should output (`early`) to the environment, independently of the number of (`output`)-messages received. Indeed, as both parties are honest, $\mathcal{F}_{\text{SFE}}$ will wait for p_1's input before producing output. The environment chooses to implement one of the two above scenarios at random; as the two scenarios are identical for p_2, the action taken by p_2 will differ from the "correct action" with probability at least 1/2, which allows $\mathcal{Z}$ to distinguish.
For the possibility, we consider the cases (I) $\delta = 1$ and (II) $\delta > 1$ separately.
Case I ($\delta = 1$): $[\mathcal{F}_{\text{SFE}}^{f,Rnd}]^{\texttt{NT}}$ can be realized from $\mathcal{F}_{\text{BD-SEC}}^{1}$ by the simple protocol in which p_1 sends its input via the channel $\mathcal{F}_{\text{BD-SEC}}^{1}$. Indeed, the simulator is informed about who is corrupted and about when messages are sent to $[\mathcal{F}_{\text{SFE}}^{f,Rnd}]^{\texttt{NT}}$ and can delay the outputs of the functionality arbitrarily, therefore, it can ensure that the exchanged outputs are only handed to the (dummy) parties when the environment expects to see them.
Case II ($\delta > 1$): To complete the proof, we show that $[\mathcal{F}_{\text{SFE}}^{f,Rnd}]^{\texttt{NT}}$ can be realized from $\mathcal{F}_{\text{BD-SEC}}^{\delta}$: Lemma 3 implies that $[\mathcal{F}_{\text{BD-SEC}}^{1}]^{\texttt{NT}}$ can be realized in the $\mathcal{F}_{\text{BD-SEC}}^{\delta}$-hybrid model. Furthermore, in case I we showed that $\mathcal{F}_{\text{BD-SEC}}^{1}$ is sufficient for $[\mathcal{F}_{\text{SFE}}^{f,Rnd}]^{\texttt{NT}}$, which extends to $[\mathcal{F}_{\text{BD-SEC}}^{1}]^{\texttt{NT}}$, as relaxing the hybrid $\mathcal{F}_{\text{BD-SEC}}^{1}$ to its non-terminating relaxation does not have any effect on the statement as the functionality we want to achieve is already of this type. In particular, we use the same protocol as in case I, the only difference in the simulation is that whenever the adversary delays some output in the $[\mathcal{F}_{\text{BD-SEC}}^{1}]^{\texttt{NT}}$, the simulator also delays the outputs of the ideal functionality accordingly. □

In reality, synchronous clocks alone are not sufficient for synchronous computation if there is no *known* upper bound on the delay of the channels (even with guaranteed *eventual* delivery); this statement is formalized using the clock functionality $\mathcal{F}_{\text{CLOCK}}$ and the channels $\mathcal{F}_{\text{ED-SEC}}$ in the following lemma. The proof is similar to the proof of Lemma 1.

[10] This capability of distinguishing "honest" from "dishonest" behavior is key in synchronous models: as the honest parties are guaranteed that they can send their messages on time, dishonest parties will blow their cover by not sending a message prior to the deadline.

Lemma 6. *There is a function f such that for any polynomial Rnd: $[\mathcal{F}_{\text{SFE}}^{f,Rnd}]^{\texttt{NT}}$ can be realized in the $\{\mathcal{F}_{\text{ED-SEC}}, \mathcal{F}_{\text{CLOCK}}\}$-hybrid model, but $\mathcal{F}_{\text{SFE}}^{f,Rnd}$ cannot.*

Proof (sketch). We prove the statement for f being the two-party function used in the proof of Lemma 5, i.e., on input a bit $x_1 \in \{0,1\}$ from p_1 (and no input from p_2) f outputs x_1 to p_2 (f outputs $y_2 := 0$ if $x_1 = \bot$) and nothing to p_2.

The sufficiency of $\{\mathcal{F}_{\text{ED-SEC}}, \mathcal{F}_{\text{CLOCK}}\}$ for $[\mathcal{F}_{\text{SFE}}^{f,Rnd}]^{\texttt{NT}}$ follows along the lines of the sufficiency condition in case II in the proof of Lemma 5. Indeed, the channels $\mathcal{F}_{\text{ED-SEC}}$ are sufficient even without $\mathcal{F}_{\text{CLOCK}}$.

We show that $\mathcal{F}_{\text{SFE}}^{f,Rnd}$ cannot be realized by an $\{\mathcal{F}_{\text{ED-SEC}}, \mathcal{F}_{\text{CLOCK}}\}$-hybrid protocol. Consider the dummy adversary $\mathcal{A}$ and the environment $\mathcal{Z}$ which corrupts no party and has the following distinguishing strategy: $\mathcal{Z}$ starts by giving a uniformly random two-bit input $m \in \{0,1\}^2$ to p_1. As soon as it receives the leakage $\ell(m)$ from the functionality (forwarded by $\mathcal{A}$), $\mathcal{Z}$ instructs $\mathcal{A}$ to give delay $T = Rnd \cdot |\mathcal{P}| + 2 = 2Rnd + 2$ to $\mathcal{F}_{\text{ED-SEC}}$. Subsequently, $\mathcal{Z}$ activates p_2 with input $(\texttt{fetch})$ $2Rnd + 1$ times in a row; as a result, if $\mathcal{Z}$ is witnessing the ideal-world execution, it will receive the message m from p_2 (note that $\mathcal{F}_{\text{SFE}}^{f,Rnd}$ does not allow the simulator to stall this reply). Hence, p_2's protocol (in the real world) also has to output the message m to $\mathcal{Z}$ after $2Rnd + 1$ activations. However, the protocol of p_2 cannot get any information on m from $\mathcal{F}_{\text{ED-SEC}}$, because by setting the delay to $T = 2Rnd + 2$, the environment makes sure that the $2Rnd + 1$ are not sufficient for making $\mathcal{F}_{\text{ED-SMT}}$ deliver m to p_2. The only possible way out for p_2 is to interact with the functionality $\mathcal{F}_{\text{CLOCK}}$. However, as p_1 is activated only once, using the functionality $\mathcal{F}_{\text{CLOCK}}$ allows for at most one bit of communication (i.e., the bit indicating whether or not p_1 has switched his indicator bit d_1 in this one activation). Because m is a two-bit message, the probability that p_2 outputs m is at most 1/2, which gives $\mathcal{Z}$ a noticeable distinguishing advantage. $\square$

4.4 Atomicity of Send/Receive Operations and Rushing

Hirt and Zikas [24] pointed out that the standard formulation of a "rushing" adversary [11] in the synchronous setting puts a restriction on the order of the send/receive operations within a synchronous round. The modularity of our framework allows to pinpoint this restriction by showing that the rushing assumption corresponds to a "simultaneous multi-send" functionality which cannot even be realized using $\mathcal{F}_{\text{CLOCK}}$ and $\mathcal{F}_{\text{BD-SEC}}$.

Intuitively, a rushing adversary [11] cannot preempt a party while this party is sending its messages of some round. This is explicitly stated in [11], where the notion of "synchronous computation with *rushing*" is defined as follows:

> *"The computation proceeds in rounds; each round proceeds in mini-rounds, as follows. Each mini-round starts by allowing $\mathcal{A}$ to corrupt parties one by one in an adaptive way, as long as at most t parties are corrupted altogether. (The behavior of the system upon corruption of a party is described below.) Next $\mathcal{A}$ chooses an uncorrupted party, p, that was not yet activated in this round and activates it. Upon activation, p receives the messages sent to it in the previous round,* generates its messages for this round*, and the next mini-round begins."*

In reality, it is arguable whether we can obtain the above guarantee by just assuming bilateral bounded-delay channels and synchronized clocks. Indeed, sending multiple messages is typically not an atomic operation, as the messages are buffered on the network interface of the computer and sent one-by-one. Hence, to achieve the simultaneity, one has to assume that the total time it takes for the sender to put all the messages on the network minus the *minimum* latency of the network is not sufficient for a party to become corrupted.

The "simultaneous multi-send" guarantee is captured in the following (UC) functionality, which is referred to as the *simultaneous multi-send* channel, and denoted by $\mathcal{F}_{\text{MS}}$. On a high level, $\mathcal{F}_{\text{MS}}$ can be described as a channel allowing a sender p_i to send a vector of messages $(x_1, \ldots, x_n)$ to the respective receivers $p_1, \ldots, p_n$ as an atomic operation. The formal description of $\mathcal{F}_{\text{MS}}$ is similar to $\mathcal{F}_{\text{BD-SMT}}$ with the following modifications: First, instead of a single receiver, there is a set $\mathcal{P}$ of receivers, and instead of a single message, the sender inputs a vector of $|\mathcal{P}|$ messages, one for each party in $\mathcal{P}$. As soon as some party receives its message, the adversary cannot replace any of the remaining messages that correspond to honest receivers, not even by corrupting the sender. As in the case of bounded-delay channels, we denote by $\mathcal{F}_{\text{MS-AUTH}}$ the multi-send channel which leaks the transmitted vector to the adversary. The following lemma states that the delayed relaxation of $\mathcal{F}_{\text{MS-AUTH}}$ cannot be realized from $\mathcal{F}_{\text{BD-SEC}}$ and $\mathcal{F}_{\text{CLOCK}}$ when arbitrary many parties can be corrupted. This implies that $\mathcal{F}_{\text{MS-AUTH}}$ can also not be realized from $\mathcal{F}_{\text{BD-SEC}}$ and $\mathcal{F}_{\text{CLOCK}}$.

Functionality $\mathcal{F}^{\ell}_{\text{MS}}(p_i, \mathcal{P})$

- Upon receiving a vector of messages $\boldsymbol{m} = (m_1, \ldots, m_n)$ from p_i, record $\boldsymbol{m}$ and send a message $(\texttt{sent}, \ell(\boldsymbol{m}))$ to the adversary.
- Upon receiving $(\texttt{fetch})$ from $p_j \in \mathcal{P}$, output $(\texttt{sent}, m_j)$ to p_j (with $m_j = \bot$ if $\boldsymbol{m}$ has not been recorded).
- Upon receiving $(\texttt{replace}, \boldsymbol{m}')$ from the adversary if no (honest or corrupted) p_j received m_j *before* p_i became corrupted, then replace $\boldsymbol{m}$ by $\boldsymbol{m}'$.

Lemma 7. *Let $\mathcal{P}$ be a player set with $|\mathcal{P}| > 3$. Then there exists no protocol which UC-realizes $[\mathcal{F}_{\text{MS-AUTH}}]^{\texttt{NT}}$ in the $\{\mathcal{F}_{\text{CLOCK}}, \mathcal{F}_{\text{BD-AUTH}}\}$-hybrid model and tolerates a corrupted majority.*

Proof (sketch). Garay et al. [21] showed that if atomic multi-send (along with a setup for digital signatures) is assumed, then the broadcast protocol from Dolev and Strong [19] UC-realizes broadcast (without guaranteed termination) in the presence of an adaptive adversary which corrupts any number of parties. Hence, if there exist a protocol for realizing $[\mathcal{F}_{\text{MS-AUTH}}]^{\texttt{NT}}$ in the synchronous model, i.e., in the $\{\mathcal{F}_{\text{CLOCK}}, \mathcal{F}_{\text{BD-AUTH}}\}$-hybrid world, with corrupted majority and adaptive adversary, then one could also realize broadcast in this model, contradicting the impossibility result of [24]. $\square$

The above lemma implies that the traditional notion of "synchronous computation with *rushing*" cannot be, in general, achieved in the UC model unless some non-trivial property is assumed on the communication channel. Yet, $[\mathcal{F}_{\text{MS-AUTH}}]^{\texttt{NT}}$ *can* be UC-realized from $\{[\mathcal{F}_{\text{BD-AUTH}}]^{\texttt{NT}}, [\mathcal{F}_{\text{COM}}]^{\texttt{NT}}\}$, where $\mathcal{F}_{\text{COM}}$ denotes the standard (UC) commitment functionality [13]. The idea is the following: In order to simultaneously multi-send a vector $(x_1, \ldots, x_n)$ to the parties $p_1, \ldots, p_n$, the sender sends an independent commitment on x_i to every recipient p_i, who acknowledges the receipt (using the channel $[\mathcal{F}_{\text{BD-AUTH}}]^{\texttt{NT}}$). After receiving all such acknowledgments, the sender, in a second round, opens all commitments. The functionality $\mathcal{F}_{\text{COM}}$ ensures that the adversary (unless the sender is corrupted in the first round) learns the committed messages x_i only after every party has received the respective commitment; but, from that point on, $\mathcal{A}$ can no longer change the committed message. For completeness we state the above in the following lemma.

Lemma 8. *There is a synchronous $\{[\mathcal{F}_{\text{BD-AUTH}}]^{\texttt{NT}}, [\mathcal{F}_{\text{COM}}]^{\texttt{NT}}\}$-hybrid protocol UC-realizing $[\mathcal{F}_{\text{MS-AUTH}}]^{\texttt{NT}}$.*

Proof. Let π be the protocol that proceeds as follows (we assume that the designated sender $p_i \in \mathcal{P}$ is specified in the session ID, and denote the corresponding protocol machine by π_i). On input a message vector $\boldsymbol{m}$, the machine π_i initializes one copy $[\mathcal{F}_{\text{COM}}]^{\texttt{NT}}_j$ and issues $(\texttt{commit}, m_j)$ to $[\mathcal{F}_{\text{COM}}]^{\texttt{NT}}_j$ for each receiver $p_j \in \mathcal{P} \setminus \{p_i\}$. In the following activations, the machine π_i repeatedly polls the channels $[\mathcal{F}_{\text{BD-SEC}}]^{\texttt{NT}}$ and waits to receive a message $(\texttt{ack})$ from each p_j. Afterward, π_i proceeds by sending $(\texttt{open})$ messages to all $[\mathcal{F}_{\text{COM}}]^{\texttt{NT}}_j$. The protocol machines of the receiving parties p_j wait for the notification $(\texttt{receipt})$ from the functionality $[\mathcal{F}_{\text{COM}}]^{\texttt{NT}}_j$ and react by sending a message $(\texttt{ack})$ via the channel $[\mathcal{F}_{\text{BD-SEC}}]^{\texttt{NT}}$ to π_i. After receiving both the message $(\texttt{open}, m_j)$ from $[\mathcal{F}_{\text{COM}}]^{\texttt{NT}}_j$ and the query $(\texttt{fetch})$ on the input tape, π_j outputs m_j locally.

We describe the simulator $\mathcal{S}$ that is used to prove that π achieves the desired goals. Upon receiving $(\texttt{sent}, \boldsymbol{m})$ from $[\mathcal{F}_{\text{MS-AUTH}}]^{\texttt{NT}}$, $\mathcal{S}$ simulates (in each of the subsequent activations of π_i) a public delayed message $(\texttt{receipt})$ from $[\mathcal{F}_{\text{COM}}]^{\texttt{NT}}_j$ for one of the $p_j \neq p_i$. Whenever the environment acknowledges the delivery of such a message for $[\mathcal{F}_{\text{COM}}]^{\texttt{NT}}_j$, $\mathcal{S}$ simulates the transmission of the message $(\texttt{ack})$ on the channel $[\mathcal{F}_{\text{BD-AUTH}}]^{\texttt{NT}}$ from p_j to p_i. On every subsequent activation of p_i, the machine $\mathcal{S}$ simulates the behavior of the channels $[\mathcal{F}_{\text{BD-AUTH}}]^{\texttt{NT}}$ that π_i is polling, until all acknowledgments (would) have arrived—this is by simply counting the respective activations. On the following activations of p_i, $\mathcal{S}$ simulates the notifications $(\texttt{open}, m_j)$ from the functionalities $[\mathcal{F}_{\text{COM}}]^{\texttt{NT}}_j$, and after the environment acknowledged this opening and $\mathcal{S}$ obtained the notification for the output from $[\mathcal{F}_{\text{MS-AUTH}}]^{\texttt{NT}}$, $\mathcal{S}$ acknowledges this output. If the environment requests to corrupt a party, the simulator can easily construct the expected internal state since all data that does not solely represent the state of the protocol is the message vector $\boldsymbol{m}$ which is provided to $\mathcal{S}$ in the first activation.

Note that the two executions are indeed indistinguishable: The messages produced by the simulator have exactly the correct distribution (except for the opening message, all messages only represent state transitions of the protocol, and the messages m_j in the opening messages are exactly the ones obtained from $[\mathcal{F}_{\text{MS-AUTH}}]^{\texttt{NT}}$). Also, the restriction that $\mathcal{S}$ cannot replace the message vector if p_i was honest while sending $\boldsymbol{m}$ *and* the first message was already delivered to a p_j is consistent with the restrictions in the real execution, as if p_i is corrupted *after* the first commitment is opened, all other messages to the honest receivers are already immutable in the respective commitment functionalities (by the strict ordering of events enforced by π_i and the acknowledgments sent upon receiving a (`receipt`)) and can not be changed unless the respective receivers are corrupted. Hence, the same restrictions apply in the real model. □

Using Lemmas 7 and 8, and the fact that the delayed relaxation of any $\mathcal{F}$ can be realized in the $\mathcal{F}$-hybrid model, we can extend the result of [13] on impossibility of (UC) commitments to our synchronous setting.

