Concurrent Composition of Secure Protocols in the Timing Model

Yael Tauman Kalai[†*]
M.I.T
yael@csail.mit.edu

Yehuda Lindell[†]
Dept. of Computer Science
Bar-Ilan University, Israel
lindell@cs.biu.ac.il

Manoj Prabhakaran[†]
Princeton University and UCLA
mp@cs.princeton.edu

July 28, 2005

Abstract

In the setting of secure multiparty computation, a set of mutually distrustful parties wish to securely compute some joint function of their inputs. In the stand-alone case, it has been shown that *every* efficient function can be securely computed. However, in the setting of concurrent composition, broad impossibility results have been proven for the case of no honest majority and no trusted setup phase. These results hold both for the case of general composition (where a secure protocol is run many times concurrently with arbitrary other protocols) and self composition (where a single secure protocol is run many times concurrently).

In this paper, we investigate the feasibility of obtaining security in the concurrent setting, assuming that each party has a local clock and that these clocks proceed at approximately the same rate. We show that under this mild timing assumption, it is possible to securely compute *any* multiparty functionality under concurrent *self* composition. We also show that it is possible to securely compute *any* multiparty functionality under concurrent *general* composition, as long as the secure protocol is run only with protocols whose messages are delayed by a specified amount of time. On the negative side, we show that it is impossible to achieve security under concurrent general composition with no restrictions whatsoever on the network (like the aforementioned delays), even in the timing model.

Keywords: theory of cryptography, secure multiparty computation, concurrent composition, timing assumptions.

[*]Supported in part by NSF CyberTrust grant CNS-0430450

[†]Part of this work was carried out while the authors were all at IBM T.J.Watson Research, New York.

1 Introduction

1.1 Background

In the setting of secure multiparty computation, a set of parties with private inputs wish to jointly compute some functionality of their inputs. Loosely speaking, the security requirements of such a computation are that nothing is learned from the protocol other than the output (privacy), and that the output is distributed according to the prescribed functionality (correctness). More exactly, the result of an execution of a secure protocol must be like the result of an "ideal execution" with an incorruptible trusted party who (honestly) computes the function for the parties (cf. [10] or [25, Section 7.1]). These security requirements must hold in the face of a malicious adversary who controls some subset of the parties and can arbitrarily deviate from the protocol instructions. Powerful feasibility results have been shown for this problem, demonstrating that *any* multiparty probabilistic polynomial-time functionality can be securely computed for any number of corrupted parties, assuming the existence of enhanced trapdoor permutations [44, 27, 25].

Security under concurrent composition. The above-described feasibility results relate only to the stand-alone setting, where a single protocol is run in isolation. However, in modern network settings, protocols must remain secure even when many protocol executions take place concurrently and are being attacked in a coordinated manner. Unfortunately, the security of a protocol in the stand-alone setting does not necessarily imply its security under concurrent composition. Therefore, an important research goal is to re-establish the feasibility results of the stand-alone setting for the setting of concurrent composition. There are two main types of concurrent composition that have been considered:

1. *Concurrent self composition:* In this setting, a single protocol is run many times concurrently in a network. Formally, "concurrency" means that the adversary has full control over the scheduling of all messages sent by the honest parties.

2. *Concurrent general composition:* In this setting, a protocol is run many times in an arbitrary network. That is, the protocol is run many times concurrently, alongside other secure and insecure protocols, again with the scheduling being fully controlled by the adversary.

On the positive side, it has been shown that in the case of an honest majority, or a trusted setup phase (e.g., for generating a common reference string or for generating a secure public-key infrastructure), any functionality can be securely computed under concurrent general composition [11, 14, 2]. Thus, in these cases, we obtain the same broad feasibility results of the stand-alone model (except that in the stand-alone model, neither an honest majority nor a trusted setup phase is needed).

When considering the case of *no honest majority* and *no trusted setup* in the setting of concurrent composition, the situation is completely different. Recent impossibility results have demonstrated that in such a setting, large classes of functionalities cannot be securely computed [12, 11, 13, 34, 35]. These results hold for both concurrent general composition *and* concurrent self composition. In fact, these two types of composition have been shown to be (almost) equivalent [35]. Therefore, in the natural setting of no trusted setup and no honest majority (including the important *two-party case*), it is impossible to construct protocols that remain secure in the setting of full concurrency.

There are a number of possible ways to overcome these impossibility results. One direction is to weaken the security requirements; this approach was taken by [38, 42]. Another direction, and the one taken in this paper, is to introduce realistic assumptions on the adversary or network, while providing the same strong security guarantees as for the stand-alone setting. Needless to say, it is best to not assume any restriction whatsoever. However, as we have mentioned, this is not possible. We therefore consider a very reasonable network restriction that holds in real networks today.

Timing assumptions. The network restriction that we consider is a *timing* assumption on the network. Timing assumptions were first used in the context of secure protocol composition by [19] who used them to achieve (efficient) zero-knowledge protocols that remain secure under concurrent self composition. (An equivalent formulation of these assumptions was given in [26], and our presentation is more according to this latter formulation.) There are two specific assumptions involved here:

- *Assumption 1 – bounded clock drift:* It is assumed that the parties' local clocks proceed at approximately the same rate. Specifically, there exists a *global* bound $\epsilon \geq 1$ such that when one local clock advances t time units, every other local clock advances t' time units where $t/\epsilon \leq t' \leq t\epsilon$. We stress that there is *no assumption* regarding the synchronization of the parties' local clocks with respect to each other (and, in particular, they may read completely different times).
- *Assumption 2 – maximum latency:* It is assumed that an upper bound Δ is known on the time it takes for a message to be computed, sent and delivered from one party to another. In other words, Δ is the *maximum latency* over the network (plus the time it takes to carry out the local computation for generating the message that is sent). For simplicity, we assume that all local computation is instantaneous, and that Δ measures the latency only (or, in other words, the time that it takes for the adversary to deliver messages).

The second of these two assumptions is far more problematic than the first. This is due to the fact that in real settings, the variance of network latency can be very large. Thus, a global upper bound would have to be very large. As we will see, taking such a high upper bound would greatly hinder performance. In addition, any reasonable bound is unlikely to always hold, thus potentially compromising the security of the protocol. In contrast, local clocks are usually very accurate, at least with respect to the drift.

Motivated by this observation, we relate to these assumptions differently. More specifically, our definition of *security* for the timing model relies *only* on the first assumption regarding the clock drift. Therefore, security holds as long as the drifts of the clocks are not too far apart, and *irrespective of the network latency* (which may, however, cause the execution to terminate unsuccessfully). The latency assumption is only used to ensure *liveness* (or non-triviality of protocols). Namely, we only require that the protocol terminates successfully if it does not come under attack and the latency is indeed lower than Δ.

The use of timing assumptions. As in other works, the timing assumptions are used for introducing time-out and delay operations in the protocol instructions. A time-out command is of the form: "if more than $f(\Delta, \epsilon)$ time units have passed since message x was sent (or received), and message y has not yet been received, then output time-out and halt the execution" (where f is a function specified by the protocol). A delay command is of the form: "wait $g(\Delta, \epsilon)$ time units before sending message y". Typically, the use of these operations is to limit the interleaving of different protocol executions. Specifically, delay and time-out commands are used to ensure properties of the form: if part A of execution i begins after part B of execution j, then part B of execution j is completed before part A of execution i. This is achieved by timing-out if B takes too long and delaying to makes sure that A takes long enough, as depicted in Figure 1. The differences in the lengths of part A and part B in the different executions shown in the figure are due to the control that the adversary has over message delivery. We stress that the time-out and delay instructions depend on the parties' local clocks only, and so do not rely on any global synchronization.

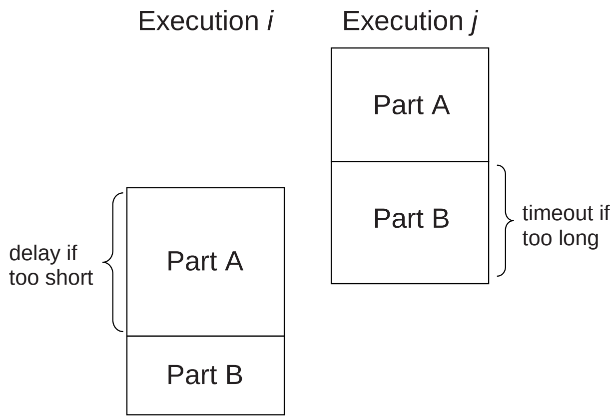

Figure 1: limiting the interleaving (notice that part A must take longer than part B).

Limiting time-out damage. As we have described, time-out instructions are used in order to limit the adversary's power in scheduling the executions. However, if the network latency during a protocol execution is higher than usual (say, due to high network traffic), then a time-out can occur even when no adversary is present.[1] This therefore raises the question of what actions should honest parties take after a time-out occurs. In particular, can a timed-out protocol execution be safely restarted? Fortunately, our protocols have the property that they remain secure if they are restarted from scratch after a time-out.

In order to further clarify this point, we distinguish between two types of failures: abort failures that occur due to foul play by participating parties (and are present even in the stand-alone case when there is no honest majority), and time-out failures that occur due to high network latency (or by the adversary stalling or blocking messages that are sent). In the case of an *abort failure* (again, even in the stand-alone case), security is *not* guaranteed if the honest parties restart the protocol execution. This is due to the fact that the adversary may have received its own output, and based on this output has decided to cause the honest parties to abort.[2] In contrast, we argue that it should be possible to restart protocol executions that are halted due to *time-out failures*. This is due to the fact that such a failure can occur even if there is no adversarial interference, and just due to the network latency being high at the time of the execution. In order to ensure that timed-out executions can be restarted without any damage to security, our definition of security requires that time-outs (by honest participants) are only allowed to occur in an early stage of the protocol, before any information about the output is revealed. We note that since timed-out protocols can be restarted safely, a relatively optimistic estimate on the network latency can be taken with the cost being that timed-out protocol executions are simply restarted. (There is a tradeoff here between choosing a large Δ that slows down all protocol executions and choosing a small Δ that results in more executions being timed-out and restarted.) We remark that previous

[1] Of course, one could set Δ to be an upper bound that includes latencies that are far higher than the average. However, this will have the effect of significantly delaying all executions.

[2] For the sake of concreteness, consider the case that parties run a coin-tossing protocol. Then, the first party to receive output can cause an abort if the first bit of the output is not 0. By re-executing upon abort, this party can bias the outcome so that the resulting string always has the first bit set to 0.

works that used timing assumptions considered only the problem of concurrent zero-knowledge, where essentially no output is generated. The above discussion is therefore a "non-issue" in that case, and protocol executions can always be restarted.

1.2 Our Main Results

We investigate the feasibility of constructing protocols that are secure under concurrent composition in the timing model. We consider both self and general composition, with the following main results:

1. *Concurrent self composition:* We show that in the timing model, every multiparty functionality $\mathcal{F}$ can be securely computed under concurrent self composition. We note that the model of concurrent self composition that we consider here is one *without* fixed roles. Thus, for example, in the setting of zero-knowledge, parties can play both the prover and verifier roles simultaneously. This is the first zero-knowledge protocol (for the setting of unbounded concurrency) that has this property.

2. *Concurrent general composition:* For this setting, we have both positive and negative results:

   (a) *Positive result:* We show that in the timing model, every multiparty functionality $\mathcal{F}$ can be securely computed under concurrent general composition, as long as the arbitrary protocols that are running in the network all have the property that their messages are delayed by some specified amount of time (the exact delay required is given by the secure protocol computing $\mathcal{F}$).

   (b) *Negative result:* We show that it is impossible to achieve security under concurrent general composition if no timing restrictions (like delays) are imposed on the arbitrary protocols running in the network.