Corollary 1. *There exists no protocol which UC-realizes the functionality $\mathcal{F}_{\text{COM}}$ in the $\{\mathcal{F}_{\text{BD-AUTH}}, \mathcal{F}_{\text{CLOCK}}\}$-hybrid model.*

5 Existing Synchronous Models as Special Cases

In this section, we revisit existing models for synchronous computation. We show that the $\mathcal{F}_{\text{SYN}}$-hybrid model as specified in [10] and the Timing model from [26] are sufficient only for non-terminating computation (which can be achieved even in a fully asynchronous environment). We also show that the models of [33] and [25] can be expressed as special cases in our model.

5.1 The $\mathcal{F}_{\text{SYN}}$-Hybrid Model

In [10], a model for synchronous computation is specified by a *synchronous network* functionality $\mathcal{F}_{\text{SYN}}$. On a high level, $\mathcal{F}_{\text{SYN}}$ corresponds to an authenticated network with storage, which proceeds in a round-based fashion; in each round r, every party associated to $\mathcal{F}_{\text{SYN}}$ inputs a vector of messages, where it is guaranteed that (1) the adversary cannot change the message sent by an honest party without corrupting this party, and (2) the round index is only increased after every honest party as well as the adversary have submitted their messages for that round. Furthermore, $\mathcal{F}_{\text{SYN}}$ allows the parties to query the current value of r along with the messages of that round r. For completeness, we have included the detailed description of the functionality $\mathcal{F}_{\text{SYN}}$ in Appendix B.

$\mathcal{F}_{\text{SYN}}$ requires the adversary to explicitly initiate the round switch; this allows the adversary to stall the protocol execution (by not switching rounds). Hence, $\mathcal{F}_{\text{SYN}}$ cannot be used for evaluating a not locally computable function f with guaranteed termination: because we require termination, for every protocol which securely realizes $\mathcal{F}_{\text{SFE}}$ and for every adversary, the environment $\mathcal{Z}$ which gives inputs to all honest parties and issues sufficiently many `fetch` requests has to be able to make π generate its output from the evaluation. This must, in particular, hold when the adversary never commands $\mathcal{F}_{\text{SYN}}$ to switch rounds, which leads to a contradiction.

Lemma 9. *For every not locally computable n-party function f and for every polynomial Rnd: there is no $\mathcal{F}_{\text{SYN}}$-hybrid protocol which securely realizes $\mathcal{F}_{\text{SFE}}^{f,Rnd}$ with guaranteed termination.*

Proof. Let f be a function that is not locally computable and assume, towards contradiction that there is a protocol π which securely realizes $\mathcal{F}_{\text{SFE}}^{f,Rnd}$ in the $\mathcal{F}_{\text{SYN}}$-hybrid model. As f is not locally computable, there are input vectors $\boldsymbol{x}$ and $\boldsymbol{x'}$ and a party $p_i \in \mathcal{P}$ such that $x_i = x'_i$ but $(f(\boldsymbol{x}, R))_i \not\approx (f(\boldsymbol{x'}, R))_i$. Consider the following environment: $\mathcal{Z}$ corrupts no party, tosses a coin and hands inputs either $\boldsymbol{x}$ or $\boldsymbol{x'}$ to the parties. Subsequently, $\mathcal{Z}$ activates the parties $Rnd \cdot |\mathcal{P}|$ times in a round-robin fashion using (`output`)-queries. Throughout this computation, $\mathcal{Z}$ ignores all activations from the adversary and never activates the adversary, but $\mathcal{Z}$ records the values y_i provided as output by the honest parties in the last round of the computation.

In the ideal-world execution (i.e., in the $\mathcal{F}_{\text{SFE}}^{f,Rnd}$-hybrid model with some simulator $\mathcal{S}$), by the definition of $\mathcal{F}_{\text{SFE}}^{f,Rnd}$, this behavior will result in an output vector $\boldsymbol{y}$ that is distributed as either $f(\boldsymbol{x}, R)$ or $f(\boldsymbol{x'}, R)$, depending on the inputs provided by the environment. In contrast, in the $\mathcal{F}_{\text{SYN}}$-hybrid world with the dummy adversary $\mathcal{A}$, the network $\mathcal{F}_{\text{SYN}}$ will not allow the parties to exchange any message (as the round does not advance). As a result, the distribution of the output y_i is the same independent of whether the input provided by $\mathcal{Z}$ is $\boldsymbol{x}$ or $\boldsymbol{x'}$ (as all messages received by the protocol machine of party p_i are exactly the same). $\mathcal{Z}$ uses the distinguisher guaranteed for $(f(\boldsymbol{x}, R))_i$ and $(f(\boldsymbol{x'}, R))_i$ and obtains half the distinguishing advantage (which is still non-negligible). $\square$

The only advantage that $\mathcal{F}_{\text{SYN}}$ offers on top of what can be achieved from (asynchronous) channels is that $\mathcal{F}_{\text{SYN}}$ defines a point in the computation (chosen by $\mathcal{A}$) in which the round index advances for all parties *simultaneously.* This, however, is also achieved by the (ideal) clock functionality $\mathcal{F}_{\text{CLOCK}}$. Indeed, the functionality $\mathcal{F}_{\text{SYN}}$ can be realized by a protocol using $\mathcal{F}_{\text{CLOCK}}$ and fully asynchronous channels. This protocol follows the ideas of [2,26]: In each round r, each party p_i sends a message to all other parties. After receiving messages from all p_j, p_i uses the clock $\mathcal{F}_{\text{CLOCK}}$ to synchronize with all other p_j. Once all parties have acknowledged the reception, p_i prepares the messages received in that round for local output (upon request) and starts the next round (as soon as messages have been provided as local input). This proves the following lemma.[11]

Lemma 10. *There exists a protocol that UC-realizes the functionality $\mathcal{F}_{\text{SYN}}$ in the $\{\mathcal{F}_{\text{CLOCK}}, \mathcal{F}_{\text{AUTH}}\}$-hybrid model.*

5.2 The Timing Model

The "Timing model" [23,26] integrates a notion of time into the protocol execution by extending the model of computation. Each party, in addition to its communication and computation tapes, has a *clock tape* that is writable for the adversary in a monotone and "bounded drift"-preserving manner: The adversary can only increase the value of the clocks, and, for any two parties, the distance ϵ of their clocks' speed (drift) at any point in time is bounded by a known constant. The value of a party's clock tape defines the local time of this party. Depending on this time, protocols delay sending messages or time out if a message has not arrived as expected. We formalize this modeling of time in UC in the following straightforward manner: We introduce a functionality $\mathcal{F}_{\text{TIME}}$ that maintains a clock value for each party and allows the adversary to advance this clock in the same monotone and "bounded drift"-preserving way. Instead of reading the local clock tape, $\mathcal{F}_{\text{TIME}}$-hybrid protocols obtain their local time value by querying $\mathcal{F}_{\text{TIME}}$.

Functionality $\mathcal{F}^{\epsilon}_{\text{TIME}}(\mathcal{P})$

For every party $p_i \in \mathcal{P}$, the functionality $\mathcal{F}_{\text{TIME}}$ maintains an integer t_i (p_i's current time) and two bits $a_i, s_i \in \{0, 1\}$. For initialization, $\mathcal{F}_{\text{TIME}}$ sets $a_i := 0$ and $s_i := 1$ and expects to receive a vector $\boldsymbol{T}$ of $|\mathcal{P}|$ integers from $\mathcal{A}$ and $t_i := \boldsymbol{T}_i$ (otherwise $t_i = 0$ for all $p_i \in \mathcal{P}$).

– Upon receiving $(\texttt{deactivate}, p_i)$ from $\mathcal{A}$, set $a_i := 0$ and $s_i := 0$, and send $(\texttt{ok})$ to $\mathcal{A}$.
– Upon receiving $(\texttt{set}, (\rho_i)_{p_i \in \mathcal{P}})$ from $\mathcal{A}$, check whether $a_i = 0$ for all $p_i \in \mathcal{P}$ and whether setting t_i to ρ_i for all $p_i \in \mathcal{P}$ preserves the property of being ϵ-drift preserving. If this is the case, set $t_i := \rho_i$ and $s_i := 1$ for all $p_i \in \mathcal{P}$. Send $(\texttt{ok})$ to $\mathcal{A}$.
– Upon receiving $(\texttt{activate}, p_i)$ from $\mathcal{A}$, if $s_i = 1$ then set $a_i := 1$. Send $(\texttt{ok})$ to $\mathcal{A}$.
– Upon receiving $(\texttt{time})$ from p_i, if $a_i = 1$ then send $(\texttt{time}, t_i)$ to p_i, otherwise send $(\texttt{inactive})$ to p_i.

The different scheduling schemes used in [10] and [26] impose some further technical details on $\mathcal{F}_{\text{TIME}}$: While in [10], the "master scheduler" activated by default is defined to be the environment $\mathcal{Z}$, this task is fulfilled by the adversary $\mathcal{A}$ in [26]. To be able to show the relation between security statements in the two models, we have to define the functionality $\mathcal{F}_{\text{TIME}}$ such that it "protects" parties from being activated without the acknowledgment of the adversary.

Lemma 11 shows that $\mathcal{F}_{\text{TIME}}$ can be realized from fully asynchronous authenticated communication. Together, Lemmas 11 and 12 demonstrate that "timed" protocols cannot UC-realize more functionalities than non-"timed" protocols, which is consistent with [26, Theorem 2]. In contrast, "timed" protocols *can* securely realize arbitrary functionalities under composition with δ-delayed protocols [26, Theorem 1].

[11] In previous versions of the work, we proved that a *relaxed* version of $\mathcal{F}_{\text{SYN}}$ is achieved even without $\mathcal{F}_{\text{CLOCK}}$. Adapting the statement allows us to omit this additional functionality, while preserving the high-level result due to Lemma 6.

Lemma 11. *Let $\mathcal{P}$ be a player set and $\epsilon \geq 1$. The functionality $\mathcal{F}^{\epsilon}_{\text{TIME}}(\mathcal{P})$ can be UC-realized from pairwise authenticated channels $\mathcal{F}_{\text{AUTH}}$.*

Proof. The protocol τ realizes $\mathcal{F}_{\text{TIME}}$ in the $\mathcal{F}_{\text{AUTH}}$-hybrid model.

Protocol $\tau(\mathcal{P})$

For (each) party p_i, initialize variables $t_j^{(i)} := 0$ and $a_j^{(i)} := 1$ for all $j \in \mathcal{P}$. Initialize a channel $\mathcal{F}_{\text{AUTH}}(p_i, p_j)$ to each $p_j \in \mathcal{P}$. If several messages are to be sent in a single activation, they are stored in a buffer and sent during the following activations.

– Upon a $(\texttt{set-inactive}, \rho_j)$ message from p_j, set $a_j^{(i)} := 0$.
  - If $a_i^{(i)} = 1$ and $t_i^{(i)} \leq t_l^{(i)}$ for all $p_l \in \mathcal{P}$, then become inactive.
  - If for all $l \in \mathcal{P} : ((a_l^{(i)} = 0 \ \wedge \ t_l^{(i)} \geq t_i^{(i)}) \ \vee \ t_l^{(i)} > t_i^{(i)})$, then set $t_i^{(i)} := t_i^{(i)} + 1$ and $a_i^{(i)} := 1$, and send $(\texttt{set-active}, t_i^{(i)})$ to all $p_l \in \mathcal{P} \setminus \{p_i\}$.

– Upon a $(\texttt{set-active}, \rho_j)$ message from p_j with $\rho_j > t_j^{(i)}$, set $a_j^{(i)} := 1$ and $t_j^{(i)} := \rho_j$.

– Upon local input $(\texttt{time})$, if $a_i^{(i)} = 0$ then output $(\texttt{inactive})$, otherwise output $(\texttt{time}, t_i^{(i)})$.

– Upon any activation via the communication tape while no outgoing messages are pending and $a_j^{(i)} = 1$ for all $p_j \in \mathcal{P}$, become inactive.

– Becoming inactive: Set $a_i^{(i)} := 0$ and send a message $(\texttt{set-inactive}, t_i^{(i)})$ to all $p_j \in \mathcal{P} \setminus \{p_i\}$.

Messages that do not match any of the rules are ignored.

The simulator $\mathcal{S}$ for the protocol τ and the dummy adversary $\mathcal{D}$ proceeds as follows. All messages between $\mathcal{Z}$ and (the simulated) $\mathcal{D}$ are simply forwarded. $\mathcal{S}$ also simulates protocol machines τ_i running protocol τ for each party p_i, and maintains buffers for the (simulated) functionalities $\mathcal{F}_{\text{AUTH}}(p_i, p_j)$. During the execution, $\mathcal{S}$ behaves as follows:

– For a message m sent among (the simulated) τ_i and τ_j, simulate $(\texttt{leak}, m)$ from $\mathcal{F}_{\text{AUTH}}(p_i, p_j)$ to $\mathcal{D}$.
– If $\mathcal{D}$ activates τ_j or issues a message $(\texttt{deliver})$ to a (non-empty) channel $\mathcal{F}_{\text{AUTH}}(p_i, p_j)$, run the simulated τ_j with the respective input, and:
  - if τ_j changes $a_j^{(j)}$ from 1 to 0, then send $(\texttt{deactivate}, p_j)$ to $\mathcal{F}_{\text{TIME}}$. If τ_j was the last (honest) protocol machine to make this transition *for the current round value*, then increase the clock counter for all $p_i \in \mathcal{P}$ using the $(\texttt{set}, (\rho_i)_{p_i \in \mathcal{P}})$ message to $\mathcal{F}_{\text{TIME}}$.
  - if τ_j changes $a_j^{(j)}$ from 0 to 1, then send $(\texttt{activate}, p_j)$ to $\mathcal{F}_{\text{TIME}}$.
– Corruption of parties can be handled easily because $\mathcal{S}$ internally simulates the complete state of the protocol.

The described simulator is perfect. In particular, we show that any query by $\mathcal{Z}$ will lead to the same state transitions and replies in the two executions.

There are three main arguments that build the proof: First, the transitions of the variables $a_j^{(i)}$ and $t_j^{(i)}$ within the parties in the real execution and the simulated protocol machines in the ideal execution are consistent for every query of $\mathcal{Z}$. Second, throughout the complete ideal execution, the invariant $a_i = a_i^{(i)}$, $a_i = 1 \Rightarrow t_i = t_i^{(i)}$, $(a_i = 0 \ \wedge s_i = 0) \Rightarrow t_i = t_i^{(i)}$, $(a_i = 0 \ \wedge s_i = 1) \Rightarrow t_i = t_i^{(i)} + 1$, and $t_l^{(j)} \leq t_l$ is always preserved, where a_i, t_i are the values in $\mathcal{F}_{\text{TIME}}$ and $a_i^{(j)}$, $t_i^{(j)}$ are the values within the simulated protocol machine τ_j. Third, the responses of the real and ideal model on the same queries are always identical, given that the state is consistent. (The fact that the protocol is deterministic simplifies some arguments.)

The environment can initiate the following queries at its interfaces (and those of $\mathcal{D}$):

Query $(\texttt{time})$ **at** τ_j**:** In both cases, the response is $(\texttt{inactive})$ if $a_j = 0$ and $(\texttt{time}, t_j)$ if $a_j = 1$ (consistently for the values $a_j^{(j)}$ and $t_j^{(j)}$).

Query (`deliver`) at $\mathcal{F}_{\text{AUTH}}(p_i, p_j)$: If the buffer has the same contents in both cases prior to this query, then the contents is the same after this query. Also, if the buffer is empty, then $\mathcal{Z}$ is activated in both cases.

If the first message in the buffer is a $(\texttt{set-active}, \rho_i)$ message, then both the real and the simulated τ_j will update their internal view of τ_i's state, but will not change their own state.

If the first message in the buffer is a $(\texttt{set-inactive}, \rho_j)$ message, then the state transition of the real and the ideal τ_j will also be the same. We have to argue that the effect on the behavior of $\mathcal{F}_{\text{TIME}}$ coincides with the behavior of τ_j for $(\texttt{time})$-queries.

$a_j^{(j)} = 1 \rightsquigarrow a_j^{(j)} = 0$**:** $\mathcal{S}$ sends $(\texttt{deactivate}, p_j)$ to $\mathcal{F}_{\text{TIME}}$, which sets in $a_j = 0$ and $s_j = 0$. Since neither t_j nor $t_j^{(j)}$ changed, the invariant is preserved.

If $a_l = 0$ and $s_l = 0$ for all $p_l \in \mathcal{H}$, then the invariant implies that $t_l = t_l^{(l)}$. Also, $\mathcal{S}$ (which can easily keep track of this) will advance $t_l := t_l + 1$ for all $p_l \in \mathcal{P}$ by sending $\big(\texttt{set}, (t_l + 1)_{p_l \in \mathcal{P}}\big)$, which will also set $s_l := 1$ for all $p_l \in \mathcal{P}$. Hence, after the switch, we have $a_l = 0$, $s_l = 1$, and $t_l = t_l^{(l)} + 1$ for all $p_l \in \mathcal{H}$, so the invariant is preserved.

$a_j^{(j)} = 0 \rightsquigarrow a_j^{(j)} = 1$**:** $\mathcal{S}$ will send $(\texttt{activate}, p_j)$ to $\mathcal{F}_{\text{TIME}}$, which induces $a_j := 1$.

Note that τ_j only switches if $a_l^{(j)} = 0$ or $t_l^{(j)} > t_j^{(j)}$ for all $p_l \in \mathcal{P}$. If there is an $p_l \in \mathcal{H}$ with $a_l^{(j)} = 1$, then $t_j = t_l \geq t_l^{(j)} > t_j^{(j)}$ and hence $s_j = 1$ and $t_j = t_j^{(j)} + 1$. Otherwise, if $a_l^{(j)} = 0$ for all $p_l \in \mathcal{H}$, then we must have $t_l \geq t_l^{(j)} \geq t_j^{(j)}$, so all parties are inactive and (at least) in the "current" round, so we also have $s_j = 1$ and $t_j = t_j^{(j)} + 1$. Hence, after this transition, we have $a_j = a_j^{(j)} = 1$ and $t_j^{(j)} = t_j$.

Empty activation at τ_j: This is a special case of the above.

The initial state of the system also fulfills the invariant. Note that while corrupted parties can easily stall the computation (by simply withholding the respective messages), they cannot undermine the consistency guarantees of the protocol. Also, note that the protocol is non-trivial: in presence of an adversary that forwards messages among the protocol machines and occasionally activates them via the network, the protocol will indeed make progress. □

Translating functionalities. As for the different definitions of protocol executions, there are several differences between the definitions of ITMs in [26] and those in [10]. In particular, ITMs in [26] have an explicit outgoing communication tape, on which all messages (both "communication" messages intended for communication over the network and "ideal" messages directed to an ideal functionality) are written (albeit the adversary cannot access the contents of the ideal messages, but merely sees their length). ITMs can write an (a priori) arbitrary number of messages for arbitrary receivers to their outgoing communication tape in a single activation. In [10], in contrast, communication is defined as writing directly to the respective input tape of the receiving ITI. This implies that ideal messages are not communicated via $\mathcal{A}$, and that, in a single activation, messages can only be sent to a single receiver. For ideal functionalities, the differences that must be leveraged by the transformation T are: For inputs to and outputs from parties, the lengths of the respective messages are leaked to $\mathcal{A}$. Also, $\mathcal{A}$ may schedule the delivery of any such "ideal" message.

We define the transformation mapping T_T as follows. The functionality $T_T(\mathcal{F})$, upon receiving an input message x from any honest party p_i, sends a notification $(\texttt{input}, p_i, |x|)$ to the adversary. After the adversary has acknowledged this input, $T_T(\mathcal{F})$ simulates x as an input to $\mathcal{F}$. All outputs y of $\mathcal{F}$ to a party p_i are sent by $T_T(\mathcal{F})$ as (private) delayed outputs, where additionally the length $|y|$ is leaked.

Translating protocols. The protocol compiler C_T translates the ITMCs (interactive Turing machines with clocks) from [26] to $\mathcal{F}_{\text{TIME}}$-hybrid ITMs. At the beginning of each activation (except for replies from $\mathcal{F}_{\text{TIME}}$), $C_T(\pi)$ sends a message $(\texttt{time})$ to the functionality $\mathcal{F}_{\text{TIME}}$ and writes the returned time value to the simulated clock tape of π. If $\mathcal{F}_{\text{TIME}}$ replies with $(\texttt{inactive})$, then $C_T(\pi)$ buffers the input (if necessary) and halts. The protocol compiler also leverages for the different semantics of communication. This means that the $\mathcal{F}_{\text{AUTH}}$-hybrid protocol $C_T(\pi)$, instead of writing network messages to the outgoing communication tape, inserts them into the authenticated channel $\mathcal{F}_{\text{AUTH}}(p_i, p_j)$ that p_i shares with the receiver p_j. After the initial input is obtained, $C_T(\pi)$ ignores all further activations via the input tape.[12]

Lemma 12 then shows that any security statement about a functionality $\mathcal{F}$ in the Timing model can be translated into a statement about $T_T(\mathcal{F})$ in the $\{\mathcal{F}_{\text{TIME}}, \mathcal{F}_{\text{AUTH}}\}$-hybrid model (in UC). The proof appears in Section C of the appendix. Following [10, Claim 12], it is sufficient to show the that a protocol is secure in the Timing model if and only if it is secure in UC with respect to specialized simulators.

Lemma 12. *For an arbitrary functionality $\mathcal{F}$ and a protocol π in the Timing model, π securely realizes $\mathcal{F}$ (in the Timing model) if and only if the compiled protocol $C_T(\pi)$ UC-realizes $T_T(\mathcal{F})$ in the $\{\mathcal{F}_{\text{TIME}}, \mathcal{F}_{\text{AUTH}}\}$-hybrid model in the presence of a static*[13] *adversary and with respect to specialized simulators.*

5.3 Models with Explicit Round-Structure

Nielsen's framework [33]. The framework described in [33] is an adaptation of the asynchronous framework of [12] to authenticated synchronous networks. While the general structure of the security definition is adopted, the definition of protocols and their executions differs considerably. For instance, the "subroutine" composition of two protocols is defined in a "lock-step" way: the round switches occur at the same time. Similarly to our bounded-delay channels, messages in transfer can be replaced if the sender becomes corrupted. Lemma 13 allows to translate, along the lines of Section 5.2, any security statement in the model of [33] into a security statement about a synchronous protocol in the $\{\mathcal{F}_{\text{CLOCK}}, \mathcal{F}_{\text{BD-AUTH}}\}$-hybrid model. As in the previous section, the translation is done by a functionality compiler $T_N(\cdot)$ that resolves the type mismatch between the functionalities in UC and in [33], and a corresponding protocol compiler $C_N(\cdot)$. The descriptions of these compilers along with the proof of Lemma 13 are provided in Appendix D, where we also briefly describe the model of [33]. We emphasize that the converse statement of Lemma 13 does *not* hold, i.e., there are UC statements about synchronous protocols that cannot be modeled in the [33] framework. For instance, our synchronous UC model allows protocols to use further functionalities that run mutually asynchronously with the synchronous network, which cannot be modeled in [33].

Lemma 13. *For an arbitrary functionality $\mathcal{F}$ and a protocol π in [33], π securely realizes $\mathcal{F}$ (in [33]) if and only if the compiled protocol $C_N(\pi)$ UC-realizes $T_N(\mathcal{F})$ in the $\{\mathcal{F}_{\text{CLOCK}}, \mathcal{F}_{\text{BD-AUTH}}\}$-hybrid model.*

Hofheinz and Müller-Quade's framework [25]. The framework of [25] also models authenticated synchronous networks based on the framework of [12], but the rules of the protocol execution differ considerably: The computation proceeds in rounds, and each round is split into three phases. In each phase, only a subset of the involved ITIs are activated, and the order of the activations follows a specific scheme. The adversary has a relaxed *rushing* property: while being the last to specify the messages for a round, he cannot corrupt parties within a round. This corresponds to a network with guarantees that are stronger than simultaneous multi-send: once the first message of an honest party is provided to the adversary, all messages of honest parties are guaranteed to be delivered correctly.[14] We model this relaxed rushing property in UC by the functionality $\mathcal{F}_{\text{MS+}}$ (see Appendix E), which is a modified version of $\mathcal{F}_{\text{MS}}$ and exactly captures this guarantee. As before, we translate the security statements of [25] to our model (where $\mathcal{F}_{\text{MS+}}$ is used instead of $\mathcal{F}_{\text{AUTH}}$) through a pair of compilers $(T_H(\cdot), C_H(\cdot))$.

Lemma 14. *For an arbitrary functionality $\mathcal{F}$ and a protocol π in [25], π securely realizes $\mathcal{F}$ (in [25]) if and only if the compiled protocol $C_H(\pi)$ UC-realizes $T_H(\mathcal{F})$ in the $\{\mathcal{F}_{\text{CLOCK}}, \mathcal{F}_{\text{MS+}}\}$-hybrid model.*

[12] In contrast to the Timing model, protocols in the model of [10] can only send a single message within an activation. On the other hand, this property of the Timing model leaks some additional information to the adversary, namely the information that the messages on the outgoing communication tape are all messages to be sent prior to the next input. In order to equip the UC-adversary with the same power, we define that the protocol π sends an empty message to the adversary if it is activated for sending but there are no pending messages.

[13] The Timing model [26] considers only static adversaries.

[14] In the context of [25], this is an advantage as it strengthens the impossibility result.

6 Conclusion

We described a modular security model for synchronous computation within the (otherwise inherently asynchronous) UC framework by specifying the real-world synchrony assumptions of bounded-delay channels and loosely synchronized clocks as functionalities. The design principle that underlies these functionalities allows us to treat guaranteed termination; previous approaches for synchronous computation within UC either required fundamental modifications of the framework (which also required re-proving fundamental statements) or did not allow to make such statements altogether.

The framework faithfully models the guaranteed expected from synchronous networks within the formal UC framework, which we demonstrated by revisiting and translating basic results from the literature on synchronous protocols. Furthermore, the flexibility of our framework allows to cast previous specialized frameworks as special cases of our model by introducing network functionalities that provide the guarantees formalized in those models.

Acknowledgments. We thank Ran Canetti for interesting discussions and comments on the comparison of our model with the $\mathcal{F}_{\text{SYN}}$-hybrid formulation. The first author was supported by NSF awards #0447075, #1111599, and #1223623. The second and third authors were supported by the Swiss National Science Foundation (SNF), project no. 200020-132794. The fourth author was partially supported by a fellowship from the SNF, project no. PBEZP2-134445.

References

1. Asharov, G., Lindell, Y., Rabin, T.: Perfectly-Secure Multiplication for Any $t < n/3$. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 240–258. Springer, Heidelberg (2011)
2. Awerbuch, B.: Complexity of Network Synchronization. Journal of the ACM 32, pp. 804–823 (1985)
3. Backes, M.: Unifying Simulatability Definitions in Cryptographic Systems under Different Timing Assumptions. In: Amadio, R., Lugiez, D. (eds.) CONCUR 2003. LNCS, vol. 2761, pp. 350–365. Springer, Heidelberg (2003)
4. Backes, M., Hofheinz, D., Müller-Quade, J., Unruh, D.: On Fairness in Simulatability-based Cryptographic Systems. In: Proceedings of FMSE, pp. 13–22. ACM (2005)
5. Backes, M., Pfitzmann, B., Steiner, M., Waidner, M.: Polynomial Fairness and Liveness. In: Proceedings of the 15th Annual IEEE Computer Security Foundations Workshop, pp. 160–174. IEEE (2002)
6. Backes, M., Pfitzmann, B., Waidner, M.: The Reactive Simulatability (RSIM) Framework for Asynchronous Systems. Information and Computation 205, pp. 1685–1720 (2007)
7. Ben-Or, M., Canetti, R., Goldreich, O.: Asynchronous Secure Computation. In: Proceedings of the 25th Annual ACM Symposium on Theory of Computing, pp. 52–61. ACM (1993)
8. Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. In: Proceedings of the 20th Annual ACM Symposium on Theory of Computing, pp. 1–10. ACM (1988)
9. Canetti, R.: Studies in Secure Multiparty Computation and Applications. PhD thesis, The Weizmann Institute of Science (1996)
10. Canetti, R.: Universally Composable Security: A New Paradigm for Cryptographic Protocols. In: Cryptology ePrint Archive, Report 2000/067 (2005)
11. Canetti, R.: Security and Composition of Multiparty Cryptographic Protocols. In: Journal of Cryptology 13, pp. 143–202 (2000)
12. Canetti, R.: Universally Composable Security: A New Paradigm for Cryptographic Protocols. In: Proceedings of the 42nd Annual IEEE Symposium on Foundations of Computer Science, pp. 136–145. IEEE (2001)
13. Canetti, R., Fischlin, M.: Universally Composable Commitments. In: Kilian, J. (ed.) CRYPTO 2001, LNCS, vol. 2139, pp. 19–40. Springer, Heidelberg (2001)
14. Canetti, R., Krawczyk, H.: Universally Composable Notions of Key Exchange and Secure Channels. In: Knudsen, L. R. (ed.) EUROCRYPT 2002, LNCS, vol. 3027, pp. 337–351. Springer, Heidelberg (2002)
15. Canetti, R., Lindell, Y., Ostrovsky, R., Sahai, A.: Universally Composable Two-Party and Multi-Party Secure Computation. In: Proceedings of the 34th Annual ACM Symposium on Theory of Computing, pp. 494–503. ACM (2002)
16. Chaum, D., Crépeau, C., Damgård, I.: Multiparty Unconditionally Secure Protocols. In: Proceedings of the 20th Annual ACM Symposium on Theory of Computing, pp. 11–19. ACM (1988)
17. Chor, B., Moscovici, L.: Solvability in Asynchronous Environments. In: Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science, pp. 422–427. IEEE (1989)
18. Dodis, Y., Micali, S.: Parallel Reducibility for Information-Theoretically Secure Computation In: Bellare, M. (ed.) CRYPTO 2000. LNCS, vol. 1880, pp. 74–92. Springer, Heidelberg (2000)
19. Dolev, D., Strong, H. R.: Polynomial Algorithms for Multiple Processor Agreement. In: Proceedings of the 14th Annual ACM Symposium on Theory of Computing, pp. 401–407. ACM (1982)
20. Dwork, C., Naor, M., Sahai, A.: Concurrent Zero-Knowledge. In: Proceedings of the 30th Annual ACM Symposium on Theory of Computing, pp. 409–418. ACM (1998)
21. Garay, J. A., Katz, J., Kumersan, R., Zhou, H.-S.: Adaptively Secure Broadcast, Revisited. In: Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing, pp. 179–186. ACM (2011)
22. Goldreich, O., Micali, S., Wigderson, A.: How to Play any Mental Game—A Completeness Theorem for Protocols with Honest Majority. In: Proceedings of the 19th Annual ACM Symposium on Theory of Computing, pp. 218–229. ACM (1987)
23. Goldreich, O.: Concurrent Zero-Knowledge with Timing, Revisited. In: Proceedings of the 34th Annual ACM Symposium on Theory of Computing, pp. 332–340. ACM (2002)
24. Hirt, M., Zikas, V.: Adaptively Secure Broadcast. In: Gilbert, H. (ed.) EUROCRYPT 2010, LNCS, vol. 6110, pp. 466–485. Springer, Heidelberg (2010)
25. Hofheinz, D., Müller-Quade, J.: A Synchronous Model for Multi-Party Computation and the Incompleteness of Oblivious Transfer. In: Proceedings of Foundations of Computer Security — FCS'04, pp. 117–130. (2004)
26. Kalai, Y. T., Lindell, Y., Prabhakaran, M.: Concurrent General Composition of Secure Protocols in the Timing Model. In: Proceedings of the 37th Annual ACM Symposium on Theory of Computing, pp. 644–653. ACM (2005)
27. Künzler, R., Müller-Quade, J., Raub, D.: Secure Computability of Functions in the IT Setting with Dishonest Majority and Applications to Long-Term Security. In: Reingold, O. (ed.) Theory of Cryptography, LNCS, vol. 5444, pp. 238–255. Springer, Heidelberg (2009)

28. Kushilevitz, E., Lindell, Y., Rabin, T.: Information-theoretically Secure Protocols and Security under Composition. In: Proceedings of the 38th Annual ACM Symposium on Theory of Computing, pp. 109–118. ACM (2006)
29. Maji, H., Prabhakaran, M., Rosulek, M.: Cryptographic Complexity Classes and Computational Intractability Assumptions. In: Innovations in Computer Science. Tsinghua University Press (2010)
30. Maurer, U.: Constructive Cryptography: A New Paradigm for Security Definitions and Proofs. In: Mödersheim, S., Palamidessi, C. (eds.) TOSCA, LNCS, vol. 6993, pp. 33–56. Springer, Heidelberg (2011)
31. Maurer, U., Renner, R.: Abstract Cryptography. In: Innovations in Computer Science. Tsinghua University Press (2011)
32. Maurer, U., Tackmann, B.: Synchrony Amplification. In: International Symposium on Information Theory Proceedings, pp. 1583–1587. IEEE (2012)
33. Nielsen, J. B.: On Protocol Security in the Cryptographic Model. PhD thesis, University of Aarhus (2003)
34. Rabin, T., Ben-Or, M.: Verifiable Secret Sharing and Multiparty Protocols with Honest Majority. In: Proceedings of the 21st Annual ACM Symposium on Theory of Computing, pp. 73–85. ACM (1989)

A Supplementary Material

This section contains supplementary material that has been deferred from the main paper, but which is useful to consult while reading the paper. Some of the functionalities presented here are material imported from other papers and restated using our conventions, and some of the functionalities are only described informally in the body of the paper.

A.1 Secure Function Evaluation

An (albeit hard to implement) version of the secure function evaluation (SFE) functionality $\mathcal{F}_{\text{SFE}}$ appears in [10]. The functionality models the strictest version of termination one can hope for: once all honest parties have provided input, the functionality will produce output in the next activation. This functionality cannot be implemented if the network leaks any information about transferred messages (e.g., the fact that a message is transferred). Also, depending on the set-up used by protocols, the functionality must potentially be weakened to drop guarantees if too many parties become corrupted.

Functionality $\mathcal{F}^f_{\text{SFE}}(\mathcal{P})$

$\mathcal{F}^f_{\text{SFE}}$ proceeds as follows, given a function $f : (\{0,1\}^* \cup \{\bot\})^n \times R \to (\{0,1\}^*)^n$ and a player set $\mathcal{P}$. Initialize the variables $x_1, \dots, x_n, y_1, \dots, y_n$ to a default value $\bot$.