Positive results. We now elaborate on our main positive results. We note that all of our results relate to the setting of no honest majority. Therefore, our definition of security does not guarantee fairness. That is, the adversary may receive output while the honest parties do not. This is standard for the case of no honest majority, even in the stand-alone model. Our result regarding concurrent self composition is stated as follows.

Theorem 1.1 *Assume that there exist enhanced trapdoor permutations.*[3] *Then, in the timing model, any multiparty functionality $\mathcal{F}$ can be securely computed under concurrent self composition, in the presence of static adversaries.*

The proof of Theorem 1.1 gives the first construction of a protocol that achieves security (in the standard sense) under concurrent self composition, without relying on a trusted setup phase, an assumed honest majority, or an assumed a-priori bound on the number of executions taking place concurrently. As we have mentioned above, our definition for concurrent self composition makes no limitation on the roles played by the parties. Thus, Theorem 1.1 implies the existence of a concurrent zero-knowledge protocol (in the timing model) that remains secure even when the adversary can carry out a man-in-the-middle attack (i.e., where the adversary may simultaneously verify one proof and prove another). This is the first zero-knowledge protocol with this property.

[3] An *enhanced* trapdoor permutation has the property that it is hard to invert even if you know the coins used to sample the value from the domain; see [25, Appendix C.1].

(We note that an analogous result is known in the model of bounded concurrency, where the number of concurrent executions is a priori bounded [39].)

In order to state our positive result for concurrent general composition in more detail, we introduce the following terminology. We say that a protocol π is δ-delayed if every message in π is delayed by at least δ units of time before it is sent. We stress that the honest parties must carry out these delays, but there is no restriction on the adversary's behavior. We also stress that the protocol π may have other timing instructions; the only requirement is that it be δ-delayed. We note that if the protocol π is not already δ-delayed, then it may be converted into such a protocol by adding delays as required. This conversion is straightforward if the original (not δ-delayed) π does not contain any timing instructions. However, if it does, adding delays may be problematic. We discuss this issue in more length below (in the paragraph "Using Theorem 1.2", and in Section 3.2).

We say that a protocol ρ is secure under concurrent general composition with delays if there exists a value δ (that may depend on ρ) such that ρ remains secure when run many times concurrently with any protocol π that is δ-delayed. We stress that ρ itself is not necessarily δ-delayed; rather, it is the arbitrary protocol π running alongside the many executions of ρ that must be δ-delayed. Note that this notion of composition restricts the set of protocols that can be run concurrently with the secure protocol ρ to those protocols π that are δ-delayed. Thus, it is not concurrent general composition in the full sense, but a restriction of it. We are now ready to state the theorem:

Theorem 1.2 *Assume that there exist enhanced trapdoor permutations. Then, in the timing model, any multiparty functionality $\mathcal{F}$ can be securely computed under concurrent general composition with delays, in the presence of static adversaries.*

We note that Theorem 1.1 follows immediately from Theorem 1.2. This is due to the fact that concurrent self composition is a special case of concurrent general composition. (Recall that in the setting of concurrent general composition, the secure protocol ρ is run many times concurrently, alongside any arbitrary other protocol. Thus, the setting of concurrent self composition is just where the arbitrary protocol running alongside is empty. See Section 2.3 for the actual definition.)

We prove Theorems 1.1 and 1.2 by first constructing a protocol that securely realizes the common random string (CRS) functionality under concurrent general composition with delays. This functionality simply hands each party a uniformly distributed string, and as such is essentially a multiparty coin-tossing functionality. We then rely on the fact that any efficient functionality can be securely computed in the common random string model [14]. Thus combining our protocol for securely realizing the CRS functionality together with a delayed version of the protocol of [14], we obtain that any efficient functionality can be securely computed under concurrent general composition with delays. We note that we actually obtain a stronger result than that stated in Theorem 1.2. Specifically, we show that it is possible to construct a set of protocols that are secure in the timing model and can be run concurrently with each other and with any δ-delayed protocol. This is achieved by having all the protocols first use our protocol for securely computing the CRS functionality. Then, each protocol uses a different instantiation of the protocol of [14], depending on the functionality to be computed.[4] See the end of Section 3.2 for more discussion.

Using Theorem 1.2. On the one hand, Theorem 1.2 is strong in that it guarantees security in the face of any δ-delayed protocol π. On the other hand, "most" protocols are *not* δ-delayed. Despite this, we argue that in principle, Theorem 1.2 provides a general way to achieve security in

[4] We note that Theorem 1.2 itself does not imply that different instantiations of our protocol can be run together securely. The reason for this is that our protocols are only guaranteed to be secure when run together with δ-delayed protocols, and our protocols themselves are *not* δ-delayed.

real network settings. Consider first the case of an arbitrary protocol π that itself does not use any timing instructions (in particular, π never refers to a clock). We call such a protocol timing-free. Then, π may be modified so that each message is delayed by some δ units of time, and this makes no difference to its result. In particular, if the original π was secure, then so is the modified version. Now, such a modification is easily made to such protocols, and does not even require any "internal" tampering (rather, honest parties can just stall messages on their outgoing communication ports). We conclude that honest parties may continue to run any arbitrary timing-free protocol that they wish, with the only price being that of *efficiency* due to the fact that they now need to delay all the messages of these arbitrary protocols. Of course, such a severe slow-down is unlikely to be tolerated in real settings. Thus reducing the number and length of the delays imposed by our protocol is an important question for future research. Nevertheless, these are secondary issues; our focus is on the question of *feasibility.* We stress again that there is no assumption regarding the delaying of messages by corrupted parties. Rather, security is guaranteed as long (*and only as long*) as the *honest parties* delay all the messages of π, as instructed. Thus, the modification of a timing-free protocol into a δ-delayed protocol can easily be achieved *in principle* in real network settings.

The above discussion relates to timing-free protocols π. Regarding protocols π that also use timing instructions, the situation is more problematic. First, note that if a protocol π that contains timing instruction is *already* δ-delayed, then it can be run concurrently with our secure protocol ρ. In contrast, if π is *not* δ-delayed, then one cannot naively add delays before every message in order to convert it into a δ-delayed protocol, because this may adversely affect the existing time-based instructions in π. Rather, these timing instructions in π must also be adjusted. Loosely speaking, this can be achieved by modifying the clock used by π so that every δ units of time are interpreted in π as a single unit of time. This results in a δ-slowdown of π, and enables its modification to a δ-delayed protocols. A more exact description of how this works is provided at the end of Section 3.2.

Negative results. As we have mentioned, the timing assumptions are used for introducing time-out and delay instructions in the protocol. Furthermore, our use of delays is extensive, since we do not only insert delay instructions into our secure protocol ρ, but we also require that *every* message of every arbitrary protocol π that runs concurrently to ρ is also delayed. This is clearly a drawback of our result. However, we show that some sort of "time-based" modification of π is *essential* for achieving security. Recall that a protocol π is timing-free if it never looks at its clock and so contains no time-based instruction (and in particular no delay or time-out instructions). We say that a protocol ρ is secure under concurrent general composition with timing-free protocols, if it is secure when run concurrently with any timing-free protocol π. We prove the following theorem (stated informally here):

Theorem 1.3 *There exist large classes of efficient functionalities that cannot be securely computed under concurrent general composition with timing-free protocols, even in the timing model, unless an honest majority or a trusted setup phase are assumed.*

We conclude that some timing-based modification must be introduced into π. The question of how many delays must be introduced into π, and of what length, is left open by this work.

1.3 Related Work

Secure computation was first studied in the stand-alone model, where it was shown that any multi-party functionality can be securely computed [44, 27, 6, 15]. The study of concurrent composition of protocols was initiated by [23] in the context of witness indistinguishability, and was next considered by [18] in the context of non-malleability. Until recently, the majority of work on concurrent composition was in the context of concurrent zero-knowledge [19, 43]. The concurrent composition of protocols for *general secure computation* was only considered much later. Specifically, the first definition and composition theorem for security under concurrent general composition was presented by [17] for the case of perfect security in the information-theoretic setting. Next, [41] considered the computational setting and the case that a secure protocol is executed once in an arbitrary network. The general case for the computational setting, where many secure protocol executions may take place (again, in an arbitrary network) was then considered in the definition (and composition theorem) of universal composability [11]. It was also shown that any functionality can be securely realized in this setting assuming an honest majority [11], or assuming a trusted setup phase in the form of a common random string [14] or in the form of a key registration functionality [2]. However, in the case of no honest majority or trusted setup, broad impossibility results have been demonstrated for universal composability, concurrent general composition and concurrent self composition [13, 34, 35].

These impossibility results justify and provide motivation for considering restricted network settings and weaker notions of security. One type of restriction that has been considered for concurrent self composition is that of m-bounded concurrency, where an upper bound m on the global number of concurrent executions is assumed [1]. In this model, both positive results [33, 40, 39] and lower bounds [33, 35] have been demonstrated. In our opinion, the timing model is a more realistic assumption than that of bounded concurrency. A different way of bypassing the aforementioned impossibility results (and one not taken in this paper) is to consider weaker notions of security. This approach was taken by [38] and [42] who both provide "additional power" to the ideal adversary (i.e., they allow the simulator to run in more time than the real adversary). We remark that such solutions provide weaker security guarantees. In particular, a protocol proven secure while using a super-polynomial simulation strategy may have adverse effects on the arbitrary in which it is being run.

As we have mentioned, timing assumptions were introduced in the cryptographic context by [19]. Subsequently, they have been used in a number or works, including [21, 20, 31, 26]. However, all of these works considered the security of specific cryptographic tasks (namely, zero-knowledge and authentication-type protocols). Furthermore, they all considered security under a limited form of concurrent self composition. This paper is the first to use timing assumptions in order to construct a secure protocol for *any* multiparty functionality, that remains secure under concurrent self composition, and under concurrent general composition with delays.

2 Definitions and Tools

2.1 Preliminaries

We denote the security parameter by n. A function $\mu(\cdot)$ is negligible in n (or just negligible) if for every polynomial $p(\cdot)$ there exists a value N such that for all $n > N$ it holds that $\mu(n) < 1/p(n)$. Let $X = \{X(n,a)\}_{n\in\mathsf{N},a\in\{0,1\}^*}$ and $Y = \{Y(n,a)\}_{n\in\mathsf{N},a\in\{0,1\}^*}$ be distribution ensembles. Then, we say that X and Y are computationally indistinguishable, denoted $X \stackrel{c}{\equiv} Y$, if for every probabilistic polynomial-time distinguisher D there exists a function $\mu(\cdot)$ that is negligible in n, such that for *every* $a \in \{0,1\}^*$,

$$|\Pr[D(X(n,a)) = 1] - \Pr[D(Y(n,a)) = 1]| < \mu(n)$$

Typically, the distributions X and Y will denote the output vectors of the parties in real and ideal executions, respectively. In this case, a denotes the parties' inputs.

A machine is said to run in polynomial-time if its number of steps is polynomial in the *security parameter*, irrespective of the length of its input. Formally, each machine has a security-parameter tape upon which 1^n is written. The machine is then polynomial in the contents of this tape.