- Upon receiving input $(\texttt{input}, v)$ from some party $p_i \in \mathcal{P}$, set $x_i := v$ and send a message $(\texttt{input}, p_i)$ to the adversary.
- Upon receiving input $(\texttt{output})$ from some party p_i with $p_i \in \mathcal{P}$, do:
  - If x_j has been set for all $p_j \in \mathcal{H}$, and $y_1, \dots, y_n$ have not yet been set, then choose $r \overset{R}{\leftarrow} R$ and set $(y_1, \dots, y_n) := f(x_1, \dots, x_n, r)$.
  - Output y_i to p_i.

A.2 Secure (Reactive) Multi-Party Computation

We next describe the functionality $\mathcal{F}_{\text{MPC}}$ for reactive multi-party computation. For simplicity, we first describe the extension of $\mathcal{F}_{\text{SFE}}$ from [10] to the reactive case. As with the non-reactive function evaluation (SFE), this MPC functionality is too strong to be implemented in "reasonable" synchronous networks. However, introducing it will allow us to deal with reactiveness separately from the problems that occur due to delays and synchronization issues.

A reactive computation can be specified as an ordered sequence of secure function evaluations which can maintain a joint state. The state used to evaluate any function in this sequence is passed on to the subsequent functions. For simplicity we assume, as in the case of $\mathcal{F}_{\text{SFE}}$, that each time a party is expected to give input to (resp. receive output from) the computation, *every* party gives an—potentially void—input (resp. receives output). More precisely, the computation is described as a vector of functions $\boldsymbol{f} = (f_1, \dots, f_m)$, where each $f_\lambda \in \boldsymbol{f}$ takes as input a vector of values from $\{0,1\}^* \cup \{\bot\}$, a uniformly random value r from a known domain R, as well as a *state vector* $\boldsymbol{\mathcal{S}}_\lambda \in ((\{0,1\}^* \cup \{\bot\})^n \times R)^{(\lambda-1)}$. The state vector $\boldsymbol{\mathcal{S}}_\lambda$ contains the inputs and randomness used for evaluating the functions $f_1, \ldots, f_{\lambda-1}$; each $f_\lambda \in \boldsymbol{f}$ outputs a vector of strings $\boldsymbol{y}_\lambda \in \{0,1\}^n$.

The functionality $\mathcal{F}_{\text{MPC}}^{\boldsymbol{f}}$ is parametrized by the vector of functions $\boldsymbol{f} = (f_1, \ldots, f_m)$. For each $f_\lambda \in \boldsymbol{f}$, $\mathcal{F}_{\text{MPC}}$ might receive an input $x_{i,\lambda}$ from party i at any point before f_λ is evaluated (as soon as f_λ is evaluated and the output has been given to some party, $\mathcal{F}_{\text{MPC}}^{\boldsymbol{f}}$ stops accepting inputs for f_λ). In that case, $\mathcal{F}_{\text{MPC}}$ records $x_{i,\lambda}$ as p_i's input to the evaluation of f_λ. We denote the vector of the parties' inputs to the function f_λ as $\boldsymbol{x}_\lambda = (x_{1,\lambda}, \ldots, x_{n,\lambda})$. Similarly, we denote the vector of the parties' outputs from f_λ as $\boldsymbol{y}_\lambda = (y_{1,\lambda}, \ldots, y_{n,\lambda})$. We say that $\boldsymbol{x}_\lambda$ (resp. $\boldsymbol{y}_\lambda$) *has been set* if all the *honest* parties have handed $\mathcal{F}_{\text{MPC}}$ their input for f_λ (resp. the output of f_λ has already been computed). When a request for producing output for some f_λ is received from some honest p_j, $\mathcal{F}_{\text{MPC}}$ checks that the input $\boldsymbol{x}_\lambda$ and all outputs $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_{\lambda-1}$ have been set; if so, $\mathcal{F}_{\text{MPC}}$ computed f_λ's outcome $\boldsymbol{y}_\lambda = (y_{1,\lambda}, \ldots, y_{n,\lambda})$ using the inputs and the state vector $\boldsymbol{\mathcal{S}}_\lambda$, records the output, and hands to p_j his part of the output, i.e., $y_{i,\lambda}$. Recall that the state consist of the inputs to all previous functions and the corresponding randomness, i.e. $\boldsymbol{\mathcal{S}}_\lambda = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{\lambda-1}, r_1, \ldots, r_{\lambda-1})$. A detailed description of $\mathcal{F}_{\text{MPC}}^{\boldsymbol{f}}$ appears in the following.

Functionality $\mathcal{F}_{\text{MPC}}^{\boldsymbol{f}}(\mathcal{P})$

$\mathcal{F}_{\text{MPC}}^{\boldsymbol{f}}$ proceeds as follows, given a vector of functions $\boldsymbol{f} = (f_1, \ldots, f_m)$ and a player set $\mathcal{P}$, where $|\mathcal{P}| = n$, and for $\lambda = 1, \ldots, m$, $f_\lambda : (\{0,1\}^* \cup \{\bot\})^n \times ((\{0,1\}^* \cup \{\bot\})^n \times R)^{(\lambda-1)} \times R \to (\{0,1\}^*)^n$. Initialize the variables $x_{1,1}, \ldots, x_{n,m}, y_{1,1}, \ldots, y_{n,m}$ to a default value $\bot$ and $\boldsymbol{\mathcal{S}}_0 := (\bot, \ldots, \bot)$.

– Upon receiving input $(\texttt{input}, \lambda, v)$ from some party $p_i \in \mathcal{P}$ and $\lambda \in \{1, \ldots, m\}$, if $\boldsymbol{y}_\lambda$ has not yet been set, then set $x_{i,\lambda} := v$ and send a message $(\texttt{input}, p_i, \lambda)$ to the adversary.
– Upon receiving input $(\texttt{output}, \lambda)$ from some party $p_i \in \mathcal{P}$ and $\lambda \in \{1, \ldots, m\}$, do:
  - If $\boldsymbol{x}_\lambda$ and $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_{\lambda-1}$ have been set, and $\boldsymbol{y}_\lambda$ has not yet been set, then choose $r_\lambda \overset{R}{\leftarrow} R$ and set $(y_{1,\lambda}, \ldots, y_{n,\lambda}) := f(\boldsymbol{x}_\lambda, \boldsymbol{\mathcal{S}}_{\lambda-1}, r_\lambda)$; also set $\boldsymbol{\mathcal{S}}_\lambda := (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_\lambda, r_1, \ldots, r_\lambda)$.
  - Output $y_{i,\lambda}$ to p_i.

As already mentioned, the above functionality cannot be realized under "reasonable" synchrony assumptions. Indeed, one can verify that this would contradict Lemma 4, as $\mathcal{F}_{\text{SFE}}^{f}$ is the same as $\mathcal{F}_{\text{MPC}}^{\boldsymbol{f}'}$, where $\boldsymbol{f}' = (f)$. For that reason, we modify it along the ways of the modification we did for the SFE functionality from [10]: the MPC functionality is parametrized, in addition to the vector of functions $\boldsymbol{f} = (f_1, \ldots, f_m)$, by a vector of round functions $\boldsymbol{Rnd} = (Rnd_1(\cdot), \ldots, Rnd_m(\cdot))$, where each $Rnd_\lambda \in \boldsymbol{Rnd}$ is a function of the security parameter which specifies the number of rounds that are used for evaluating f_λ. As in the case of $\mathcal{F}_{\text{SFE}}$, in each round the simulator is given $|\mathcal{P}|$ activations for each party p_i, to simulate the messages sent from p_i in that round. Because Rnd_λ corresponds to the number of rounds needed for the evaluation of f_λ, and the output of f_λ should be generated only after the outputs of $f_1, \ldots, f_{\lambda-1}$ have been set, if the MPC functionality receives a request for output (from some $p_i \in \mathcal{H}$) for f_λ in round $\ell < \sum_{\rho=1}^{\lambda} Rnd_\rho$, then it notifies p_i that it is too early for generating this output.

Definition 2 (Guaranteed Termination—Reactive Computation). *We say that a protocol UC-securely realizes a (possibly reactive) computation described by the function vector* $\boldsymbol{f}$ with guaranteed termination, *if it UC-realizes a functionality* $\mathcal{F}_{\text{MPC}}^{\boldsymbol{f},\boldsymbol{Rnd}}$ *for some vector of round functions* $\boldsymbol{Rnd}$.

All the results of Section 4.2 that are stated for $\mathcal{F}_{\text{SFE}}^{f}$ and for $\mathcal{F}_{\text{SFE}}^{f,Rnd}$, i.e., Lemma 4 and Theorem 1, apply also to $\mathcal{F}_{\text{MPC}}^{\boldsymbol{f}}$ and $\mathcal{F}_{\text{MPC}}^{\boldsymbol{f},\boldsymbol{Rnd}}$, respectively.

B Canetti's Universally Composable Synchronous Network

In the 2005 version of the UC framework [10], Canetti describes a functionality that models a synchronous network. We describe this functionality $\mathcal{F}_{\text{SYN}}$ adapted to our conventions. We assume that all the parties in $\mathcal{P}$ are aware that the corresponding synchronous session has started. (If we do not want to make this assumption, we can have $\mathcal{F}_{\text{SYN}}$ send an initialization message to every player in $\mathcal{P}$.)

Functionality $\mathcal{F}_{\text{MPC}}^{\boldsymbol{f},\boldsymbol{Rnd}}(\mathcal{P})$

$\mathcal{F}_{\text{SFE}}^{\boldsymbol{f},\boldsymbol{Rnd}}$ proceeds as follows, given a vector of functions $\boldsymbol{f} = (f_1, \ldots, f_m)$, a vector of round functions $\boldsymbol{Rnd} = (Rnd_1, \ldots, Rnd_m)$, and a player set $\mathcal{P}$, where $|\mathcal{P}| = n$, and for $\lambda = 1, \ldots, m$, $f_\lambda : (\{0,1\}^* \cup \{\perp\})^n \times ((\{0,1\}^* \cup \{\perp\})^n \times R)^{(\lambda-1)} \times R \to (\{0,1\}^*)^n$. Initialize the variables $x_{1,1}, \ldots, x_{n,m}, y_{1,1}, \ldots, y_{n,m}$ to a default value $\perp$, initialize $\boldsymbol{\mathcal{S}}_0 := (\perp, \ldots, \perp)$, and for each pair $(p_i, \lambda) \in \mathcal{P} \times \{1, \ldots, m\}$ initialize the delay $t_{i,\lambda} := |\mathcal{P}|$. Moreover, initialize a global round counter $\ell := 1$.

- Upon receiving input $(\texttt{input}, \lambda, v)$ from some party $p_i \in \mathcal{P}$ and $\lambda \in \{1, \ldots, m\}$, if $\boldsymbol{y}_\lambda$ has not yet been set, then set $x_{i,\lambda} := v$ and send a message $(\texttt{input}, p_i, \lambda)$ to the adversary.
- Upon receiving input $(\texttt{output}, \lambda)$ from some party $p_i \in \mathcal{P}$ and $\lambda \in \{1, \ldots, m\}$, if $p_i \in \mathcal{H}$ and for some $\rho \in \{1, \ldots, \lambda\}$ the input $x_{i,\rho}$ has not yet been set then ignore p_i's message, else do:
  - If $t_{i,\lambda} > 1$, then set $t_{i,\lambda} := t_{i,\lambda} - 1$. If (now) $t_{j,\lambda} = 1$ for all $p_j \in \mathcal{H}$, then set $\ell := \ell + 1$ and $t_{j,\lambda} := |\mathcal{P}|$ for all $p_j \in \mathcal{P}$. Send $(\texttt{activated}, p_i, \lambda)$ to the adversary.
  - Else, if $t_{i,\lambda} = 1$ but $\ell < \sum_{\rho=1}^{\lambda} Rnd_\rho$, then send $(\texttt{early})$ to p_i.
  - Else,
    - * If $\boldsymbol{x}_\lambda$ and $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_{\lambda-1}$ have been set, and $\boldsymbol{y}_\lambda$ has not yet been set, then choose $r_\lambda \overset{R}{\leftarrow} R$ and set $(y_{1,\lambda}, \ldots, y_{n,\lambda}) := f(\boldsymbol{x}_\lambda, \boldsymbol{\mathcal{S}}_{\lambda-1}, r_\lambda)$; also set $\boldsymbol{\mathcal{S}}_\lambda := (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_\lambda, r_1, \ldots, r_\lambda)$.
    - * Output $y_{i,\lambda}$ to p_i.

Functionality $\mathcal{F}_{\text{SYN}}$ $(\mathcal{P})$

Initialize a round counter $r := 1$.

- Upon receiving input $(\texttt{send}, \boldsymbol{M})$ from a party $p \in \mathcal{P}$, where $\boldsymbol{M}$ is a vector of n messages (one for each party in $\mathcal{P}$), record $(p, \boldsymbol{M}, r)$ and output $(p, \boldsymbol{M}, r)$ to the adversary. (If p later becomes corrupted then the record $(p, \boldsymbol{M}, r)$ is deleted.)
- Upon receiving a message $(\texttt{Advance-Round}, \boldsymbol{N})$ from the adversary, do: If there exists $p \in \mathcal{H}$ for which no record $(p, \boldsymbol{M}, r)$ exists then ignore the message. Else:
  1. Interpret $\boldsymbol{N}$ as the list of messages sent by corrupted parties in the current round r.
  2. Prepare for each $p \in \mathcal{P}$ the list L_p^r of messages that were sent to it in round r.
  3. Update $r := r + 1$.
- Upon receiving input $(\texttt{receive})$ from party p, output $(\texttt{sent}, L_p^{r-1}, r)$ to p. (Let $L_p^0 = \perp$).

We begin by describing the protocol in more detail. Each protocol machine keeps internally a vector of "current" received messages which is initialized to $\perp$. Whenever a protocol machine obtains an input $(\texttt{receive})$ on the input tape, it writes this "current" vector on the subroutine output tape of it's "parent" ITI. For each party p_i, each round r starts by obtaining input $(\texttt{send}, \boldsymbol{M})$ on the input tape. (Ignore all further such requests until the next round switch.) For each $p_j \in \mathcal{P} \setminus \{p_i\}$, the party p_i will then send the message M_j to p_j. This process takes $n-1$ activations, where the first one comes from the $\texttt{send}$-request and the following $n-2$ activations are obtained from $\mathcal{A}$. The party p_i then waits until it obtains all messages from $p_j \neq p_i$ on the channels $\mathcal{F}_{\text{AUTH}}(p_j, p_i)$, and then sends $(\texttt{RoundOK})$ to $\mathcal{F}_{\text{CLOCK}}$. All subsequent activations are used to initially query $(\texttt{RequestRound})$ at $\mathcal{F}_{\text{CLOCK}}$, after such a query returns $d_i = 0$, the received messages are stored to be the "current" messages and further sending requests on the input tape are processed.

Lemma 10. There exists a protocol that UC-realizes the functionality $\mathcal{F}_{\text{SYN}}$ in the $\{\mathcal{F}_{\text{CLOCK}}, \mathcal{F}_{\text{AUTH}}\}$-hybrid model.

Proof (idea). We describe a simulator $\mathcal{S}$ as follows:

- Receiving the message $(p, \boldsymbol{M}, r)$ from $\mathcal{F}_{\text{SYN}}$ for some party $p \in \mathcal{P}$ in round r, simulate the first message $(\texttt{sent}, M_j)$ for some $p_j \neq p$ on the $\mathcal{F}_{\text{AUTH}}$ channel from p to p_j. The following $n-2$ activations of p are handled by simulating the next $n-2$ such messages.
- For further activations of p, until all messages intended for p in the current round have been delivered on the simulated channels $\mathcal{F}_{\text{AUTH}}(p_i, p)$ for $p_i \neq p$, the corresponding activations are merely dropped. The last such message leads to the simulation of $(\texttt{switch}, p)$ from $\mathcal{F}_{\text{CLOCK}}$.

– Once the messages $(\texttt{switch}, p)$ have been simulated for all $p \in \mathcal{H}$, input $(\texttt{Advance-Round}, \boldsymbol{N})$ with the vector $\boldsymbol{N}$ of messages sent by the corrupted parties to $\mathcal{F}_{\text{SYN}}$.

Corruption is treated in the obvious way: message replacements are allowed only if the messages are not supposed to be delivered to the parties already. (This can be handled by $\mathcal{S}$ easily, as the activations via $\mathcal{A}$ let $\mathcal{S}$ track which messages have been obtained by the parties and which have not. Moreover, at the point where $\mathcal{S}$ has to hand the vector $\boldsymbol{N}$ to $\mathcal{F}_{\text{SYN}}$, all messages are from the honest parties are determined already.) □

C Transferring Statements from the Timing Model

The goal of the "Timing model" [26] is to examine the guarantees that can be used by cryptographic protocols if certain guarantees concerning time are assumed. The respective guarantees considered by the model are two-fold.

Bounded clock drift: The parties have clocks that are, for some global parameter $\epsilon \geq 1$, *ϵ-drift preserving*: When some local clock advanced by time δ, all other local clocks must have advanced by time δ' with $\delta/\epsilon < \delta' < \epsilon \cdot \delta$.

Maximum latency: There is an upper bound Δ on the time it takes to compute, send, and deliver messages between parties via the assumed channels.

To be able to compare our security model to the Timing model from [26], we provide an analogous extension of the protocol machines. The model of [26] defines the protocol machines to be ITMCs—interactive Turing machines with clocks. These protocol machines have a specific *clock tape* that can be written to by the adversary, under the restriction that the values are advanced according to the *ϵ-drift preserving* property. Unlike for other tapes, an ITM is not activated when the adversary writes to the clock tape. We model this extension of the ITMs as an ideal functionality $\mathcal{F}_{\text{TIME}}$ that is available to the protocol machines, where instead of accessing the local clock tape, the protocol machines access the ideal functionality $\mathcal{F}_{\text{TIME}}$. As in [26], the adversary fulfills the task of actually advancing the value of the clock, where the functionality $\mathcal{F}_{\text{TIME}}$ ensures that the ϵ-drift preserving property is preserved.

Lemma 12. For an arbitrary functionality $\mathcal{F}$ and a protocol π in the Timing model, π securely realizes $\mathcal{F}$ (in the Timing model) if and only if the compiled protocol $C_T(\pi)$ UC-realizes $T_T(\mathcal{F})$ in the $\{\mathcal{F}_{\text{TIME}}, \mathcal{F}_{\text{AUTH}}\}$-hybrid model in the presence of a static[15] adversary and with respect to specialized simulators.

Proof. We describe how the adversarial interfaces in Timing model and in the $\{\mathcal{F}_{\text{TIME}}, \mathcal{F}_{\text{AUTH}}\}$-hybrid model of UC can be translated. Adversaries or simulators can be translated between the Timing model and UC by applying the suitable conversion strategies at both the interface to the "execution" and the interface to the environment.

The interface to the real execution. In the execution in the Timing model, the adversary can communicate with the protocol machines π_i (by sending a message "earmarked" for π_i to p_i, respectively receiving a message from π_i via p_i), and can write (valid) time values to the clock tapes of the p_i. In UC, the adversary can communicate with the protocol machines $C_T(\pi)_i$, the authenticated channels $\mathcal{F}_{\text{AUTH}}(p_i, p_j)$, and the time functionality $\mathcal{F}_{\text{TIME}}$ via the respective communication tapes. The actions in the Timing model translate as follows.

Updating the clock tapes of $(\pi_i)_{p_i \in \mathcal{P}}$ to $(\rho_i)_{p_i \in \mathcal{P}}$: sending $(\texttt{deactivate}, p_i)$ to $\mathcal{F}_{\text{TIME}}$ for all $p_i \in \mathcal{P}$, sending a message $(\texttt{set}, (\rho_i)_{p_i \in \mathcal{P}})$ to $\mathcal{F}_{\text{TIME}}$, and activating all p_i by sending $(\texttt{activate}, p_i)$ to $\mathcal{F}_{\text{TIME}}$.

Obtaining messages sent by π_i: obtaining the messages one-by-one by activating $C_T(\pi)_i$ until no further message is sent. Messages for π_j correspond to $(\texttt{leak}, m)$ from $\mathcal{F}_{\text{AUTH}}(p_i, p_j)$.

Delivering message from π_i to π_j: issuing $(\texttt{deliver}, m)$ to $\mathcal{F}_{\text{AUTH}}(p_i, p_j)$.

Sending a message for a corrupt π_i: inserting the message into the corresponding $\mathcal{F}_{\text{AUTH}}(p_i, p_j)$.

In between two activations of players, the Timing model adversary must update either the clock tapes of all parties or none at all. This stems from the definition of the ϵ-drift preserving property. In UC, this is captured by the functionality $\mathcal{F}_{\text{TIME}}$ that takes a vector of values for all parties.

[15] The Timing model [26] considers only static adversaries.

The interface to the ideal execution. In the execution in the Timing model, the adversary can communicate with the ideal functionality $\mathcal{F}$ using the communication tapes of $\mathcal{F}$. Also, the adversary can forward ideal messages from $\mathcal{F}$ to p_i and vice versa by copying them to the corresponding tape. In UC, the adversary can communicate with $T_T(\mathcal{F})$ via communication tapes.

Obtaining ideal messages from $\mathcal{F}$ for ψ_i: obtaining notifications (including message length) for the delayed output from $T_T(\mathcal{F})$.
Obtaining ideal messages from ψ_i for $\mathcal{F}$: obtaining notifications $(\texttt{input}, p_i, |x|)$ from $T_T(\mathcal{F})$.
Delivering output from $\mathcal{F}$: acknowledging the delivery of the delayed output at $T_T(\mathcal{F})$.
Delivering input to $\mathcal{F}$: acknowledging the delivery of the delayed input at $T_T(\mathcal{F})$.
Communication with $\mathcal{F}$: communication with $T_T(\mathcal{F})$.

The interface to the environment. In UC, the adversary communicates with the environment directly via its local input and the environment's subroutine output tape. In the Timing model, the role of the environment is taken by the malicious protocol ψ, which communicates with the adversary using the generic communication tapes. The translation is straightforward: a message from or to $\mathcal{Z}$ corresponds directly to a message from or to some protocol machine running ψ. Note that, in UC, the environment is the master scheduler, whereas this task is fulfilled by the adversary in the Timing model. This difference must be accounted for by carefully designing input/output behavior of all involved systems.

Security in the Timing model implies security in UC. To show security in UC with respect to specialized simulators, we have to show that for each adversary $\mathcal{A}$ and environment $\mathcal{Z}$, there is a simulator $\mathcal{S}$ such that the outcomes of the real and the ideal execution are indistinguishable. We convert $\mathcal{A}$ into a Timing model adversary $\mathcal{A}'$ and $\mathcal{Z}$ into a malicious protocol $\psi^{\mathcal{Z}}$ and conclude that there is a good Timing model simulator $\mathcal{S}'$. From this simulator, we construct a UC-simulator $\mathcal{S}$ and show that the output of $\mathcal{Z}$ and the Timing model distinguisher D are the same for both the real and the ideal executions.

The adversary $\mathcal{A}'$ is constructed from the adversary $\mathcal{A}$ by the above described interface transformations. We detail the translation of $\mathcal{A}$'s clock tape handling to $\mathcal{A}'$: The definition of the adversarial interface of $\mathcal{F}_{\text{TIME}}$ guarantees that the values of the clock tapes of all parties advance simultaneously, so $\mathcal{A}'$ can write the values $(\rho_i)_{p_i \in \mathcal{P}}$ to the clock tapes of $(\pi_i)_{p_i \in \mathcal{P}}$ after seeing the $(\texttt{set}, (\rho_i)_{p_i \in \mathcal{P}})$ message from $\mathcal{A}$ between two activations of parties (this is necessary for $\mathcal{A}'$ to be ϵ-drift preserving). Also, the fact that $\mathcal{A}'$ controls the activations of the π_i (by means of the $\psi_i^{\mathcal{Z}}$, see below) allows $\mathcal{A}'$ to only activate p_i if reading the time tape would succeed for the protocol $C_T(\pi_i)$ and can otherwise activate protocol machine $\psi_0^{\mathcal{Z}}$.

The distinguisher $\mathcal{Z}$ is converted into a malicious protocol $\psi^{\mathcal{Z}}$ as follows. We define $\psi_0^{\mathcal{Z}}$ to run the ITM $\mathcal{Z}$ and relay all messages intended for $\mathcal{A}$ to $\mathcal{A}'$, and mask the messages from $\mathcal{A}'$ as messages from $\mathcal{A}$. Inputs to and outputs from $C_T(\pi)_0$ can be processed locally by $\psi_0^{\mathcal{Z}}$, inputs for $C_T(\pi)_i$ with $i \neq 0$ are sent via $\mathcal{A}'$ to the respective $\psi_i^{\mathcal{Z}}$, which only acts as a forwarder between $C_T(\pi)_i$ and $\psi_0^{\mathcal{Z}}$. The protocol machine $\psi_0^{\mathcal{Z}}$ allows the adversary to schedule the input to π_0. When $\mathcal{Z}$ generates local output, $\psi_0^{\mathcal{Z}}$ outputs the same message (the local output of $\psi_i^{\mathcal{Z}}$ for $i \neq 0$ is constant).

By the fact that π is secure in the Timing model, we know that there is a good simulator $\mathcal{S}'$. From this simulator, we obtain a simulator $\mathcal{S}$ in UC using the translations described above.

Finally, we track the messages sent (by $\mathcal{Z}$) in both executions and verify that the messages returning to $\mathcal{Z}$ are computed using the same functions on identically distributed inputs in the execution in UC and the execution within $\psi_0^{\mathcal{Z}}$ in the Timing model, in both the real and the ideal cases. Note that, in UC, the environment $\mathcal{Z}$ is the master scheduler that is also activated in the beginning of the execution, whereas this task is fulfilled by the adversary $\mathcal{A}'$ in the Timing model.

In more detail, we have to argue that, for each activation, the inputs and the state of the ITMs π_i, $\mathcal{A}$, and $\mathcal{Z}$ have the same distribution in the executions $\text{EXEC}_{C_T(\pi),\{\mathcal{F}_{\text{TIME}},\mathcal{F}_{\text{AUTH}}\},\mathcal{A},\mathcal{Z}}$ in UC and $\text{EXEC}_{\pi,\mathcal{A}',\psi^{\mathcal{Z}}}$ in the Timing model. We then conclude the same for the outputs by an inductive argument. Note that we also have to argue that the clock tape of π_i has the same contents. The following types of activations occur in the real execution.

Local input at π_i: π_i computes based on the input from $\mathcal{Z}$ and the clock tape, which is guaranteed to have consistent contents by potentially buffering the input (either within $C_T(\pi)$ or within $\psi^{\mathcal{Z}}$ by the fact that $\psi_i^{\mathcal{Z}}$ awaits the acknowledgment of $\mathcal{A}'$). Moreover, it is easy for $\mathcal{A}'$ to track which π_i can be activated and what the contents of their clock tapes must be like.

Receive message at π_i: The (new) values considered by π_i are the contents of the message and the clock tape. Again, the consistency of the clock tapes is guaranteed by providing the messages to π_i only if p_i is marked as active in $\mathcal{F}_{\text{TIME}}$.

Leaked message from $\mathcal{F}_{\text{AUTH}}$ at $\mathcal{A}$: The contents of the messages is generated by π_i and the "envelopes" can be easily simulated by $\mathcal{A}'$.

Reply from $\mathcal{F}_{\text{TIME}}$ to $\mathcal{A}$: The messages are constant (the purpose is to return the activation).

Communication between $\mathcal{Z}$ at $\mathcal{A}$: The correct "forwarding" of these messages is guaranteed by the "protocol" used among $\psi^{\mathcal{Z}}$ and $\mathcal{A}'$.

Empty activation from $C_T(\pi_i)$ at $\mathcal{A}$: This can easily be simulated as $\mathcal{A}'$ keeps track of the messages that are otherwise buffered by $C_T(\pi_i)$.

Output of π_i at $\mathcal{Z}$: The output is simply forwarded by $\psi_i^{\mathcal{Z}}$.

Of course, we must also provide the same arguments for the ideal executions.

Inputs to $\mathcal{F}$ (from $\psi_i/\mathcal{Z}$): The input notifications of $T_T(\mathcal{F})$ are wrapped as messages among the $\psi_i^{\mathcal{Z}}$ by $\mathcal{S}$, and the input is acknowledged by $\mathcal{S}$ as soon as $\mathcal{S}'$ decides to deliver the messages (among the $\psi_i^{\mathcal{Z}}$ and the ideal input message to $\mathcal{F}$). This guarantees that $\mathcal{F}$ will obtain the same inputs in the same order in both cases.

Outputs of $\mathcal{F}$ to $\psi_i/\mathcal{Z}$: The notifications for delayed outputs from $T_T(\mathcal{F})$ are collected by $\mathcal{S}$ and simulated as ideal messages from $\mathcal{F}$ to ψ_i. $\mathcal{S}$ asks $T_T(\mathcal{F})$ to deliver the outputs once $\mathcal{S}'$ delivers the ideal messages to $\psi_i^{\mathcal{Z}}$ and the (forwarded) message to $\psi_0^{\mathcal{Z}}$. This guarantees that $\mathcal{Z}$ obtains the outputs in the correct order.

Communication between $\mathcal{F}$ and $\mathcal{S}'$: The communication is simply forwarded by the definition of $T_T(\mathcal{F})$ and $\mathcal{S}$.

Messages among $\mathcal{S}'$ and $\psi_i/\mathcal{Z}$: Messages from $\mathcal{Z}$ are "wrapped" as messages from ψ_i by $\mathcal{S}$ before handing them to $\mathcal{S}'$. This means that messages from $\mathcal{Z}$ to $\mathcal{A}$ are wrapped as special messages from ψ_0 (as done by $\psi_0^{\mathcal{Z}}$), and the input notifications from $T_T(\mathcal{F})$ are wrapped as described above. On the other hand, $\mathcal{S}$ also "unwraps" the messages originating from $\mathcal{S}'$ before handing them to $\mathcal{Z}$. The consistent behavior of $\mathcal{S}$ and $\psi_0^{\mathcal{Z}}$ guarantees that communication between $\mathcal{S}'$ and $\mathcal{Z}$ works identically in both settings.

The input to the UC-execution is one auxiliary input string for $\mathcal{Z}$, whereas in the Timing model, the input is one such string for each party p_i, and one for the adversary $\mathcal{A}'$. Hence, the auxiliary input for $\mathcal{Z}$ is provided to p_0, while the inputs to all other parties as well as the adversary are empty. Note that all parties p_i with $i \neq 0$ provide empty output, while p_0 forwards the output of $\mathcal{Z}$. As $\mathcal{Z}$'s "view" is identical in both the ideal executions in the Timing model and UC and in the real executions in the Timing model and UC, the distinguishing advantage of $\mathcal{D}'$ simply forwarding $\mathcal{Z}$'s decision in the Timing model is at least as large as the advantage of $\mathcal{Z}$ in specialized simulator UC.

Security in the Timing model is defined as

$$\forall \psi, \mathcal{A}'\ \exists \mathcal{S}' : \left\{ \text{EXEC}_{\pi,\mathcal{A}',\psi}(k,x,z) \right\}_{k\in\mathbb{N}, x\in(\{0,1\}^*)^m, z\in\{0,1\}^*} \overset{c}{\approx} \left\{ \text{EXEC}_{\mathcal{F},\mathcal{S}',\psi}(k,x,z) \right\}_{k\in\mathbb{N}, x\in(\{0,1\}^*)^m, z\in\{0,1\}^*}.$$

Following the above descriptions, we transform any pair $\mathcal{Z}, \mathcal{A}$ of a UC-environment and a UC-adversary into such a pair $\psi^{\mathcal{Z}}, \mathcal{A}'$. The simulator $\mathcal{S}'$ guaranteed by the above definition can be converted into a UC-simulator $\mathcal{S}$, such that the (computational) distance of the output vectors of the two Timing model experiments is as large as the (statistical) distance of the outcomes of the two UC experiments, meaning that we have

$$\forall \mathcal{Z}, \mathcal{A}\ \exists \mathcal{S} : \left\{ \text{EXEC}_{C_T(\pi),\{\mathcal{F}_{\text{TIME}},\mathcal{F}_{\text{AUTH}}\},\mathcal{A},\mathcal{Z}}(k,x) \right\}_{k\in\mathbb{N}, x\in\{0,1\}^*} \approx \left\{ \text{EXEC}_{T_T(\mathcal{F}),\mathcal{S},\mathcal{Z}}(k,x) \right\}_{k\in\mathbb{N}, x\in\{0,1\}^*}.$$

Security in UC implies security in the Timing model. We employ the above described interface conversion strategy to convert the Timing model adversary $\mathcal{A}'$ and malicious protocol ψ into a UC-adversary $\mathcal{A}$ and environment $\mathcal{Z}^{\psi}$. The adversary $\mathcal{A}$ notifies the environment $\mathcal{Z}^{\psi}$ upon the first activation originating from $\mathcal{A}'$ for each party ψ_i such that $\mathcal{Z}^{\psi}$ can take care of providing ψ_i with the correct input. The environment $\mathcal{Z}^{\psi}$ emulates ψ_i for all p_i, forwards the local input to the $C_T(\pi)_i$ and the messages of the ψ_i to the adversary.

Finally, $\mathcal{Z}$ tailors a transcript of the corresponding Timing model execution and outputs it. The distinguisher $\mathcal{D}'$ of from the Timing model can hence be used to distinguish the outputs of $\mathcal{Z}^{\psi}$ in the real and ideal cases. Also, we employ the inverse conversion to transform the guaranteed UC-simulator $\mathcal{S}$ into a Timing model simulator $\mathcal{S}'$.

Translating the handling of the clock tapes is straightforward: once $\mathcal{A}'$ has finished the writing operation (i.e., proceeds with the next activation), $\mathcal{A}$ deactivates all parties at $\mathcal{F}_{\text{TIME}}$, submits a vector of new clock values, and re-activates all parties.

The remainder of the argument is as above: The distribution of the inputs obtained and outputs generated by the ITMs π_i, ψ_i, and $\mathcal{A}'$ is the same in the execution in the Timing model and the one in UC. The transcript for $\mathcal{D}'$ is obtained in a deterministic way (and also has the same distribution). As the transcripts have the same distribution, the distinguishing advantage achieved by $\mathcal{D}'$ is the same as well.

More detailed, what are the actions that can happen within the system.

Local input to π_i: The ITM π_i computes based on this local input (which is easily forwarded from the simulated ψ_i by $\mathcal{Z}^{\psi}$) and the contents of the clock tape (which is kept in a consistent state by the above described strategy).