2.2 Security Under Concurrent General Composition in the Timing Model

In this section, we present the definition of concurrent general composition in the timing model. This is a direct extension of the definition of concurrent general composition in the standard (non-timed) model, as defined for example in [11, 34]. Informally speaking, concurrent general composition considers the case that a protocol ρ for securely computing some ideal functionality $\mathcal{F}$, is run concurrently (many times) with arbitrary protocols running in the network. This arbitrary network is modelled as a "calling protocol" π with respect to the functionality $\mathcal{F}$. That is, π is a protocol that contains, among other things, "ideal calls" to a trusted party that computes the functionality $\mathcal{F}$. This means that in addition to standard messages sent between the parties, protocol π's specification contains instructions of the type "send the value x to the trusted party and receive back output y". Then, the real-world scenario is obtained by replacing the ideal calls to $\mathcal{F}$ in protocol π with real executions of protocol ρ. (When we say that an ideal call to $\mathcal{F}$ is replaced by an execution of ρ, this means that the parties run ρ upon the same inputs that π instructs them to send to the trusted party computing $\mathcal{F}$.) The composed protocol is denoted by π^ρ and it takes place without any trusted help. We note that in this composed protocol, messages of π may be sent concurrently to the executions of ρ (even though π "calls" ρ). In addition, the inputs are determined by π and may therefore be influenced by previous ρ-outputs and the party's overall view in the arbitrary network. Security is defined by requiring that for every protocol π that contains ideal calls to $\mathcal{F}$, an adversary interacting with the composed real protocol π^ρ (where there is no trusted help) can do no more harm than in an execution of π where a trusted party computes all the calls to $\mathcal{F}$. This therefore means that ρ behaves just like an ideal call to $\mathcal{F}$, even when it is run concurrently with any arbitrary protocol π.

The above informal description is the same in the standard (non-timed) model and in the timing model. The only difference is that in the timing model, the parties have access to local clocks. This will be described below.

Secure multiparty computation. A multiparty computation task for a set of parties $P_1, \ldots, P_m$ is cast by specifying a (probabilistic polynomial-time) multiparty ideal functionality machine $\mathcal{F}$ that receives inputs from parties and provides outputs. The aim of the computation is for the parties to jointly compute the functionality $\mathcal{F}$. According to the standard ideal/real model paradigm [28, 4, 36, 10, 25], a real protocol execution is compared to an ideal execution where a *trusted third party* computes $\mathcal{F}$ for the parties. Instead of explicitly considering such a trusted party, we sometimes talk about the parties (and adversary) communicating directly with the ideal functionality. This is just shorthand for saying that the parties communicate with the trusted party computing the functionality.

Adversarial behavior. In this work we consider malicious, static adversaries. That is, the adversary controls an a priori fixed subset of the parties who are said to be corrupted. The corrupted parties follow the instructions of the adversary in their interaction with the honest parties, and may arbitrarily deviate from the protocol specification. The adversary also receives the view of the corrupted parties at every stage of the computation. In our model, the adversary also has full control over the scheduling of the delivery of all messages. Thus, the network is asynchronous. Finally, as we will see below, the adversary has *some* control over the clocks of honest parties.

The $\mathcal{F}$-hybrid model. Let π be an arbitrary protocol that utilizes ideal interaction with a trusted party computing the multiparty functionality $\mathcal{F}$ (recall that π actually models arbitrary network activity). There may be many copies of the functionality, and so these copies are differentiated by unique session identifiers. A protocol π that runs in the $\mathcal{F}$-hybrid model contains two types of messages: standard messages and ideal messages. A standard message is one that is sent between two parties that are participating in the execution of π, using the point-to-point network (or broadcast channel, if assumed). An ideal message is one that is sent by a participating party (or the adversary) to the ideal functionality $\mathcal{F}$, or from the ideal functionality to a participating party (or the adversary). Ideal messages are typically *inputs* and *outputs* for the functionality being computed by the trusted party. However, in order to model execution failures, there are two "special" ideal messages (or instructions) that the adversary can send to the trusted party:

1. Abort instructions: These instructions play the same role as in stand-alone secure computation for the case of no honest majority. That is, in the case that an honest party receives an invalid message in a real protocol execution, it would halt and output abort (meaning that malicious behavior has been detected). The ideal adversary must therefore also be able to cause an honest party to output abort in the ideal (or hybrid) model.

   Thus, the adversary can issue instructions of the type $(\mathsf{abort}, sid, P_i)$ to the trusted party. Upon receiving such a message, the trusted party forwards (abort, sid) to P_i, who in turn sets its output from execution sid to abort.[5] We stress that an abort instruction can be issued at any time and for any party. Furthermore, once an honest party receives abort, it halts the execution (and also refuses to restart it).

2. Time-out instructions: These instructions are included in order to model the case that an honest party is instructed to output time-out in a protocol execution (in the timing model). As with abort, the ideal adversary must also be able to cause an honest party to output time-out in an ideal/hybrid execution. However, the mechanism for time-outs is different than for aborts. In particular, a time-out can only be issued *before* any output was generated.

   Formally, the adversary may issue an instruction of the type $(\mathsf{time\text{-}out}, sid)$ to the trusted party. Upon receiving such a message, the trusted party checks if it previously sent any output in execution sid.[6] If no outputs have been sent, then the trusted party sends $(\mathsf{time\text{-}out}, sid)$ to all parties and halts the execution. Otherwise (if outputs have been sent), the trusted party just ignores the time-out instruction.

We remark that if one party receives time-out, then no parties receive output. This is due to the fact that the functionality sends time-out to all parties (and so if the adversary delivers an output to an honest party, it can only be time-out). This is in contrast to aborts, where some parties may receive abort and some may receive their prescribed output. (The issue of whether all parties receive abort together or not is discussed in [29].)

[5]We note that in stand-alone secure computation in the synchronous model, abort is also output when a party doesn't receive all of its messages in a given round. In our model, a party will either just continue waiting (possibly forever), or will output time-out if so instructed by the protocol.

[6]In the case that the functionality is a simple function, all outputs are generated and sent at the same time. Thus, this reduces to the trusted party just checking if it has computed the function output yet. In the case of *reactive functionalities* where computation takes place over a number of phases, outputs are generated at different times. Here, the trusted party checks that *no outputs* were generated until this time.

Notice that the computation of π is a "hybrid" between the ideal model (where a trusted party carries out the entire computation) and the real model (where the parties interact with each other only). Specifically, the messages of π are sent directly between the parties, and the trusted party is only used in the ideal calls to $\mathcal{F}$.

As is standard for concurrent settings, the adversary controls the scheduling of all messages, including both standard and ideal messages. As usual, we assume that the parties are connected via authenticated channels. Therefore, the adversary can read all standard messages, and may use this knowledge to decide when, if ever, to deliver a message. (We remark that the adversary cannot, however, modify messages or insert messages of its own.) In contrast, the channels connecting the participating parties and the trusted third party are both authenticated *and* private. More precisely, ideal messages are comprised of a public header and a private body. The contents of a message that belong in the header or body is part of the functionality definition. In general, the public header contains information like the name and session identifier of the functionality for which the message is intended. We stress that although the adversary delivers the entire message, it can only read the public header, and cannot read the private body. However, we adopt the convention that the *length* of this private body is given to the adversary. (This models the fact that the lengths of inputs and outputs cannot be fully hidden from the adversary.)[7]

Computation in the $\mathcal{F}$-hybrid model proceeds as follows. The computation begins with the adversary receiving the inputs and random tapes of the corrupted parties. Throughout the execution, the adversary controls these parties and can instruct them to send any standard and ideal messages that it wishes. In addition to controlling the corrupted parties, the adversary delivers all the standard and ideal messages by copying them from outgoing communication tapes to incoming communication tapes. The series of activations is sequential. That is, the adversary is activated first, at which time it can carry out any arbitrary computation. It concludes its activation by writing a message to the incoming communication tape of either a party or an ideal functionality. A party (or an ideal functionality) that receives a message on its incoming communication tape is immediately activated. When it halts, the adversary is activated once again.[8] Upon being activated, the honest parties always follow the specification of protocol π. Specifically, upon receiving a message (delivered by the adversary), the party reads the message, carries out a local computation as instructed by π, and writes standard and/or ideal messages to its outgoing communication tape, as instructed by π. Likewise, the ideal functionality follows its prescribed instructions (and is never corrupted). At the end of the computation, the honest parties write the output value prescribed by π on their output tapes, the corrupted parties output a special corrupted symbol and the adversary outputs an arbitrary function of its view. Let n be the security parameter, let $\mathcal{S}$ be an adversary for the $\mathcal{F}$-hybrid model with auxiliary input $z \in \{0,1\}^*$, let $I \subseteq [m]$ be the set of corrupted parties, and let $\overline{x} = (x_1, \ldots, x_m) \in (\{0,1\}^*)^m$ be the vector of the parties' inputs to π. Then, the hybrid execution of π with ideal functionality $\mathcal{F}$, denoted $\text{HYBRID}^{\mathcal{F}}_{\pi,\mathcal{S},I}(n, \overline{x}, z)$, is defined as the output vector of all parties and $\mathcal{S}$ from the above hybrid execution.

The real model. Let ρ be a multiparty protocol. Intuitively, the composition of protocol π with ρ is such that a real execution of protocol ρ takes the place of an ideal call to $\mathcal{F}$. Formally, each party holds the code of a probabilistic interactive Turing machine (ITM) that works according to the specification of the protocol ρ.[9] When π instructs a party to send an ideal message α to the

[7]In this work, the ideal functionality that we consider generates a public common random string. Therefore, all communication between the parties and functionality can be made part of the public header.

[8]The adversary can activate parties at the beginning of the execution, before there are messages to deliver, by sending them a special "begin computation" message.

[9]Note that each party receives the same machine and thus the same set of instructions for ρ. This means that

ideal functionality $\mathcal{F}$ with session identifier sid, the party creates a new instantiation of the ITM for ρ, associates the identifier sid with this machine, and invokes it with input α. Any message that it later receives that is earmarked for ρ with identifier sid, it forwards to this ITM. All other messages (that are not earmarked for ρ) are answered according to π. Finally, when the execution of ρ with identifier sid concludes and a value β is written on the output tape of the ITM, the party copies β to the incoming communication tape for π, as if β is an ideal message (i.e., output) received from the copy of the ideal functionality $\mathcal{F}$ with identifier sid. This composition of π with ρ is denoted π^ρ and takes place without any trusted help. Thus, the computation proceeds in the same way as in the hybrid model, except that all messages are standard. (Like in the hybrid model, the adversary controls message delivery and can also read messages sent, but cannot modify or insert messages.) Let n be the security parameter, let $\mathcal{A}$ be an adversary for the real model with auxiliary input z, let $I \subseteq [m]$ be the set of corrupted parties, and let $\overline{x}$ be the vector of the parties' inputs to π. Then, the real execution of π with ρ, denoted $\text{REAL}_{\pi^\rho,\mathcal{A},I}(n, \overline{x}, z)$, is defined as the output vector of all the parties and $\mathcal{A}$ from the above real execution.

Interactive Turing machines with clocks. The above description of the $\mathcal{F}$-hybrid and real models is consistent with both the standard non-timed model and with the timing model (except for the time-out instructions which are really only relevant in the timing model). As we have mentioned, the difference between these models is that in the timing model, the parties have clocks. In order to model these clocks, we add a clock tape to the interactive Turing machines that model the parties in the network; we call such a modified machine an ITMC (interactive Turing machine with a clock). As we will see below, the adversary is the only machine to update the clock tapes of the parties. The leeway given to the adversary in its control over these tapes determines the model being considered. For example, if the adversary has full control and can write any values that it wishes to the clock tapes, then this is equivalent to a non-timed, fully asynchronous model. On the other extreme, if the adversary initializes all clocks to 0 and adds 1 to each clock at the same time, then this is equivalent to the fully synchronous model.[10] In the timing model, as introduced by [19], the adversary is somewhat limited in its power over the clock tapes. Specifically, the adversary can initialize the values of the clock tapes to any values that it wishes (this initialization takes place at the onset of the computation and models the fact that we do *not* require synchronized clocks). Following this initialization step, the adversary may update the clock of any party that it wishes, under the constraint that a bound on the *clock drift* is preserved. Loosely speaking, this restriction states that the clocks of all machines proceed at approximately the same rate (give or take ϵ).