Local input to ψ_i: This is derived from $\mathcal{Z}^{\psi}$'s auxiliary input and is provided by $\mathcal{Z}^{\psi}$ at the correct point in time (according to the clock tape of ψ_i), which is ensured by the collaboration of $\mathcal{A}$ and $\mathcal{Z}^{\psi}$, once $\mathcal{A}'$ schedules p_i for the first time.

Message delivered for π_i, ψ_i, or local output from π_i to ψ_i: The consistency of both these messages and the contents of p_i's clock tape is argued as above. In the Timing model, the messages among the ψ_i are scheduled and delivered via $\mathcal{A}'$. Hence, $\mathcal{Z}^{\psi}$ (and $\mathcal{A}$) forward all messages among the simulated ψ_i via $\mathcal{A}'$ to guarantee the correct scheduling.

Local output of ψ_i: This output is faithfully included in the transcript by $\mathcal{Z}^{\mathcal{D}',\psi}$.

Messages obtained by $\mathcal{A}'$: The messages generated by the (simulated) ψ_i are easily obtained and forwarded to $\mathcal{A}$ by $\mathcal{Z}^{\psi}$, the messages generated by π_i can be extracted from $C_T(\pi_i)$ by the above described scheme.

Of course, we have to provide a similar analysis for the ideal setting (the involved objects are $\mathcal{F}$, ψ, $\mathcal{S}$, and $\mathcal{D}'$).

Input from ψ_i to $\mathcal{F}$: The protocol machine ψ_i sends its input to $\mathcal{F}$ by means of an ideal message. In $\mathcal{Z}^{\psi}$, this is translated into an input to $T_T(\mathcal{F})$ at interface i, which yields an input notification to $\mathcal{S}$. Once $\mathcal{S}$ acknowledges this input to $\mathcal{F}$, $\mathcal{S}'$ will deliver the ideal message to $\mathcal{F}$.

Communication between $\mathcal{F}$ and $\mathcal{S}$: The conversion between the two different communication modes is done identically by $T_T(\mathcal{F})$ and $\mathcal{S}'$.

Local input to ψ_i: As in the real case.

Message delivered for ψ_i: In the execution in the Timing model, the communication among the ψ_i is scheduled by $\mathcal{S}'$. In UC, $\mathcal{Z}^{\psi}$ simulates the protocol machines ψ_i, providing the messages to $\mathcal{A}$ to obtain the scheduling via $\mathcal{A}'$. The simulator $\mathcal{S}'$ behaves consistently: It provides $\mathcal{S}$ with the notifications that would have been provided by $\mathcal{Z}^{\psi}$ and also acts according to $\mathcal{S}$'s replies.

Output from $\mathcal{F}$ to ψ_i: The messages from $\mathcal{F}$ to ψ_i are ideal messages that are scheduled by $\mathcal{S}'$. In UC, the functionality $T_T(\mathcal{F})$ first sends the notification for the delayed output to $\mathcal{S}$ (which has to acknowledge the delivery). Hence, $\mathcal{S}'$ can deliver the message once $\mathcal{S}$ sends the acknowledgment.

As the outputs of the simulated ψ are used to construct the transcript, the input to $\mathcal{D}'$ has the same distribution in both cases.

The input to the Timing model execution consists of one auxiliary string for each p_i along with one such string for the adversary $\mathcal{A}'$. These inputs can all be encoded into one single auxiliary string provided to $\mathcal{Z}^{\psi}$ in UC, which recovers the original contents and provides it as an input to the ψ_i and (via $\mathcal{A}$) to $\mathcal{A}'$.

Security in specialized simulator UC is defined as

$$\forall \mathcal{A}, \mathcal{Z}\ \exists \mathcal{S} : \left\{ \text{EXEC}_{C_T(\pi),\{\mathcal{F}_{\text{TIME}},\mathcal{F}_{\text{AUTH}}\},\mathcal{A},\mathcal{Z}}(k,x) \right\}_{k\in\mathbb{N},x\in\{0,1\}^*} \approx \left\{ \text{EXEC}_{T_T(\mathcal{F}),\mathcal{S},\mathcal{Z}}(k,x) \right\}_{k\in\mathbb{N},x\in\{0,1\}^*}.$$

Our goal is to conclude security in the Timing model, which is

$$\forall \psi, \mathcal{A}' \; \exists \mathcal{S}' : \left\{ \text{EXEC}_{\pi,\mathcal{A}',\psi}(k,x,z) \right\}_{k\in\mathbb{N},x\in(\{0,1\}^*)^m,z\in\{0,1\}^*} \overset{c}{\approx} \left\{ \text{EXEC}_{\mathcal{F},\mathcal{S}',\psi}(k,x,z) \right\}_{k\in\mathbb{N},x\in(\{0,1\}^*)^m,z\in\{0,1\}^*}.$$

Starting from a Timing model adversary $\mathcal{A}'$ (and a protocol ψ), we obtain a UC-adversary $\mathcal{A}$ and a UC environment $\mathcal{Z}^{\psi}$, and transform the guaranteed UC-simulator $\mathcal{S}$ into a Timing model simulator $\mathcal{S}'$. A distinguisher $\mathcal{D}'$ for Timing model transcripts can then also be applied to the output of the UC environment $\mathcal{Z}^{\psi}$, and achieves a distinguishing advantage at least as good as $\mathcal{D}'$. Hence, by the fact that the protocol is UC-secure with respect to specialized simulators, we can conclude that it is also secure in the Timing model. □

By combining the above lemma, namely that statements from the Timing model can be faithfully translated into the $\{\mathcal{F}_{\text{TIME}}, \mathcal{F}_{\text{AUTH}}\}$-hybrid model of UC, with Lemma 11 from the main paper, that the Timing model does not allow to prove that a protocol achieves guaranteed termination.

D Nielsen's Model

Nielsen [33] adapts the asynchronous UC framework of Canetti [12] to the setting of authenticated synchronous networks. While the general structure of the security definition is adopted, the frameworks differ considerably in their concrete definitions. For instance, Nielsen defines the composition of two synchronous protocols in a "lock-step" way: the round switches of all protocols occur at the same time. In Section D.1, we sketch the basic structure of Nielsen's framework [33] and point out an ambiguity in the original formulation. In Section D.2, we show the relation of this framework to our $\{\mathcal{F}_{\text{CLOCK}}, \mathcal{F}_{\text{BD-AUTH}}\}$-hybrid model (in the framework of [12]). In particular, we show that the framework also allows to capture guaranteed termination. For a more detailed introduction to the model, refer to Chapter 3 of [33].

D.1 Description of the Model

The fundamental structure of the security definition in [33] is adopted from the framework of *universally composable security* in [12]: the security of a protocol π is defined by an ideal functionality $\mathcal{F}$, and π is deemed secure if there is a simulator $\mathcal{S}$ such that an execution of the protocol π in the synchronous network is indistinguishable from an execution of $\mathcal{F}$ with the simulator $\mathcal{S}$.

The synchronous protocol π among parties $\mathcal{P} = \{p_1, \ldots, p_n\}$ is executed in rounds, and the model guarantees that each party p_i is activated exactly once per round. In each round r, the protocol machine π_i of party p_i obtains local input as well as the local output generated by sub-protocols and, for each party $p_j \in \mathcal{P}$, one message that π_j sent in round $r-1$. During the activation, π_i produces local output, local input for each sub-protocol, and, for each $p_j \in \mathcal{P}$, one message that will be delivered to π_j in round $r+1$.

The course of the execution is directed by the environment $\mathcal{Z}$. In each round r, $\mathcal{Z}$ may choose the order in which the (honest) parties p_i with $p_i \in \mathcal{H}$ are activated. $\mathcal{Z}$ chooses the messages sent to p_i by the corrupted parties immediately before the activation, which corresponds to the *strongly rushing* property. Also, the messages generated by p_i for other parties are given to $\mathcal{Z}$ immediately after the activation. After all activations of a round are completed, $\mathcal{Z}$ may switch the computation to the next round. At any point of the computation, $\mathcal{Z}$ may *corrupt* honest parties p_j, obtaining the internal state[16] of π_j. The fact that $\mathcal{Z}$ controls p_j in the further execution is modeled by having $\mathcal{Z}$ specify all future messages sent by p_j. Ideal functionalities must explicitly specify the gained capabilities of $\mathcal{Z}$ in terms of messages leaked to and received from the simulator.

The composition of two ideal functionalities is defined as executing both functionalities in a "lock-step" way. This means that the rounds are defined globally for the complete execution, and that all ideal functionalities switch rounds synchronously. Hence, this also holds for protocols executed both in parallel or as

[16] The original version of [33] was based on an earlier version of [10] and only leaked the random tape. Leaking the complete internal state is consistent with the "standard corruption" of [10, Page 68], possibly including also the local inputs and messages received in previous executions. Also, corruption in [33] is PID-wise.

sub-protocols. Note that the communication between a protocol and a sub-protocol also adheres to these rounds of execution, which implies that if a protocol is composed of several layers of sub-protocols, each such layer introduces one round of delay for messages propagated from the functionalities to the player's interfaces.

Ambiguity of Corruption. Upon corruption of a party p_i in round r, the model sets $\mathcal{C} := \mathcal{C} \cup \{p_i\}$ to store the information that p_i has been corrupted. At a later activation $(\texttt{activate}, p_j, x_{j,r}, (m_{i,j,r-1})_{p_i \in \mathcal{C}})$ in the same round r, the adversary is—technically—also allowed to specify the message $m_{i,j,r-1}$ which was sent by p_i *in round* $r-1$. We assume that this behavior is not intended, and propose fixing the ambiguity by explicitly specifying whether p_i was corrupted in round $r-1$, for instance by keeping a sequence of corruption sets $(\mathcal{C}^{r'})_{r' \in \mathbb{N}}$ with $\mathcal{C}^r \subseteq \mathcal{C}^{r+1}$.

D.2 Relation to our Model

In this section, we prove that, for a protocol π and a functionality $\mathcal{F}$ defined in Nielsen's model, there is a compiled protocol $C_N(\pi)$ that implements the translated functionality $T_N(\mathcal{F})$ in the $\{\mathcal{F}_{\text{CLOCK}}, \mathcal{F}_{\text{BD-AUTH}}\}$-hybrid model of [12] if and only if π implements $\mathcal{F}$ in the model of [33]. Moreover, the translation $T_N(\cdot)$ preserves the termination guarantees (if a functionality achieves guaranteed termination in [33], then the transformed functionality $T_N(\mathcal{F})$ also achieves this in our model).

Translating the synchronous functionalities from [33]

For a player set $\mathcal{P}$, the functionality initializes $r := 1$, $\mathcal{H} = \mathcal{P}$, as well as $r_i := 1$, $c_i := 0$, and $d_i := 1$ for all $p_i \in \mathcal{P}$.

- On activation while $r_i > r$ and $c_i \geq |\mathcal{P}|$ for all $p_i \in \mathcal{H}$:
  - If the received message is $(\texttt{end round}, v'_{\mathcal{F}})$ from $\mathcal{A}$, then produce outputs $y_{i,r}$ for all $p_i \in \mathcal{P}$ as defined by the functionality $\mathcal{F}$. Set $r := r + 1$, $c_i := 0$ for all $p_i \in \mathcal{P}$, and send $(y_{j,r})_{p_j \in \mathcal{P} \setminus \mathcal{H}}$ to $\mathcal{A}$.
  - Any other activation is interpreted as $(\texttt{end round}, \perp)$ message[a] from $\mathcal{A}$ (except that the functionality does not send the corrupted parties' outputs to $\mathcal{A}$).
- On input $(\texttt{input}, x_{i,r_i})$ with $r = r_i = d_i$ from party $p_i \in \mathcal{H}$, compute the value $v_{\mathcal{F}}$ as specified by the functionality $\mathcal{F}$ and set $r_i := r_i + 1$ and $c_i := 0$. Send $(\texttt{leaked}, i, v_{\mathcal{F}})$ to $\mathcal{A}$.
- On an empty activation from p_i with $r_i > r$, set $c_i := c_i + 1$. If $r_i = r > d_i$, set $d_i := r_i$. Send $(\texttt{activated}, p_i)$ to $\mathcal{A}$.
- On input $(\texttt{output})$ from party p_i or $(\texttt{output}, p_j)$ with $p_j \in \mathcal{P} \setminus \mathcal{H}$ from $\mathcal{A}$, reply with $(\texttt{output}, y_{i,r})$.
- On input $(\texttt{corrupt}, p_i)$ from $\mathcal{A}$ with $p_i \in \mathcal{H}$, set $\mathcal{H} := \mathcal{H} \setminus \{p_i\}$, obtain the value $v_{\mathcal{F}}$ from $\mathcal{F}$ by simulating the corresponding corruption message for $\mathcal{F}$ and output $v_{\mathcal{F}}$ to $\mathcal{A}$.

[a] Assumed to be the default input.

Translating Protocols. A protocol machine π_i in the model of [33], in each round r, takes a local input $x_{i,r}$ and a set of messages $(m_{j,i,r-1})_{p_j \in \mathcal{P}}$ (where the message $m_{i,i,r-1}$ models keeping state), and produces local output $y_{i,r}$ as well as messages $(m_{i,j,r})_{p_j \in \mathcal{P}}$. The protocol π is transformed into an ITM π' in the $\{\mathcal{F}_{\text{CLOCK}}, \mathcal{F}_{\text{BD-AUTH}}\}$-hybrid model as follows. The ITM π' keeps an (initially empty) internal buffer $\mathcal{B}$ with messages that have been computed but not yet output or sent via a channel. Set the round counter to $r_i = 1$ and define $m_{j,i,0} = \perp$ for $p_j \in \mathcal{P}$.

- On local input $(\texttt{output})$, output the latest value $y_{i,r'}$ that is marked as ready for output.
- On local input $(\texttt{input}, x_{i,r_i})$, use π to compute $\big((m_{i,j,r_i})_{p_j \in \mathcal{P}}, y_{i,r_i}\big) := \pi\big((m_{j,i,r_i-1})_{p_j \in \mathcal{P}}, x_{i,r_i}\big)$ and store the messages $m_{i,j,r}$ in $\mathcal{B}$. Deliver the first message.
- Upon an empty activation, deliver one message from $\mathcal{B}$. That is, send $(\texttt{send}, m_{i,j,r})$ to the channel $\mathcal{F}_{\text{BD-AUTH}}(p_i, p_j)$. Once $\mathcal{B}$ is empty, send $(\texttt{RoundOK})$ to $\mathcal{F}_{\text{CLOCK}}$.
- Upon an empty activation, check via $(\texttt{RequestRound})$ with $\mathcal{F}_{\text{CLOCK}}$ whether the round switch occurred (i.e., $d = 0$). In this case, mark $y_{i,r}$ as ready for output and send $(\texttt{RoundOK})$ to $\mathcal{F}_{\text{CLOCK}}$.

– Upon each activation, check via $(\texttt{RequestRound})$ with $\mathcal{F}_{\text{CLOCK}}$ whether the next round switch occurred (i.e., $d = 0$). In this case, set $r_i := r_i + 1$ and obtain the messages m_{j,i,r_i-1} by sending $(\texttt{fetch})$ to $\mathcal{F}_{\text{BD-AUTH}}(p_j, p_i)$ for all $p_j \in \mathcal{P} \setminus \{p_i\}$ and proceed with the first step.

Lemma 13 is an immediate consequence of the following lemma.

Lemma 15. *Let π be a protocol and $\mathcal{F}$ be an ideal functionality in the model of [33]. Then, the protocol π implements the functionality $\mathcal{F}$ in the model of [33] if and only if $C_N(\pi)$ UC-implements the functionality $T_N(\mathcal{F})$ in the $\{\mathcal{F}_{\text{CLOCK}}, \mathcal{F}_{\text{BD-AUTH}}\}$-hybrid model.*

Proof. We first show that any environment that is a good synchronous distinguisher for the model of [33] is also a good distinguisher for the $\{\mathcal{F}_{\text{CLOCK}}, \mathcal{F}_{\text{BD-AUTH}}\}$-hybrid model (i.e., security in UC implies security in [33]). Hence, we describe a transformation that converts a synchronous environment into a UC-environment. This resulting environment $\mathcal{Z}'$ proceeds as follows.

– A query $(\texttt{activate}, p_i, x_{i,r}, (m_{j,i,r-1})_{p_j \in \mathcal{C}})$ from $\mathcal{Z}$ is handled by injecting the messages $m_{j,i,r-1}$ into the channels $\mathcal{F}_{\text{BD-AUTH}}(p_j, p_i)$ and providing local input $(\texttt{input}, x_{i,r})$ to π'_i. Then, repeatedly activate π'_i until all messages for round r are in the channels $\mathcal{F}_{\text{BD-AUTH}}(p_i, p_j)$. Provide the vector $(m_{i,j,r})_{p_j \in \mathcal{P} \setminus \{p_i\}}$ of messages leaked on the channels $\mathcal{F}_{\text{BD-AUTH}}(p_i, p_j)$ to $\mathcal{Z}$.
– On query $(\texttt{end round})$ from $\mathcal{Z}$, request the local outputs $y_{j,r}$ by first activating π'_j and receiving $(\texttt{switch}, p_j)$ from $\mathcal{F}_{\text{CLOCK}}$ for each $p_j \in \mathcal{H}$ (this is done twice in rounds), and then querying $(\texttt{output})$ again at each π'_j for $p_j \in \mathcal{H}$. Return the vector $(y_{j,r})_{p_j \in \mathcal{H}}$ to $\mathcal{Z}$.
– On query $(\texttt{corrupt}, p_i)$ from $\mathcal{Z}$, issue the same request to $\mathcal{F}_{\text{CLOCK}}$, all $\mathcal{F}_{\text{BD-AUTH}}(p_i, p_j)$ and $\mathcal{F}_{\text{BD-AUTH}}(p_j, p_i)$ with $p_j \in \mathcal{P}$, and instruct the adversary to corrupt π'_i. The internal state of π'_i contains the internal state of π_i. Extract this information and return it to $\mathcal{Z}$.
– If $\mathcal{Z}$ issues a query that would not be allowed in [33], then output 0 and halt. Otherwise, once $\mathcal{Z}$ provides local output, write the same value on the output tape and halt.