More formally, let $M_1, \ldots, M_\ell$ be the ITMC's in the network and let $a_1, a_2, \ldots$ be the series of global states of all machines in the network, where a_j denotes the global state after the j^{th} activation of a machine by the real-model adversary. (Note that we do not include activations of the adversary, but just of the participating parties.) Denote the contents of the clock tape of machine M_i in activation a_j by $\mathsf{clock}_i(a_j)$, and let $\mathsf{clock}_i(a_0)$ be the initial value of its clock tape. Then, adversarial control over the clocks is modelled as follows:

1. Before the computation begins, the adversary is allowed to write any values that it wishes to the parties' clock tapes (if a value is not written, then the default is 0). These are the *initial* clock values.

separate, fixed roles are not defined for the different parties. Rather, the assignment of different roles (if they exist, like for example in zero-knowledge where there are separate prover and verifier roles) is assumed to be part of the functionality definition and protocol execution.

[10] Of course, just updating the clocks together does not necessarily force the adversary to activate all the parties in parallel (or essentially in parallel, by activating the parties sequentially in a round robin fashion). However, a protocol can force this by having a party abort if it does not receive its round i messages when its clock reads i.

2. Every time that the adversary is activated, it is given write access to the clock tapes of all the parties. This write access is limited in a natural way in that the adversary is only allowed to increase the current value. We stress that writing to a party's clock tape does *not* activate it (in this way, it is different to writing to a party's incoming communication tape).

The above describes *how* the adversary updates the clock tapes; it does not specify any limitations over these updates. In the timing model, it is assumed that the clocks all proceed within ϵ units of each other. That is, let $\epsilon \geq 1$ be a constant. We say that an adversary is ϵ-drift preserving if for every pair of parties P_i and P_j and for every $k = 1, 2, \ldots$,

$$\frac{1}{\epsilon} \cdot (\mathsf{clock}_j(a_k) - \mathsf{clock}_j(a_{k-1})) \leq \mathsf{clock}_i(a_k) - \mathsf{clock}_i(a_{k-1}) \leq \epsilon \cdot (\mathsf{clock}_j(a_k) - \mathsf{clock}_j(a_{k-1})) \quad (1)$$

In other words, whenever a party's clock is increased by some value δ, then all other clocks must be increased by some value between δ/ϵ and $\delta\epsilon$. An *equivalent* and more explicit way of stating this requirement is as follows.

Let $\epsilon \geq 1$ be a constant. Then, we say that an adversary is ϵ-drift preserving if there exist a series of values $\delta_1, \delta_2, \ldots$ so that for every i and every $k = 1, 2, \ldots$

$$\delta_k \leq \mathsf{clock}_i(a_k) - \mathsf{clock}_i(a_{k-1}) \leq \delta_k \cdot \epsilon$$

This means that between activation a_{k-1} and activation a_k, the clocks of all parties have increased by a value which is between δ_k and $\delta_k\epsilon$.[11]

Intuitively, one can think of δ_k as being the objective real time (although there may be a number of values δ_k that fulfill this condition, and in real life clocks can also be slower than the real time, so this is not really the case).

The timed $\mathcal{F}$-hybrid and real models. The only augmentation that we need to make to the above description of the $\mathcal{F}$-hybrid and real models is to incorporate the clocks and the adversary's limitations with respect to updating these clocks. Formally, we model the honest parties as interactive Turing machines with clocks, and require that the adversary (in both the $\mathcal{F}$-hybrid and real models) be ϵ-drift preserving. Everything else remains the same. Using the same notation as above, we denote the output of a hybrid execution of π with an ϵ-drift preserving adversary $\mathcal{S}$ and an ideal functionality $\mathcal{F}$ by $\text{HYBRID}^{\mathcal{F},\epsilon}_{\pi,\mathcal{S},I}(n, \overline{x}, z)$. Likewise, we denote the output of a real execution of π with ρ and an ϵ-drift preserving adversary $\mathcal{A}$ by $\text{REAL}^{\epsilon}_{\pi^\rho,\mathcal{A},I}(n, \overline{x}, z)$.

Security as emulation of a real execution in the hybrid model. Having defined the hybrid and real models, we can now define security of protocols. Generally speaking, the definition of security under concurrent general composition asserts that for any context, or calling protocol π, the real execution of π^ρ emulates the hybrid execution of π with ideal calls to $\mathcal{F}$. This is formulated by saying that for every real-model adversary there exists a hybrid model adversary for which the output distributions are computationally indistinguishable. As we have discussed in the introduction, we consider here concurrent general composition *with delays.* In order to define this notion, we define the set of protocols Π_δ so that $\pi \in \Pi_\delta$ if every message in π is delayed by *at least* δ units of time before being sent. Formally:

Definition 1 (delayed protocols): *Let $\delta(n)$ be a function. Then, the class $\boldsymbol{\Pi_\delta}$ is defined to be the set of protocols π with the property that upon security parameter n, every (standard or ideal) message in π is preceded by a* delay *instruction of length at least $\delta(n)$ units of time. Any protocol $\pi \in \Pi_\delta$ is called δ*-delayed.

[11] Notice that taking $\delta_k = \min_j\{\mathsf{clock}_j(a_k) - \mathsf{clock}_j(a_{k-1})\}$, Eq. (1) implies that $\max_j\{\mathsf{clock}_j(a_k) - \mathsf{clock}_j(a_{k-1})\} \leq \delta_k\epsilon$, which in turn implies that for every i, $\delta_k \leq \mathsf{clock}_i(a_k) - \mathsf{clock}_i(a_{k-1}) \leq \delta_k\epsilon$.

Security is formulated as described in the above informal discussion, but we require the "calling protocol" to be within the set Π_δ. We stress that security must hold for *every* delayed protocol $\pi \in \Pi_\delta$. Thus, we obtain general composition (at least, with respect to the set of protocols Π_δ).

Definition 2 (security under concurrent general composition with delays): *Let ρ be a polynomial-time protocol and let $\mathcal{F}$ be an ideal functionality. Then, ρ* securely realizes $\mathcal{F}$ under concurrent general composition with delays in the timing model with ϵ *if there exists a polynomial $\delta(\cdot)$ such that for every ϵ-drift preserving non-uniform polynomial-time real-model adversary $\mathcal{A}$, there exists an ϵ-drift preserving non-uniform probabilistic polynomial-time hybrid-model adversary $\mathcal{S}$ such that for every $I \subseteq [m]$:*

$$\left\{\text{HYBRID}^{\mathcal{F},\epsilon}_{\pi,\mathcal{S},I}(n,\overline{x},z)\right\}_{n\in\mathsf{N};\pi,\overline{x},z} \stackrel{\text{c}}{\equiv} \left\{\text{REAL}^{\epsilon}_{\pi^\rho,\mathcal{A},I}(n,\overline{x},z)\right\}_{n\in\mathsf{N};\pi,\overline{x},z}$$

where $\pi \in \Pi_{\delta(n)}$, $\overline{x} \in (\{0,1\}^*)^m$, and $z \in \{0,1\}^*$. We call δ the delay parameter *of ρ, and also say that ρ* securely realizes $\mathcal{F}$ under concurrent general composition with Π_δ in the timing model with ϵ.

As we have discussed in the Introduction, the timing model relies on two assumptions: the clock-drift ϵ and the maximum network latency Δ. However, the security of a protocol relies solely on the seemingly weaker assumption regarding clock drift. Therefore, our above definition refers to the clock drift, but makes no mention of the network latency. Rather, the latency assumption is only used in order to guarantee *non-triviality.*

Non-trivial protocols in the timing model. Loosely speaking, a protocol is non-trivial if the honest parties are guaranteed to receive their outputs (according to the functionality definition) in executions where the adversary is "well-behaved". More specifically, in the context of the timing model, a protocol is non-trivial for Δ and ϵ if in each execution in which the adversary is ϵ-drift preserving, delivers all messages in time at most Δ, and does not corrupt any party, none of the parties output time-out or abort.

Definition 3 (non-triviality): *We say that a protocol ρ is* non-trivial under timing assumptions (Δ, ϵ) *if in any execution of ρ where:*

1. *The real adversary $\mathcal{A}$ has not corrupted any of the participating parties, and*
2. *The real adversary $\mathcal{A}$ is ϵ-drift preserving and delivers all the messages of ρ within Δ time units (according to the clocks of all the parties),*

it holds that all parties receive output that does not equal time-out *or* abort.

Notice that item (2) in Definition 3 refers to delivery within Δ time units according to the clocks of *all parties.* This means that Δ is an upper bound on the latency with respect to all local clocks (and not with respect to some specific clock).

Modelling delays and time-outs. As we have discussed in the Introduction, our secure protocols utilize the clocks by introducing delay and time-out instructions. Such instructions can be carried out in our model as follows:

1. *Delay instructions:* If a party P_i is instructed to delay sending a message x by c time units, then it chooses a random identifier *delay-id* and writes (x, *delay-id*, c, *time*) on its work tape, where *time* is the current contents of its clock tape. It then writes (delay,*delay-id*,c) on its outgoing communication tape concluding the activation. Upon receiving a message (send,*delay-id*) from the adversary in a future activation, party P_i first checks that c units have passed

according to its clock (i.e., that the current contents of its clock is at least $time + c$, where c and *time* are the values in the tuple indexed by *delay-id*). If not, then it halts this activation. If yes, then it writes the delayed message x on its outgoing communication tape, concluding the activation. (We note that our decision to write the length c of the delay on the outgoing communication tape is arbitrary and makes no difference to our result.)

2. *Time-out instructions:* If a party P_i (or an ITMC that it runs as a subprotocol) has an instruction to time-out if it doesn't receive a specific message within c time units from the present time, then P_i writes the current contents of its clock tape on its work tape. Then, when it receives the specific message, it outputs time-out if the current contents of its clock tape is greater than the previously recorded value plus c.

Discussion – local computation time. In our definitions, we have included a local clock on machines and use this to measure the time that it takes for messages to be sent and received over the network. A more general model would also include issues such as the time that it takes for local computation. The focus of this paper is a secure protocol that utilizes timing assumptions, and not the issue of modelling time in its most general fashion. Our model therefore assumes that local computation is immediate (this can be seen because the adversary is not activated while local computations take place and so cannot update the clocks). One approach for generalizing the model is to have the adversary be activated after every single step of the transition function of an ITMC. We leave these questions of modelling for future work.

2.3 Security Under Concurrent Self Composition in the Timing Model

In the setting of concurrent self composition, a secure protocol is run many times concurrently, but it is the only protocol running in the network. This setting is a special case of the setting of concurrent general composition, where the "arbitrary" protocol π contains calls to the ideal functionality, and nothing else (and thus is no longer arbitrary). We let λ_δ denote the set of δ-delayed protocols that contain only calls to ideal functionalities (and in particular have no standard messages). We have the following definition, that basically states that a protocol is secure under concurrent self composition if it is secure under concurrent general composition with the set of protocols λ_δ:

Definition 4 (security under concurrent self composition in the timing model): *Let ρ be a polynomial-time protocol and let $\mathcal{F}$ be an ideal functionality. Then, ρ* securely realizes $\mathcal{F}$ under concurrent self composition in the timing model with ϵ *if there exists a polynomial $\delta(\cdot)$ such that for every ϵ-drift preserving non-uniform polynomial-time real-model adversary $\mathcal{A}$, there exists an ϵ-drift preserving non-uniform probabilistic polynomial-time hybrid-model adversary $\mathcal{S}$ such that for every $I \subseteq [m]$:*

$$\left\{\mathrm{HYBRID}^{\mathcal{F},\epsilon}_{\pi,\mathcal{S},I}(n,\overline{x},z)\right\}_{n\in\mathsf{N};\pi,\overline{x},z} \stackrel{\mathrm{c}}{\equiv} \left\{\mathrm{REAL}^{\epsilon}_{\pi^\rho,\mathcal{A},I}(n,\overline{x},z)\right\}_{n\in\mathsf{N};\pi,\overline{x},z}$$

where $\pi \in \lambda_{\delta(n)}$, $\overline{x} \in (\{0,1\}^*)^m$, and $z \in \{0,1\}^*$.