Claim. The described "adapter" perfectly emulates the environment's view. Formally,

$$\forall\, \mathcal{Z} : \ \text{REAL}_{\pi,\mathcal{Z}} \equiv \text{HYB}^{\{\mathcal{F}_{\text{CLOCK}}, \mathcal{F}_{\text{BD-AUTH}}\}}_{\pi', \mathcal{D}, \mathcal{Z}'},$$

with the real model of [33] and the UC-hybrid model.

We use an inductive argument: Prior to each query, all random variables in the executions have the same distribution (all messages, inputs, and outputs can be interpreted as random variables defined on the random tapes) in the two cases. During each query, the exact same transformations are applied to these random variables in either case; so the equivalence is extended to the outputs. By induction on the number of queries, we conclude that the output of the two execution also has the same distribution. Again, we differentiate between the three different types of queries.

– For each $p_i \in \mathcal{H}$, one query $(\texttt{activate}, p_i, x_{i,r}, (m_{j,i,r-1})_{p_j \in \mathcal{C}})$ is allowed for each round r. In $\text{REAL}_{\pi,\mathcal{Z}}$, such a query results in the output $(m_{i,j,r})_{p_j \in \mathcal{P} \setminus \{p_i\}}$. In the converted execution, injecting the messages $m_{j,i,r-1}$ into the corresponding channels is allowed for $p_j \in \mathcal{C}$, as the channels allow for replacing messages (this is important in case p_j is corrupted after the activation in the current round.) Providing input $(\texttt{input}, x_{i,r})$ to π'_i results in the same distribution for inputs to π_i as in the execution $\text{REAL}_{\pi,\mathcal{Z}}$, so the values computed by π_i also have the same distribution. By making π'_i send all messages $m_{i,j,r}$ to the channels, the converter obtains the tuple $(m_{i,j,r})_{p_j \in \mathcal{P} \setminus \{p_i\}}$ with the same distribution as in the "real" execution.
– After all honest parties have been activated in round r, the $(\texttt{end round})$-query is answered with $(y_{i,r})_{p_i \in \mathcal{H}}$. The converter activates all honest parties and queries their outputs, and the equivalence follows by the same argument as above.
– Upon corruption, the converter obtains the internal state of π'_i. By the definition of π', this includes the state of π_i as expected by the environment $\mathcal{Z}$.

Altogether, this proves the above claim for all environments $\mathcal{Z}$.

Claim. The described "adapter" indistinguishably emulates the environment's view in the ideal case, if the synchronous simulator $\mathcal{T}$ is constructed from the UC-simulator as described below. Formally,

$$\forall \mathcal{S}\ \exists \mathcal{T}\ \forall \mathcal{Z} :\ \text{IDEAL}_{\mathcal{F},\mathcal{T},\mathcal{Z}} \approx \text{EXEC}_{\mathcal{F}',\mathcal{S},\mathcal{Z}'},$$

where the left random variable is defined by an execution in [33] and the right random variable is defined by a UC-execution.

Converting simulators from UC to the synchronous model

The simulator $\mathcal{T}$ is given oracle access to $\mathcal{S}$ and proceeds as follows.

- On query $(\texttt{activate}, p_i, v_{\mathcal{F}}, (m_{j,i,r-1})_{p_j \in \mathcal{C}})$, first inject the messages $m_{j,i,r-1}$ into the channels simulated by $\mathcal{S}$ (expect $\mathcal{S}$ to answer consistently with the behavior of real channels—in particular, if a some p_j was corrupted after inserting a message into the channel, the message will be replaced). Then, provide the message $(\texttt{leaked}, p_i, v_{\mathcal{F}})$ to $\mathcal{S}$ as a message from $T_N(\mathcal{F})$ and repeatedly send $(\texttt{activated}, p_i)$ until all messages $m_{i,j,r}$ are supposed to be in (and leaked on) the channels. Provide the tuple $(m_{i,j,r})_{j \in \mathcal{P} \setminus \{i\}}$ as output.
- On query $(\texttt{end round})$, send one message $(\texttt{activated}, p_i)$ for each $p_i \in \mathcal{H}$, and expect the message $(\texttt{switch}, p_i)$ from $\mathcal{F}_{\text{CLOCK}}$ to be simulated. Before $\mathcal{S}$ simulates the last such notification, it may pose the query $(\texttt{end round}, v_{\mathcal{F}})$ to $T_N(\mathcal{F})$ (otherwise $\mathcal{T}$ sets $v_{\mathcal{F}} := \bot$). Provide $v_{\mathcal{F}}$ as output.
- On input $(y_{i,r})_{p_i \in \mathcal{C}}$, record this tuple. If $\mathcal{S}$ provided a value $v_{\mathcal{F}}$, provide the tuple $(y_{i,r})_{p_i \in \mathcal{C}}$ to $\mathcal{S}$ and expect the last missing $(\texttt{switch}, p_i)$-message. Upon future $(\texttt{output}, p_j)$ queries from $\mathcal{S}$ to $T_N(\mathcal{F})$ with $p_j \in \mathcal{P} \setminus \mathcal{H}$, reply with $(\texttt{output}, y_{j,r})$. Simulate one further message $(\texttt{activated}, p_i)$ for each $p_i \in \mathcal{H}$ to $\mathcal{S}$ and expect to receive the corresponding $(\texttt{switch}, p_i)$-messages as a response.
- On query $(\texttt{corrupt}, p_i)$, generate the corruption requests for $\mathcal{F}_{\text{CLOCK}}$, all $\mathcal{F}_{\text{BD-AUTH}}(p_i, p_j)$ and $\mathcal{F}_{\text{BD-AUTH}}(p_j, p_i)$, and π'_i for $\mathcal{S}$. For each of these requests, expect $\mathcal{S}$ to answer according to the behavior of the respective systems. If $\mathcal{S}$, at any future point in time, makes a $(\texttt{corrupt}, p_i)$ request to $T_N(\mathcal{F})$, query $(\texttt{corrupt}, p_i)$ to obtain a value $v_{\mathcal{F}}$ and provide this value to $\mathcal{S}$. Expect a message from $\mathcal{S}$ to the environment that describes the internal state of π'_i, and extract and output the state of π_i.
- If any message from $\mathcal{S}$ is unexpected, halt immediately.

- On queries $(\texttt{activate}, p_i, x_{i,r}, (m_{j,i,r-1})_{p_j \in \mathcal{C}})$ for $p_i \in \mathcal{H}$, the messages returned to the environment are determined equivalently. The synchronous execution first provides $(p_i, x_{i,r})$ to $\mathcal{F}$ to obtain the value $v_{\mathcal{F}}$, before $\mathcal{T}$ injects the messages $m_{j,i,r-1}$ for $p_j \in \mathcal{C}$ into the channels simulated by $\mathcal{S}$, provides a message $(\texttt{leaked}, p_i, v_{\mathcal{F}})$ to $\mathcal{S}$, and provokes the computation for the simulated π'_i by repeatedly issuing $(\texttt{activated}, p_i)$ to $\mathcal{S}$. In the UC-execution, the same messages are injected into the channels simulated by $\mathcal{S}$, and the $(\texttt{input}, x_{i,r})$ given to $T_N(\mathcal{F})$ also triggers the output of the value $v_{\mathcal{F}}$ to $\mathcal{S}$ by $\mathcal{F}$. This results in an equally distributed message $(\texttt{leaked}, p_i, v_{\mathcal{F}})$ to $\mathcal{S}$. In both cases, $\mathcal{S}$ repeatedly obtains $(\texttt{activated}, p_i)$ messages until all messages $m_{i,j,r}$ are leaked on the channels. Since the activations of the systems π_j, $\mathcal{S}$, and $\mathcal{F}$ occur in the same order with equally distributed inputs, all variables have the same distribution after the $\texttt{activate}$-query.
- Upon the $(\texttt{end round})$-query, $\mathcal{S}$ is allowed to produce a $(\texttt{end round}, v_{\mathcal{F}})$-message to $\mathcal{F}$ (or otherwise accept that $v_{\mathcal{F}} = \bot$). In the synchronous execution, the $(\texttt{end round})$ message is given to $\mathcal{T}$, which sends one $(\texttt{activate}, p_i)$ message for each $p_i \in \mathcal{H}$ to $\mathcal{S}$ to potentially obtain $v_{\mathcal{F}}$ as a response to the last message. If $\mathcal{S}$ provides such a value $v_{\mathcal{F}}$ it is provided as an output and given to $\mathcal{F}$, and the resulting messages $(y_{i,r})_{p_i \in \mathcal{P} \setminus \mathcal{H}}$ are returned to $\mathcal{S}$ via $\mathcal{T}$. In the UC-execution, the converter activates the π'_i (in rounds), which yields messages $(\texttt{activated}, p_i)$ to $\mathcal{S}$, and (by the assumption on $\mathcal{S}$, otherwise we can easily build a good distinguisher) the simulator $\mathcal{S}$ translates these into messages $(\texttt{switch}, p_i)$. Before simulating the last such message, $\mathcal{S}$ may issue a $(\texttt{end round}, v_{\mathcal{F}})$-query to $T_N(\mathcal{F})$, which will be given to $\mathcal{F}$, and the messages $(y_{i,r})_{p_i \in \mathcal{P} \setminus \mathcal{H}}$ are returned to $\mathcal{S}$. If $\mathcal{S}$ does not send such a message, $v_{\mathcal{F}} := \bot$ provided to $\mathcal{F}$ as the simulator's input by $T_N(\cdot)$, as $\mathcal{Z}''$ issues a further round of activations to the π'_i with $p_i \in \mathcal{H}$. This is consistent with the synchronous execution. In both cases, $\mathcal{S}$ can now access the values $y_{i,r}$ for $p_i \in \mathcal{C}$ generated by $\mathcal{F}$ on input $v_{\mathcal{F}}$. In [33], the tuple $(y_{i,r})_{p_i \in \mathcal{H}}$ is directly given to $\mathcal{Z}$; in the converted execution, the converter extracts the same messages by requesting the outputs via $(\texttt{output})$ at all π'_i with $p_i \in \mathcal{H}$.

- For a $(\texttt{corrupt}, p_i)$, the simulator $\mathcal{T}$ is notified and converts it into corruption queries for $\mathcal{F}_{\text{CLOCK}}$, the channels, and π'_i. The same messages are generated and given to the simulator $\mathcal{S}$ by the converter $\mathcal{C}$. If (at any future point in time) $\mathcal{S}$ sends a $(\texttt{corrupt}, p_i)$ query to $\mathcal{F}'$, then $\mathcal{T}$ asks $\mathcal{F}$ to corrupt p_i and provides the obtained value $v_{\mathcal{F}}$ to $\mathcal{S}$. In our model, the $(\texttt{corrupt}, p_i)$ message is given to $\mathcal{F}'$, which obtains the value $v_{\mathcal{F}}$ in the same way and provides it to $\mathcal{S}$. Hence, the distribution of the representation of π'_i's internal state generated by $\mathcal{S}$ is the same, hence the state of π_i extracted by $\mathcal{T}$ and $\mathcal{C}$ also has the same distribution.

Several above arguments make the assumption that the queries or answers generated by $\mathcal{S}$ are of a certain format. These points are highlighted by explicitly making the simulator $\mathcal{T}$ fail if the answers are different. Yet, in all of these cases, the ideal execution with $\mathcal{S}$ would clearly be distinguishable from the real execution, and since $\mathcal{S}$ is assumed to be a good simulator, such a condition is violated with at most negligible probability.

Combining the above two claims, we conclude that

$$\exists\, \mathcal{S}\ \forall\, \mathcal{Z}:\ \text{REAL}_{\pi,\mathcal{Z}} \approx \text{HYB}^{\{\mathcal{F}_{\text{CLOCK}},\mathcal{F}_{\text{BD-AUTH}}\}}_{\pi',\mathcal{D},\mathcal{Z}'} \approx \text{IDEAL}_{\mathcal{F}',\mathcal{S},\mathcal{Z}'} \approx \text{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}},$$

where the first and the third indistinguishability are proven above and the middle indistinguishability follows from the security statement in the hybrid model.

Security in [33] *Implies Security in UC.* We describe a converter $\mathcal{Z}''$ that transforms a UC-environment into an synchronous environment. This converter internally simulates the functionalities $\mathcal{F}_{\text{CLOCK}}$ and $\mathcal{F}_{\text{BD-AUTH}}$ and behaves as follows.

- On input $(\texttt{input}, x_{i,r})$ at π'_i, if this is not the first such input in the current round, it is dropped. Otherwise, in any round except for the first one, assemble the vector $(m_{j,i,r-1})_{p_j\in\mathcal{C}}$ from the messages obtained at the channels (set all messages to $\bot$ in the first round). Call $(\texttt{activate}, p_i, x_{i,r}, (m_{j,i,r-1})_{p_j\in\mathcal{C}})$ to obtain $(m_{i,j,r})_{p_j\in\mathcal{P}\setminus\{p_i\}}$, and store these messages in a buffer $\mathcal{B}_i$. Simulate the first message on a channel $\mathcal{F}_{\text{BD-AUTH}}(p_i, p_j)$.
- On further activations of π'_i, if $\mathcal{B}_i$ is not empty, remove the first message from $\mathcal{B}_i$ and simulate it on the channel $\mathcal{F}_{\text{BD-AUTH}}(p_i, p_j)$. For the first activation after $\mathcal{B}_i$ is empty, simulate a message $(\texttt{switch}, p_i)$ from $\mathcal{F}_{\text{CLOCK}}$, and ignore all further such activations until this message has been simulated for all parties $p_i \in \mathcal{H}$.
- On the next empty activation, call $(\texttt{end round})$ and record the tuple $(y_{i,r})_{p_i\in\mathcal{H}}$. For each empty activation at π'_j for $p_j \in \mathcal{H}$, mark the output $y_{j,r}$ as ready for output and simulate a message $(\texttt{switch}, p_j)$ from $\mathcal{F}_{\text{CLOCK}}$.
- On input $(\texttt{output})$ at π'_i, return $(\texttt{output}, y_{i,r'})$ with the latest output $y_{i,r'}$ marked as ready.
- On a corruption message $(\texttt{corrupt}, p_i)$ to the functionality $\mathcal{F}_{\text{CLOCK}}$, issue the call $(\texttt{corrupt}, p_i)$ to obtain the internal state of π_i and mark π_i as corrupted in $\mathcal{F}_{\text{CLOCK}}$. In the further activations, simulate the execution of π_i as if it were honest. Yet, in future rounds, if $\mathcal{Z}$ did not provide messages for the channels $\mathcal{F}_{\text{BD-AUTH}}(p_i, p_j)$, set the corresponding messages $m_{i,j,r}$ to a default value $\bot$.
- On a corruption message $(\texttt{corrupt}, p_i)$ to a functionality $\mathcal{F}_{\text{BD-AUTH}}(p_i, p_j)$ or $\mathcal{F}_{\text{BD-AUTH}}(p_j, p_i)$, issue the call $(\texttt{corrupt}, p_i)$ to obtain the internal state of π_i and mark p_i as corrupted for the corresponding channel. In the further activations, simulate the execution of π'_i as if it were honest. (In particular, if p_i is not corrupted at $\mathcal{F}_{\text{CLOCK}}$, then proceed only once the simulated π'_i would have sent $(\texttt{RoundOK})$.) From here on, allow $\mathcal{Z}$ to replace messages in the corresponding channel. If p_i is corrupted in several channels or in channels and $\mathcal{F}_{\text{CLOCK}}$, it looses further guarantees in the straightforward way.
- Upon a corruption message $(\texttt{corrupt})$ to the protocol machine π'_i, issue the call $(\texttt{corrupt}, p_i)$ to obtain the internal state of π_i. Since π'_i only uses π_i in a straightforward way, the state of π'_i can be easily simulated. If p_i is not corrupted on all channels and the $\mathcal{F}_{\text{CLOCK}}$, force $\mathcal{Z}$ to use the corresponding honest interfaces.

Claim. The described "adapter" perfectly emulates the environment's view. More formally, for all UC environments $\mathcal{Z}$ there is a synchronous environment $\mathcal{Z}''$ such that

$$\text{HYB}^{\{\mathcal{F}_{\text{CLOCK}},\mathcal{F}_{\text{BD-AUTH}}\}}_{\pi',\mathcal{D},\mathcal{Z}} \equiv \text{REAL}_{\pi,\mathcal{Z}''},$$

where the left random variable is defined by a UC-execution and the right random variable is defined by a synchronous execution.
We use the same inductive argument as above to handle the queries of $\mathcal{Z}$ one-by-one.