We remark that Definition 4 is formulated in a rather non-natural way, because in this way it is clearly a special case of Definition 2 (the only difference is that now the set of protocols π is much more restricted). A more natural definition is to define an ideal model that consists of many calls to the trusted party computing the ideal functionality (and nothing else), and then compare the output distribution of the real model to this ideal model. In any case, these formulations are equivalent because the hybrid model with a protocol from λ_δ is exactly the same as this ideal model.

2.4 Tools

Our protocol uses a number of different tools and primitives. In this section, we briefly describe these tools and provide references to full definitions.

Witness indistinguishable and witness hiding proofs [23]. We consider the interaction of a probabilistic polynomial-time verifier with a probabilistic polynomial-time prover who is given an auxiliary input (typically, an NP-witness) in order to carry out the proof. Such an interactive proof is witness indistinguishable if interactions in which the prover uses different "legitimate" auxiliary-inputs are computationally indistinguishable from each other [23]. Recall that any zero-knowledge proof system is also witness indistinguishable. Furthermore, witness indistinguishable proofs remain witness indistinguishable under concurrent composition. Witness hiding proofs have the property that a verifier cannot obtain a witness from its interaction with the prover. For example, if a prover proves that it knows the preimage of some one-way function using a witness-hiding proof, then the interaction will not help any probabilistic polynomial-time verifier to compute a preimage. Witness hiding proofs can be constructed from witness indistinguishable proofs by considering "double statements" with independent witnesses, of the form "I know the preimage of one of v_1 and v_2" [23]. See [24, Section 4.6] for a full treatment of witness indistinguishable and witness hiding proofs.

Strong proofs of knowledge [24]. A proof of knowledge [30, 5] is an interactive proof which convinces a verifier that the prover "knows" a witness to a certain statement. This is in contrast to a regular interactive proof, where the verifier is just convinced of the validity of the statement. The concept of "knowledge" for machines is formalized by saying that if a prover can convince the verifier, then there exists an efficient procedure that can "extract" a witness from this prover (thus the prover knows a witness because it can run the extraction procedure on itself). More formally, a proof of knowledge has the property that for every machine P^* there exists a *knowledge extractor* K, such that if P^* convinces V with probability p, then K "extracts" a valid witness from the prover P^* with probability that is negligibly close to p. A strong proof of knowledge, as defined by Goldreich [24, Sec. 4.7.6], is a proof of knowledge where the knowledge extractor runs in *strict* polynomial-time and fulfills the following more stringent requirement: There exists a negligible function $\mu(n)$ such that if a given prover convinces the honest verifier to accept with probability greater than $\mu(n)$, then the knowledge extractor succeeds in obtaining a witness with probability at least $1 - \mu(n)$. See [24, Sec. 4.7.6] for a full treatment.

We remark that there exist witness indistinguishable strong proofs of knowledge with any *super-constant* number of rounds. (The construction of [24] uses a super-logarithmic number of sequential executions of the 3-round zero-knowledge proof for Hamiltonicity [8]. However, using the same ideas, it can be shown that by running $\log n$ parallel executions of the proof of Hamiltonicity, any super-constant number of sequential repetitions is actually enough. We can therefore reduce this to any super-constant number of rounds $\alpha(n) = \omega(1)$.) We also remark that it has been shown that under exponential hardness assumptions, there *do not exist* witness indistinguishable *strong* proofs of knowledge with a constant number of rounds, even using non-black-box techniques [3].

3 Constructing Secure Protocols in the Timing Model

In this section we prove our main positive results, which consists of proving Theorem 1.2 (and obtaining Theorem 1.1 as a corollary). We begin by formally restating Theorem 1.2.

Theorem 5 (Theorem 1.2 – restated): *Assume that there exist enhanced trapdoor permutations, and let Δ and ϵ be constants where $1 \leq \epsilon \leq \sqrt[3]{1.5}$.*[12] *Then, for every probabilistic polynomial-time multiparty functionality $\mathcal{F}$, there exists a protocol ρ that securely realizes $\mathcal{F}$ under concurrent general composition with delays in the timing model with ϵ, and in the presence of static malicious adversaries. Furthermore, ρ is non-trivial under timing assumptions (Δ, ϵ). The delay parameter of ρ is $\delta(n) = \omega(1) \cdot \Delta \cdot \epsilon$.*

The majority of the proof of Theorem 5 involves showing how to securely realize the "common random string" (CRS) functionality. We begin by formally defining the CRS functionality in Section 3.1. Next, in Section 3.2 we show that in order to prove Theorem 5, it suffices to securely realize the CRS functionality. Finally, Sections 3.3 to 3.6 are devoted to showing how to securely realize the CRS functionality.

3.1 The CRS Functionality

We now formally define the common random string functionality, denoted $\mathcal{F}_{\text{CRS}}$. Intuitively, the functionality simply chooses a random string and sends it to all parties. Any party can send the functionality a crsgen request. Once the functionality receives such a request, it generates a random string R_{CRS} and sends it to the adversary and all the parties. Recall that the adversary controls the delivery of messages between $\mathcal{F}_{\text{CRS}}$ and the parties; therefore, the fact that $\mathcal{F}_{\text{CRS}}$ sends the output does not mean that the parties receive it immediately (or even that they will ever receive it). Recall also that parties may receive time-out and abort outputs, as defined in Section 2.2. A formal description of $\mathcal{F}_{\text{CRS}}$ appears in Figure 2.

Functionality $\mathcal{F}_{\text{CRS}}$

Let n be the security parameter and let $p(\cdot)$ be a fixed polynomial.[13] Let $P_1, \ldots, P_m$ be the set of all parties, and let $\mathcal{S}$ be the adversary. The functionality $\mathcal{F}_{\text{CRS}}$ proceeds as follows:

> Upon receiving a message $(\mathsf{crsgen}, sid, \{P_{i_1}, \ldots, P_{i_k}\})$, choose a uniformly distributed string $R_{\text{CRS}} \in_R \{0,1\}^{p(n)}$, send $(\mathsf{crsgen}, sid, \{P_{i_1}, \ldots, P_{i_k}\}, R_{\text{CRS}})$ to $\mathcal{S}$ and to all parties $P_{i_1}, \ldots, P_{i_k}$, and halt.

Figure 2: The ideal multiparty $\mathcal{F}_{\text{CRS}}$ functionality

We note that the $\mathcal{F}_{\text{CRS}}$ functionality sends only *uniformly distributed* strings (in contrast to some prior definitions which allowed any efficiently samplable distribution). This is crucial for our implementation since we use a coin-tossing protocol.

3.2 Reducing the Problem to Realizing the CRS Functionality

In the $\mathcal{F}_{\text{CRS}}$-hybrid model, all parties are given access to the $\mathcal{F}_{\text{CRS}}$ functionality. As we have mentioned, it follows from [14] and from the composition theorem given in [11], that if the $\mathcal{F}_{\text{CRS}}$ functionality can be securely realized under concurrent general composition then *any* functionality $\mathcal{F}$ can be securely realized under concurrent general composition.[14]

[12] This limitation on ϵ is needed in our proof of security.

[13] $\mathcal{F}_{\text{CRS}}$ is parameterized by a polynomial $p(\cdot)$ that determines the length of the common random string generated. If desired, this can be included as input with almost no difference to the protocol (the only necessary addition is for the parties to negotiate the value of $p(\cdot)$ at the onset).

[14] Actually, the result in [14] holds only for the class of "well-formed" functionalities. However, in the case of static adversaries, this only limits the functionalities to those that are "unaware" of which parties are corrupted and which

Let $\mathcal{F}$ be any multiparty functionality. Our protocol for securely realizing $\mathcal{F}$ is constructed as follows: We first construct a protocol ρ that securely realizes the $\mathcal{F}_{\text{CRS}}$ functionality *in the timing model* under concurrent general composition with the class of protocols Π_δ. We then use [14] to obtain a (timing-free) protocol σ that securely realizes $\mathcal{F}$ in the $\mathcal{F}_{\text{CRS}}$-hybrid model. Since our protocol ρ is only secure when all other protocols are δ-delayed, and we wish to run it together with σ, the protocol σ is modified to include delays. That is, let $\hat{\sigma}$ be the delayed version of σ where the sending of each message is delayed by δ local time units (since σ is timing-free, delaying its messages makes no difference to its security). Noticing that ρ is secure under concurrent general composition with Π_δ and that $\hat{\sigma} \in \Pi_\delta$, we conclude that the composed real protocol $\hat{\sigma}^\rho$ securely realizes $\mathcal{F}$ under concurrent general composition with delays, in the timing model.

The fact that we can combine the protocol of [14] with our protocol for securely computing the $\mathcal{F}_{\text{CRS}}$ functionality is intuitively clear. We prove this fact below, but remark that the proofs are straightforward, and just involves justifying a few technical points related to the above informal reasoning. First, the hybrid model of [14] is not quite the same as the one defined here, since in our definition the adversary can send time-out instructions to the trusted party. Second, we do not securely realize the $\mathcal{F}_{\text{CRS}}$ functionality under (standard) concurrent general composition; rather, we securely realize it under concurrent general composition *with delays* (i.e., with the class of protocols Π_δ for some δ). Despite the above differences, we show that our construction is secure. We note one technicality that must be dealt with. According to our definition, a time-out can only be issued before any outputs are obtained. Thus, a policy must be determined regarding what $\hat{\sigma}$ should do in case $\mathcal{F}_{\text{CRS}}$ returns time-out (note that in the timing-free model of σ, functionality $\mathcal{F}_{\text{CRS}}$ never returns time-out; this issue only arises in the timing model). There are two possible approaches here. First, we could define that the output of σ in this case is abort (and not time-out). The drawback of this approach is that we lose the advantage of a time-out that enables parties to restart the execution. The second approach is to rely on the fact that the protocol σ from [14] can be written so that there is only a single call to $\mathcal{F}_{\text{CRS}}$, and this takes place at the very onset of the execution (before any outputs are produced). We prefer this latter approach as it results in a protocol that can be restarted in the case that high network latency is the only reason that the protocol did not terminate successfully.

We first claim that delays and time-out instructions can be added to a secure timing-free protocol σ, resulting in a δ-delayed protocol that is secure in the timing model. As discussed above, we assume that σ contains only a single call to $\mathcal{F}_{\text{CRS}}$, and this takes place at the onset of the execution.

Claim 3.1 *Let $\mathcal{F}$ be a functionality and let σ be a timing-free protocol that contains a single call to $\mathcal{F}_{\text{CRS}}$ at the onset of the protocol and securely realizes $\mathcal{F}$ under concurrent general composition in the timing-free $\mathcal{F}_{\text{CRS}}$-hybrid model of* [14]. *Then there exists a δ-delayed protocol $\hat{\sigma}$ that securely realizes $\mathcal{F}$ under concurrent general composition in the timed $\mathcal{F}_{\text{CRS}}$-hybrid model, as defined in Section 2.2.*

Proof Sketch: We construct the protocol $\hat{\sigma}$ as follows. First, if an honest party receives back time-out from the ideal $\mathcal{F}_{\text{CRS}}$ functionality, then it outputs time-out and halts. Second, every message in $\hat{\sigma}$ is preceded by an instruction to delay sending the message by $\delta(n)$ units of time (where n is the security parameter). As we have discussed, the delays inserted into $\hat{\sigma}$ make no difference to its security. This is due to the fact that σ is a timing-free protocol. Therefore, it is secure for *every* possible scheduling, including the scheduling where every honest party's message is (additionally) delayed by δ units of time. Regarding the possibility of obtaining time-out, there are two cases:

---

are honest. Since in our definition of the computational model the ideal functionality is not given this information, it follows that *all efficient functionalities* can be securely realized.

1. *Case 1 – the honest parties all receive* time-out *from $\mathcal{F}_{\text{CRS}}$:* In this case, the ideal-model adversary/simulator for $\hat{\sigma}$ sends a time-out instruction to the trusted party computing the functionality $\mathcal{F}$. Since $\mathcal{F}_{\text{CRS}}$ is called at the onset of the execution, and thus before any outputs are generated, this implies that the time-out instruction is "accepted" by the trusted party. Therefore, all honest parties output time-out in the ideal execution. Since, by the definition of $\hat{\sigma}$, all honest parties also output time-out in the execution of $\hat{\sigma}$, the simulation is correct.

2. *Case 2 – the honest parties do not receive* time-out *from $\mathcal{F}_{\text{CRS}}$:* In this case, the execution of $\hat{\sigma}$ is exactly the same as σ. Therefore, the simulator for $\hat{\sigma}$ follows exactly the same strategy as the simulator for σ.

Note that the simulator for $\hat{\sigma}$ knows whether it is in case 1 or case 2, and so can efficiently implement the above strategy. This completes the proof sketch. ■

Remark: *The above reference to the* "$\mathcal{F}_{\text{CRS}}$-hybrid model of [14]" *versus the* "timed $\mathcal{F}_{\text{CRS}}$-hybrid model" *is somewhat confusing. From here on, whenever we refer to the "$\mathcal{F}_{\text{CRS}}$-hybrid model", we mean the timed model as defined in Section 2.2.*

So far, we have shown that delaying σ and adding time-out instructions does not make any difference with respect to its security in $\mathcal{F}_{\text{CRS}}$-hybrid model. The next claim shows that it suffices to prove that the $\mathcal{F}_{\text{CRS}}$ functionality can securely realized under concurrent general composition with delays.

Claim 3.2 *Let $\hat{\sigma}$ be a δ-delayed protocol that securely realizes $\mathcal{F}$ under concurrent general composition in the $\mathcal{F}_{\text{CRS}}$-hybrid model, and let ρ be a real-model protocol that securely realizes $\mathcal{F}_{\text{CRS}}$ under concurrent general composition with Π_δ in the timing model with ϵ. Then the real protocol $\hat{\sigma}^\rho$ securely realizes $\mathcal{F}$ under concurrent general composition with Π_δ in the timing model with ϵ.*

Proof Sketch: Let $\hat{\sigma}$ be as in the claim statement, and let $\pi \in \Pi_\delta$ be an arbitrary δ-delayed protocol that contains ideal calls to $\mathcal{F}$. Consider now a real execution of π with $\hat{\sigma}^\rho$, denoted $\pi^{\hat{\sigma}^\rho}$. Now, we can write $\pi^{\hat{\sigma}^\rho} = (\pi^{\hat{\sigma}})^\rho$, and therefore consider an execution of $\pi^{\hat{\sigma}}$ with ρ. Since $\pi^{\hat{\sigma}}$ is a δ-delayed protocol, we have that ρ securely realizes $\mathcal{F}_{\text{CRS}}$ when run together with $\pi^{\hat{\sigma}}$. That is, for every real adversary $\mathcal{A}$ running $(\pi^{\hat{\sigma}})^\rho$, there exists an adversary $\mathcal{H}$ running $\pi^{\hat{\sigma}}$ in the $\mathcal{F}_{\text{CRS}}$-hybrid model, such that

$$\left\{\text{HYBRID}^{\mathcal{F}_{\text{CRS}},\epsilon}_{\pi^{\hat{\sigma}},\mathcal{H},I}(n,\overline{x},z)\right\}_{n\in\mathsf{N};\overline{x},z} \stackrel{\text{c}}{\equiv} \left\{\text{REAL}^{\epsilon}_{(\pi^{\hat{\sigma}})^\rho,\mathcal{A},I}(n,\overline{x},z)\right\}_{n\in\mathsf{N};\overline{x},z}$$

Applying the fact that $\hat{\sigma}$ securely realizes $\mathcal{F}$ under concurrent general composition in the $\mathcal{F}_{\text{CRS}}$-hybrid model, we have that there exists an ideal-model adversary $\mathcal{S}$ such that

$$\left\{\text{HYBRID}^{\mathcal{F},\epsilon}_{\pi,\mathcal{S},I}(n,\overline{x},z)\right\}_{n\in\mathsf{N};\overline{x},z} \stackrel{\text{c}}{\equiv} \left\{\text{HYBRID}^{\mathcal{F}_{\text{CRS}},\epsilon}_{\pi^{\hat{\sigma}},\mathcal{H},I}(n,\overline{x},z)\right\}_{n\in\mathsf{N};\overline{x},z}$$

Combining the above, we have that $\hat{\sigma}^\rho$ securely realizes $\mathcal{F}$ under concurrent general composition with Π_δ in the timing model with ϵ, as required. ■

Under the assumption that enhanced trapdoor permutations exist, the result of [14] provides us with a protocol σ as required in Claim 3.1. Thus, using Claim 3.2 we conclude that in order to prove Theorem 5 it suffices to prove that there exists a protocol ρ that securely realizes $\mathcal{F}_{\text{CRS}}$ under concurrent general composition with Π_δ in the timing model with ϵ. The rest of this section is devoted to this task of securely realizing $\mathcal{F}_{\text{CRS}}$. Before proceeding to this task, we discuss the utility of our results for obtaining security in real networks.

Using Theorem 5 to obtain secure protocols. Formally, Theorem 5 states that it is possible to construct a protocol that securely realizes any functionality $\mathcal{F}$ under concurrent general composition with Π_δ. However, the protocol that is obtained is *not* δ-delayed. This is because the secure protocol for computing $\mathcal{F}$ is $\hat{\sigma}^\rho$, and ρ is not δ-delayed. In fact, as will be seen below (where we show how to construct ρ), it is crucial that ρ *not* be δ-delayed. Therefore, two different protocols constructed according to Claims 3.1 and 3.2 seemingly cannot be run together (each one is only secure if it is run together with δ-delayed protocols, but neither is δ-delayed).

Despite the above, a closer look at Claims 3.1 and 3.2 yields that any number of different protocols can be run together, while maintaining security. Specifically, consider any set of protocols, some of which are secure in the plain model and some of which are secure in the $\mathcal{F}_{\text{CRS}}$-hybrid model (where security here relates to security under concurrent general composition without timing). Then, it is possible to run all of these protocols in the *plain timing model* by

1. Modifying all of the protocols (adding delays and timeouts, as in the proof of Claim 3.1), and
2. Having all of the protocols that are secure in the $\mathcal{F}_{\text{CRS}}$-hybrid model use the *same* protocol ρ that securely realizes $\mathcal{F}_{\text{CRS}}$ under concurrent general composition with delays (as in the proof of Claim 3.2).

Thus, the results of this paper can be used to construct many protocols that are simultaneously secure under concurrent general composition with delays.

Obtaining a delayed network. The protocols that we obtain are secure as long as all the protocols in the network are δ-delayed (for δ as defined in Theorem 5). As we have described above, any timing-free protocol that is being run in the network can be modified into a δ-delayed protocol, without affecting its behavior. Indeed, the "only" price is a loss of efficiency. However, if parties are running *other* protocols that use time in the network, and they are not already δ-delayed, converting them into δ-delayed protocols is no longer so trivial. In particular, adding delays before every message may have adverse affects (like causing the protocol to always time-out). Thus, the delays and timeouts must be considered together.

More specifically, consider the case that π is an arbitrary protocol, that possibly refers to its clock but is not δ-delayed. Then, in order to transform it into a protocol that is δ-delayed (so that we can run our secure protocol concurrently with it), we need to add delays in a way that will not affect the behavior of π. We will assume that π already has a delay of a *single* time unit for every message. Then, we can modify π into $\hat{\pi}$ as follows: Whenever π instructs a party to read its clock tape, the party running $\hat{\pi}$ reads the clock tape and uses the result *divided* by δ. More exactly, if the contents of the clock tape is c, then the value $\lfloor c/\delta \rfloor$ is used instead.

Note first that $\hat{\pi}$ is a δ-delayed protocol. This holds by the assumption that every message in π is delayed by a single unit of time, and by the fact that a clock must proceed δ units before a single step forward is registered in π.[15] Next, we claim that the output distributions of $\hat{\pi}$ and π are identical. This is due to the fact that time units are relative and so having all parties slow their clocks by the same amount of time makes no difference. (Note that the clock drift also remains ϵ because all values in Eq. (1) are multiplied by the same value δ, and so it cancels out.) We therefore conclude that $\hat{\pi}$ is a δ-delayed protocol and that the output distribution of a real execution of $\hat{\pi}^{\hat{\sigma}^\rho}$ (as in Claim 3.2) is computationally indistinguishable to the output distribution of an execution of the original π in the $\mathcal{F}$-hybrid model, as desired.

[15] From this it follows that if the original π already has delays of size $\tau > 1$, then it suffices to change the clock value from c to $\lfloor \tau c/\delta \rfloor$, and $\hat{\pi}$ will be δ-delayed.

3.3 Overview of the Protocol for $\mathcal{F}_{\text{CRS}}$ and its Security Analysis

Before proceeding to describe the actual protocol for securely realizing the $\mathcal{F}_{\text{CRS}}$ functionality, we provide a high-level overview of the construction. The basic structure of the protocol is an extension of the two-party coin-tossing protocol of [32] (which is in turn an extension of Blum's protocol [7]). In this protocol, each party first commits to a randomly chosen value and provides a zero-knowledge proof of knowledge of the committed value. In the next phase of the protocol, each party reveals its committed value, without actually decommitting, and provides a zero-knowledge proof that the revealed value is indeed the one that was committed to. The idea behind this construction is that due to the soundness of the proofs, a corrupted party has no choice but to reveal the value that it committed to in the first phase. Thus, the binding property of the commitment scheme is preserved. Intuitively, this means that the adversary cannot bias the outcome of the coin-tossing protocol, because it is bound to the corrupted parties' committed values before it sees the honest parties' committed values. However, in order to prove the security of the coin-tossing protocol according to the simulation paradigm, it is necessary to construct a simulator that can force the outcome of the protocol to be the exact string R_{CRS} that is generated by the ideal functionality $\mathcal{F}_{\text{CRS}}$. The zero-knowledge proofs of knowledge are included in order to facilitate such a simulation. Specifically, it is true that the adversary must reveal the correctly committed value due to the soundness of the proofs. However, the simulator can run the simulator for the zero-knowledge protocol, and can effectively cheat. Thus, the simulator can reveal any value that it wishes and is not bound by the commitment scheme (note that decommitment never actually takes place; rather the committed value is revealed and the zero-knowledge proof is used to determine that it is correct). This observation is used in the following way. In the first phase of the protocol, the simulator commits to random values for the honest parties, and extracts all of the values committed to by the corrupted parties (it does this by running the knowledge extractor on the proofs of knowledge of the committed values). Next, given the corrupted parties' values, it chooses *new* random strings for the honest parties so that the XOR of the extracted corrupted parties' values and the new honest parties values equals R_{CRS} exactly. Finally, the simulator reveals the new (fake) honest party values and simulates the zero-knowledge proofs claiming that the revealed values are indeed the committed ones (which they are not). The adversary's view in this simulation is indistinguishable from in a real protocol execution due to the hiding property of the commitments and the zero-knowledge property of the proofs.