- The $(\texttt{input}, x_{i,r})$-query is only allowed once for each $p_i \in \mathcal{H}$ and r, and ignored otherwise in both cases. The values $y_{i,r}$ and $m_{i,j,r}$ for $p_j \in \mathcal{P}$ are computed by an invocation of the protocol π_i on equivalently distributed inputs: For $x_{i,r}$, this follows by the induction, so we only have to show that the messages $m_{j,i,r-1}$ are consistent. For $p_j \in \mathcal{H}$, this is clear, and parties p_j that are corrupted only at $\mathcal{F}_{\text{CLOCK}}$ still perform their computations honestly, so $\mathcal{Z}''$ can ensure the consistency. The same argument holds if the corrupted p_j is the receiver at some channel. If p_j is the sender or the protocol machine π'_i is corrupted, then $\mathcal{Z}$ can inject messages $m_{j,i,r-1}$ only before π_i switches to round r, so $\mathcal{Z}''$ can include these messages in the $\texttt{activate}$-query.
- Activating π'_i after the $(\texttt{input}, x_{i,r})$-query results in a message $m_{i,j,r}$ being leaked to $\mathcal{Z}$, before a single message $(\texttt{switch}, p_i)$ from $\mathcal{F}_{\text{CLOCK}}$ is sent. The same output is produced by $\mathcal{Z}''$.
- For each $p_i \in \mathcal{H}$, activating π'_i after all parties have sent their messages in the current round causes π'_i to mark the freshly recorded output value $y_{i,r}$ as ready and output a message $(\texttt{switch}, p_i)$. This is done consistently by $\mathcal{Z}''$. All further activations are ignored until all parties have been activated.
- The queries $(\texttt{output})$ and $(\texttt{output}, p_j)$ with $p_j \in \mathcal{P} \setminus \mathcal{H}$ are answered consistently because the same values are marked as ready in both cases, and the distribution of $y_{i,r}$ is the same by induction.
- A corruption message $(\texttt{corrupt}, p_i)$ to either $\mathcal{F}_{\text{CLOCK}}$ or one of the functionalities $\mathcal{F}_{\text{BD-AUTH}}(p_i, p_j)$ or $\mathcal{F}_{\text{BD-AUTH}}(p_j, p_i)$ has no immediate effect on the computation; the indirect effects are described in the input-step.
- Upon a corruption message $(\texttt{corrupt})$ to the protocol machine π'_i, the (simulated) internal state of π'_i is leaked to $\mathcal{Z}$ in both cases. By the construction of the protocol machine π'_i, the state can be easily derived from the state of π_i.

Claim. The described "adapter" indistinguishably emulates the environment's view in the ideal case, if the UC-simulator $\mathcal{S}$ is obtained from the synchronous simulator as described below. More formally, for all "synchronous" simulators $\mathcal{T}$ there is a UC simulator $\mathcal{S}$ such that for all UC environments $\mathcal{Z}$ there is a "synchronous" environment $\mathcal{Z}''$ such that

$$\text{EXEC}_{\mathcal{F}',\mathcal{S},\mathcal{Z}} \approx \text{IDEAL}_{\mathcal{F},\mathcal{T},\mathcal{Z}''},$$

where the left random variable is defined by a UC-execution and the right random variable is defined by a synchronous execution.

- The $(\texttt{input}, x_{i,r})$-query is only allowed once per honest p_i and round r, and ignored otherwise in both cases. In the adapted execution, $\mathcal{Z}''$ executes the $\texttt{activate}$-query. As a result, the input is provided to $\mathcal{F}$, and the value $v_{\mathcal{F}}$ produced by $\mathcal{F}$ is given to the simulator $\mathcal{T}$ together with the messages $(m_{j,i,r-1})_{p_j \in \mathcal{C}}$. $\mathcal{T}$ produces messages $(m_{i,j,r})_{p_j \in \mathcal{P} \setminus \{p_i\}}$, which are buffered by $\mathcal{Z}''$. The first message is simulated on $\mathcal{F}_{\text{BD-AUTH}}(p_i, p_j)$. In the UC-execution, the input is provided to $T_N(\mathcal{F})$, which also provides it to $\mathcal{F}$ and leaks the resulting output $v_{\mathcal{F}}$ to $\mathcal{S}$. The simulator $\mathcal{S}$ uses $\mathcal{T}$ to compute $(m_{i,j,r})_{p_j \in \mathcal{P} \setminus \{p_i\}}$, buffers these messages, and simulates the first message on $\mathcal{F}_{\text{BD-AUTH}}(p_i, p_j)$. It remains to show that the messages $(m_{j,i,r-1})_{p_j \in \mathcal{C}}$ used by $\mathcal{S}$ have the correct distribution. But this holds since $\mathcal{S}$ and $\mathcal{Z}''$ handle the injection of messages into the channels $\mathcal{F}_{\text{BD-AUTH}}(p_j, p_i)$ for $p_j \in \mathcal{P} \setminus \mathcal{H}$ in the same way.
- Upon further activations of π'_i, the buffered messages $(m_{i,j,r})_{p_j \in \mathcal{P} \setminus \{p_i\}}$ are simulated as sent over the channels $\mathcal{F}_{\text{BD-AUTH}}(p_i, p_j)$, and by simulating one message $(\texttt{switch}, p_i)$ afterward (and ignoring all further activations until the messages have been simulated for all honest parties). This is done by $\mathcal{Z}''$ in the adapted execution, and $\mathcal{S}$ produces these messages when notified by $T_N(\mathcal{F})$ via $(\texttt{activated}, p_i)$ in the UC-execution.
- Upon the first further activation of π'_i, the simulator $\mathcal{T}$ is invoked on input $(\texttt{end round})$ in both cases, and the resulting value $v_{\mathcal{F}}$ is given to $\mathcal{F}$ (via $\mathcal{Z}''$ and the model in one case, and via $\mathcal{S}$ and $T_N(\cdot)$ in the other case). The values $(y_{i,r})_{p_j \in \mathcal{P} \setminus \mathcal{H}}$ are provided to $\mathcal{T}$ either directly by the model, or by $T_N(\cdot)$ and $\mathcal{S}$. The values $(y_{i,r})_{p_j \in \mathcal{H}}$ are stored by $\mathcal{Z}''$ and $T_N(\mathcal{F})$, respectively. Since $\mathcal{F}$ computes on equally distributed values, the values $y_{i,r}$ also have the same distribution in both cases. For the first such activation of π'_i,

Converting synchronous simulators to UC

The simulator $\mathcal{S}$ is given oracle access to $\mathcal{T}$, simulates copies of $\mathcal{F}_{\text{CLOCK}}$ and $\mathcal{F}_{\text{BD-AUTH}}$, and proceeds as follows.

- On input $(\texttt{leaked}, p_i, v_{\mathcal{F}})$ from $T_N(\mathcal{F})$, compute $(m_{i,j,r})_{p_j \in \mathcal{P}\setminus\{p_i\}} := \mathcal{T}((m_{i,j,r-1})_{p_i\in\mathcal{C}}, v_{\mathcal{F}})$ and store the messages in a buffer $\mathcal{B}_i$. Simulate the first message on a channel $\mathcal{F}_{\text{BD-AUTH}}(p_i, p_j)$.
- On input $(\texttt{activated}, p_i)$ where $\mathcal{B}_i$ is not empty, remove the first message m from $\mathcal{B}_i$ and simulate m as a message on the respective channel $\mathcal{F}_{\text{BD-AUTH}}(p_i, p_j)$. On the first such activation after $\mathcal{B}_i$ is empty, simulate a message $(\texttt{switch}, p_i)$ from $\mathcal{F}_{\text{CLOCK}}$. Once this message has been simulated for all $p_i \in \mathcal{H}$, proceed to the next step.
- On further input $(\texttt{activated}, p_i)$, if this is the first such message, send $(\texttt{end round})$ to $\mathcal{T}$ to obtain the value $v_{\mathcal{F}}$ and Send $(\texttt{end round}, v_{\mathcal{F}})$ to $T_N(\mathcal{F})$ to obtain the outputs $(y_{i,r})_{p_i\in\mathcal{C}}$, and provide these values to $\mathcal{T}$. For each such message, simulate one further message $(\texttt{switch}, p_i)$ for each $p_i \in \mathcal{H}$.
- Upon $(\texttt{corrupt}, p_i)$ for $\mathcal{F}_{\text{CLOCK}}$, issue $(\texttt{corrupt}, p_i)$ to $T_N(\mathcal{F})$ to obtain $v_{\mathcal{F}}$ and use $\mathcal{T}$ to obtain the state of π_i, from which the state of π'_i can be generated. Unless explicitly corrupted, simulate π_i as honest (but note that π'_i looses the guarantees of $\mathcal{F}_{\text{CLOCK}}$). Yet, if in a round not all messages expected by π'_i are in the channels, set the undefined $m_{i,j,r}$ to $\perp$.
- On a message $(\texttt{corrupt}, p_i)$ to $\mathcal{F}_{\text{BD-AUTH}}(p_i, p_j)$ or $\mathcal{F}_{\text{BD-AUTH}}(p_j, p_i)$, obtain the internal state of π'_i as above, mark p_i as corrupted for the channel. If, for some $p_j \in \mathcal{H}$, $\mathcal{Z}$ provides an input message m to $\mathcal{F}_{\text{BD-AUTH}}(p_i, p_j)$, record this as $m_{i,j,r-1} := m$ for the activation. (If $p_j \in \mathcal{H}$ is activated without such an input, set $m_{i,j,r-1} := \perp$.) Unless explicitly corrupted, simulate π'_i as honest.
- Upon $(\texttt{corrupt})$ to π'_i, obtain the state of π'_i as above and leak it to $\mathcal{Z}$. If p_i is not corrupted on all channels and the $\mathcal{F}_{\text{CLOCK}}$, make $\mathcal{Z}$ use the corresponding honest interfaces.

the value $y_{i,r}$ is made available for output in both cases and the message $(\texttt{switch}, i)$ is simulated, either by $\mathcal{Z}''$ or by $T_N(\mathcal{F})$ and $\mathcal{S}$, respectively.

- By the above arguments, the returned value $(\texttt{output}, y_{i,r'})$ upon input $(\texttt{output})$ at π'_i is always consistent.
- Corruption messages $(\texttt{corrupt}, p_i)$ to the functionality $\mathcal{F}_{\text{CLOCK}}$ or the channels $\mathcal{F}_{\text{BD-AUTH}}(p_i, p_j)$ or $\mathcal{F}_{\text{BD-AUTH}}(p_j, p_i)$ have no immediate effect on the computation. The indirect effects are described in the previous steps.
- Upon a corruption message $(\texttt{corrupt})$ to the protocol machine π'_i, the simulated internal state of π'_i is leaked to $\mathcal{Z}$ in both cases. As above, the state is obtained in the same way. By the construction of the protocol machine π'_i, the state can be easily derived from the state of π_i.

By the same arguments as above, this shows that if no environment can distinguish the real execution of π and the ideal execution of $\mathcal{F}$ in the model of [33], then no environment can distinguish the $\{\mathcal{F}_{\text{CLOCK}}, \mathcal{F}_{\text{BD-AUTH}}\}$-hybrid execution of π' and the ideal execution of $\mathcal{F}'$ in UC. This concludes the proof. $\square$

Both the model of [33] and UC allow for a "universal" composition operation (which is the terminology from [12] for using sub-protocols). For the full proof of Lemma 13, we have to generalize the statement of Lemma 15 to protocols defined in hybrid models.

We first generalize the mapping $T_N(\cdot)$ to $T^c(\cdot)$, where the parameter $c \in \mathbb{N}$ denotes the number of empty activations required by the functionality before providing output to the parties. This is necessary because the $T_N(\mathcal{G})$-hybrid protocol $C_N(\pi)$ must, besides sending messages and synchronizing using the clock, also provide the activations to the functionality $T_N(\mathcal{G})$. In more detail, the statement we are going to prove is that π implements the functionality $\mathcal{F}$ in the $\mathcal{G}$-hybrid model of [33] if and only if $C_N(\pi)$ UC-implements $T^{c+n+1}(\mathcal{F})$ in the $\{T^c(\mathcal{G}), \mathcal{F}_{\text{CLOCK}}, \mathcal{F}_{\text{BD-AUTH}}\}$-hybrid model. The protocols $C_N(\pi)$, after computing the messages $(m_{i,j,r})_{p_i\in\mathcal{P}}$, the output $y_{i,r}$, and the input $x^{\mathcal{G}}_{i,r}$ for $\mathcal{G}$, first provides input $x^{\mathcal{G}}_{i,r}$ as well as sufficiently many activations to $T_N(\mathcal{G})$. Then, $C_N(\pi)$ sends the messages $(m_{i,j,r})_{p_j\in\mathcal{P}\setminus\{p_i\}}$ via the channels $\mathcal{F}_{\text{BD-AUTH}}(p_i, p_j)$. In the beginning of the next round, $C_N(\pi)$ also requests the output $y_{i,r}$ from $T_N(\mathcal{G})$, which is also taken as an input to the computation of the messages for the next round.

The converters and the simulators in the proof of Lemma 15 must be adapted only slightly: the increased number of empty activations for honest parties has to be taken into account, and the values communicated at the adversarial interface of $\mathcal{G}$ must be forwarded to and from $\mathcal{Z}$.

E Hofheinz and Müller-Quade's Model

Hofheinz and Müller-Quade [25] devise another model for synchronous computation based on the paradigms of the framework of universally composable security [12]. The definition of a protocol execution differs strongly from both the models of [12] and [33]. The model assumes a network of pairwise authenticated communication channels between the honest parties.

E.1 Description of the Model

The execution is defined as an interaction between ITMs. The definition of these machines is similar to the one defined in [10], but differs slightly with respect to the communication and the definition of efficiency. As in [12], the entities involved in the execution are the environment $\mathcal{Z}$, the adversary $\mathcal{A}$, the protocol machines π, and the ideal functionalities $\mathcal{F}_j$.

The execution proceeds in rounds, each of which is further divided into three phases: the *attack phase*, the *party computation phase*, and the *ideal functionality computation phase*. In each of these phases, only a subset of the ITMs is activated, and the interaction within each of these phases follows a specific set of rules. The *attack phase* models the activities of $\mathcal{Z}$ and $\mathcal{A}$. In particular, the allowed interaction corresponds to the actions of corrupted parties; in this phase, the adversary is allowed to corrupt further parties, interact with the functionalities $\mathcal{F}_j$ in the name of the corrupted parties, and generate messages to the honest parties in the name of the corrupted ones. In the *party computation phase*, all honest parties are activated in parallel, obtain their inputs, the outputs of the $\mathcal{F}_j$, and the messages from all other parties, and generate the respective responses. After this phase, only the messages to the $\mathcal{F}_j$ and the adversary are delivered immediately. In the *ideal functionality computation phase*, all $\mathcal{F}_j$ are activated with the inputs generated by the honest parties. After this phase, the computation proceeds to the *attack phase* of the next round.

Upon corruption of a party p_i, $\mathcal{A}$ immediately obtains p_i's internal state containing the complete history with tapes and head positions. From there on, $\mathcal{A}$ may write arbitrary messages on the outgoing communication tape in the name of p_i and obtains all messages that are sent to p_i.

E.2 Relation to our Model

It turns out that the network implicitly assumed by [25] is very strong. Yet, the model *does* allow for guaranteed termination. Overall, one can show that it is embedded into the $\{\mathcal{F}_{\text{CLOCK}}, \mathcal{F}_{\text{MS+}}\}$-hybrid world of our model. Note that, in particular, the message sent by an honest party can no longer be changed after they are input to $\mathcal{F}_{\text{MS+}}$.

Functionality $\mathcal{F}_{\text{MS+}}(\mathcal{P})$

Initialize a round counter $r := 1$ and a bit $d_i := 0$ for each $p_i \in \mathcal{P}$. For each round, proceed as follows.

- Upon receiving input ($\texttt{send}, \boldsymbol{M}$) from a party $p_i \in \mathcal{P}$, where $d_i = 0$ and $\boldsymbol{M}$ is a vector of $|\mathcal{P}|$ messages, record $(p_i, \boldsymbol{M}, r)$, set $d_i := 1$ and output ($\texttt{input}, p_i$) to the adversary.
- Once $d_i = 1$ for all $p_i \in \mathcal{H}$, provide the recorded tuples $(p_i, \boldsymbol{M}, r)$ to $\mathcal{A}$ for all $p_i \in \mathcal{H}$, and allow the adversary to specify the messages to be sent in the name of the players $p_j \in \mathcal{P} \setminus \mathcal{H}$.
- Upon receiving input ($\texttt{receive}$) from party $p_j \in \mathcal{P}$: if $d_i = 1$ for all $p_i \in \mathcal{H}$, set $d_i := 0$ for all $p_i \in \mathcal{H}$ and set $r := r + 1$ (if the adversary did not specify any messages, they are defined to be $\perp$). Output all messages that have been sent to p_i in round $r - 1$.

The fact that the model of [25] assumes this network can be seen by analyzing the phases that describe the computation: messages are generated by all honest parties in parallel, without giving the adversary the possibility to interfere. Moreover, the message delivery mechanism is defined in such a way that the adversary cannot prevent messages that have been sent while a party was still honest from being delivered (to other honest parties).