A crucial point in the above security argument is that the proofs of knowledge must be run *independently of each other* (in order to ensure that the adversary does not "copy" a proof from an honest party). The same holds also for the zero-knowledge proofs of consistency in phase 2. (Here the reason is slightly different. During simulation, the simulator actually "cheats" by proving an incorrect theorem. We need to ensure that the adversary cannot use the cheating of the simulator in order to cheat itself.) In the stand-alone case, this independence is achieved by simply running the proofs *sequentially*. Technically, this enables the rewinding of the proofs of knowledge provided by the adversary (for extraction in the first phase) and the rewinding of the zero-knowledge proofs verified by the adversary (for simulation in the second phase) without overlapping and therefore without interfering with any of the other proofs. In our case, however, we must achieve security under *concurrent composition*. Therefore, it is not possible to enforce any specific scheduling that will ensure independence between the proofs (or that they don't overlap during rewinding).

As a first step towards solving this problem (and as a solution to another problem), we limit the rewinding stages to be "early" on in the protocol. In particular, rewinding takes place only before the decommitment values are revealed and so before the common reference string can be learned by the adversary. This is a necessary step because *time-out* instructions are crucial for enabling "proper rewinding," and as we have discussed in Subsection 2.2, a time-out must only occur before the adversary can learn the output. We achieve this by using the specific zero-knowledge proofs of knowledge of Feige and Shamir [22]. Loosely speaking, the Feige-Shamir proof system consists of two witness-indistinguishable proofs of knowledge (WIPOKs, for short); first the verifier proves that it knows one of two independent secrets; next, the prover proves either that it knows one of the verifier's secrets or that it knows the real witness. The soundness of this protocol follows from the fact that a WIPOK for statements with multiple independent witnesses is *witness hiding*. Therefore, the prover could not have obtained the secret from the first WIPOK and must use the real witness in the second WIPOK. The zero-knowledge property is demonstrated by first extracting a secret from the verifier in the first stage, and then proving the second WIPOK using knowledge of this secret. Note that the second stage of the simulation requires no rewinding, and that this is the only part of the proof that depends on the statement being proved.

To be more precise, our protocol consists of three phases. In Phase 1, each player runs a WIPOK that it knows one of two independent secrets (this is the first WIPOK of the Feige-Shamir proof system). Then, in Phase 2, each player commits to a random value, and runs a single WIPOK that it either knows the value that it committed to or that it knows one of the secrets of the verifier (completing the Feige-Shamir proof that was initiated in Phase 1). Thus, by the end of Phase 2, each player has committed to some value and has proved in zero-knowledge to each of the other players that it knows the value that it committed to. Notice that phases 1 and 2 correspond to the first part of the coin-tossing protocol of [32]. The coin-tossing protocol is then completed in Phase 3 where each player reveals the value that it committed to in Phase 2 (without decommitting), and proves that it is the correct value. This proof is a single WIPOK that it either knows the decommitment information that corresponds to this value or that it knows one of the secrets of the verifier. Once again, combining this WIPOK with that of phase 1, we obtain a Feige-Shamir proof. Thus, both the proofs of Phase 2 and of Phase 3 (which consist of only a single WIPOK) are actually zero-knowledge, as required by the coin-tossing protocol. An important property of this protocol is that the only rewinding needed is **(a)** to extract the secrets from the first Feige-Shamir WIPOK in Phase 1 (enabling simulation later), and **(b)** to extract the committed value from the adversary in Phase 2. This implies that all rewinding takes place before Phase 3, which is where the committed values are revealed. Furthermore, all rewinding is actually for the purpose of *extraction* only.[16]

Until now, we have focused on how to limit the rewinding to the early stage of the protocol, and to witness extraction only. However, a far more crucial issue is how we carry out this extraction (i.e., rewinding) in the concurrent setting. It is here that we use the timing assumptions, via *time-out* and *delay* instructions, in an inherent way. Informally speaking, there are two issues that must be dealt with when considering concurrent composition here: **(a)** the WIPOK protocols must self-compose (i.e., we should be able to extract and enforce independence when many WIPOK executions take place concurrently), and **(b)** the WIPOK executions should remain secure (again, enabling extraction and independence) when run concurrently with an arbitrary δ-delayed protocol π. We separately explain, at an intuitive level, the security of the WIPOKs under these two types of composition.[17]

Composition with arbitrary δ-delayed protocols. The main problem that arises when running a secure protocol ρ concurrently to an arbitrary other protocol π, is that the adversary may

[16]This strategy simplifies the proof of security, because it turns out to be "much easier" to extract than simulate. This is especially true because we use *strong* proofs of knowledge, rather than ordinary ones; see below.

[17]We caution the reader that the formal proof of security does not separate out in this fashion.

be able to generate some dependence between π and the secure protocol ρ. (For example, π messages may have the same format as ρ messages and so an adversary can just forward messages from one protocol to another). On a more technical level, the proof of security works by constructing a hybrid-model simulator who runs π externally, while internally simulating ρ. Now, if the simulator needs to rewind ρ, it cannot proceed with π because the π-messages are sent to external parties and so cannot be retracted. Thus, it is crucial that while rewinding the WIPOKs in order to extract, the simulator does not need to send any π-messages externally. By setting δ to be the amount of time that it takes to complete a WIPOK, we have that the rewinding spans only over this amount of time. Thus, if π is δ-delayed, we have that no new π messages need to be dealt with during rewinding. We note that the length of the WIPOK is forced to be at most δ by timing-out if it takes too long. Thus, as described in the introduction, the needed effect is obtained by combining time-out and delay instructions together.

Concurrent self-composition. The main concern that arises here is that of *independence*. That is, when many WIPOK executions are run concurrently, the adversary can carry out a man-in-the-middle (or mauling) attack, in which it takes messages received in one execution and forwards (or modifies) them in another execution. Such a strategy enables it to "copy" a proof provided by an honest party, and contradicts the requirement of independence.

In order to prevent such an attack, it suffices to ensure that no (relevant) WIPOK in one session occurs concurrently with any (relevant) WIPOK in another session. However, in a setting where we cannot coordinate between multiple sessions of the protocol, this is impossible. We therefore have the parties prove *many* WIPOKs in every session, according to a carefully designed *scheduling strategy*. Our scheduling is based on the Chor-Rabin scheduling [16], with modifications necessary due to the fact that we work in the concurrent setting with timing (whereas they worked in the fully synchronous model). Our scheduling has the property that for every two sessions, there exists at least one WIPOK in the first session that does not overlap with *any* of the WIPOKs of the second session. We call a scheduling that has this property a *pairwise-disjoint scheduling*, and discuss it further in Sections 3.4 and 4.[18] We note that we make essential use of the timing assumptions in order to construct this scheduling.

Use of strong proofs of knowledge. We actually use *strong* proofs of knowledge in our protocol, rather than ordinary ones. (Recall that such a proof has the property that if the prover convinces the verifier with non-negligible probability, then the extractor obtains a witness with overwhelming probability. Furthermore, the running-time of the extractor is independent of the probability that the prover convinces the verifier.) We do not know if this is essential, but we also do not know how to prove the security of our protocol otherwise.[19] Loosely speaking, we use strong proofs of knowledge in order to obtain the following effect. Our simulation strategy works by running in a "straight-line simulation mode" until a WIPOK is reached. When the beginning of such a proof is reached, we leave this mode and enter an "extraction mode," where rewinding takes place. We then run the extractor, while internally simulating the *future messages* (that is, the strategy is actually one of look-ahead, rather than rewinding back). Now, if a strong proof of knowledge is used, then after the extractor terminates, we are guaranteed that the following holds: either the extractor succeeded in obtaining a witness, or if it did not, we know that the prover will only succeed in convincing the verifier with negligible probability (in which case, we will not need the witness because the session

[18]We remark that the Chor-Rabin scheduling was also used by [18] in a concurrent-type setting in order to achieve non-malleable commitments (without timing assumptions). Our setting differs in that we have many executions (and in this way it is "harder"), but we also utilize timing assumptions (and in this way it is "easier").

[19]This is the first work that we are aware of that utilizes strong proofs of knowledge in an essential way, rather than just in order to simplify the construction and proof.

will be aborted). Thus, there is no uncertainty (of course, beyond the negligible probability that the above will not hold). In contrast, in a regular proof of knowledge, such a look-ahead would fail because even if the extractor did not obtain a witness, it may still happen that the prover will convince the verifier. Thus, we would need to use a "rewind back" strategy where after the prover convinces the verifier, we would go back and obtain the witness. This type of strategy seems to be more difficult when dealing with the external π-messages (although, as mentioned above, we do not know whether or not the difficulties are inherent).

3.4 Scheduling

Our goal is to construct a protocol that securely realizes the $\mathcal{F}_{\text{CRS}}$ functionality in the timing model, in a general multiparty network where sessions are being executed concurrently. One of the major risks in this concurrent setting is related to the notion of *malleability*. Loosely speaking, this refers to an adversary who interleaves different executions of the protocol, and chooses its messages in one execution based on messages that it receives in the other executions. Consider, for example, many interleaved executions of a (regular, stand-alone) zero-knowledge proof of knowledge. In this setting, even if an adversary succeeds in convincing a verifier that it knows some secret s, it does not necessarily mean that the adversary actually knows s. Rather, it may be the case that there is some other party that is concurrently proving to the adversary that it knows the same secret s, and the adversary is simply relaying the messages between these two executions. Such a strategy is known as a "*man-in-the-middle*" attack. In order to construct secure protocols, it is necessary to prevent such attacks.

Our idea for preventing such mauling attacks is based on [16], who introduce a method for concurrently alternating and interleaving protocol executions in the *fully synchronous model*, while preserving independence. Loosely speaking, they construct an $O(\log n)$-round n-party protocol, in which each party (concurrently) carries out several zero-knowledge proofs sequentially, so that at least one of its proofs is "independent" from the proofs of the other parties.

More specifically, they associate with each party P_i a unique identifier $id^i \in \{0,1\}^{2m}$ that contains exactly m ones and m zeros (since the number of parties is polynomial in n, the value m can be set to be $O(\log n)$). The protocol consists of $2m$ phases, where in each phase some of the parties play the role of prover (and all parties verify). A party plays the prover in a zero knowledge proof in phase k if and only if the k^{th} bit of its identifier is 1 (i.e., party P_i will play the prover in phase k if and only if $(id^i)_k = 1$). In total, every party plays the prover role during half of the phases, and for every two parties P_i and P_j, there is at least one phase in which P_i acts as a prover while P_j acts only as a verifier, and vice versa. This follows from the fact that for every $i \neq j$, id^i and id^j are distinct and they both have the same number of ones and zeros. Therefore, there exist two distinct indices k and k' such that: **(a)** $(id^i)_k = 1$ and $(id^j)_k = 0$, and **(b)** $(id^i)_{k'} = 0$ and $(id^j)_{k'} = 1$. Thus, in phase k party P_i proves and party P_j only verifies, and in phase k' party P_j proves and party P_i only verifies. Intuitively, this prevents P_i from using P_j as an oracle for supplying his proofs. While this method seems to guarantee only pairwise independence, it actually achieves mutual independence. We show that a similar idea can be used to achieve independence in a concurrent setting, in the timing model.

To this end we define the notion of a *pairwise disjoint* scheduling. In Section 4 we show how to construct a pairwise disjoint scheduling in the timing model. In Section 3.5, we show how such a scheduling can be used to design a protocol that securely realizes the $\mathcal{F}_{\text{CRS}}$ functionality under concurrent general composition with delays, in the timing model.

Pairwise-disjoint scheduling. Consider one pre-specified protocol σ, which needs to be executed concurrently in many different sessions, where each session has a unique identifier. The aim of a pairwise-disjoint scheduling is to ensure that different concurrent executions of σ are somewhat "independent". Intuitively, the idea is to achieve independence by requiring the parties to act as follows: Instead of running a single execution of Protocol σ in a given session, the parties execute σ several times in that session according to some pre-specified "*pairwise-disjoint*" scheduling S. Loosely speaking, this scheduling ensures that when looking at any two distinct sessions (each containing at least one honest party), there exists at least one execution in *each* of the sessions that does not intersect (i.e., overlap) with *any* execution in the other session.

We define the notion of a *pairwise-disjoint scheduling algorithm* S that receives for input a protocol σ, a unique session identifier sid, and the network timing assumptions Δ and ϵ. The algorithm $S(\sigma, sid, \Delta, \epsilon)$ then outputs a schedule consisting of many executions of σ with the property that for every two distinct sessions sid and sid' there exists at least one execution in $S(\sigma, sid, \Delta, \epsilon)$ that does not overlap with any of the executions of $S(\sigma, sid', \Delta, \epsilon,)$ and vice versa. We stress a crucial point here. When considering many different sessions, it may be the case that *every* execution of σ in a session sid overlaps with some other execution of σ in some other session. However, it is guaranteed that for every session sid', there exists at least *one* execution of σ in session sid that does not overlap with *any* of the executions sid'. This type of pairwise disjointness suffices since in our proof the simulator simulates all the honest provers except for one chosen prover which will be an "external prover." It is only this "external prover" that cannot be rewound. Thus, it suffices to ensure that for each session there exists one execution which does not overlap with the proofs of the "external prover." This is exactly what a pairwise disjoint scheduling ensures.

In what follows, we formally define the syntax of a scheduling algorithm. We are only interested in schedules which are polynomial-time (i.e., the number of executions is polynomial-time in the security parameter n, and the delays are polynomial in Δ, ϵ and n), and in schedules which are non-trivial (where the parties output time-out only if the network delay is too long). We therefore incorporate these requirements directly into the definition.

Definition 6 (non-trivial scheduling algorithm): *A* non-trivial scheduling algorithm *is an algorithm S that receives for input a protocol σ, a session identifier sid, and a pair (Δ, ϵ), and outputs a polynomial-time schedule Σ consisting of many executions of σ together with* delay *and* time-out *instructions that is non-trivial (as defined in Definition 3).*

Before proceeding further, we define what it means for an execution of a protocol σ to *overlap* with another execution. Let σ_1 and σ_2 be two executions of Protocol σ, and let P_1 and P_2 be any two honest participants in σ_1 and σ_2 respectively. Then, σ_1 overlaps σ_2 *according to* P_1 *and* P_2 if P_1 sends a σ_1 message *after* P_2 has sent its first σ_2 message, but *before* P_2 sends its last σ_2 message. Notice that the notion of overlapping is defined with respect to a pair of parties. This is due to the fact that parties do not necessarily begin and conclude executions at the same time in an asynchronous network (and so σ_1 and σ_2 may not overlap according to some pairs, and may overlap according to others). We therefore always refer to overlapping *according to a specified pair of parties.*

We are now ready to define what it means for a schedule to be *pairwise-disjoint.*

Definition 7 (non-trivial pairwise-disjoint scheduling): *A non-trivial scheduling algorithm S is said to be* pairwise-disjoint *if on input $(\sigma, sid, \Delta, \epsilon)$ it outputs a schedule with the following property. Let $sid_1 \neq sid_2$ be any identifiers of the same length and assume that $\Sigma_1 = S(\sigma, sid_1, \Delta, \epsilon)$ and*

$\Sigma_2 = S(\sigma, sid_2, \Delta, \epsilon)$ *are run in a network with an* ϵ*-drift preserving adversary, such that both* sid_1 *and* sid_2 *have at least one honest participant each. Then for any two honest parties* P_1 *and* P_2 *in sessions* sid_1 *and* sid_2 *respectively, there exists an execution* σ_ℓ *in* Σ_2 *such that no execution of* σ *in* Σ_1 *overlaps with* σ_ℓ *according to* P_1 *and* P_2*.*

Note that if P_2 times-out session sid_2 before some execution σ_i in $S(\sigma, sid_1, \Delta, \epsilon)$ was initiated, then in particular σ_i does not overlap with any execution in $S(\sigma, sid_2, \Delta, \epsilon)$, according to P_1 and P_2. This fact will be used in the proof of Theorem 11 in Section 4.

In Section 4, we prove the following theorem which will be used in order to construct our protocol for securely realizing the $\mathcal{F}_{\text{CRS}}$ functionality.

Theorem 8 *There exists a non-trivial pairwise-disjoint scheduling algorithm for any protocol* σ*, any network delay* Δ*, any clock-drift* ϵ *such that* $1 \leq \epsilon \leq \sqrt[3]{1.5}$*, and any set of identifiers* $sid \in \{0,1\}^{\text{poly}(n)}$*.*

Before proceeding, we explain again (but in more detail) why *pairwise* disjointness suffices. In our protocol, we use a pairwise disjoint scheduling for WIPOKs. Then, at some stage in our proof of security of our protocol, we focus on a single session sid, and argue that the simulation (i.e., extraction from WIPOK proofs) in all sessions $sid' \neq sid$ can be carried out without rewinding during any WIPOK of session sid. This can be achieved because the pairwise disjointness property of the schedule guarantees that for every session sid', there exists at least one WIPOK in sid' that does not overlap with any WIPOK in sid. We can therefore extract from the non-overlapping WIPOK in sid' without rewinding any of the WIPOK proofs in sid. Since this is true for all sessions $sid' \neq sid$, we are able to simulate without rewinding any WIPOK proof in sid, as required.

3.5 The Protocol for $\mathcal{F}_{\text{CRS}}$

The protocol below refers to a one-way function f and a commitment scheme C. We denote by $C(r; s)$ a commitment to r using random coins s. For simplicity, the description of the protocol assumes that the commitment scheme is non-interactive. Such schemes are known to exist assuming the existence of 1–1 one-way functions. However, it is also possible to use the commitment scheme of [37] where the receiver first sends a single (random) message and then the committer sends its commitment. Importantly, the scheme of [37] assumes only the existence of one-way functions. Our protocol also uses a broadcast primitive. However, as shown in [29], in the case that output delivery is not guaranteed (as in our model here), broadcast that is secure under concurrent general composition can be easily implemented in a standard point-to-point network.

As was mentioned in Subsection 3.3, the protocol is based on a natural extension of the coin-tossing protocol of [32] to the multiparty setting, with the following high-level differences. First, instead of using just any zero-knowledge proof of knowledge, we use the zero-knowledge proof of knowledge of [22] that is constructed from two witness-indistinguishable proofs of knowledge.[20] Second, we use *strong* proofs of knowledge, rather than "ordinary" ones, so that if the prover convinces an honest verifier with non-negligible probability, a witness can be extracted with overwhelming probability in polynomial time.

We now present the protocol.

[20]We note that looking at our protocol it is not clear that we use the zero-knowledge proof of knowledge of [22], since the two witness-indistinguishable proofs of knowledge appear in different phases of the protocol, and moreover, we use the first witness-indistinguishable proof of knowledge for two different zero-knowledge proofs. Thus, our protocol does not exactly follow the syntax of [22] though the concept is similar.

Protocol $\boldsymbol{\rho}$ (protocol for realizing the $\mathcal{F}_{\text{CRS}}$ functionality in a general multiparty network, assuming time bounds Δ and ϵ):

- **Participating Parties:** $P_1, \ldots, P_k$ (some subset of the parties in the entire network).
- **Common Input:** the security parameter n, a session identifier $sid \in \{0,1\}^m$, and global constants Δ and ϵ.
- **The Protocol:** The protocol proceeds in three phases.
  - Phase One:
    1. Each party P_i chooses a pair of values $w_1^i, w_2^i \in_R \{0,1\}^n$, and computes $v_1^i = f(w_1^i), v_2^i = f(w_2^i)$, where f is a one-way function.
    2. Each party P_i proves independently to all other parties that it knows either $f^{-1}(v_1^i)$ or $f^{-1}(v_2^i)$. Formally, P_i proves that it knows a witness for the relation

$$R_1^i \stackrel{\text{def}}{=} \{((v_1^i, v_2^i), w) \mid v_1^i = f(w) \text{ or } v_2^i = f(w)\}.$$

       The proofs are given according to some arbitrary order; say the party with the smallest ID proves first, then the party with the second to smallest ID, and so on.[21] Each P_i carries out a proof that has the following properties:

       (a) The proof is an $\alpha(n)$-round witness-indistinguishable strong proof of knowledge, for some pre-specified super-constant function $\alpha(\cdot)$.[22] (Henceforth, we denote this proof by WISPOK, for short).

       (b) The proof is carried out in a parallel manner. That is, P_i sends the first message of the proof to all other parties. It then waits for the responses from all the parties, and only then sends the second message to all the parties, and so on.

       (c) The first and the last messages of the proof are sent by the verifier. (This is needed for technical reasons.)

       We let σ denote such a proof system. Each party P_i repeats this proof σ several times, according to a non-trivial pairwise-disjoint scheduling $S(\sigma, sid, \Delta, \epsilon)$ (the existence of such a scheduling is guaranteed in Theorem 8).

       If a party P_i receives a `time-out` message in an execution of σ, then it broadcasts `time-out` to all the parties, outputs (`time-out`, sid) and halts. Any party receiving such a `time-out` message also outputs (`time-out`, sid) and halts.
  - Phase Two: Each party P_i operates as follows.
    1. Party P_i chooses $r_i \in_R \{0,1\}^{p(n)}$ and broadcasts a commitment $c_i = C(r_i; s_i)$ to all the parties, where C is a perfectly binding commitment scheme and s_i is a random string. P_i waits for the commitments from all other parties to arrive before proceeding.
    2. Party P_i proves in parallel to every other party P_j that it knows either $f^{-1}(v_1^j)$ or $f^{-1}(v_2^j)$ or a pair (r_i, s_i) such that $c_i = C(r_i; s_i)$, using an $\alpha(n)$-round witness-indistinguishable strong proof of knowledge. Formally, P_i proves that it knows a witness for the relation

$$R_2^{i,j} \stackrel{\text{def}}{=} \{((v_1^j, v_2^j, c_i), (w, r, s)) \mid v_1^j = f(w) \text{ or } v_2^j = f(w) \text{ or } c_i = C(r; s)\}.$$

[21] Note that by requiring the proofs to be given sequentially we automatically obtain "independence" between proofs that belong to the *same* session.

[22] Recall that such proofs are known to exist for *any* super-constant function $\alpha(\cdot)$.

Time-Out Mechanism: For every proof that party P_i participated in (either as a prover or as a verifier), it checks that no more than $\boldsymbol{\tau} \stackrel{\text{def}}{=} \boldsymbol{\alpha(n)\cdot\Delta}$ local time units have passed from the time that the proof began until the time that it ended. If more time passed, then P_i broadcasts time-out to all the parties, outputs (time-out, sid) and halts the execution. Any party receiving such a time-out message also outputs (time-out, sid) and halts the execution.

3. Once Party P_i finished its proof and verified the proofs of all other parties, it broadcasts a Phase2over message to all other parties. It then waits for the same message to arrive from all other parties before proceeding. After this it will never output (time-out, sid).

- Delay Mechanism: Before continuing to Phase 3, each party P_i waits $\boldsymbol{\tau\epsilon}$ local time units.
- Phase Three: Party P_i broadcasts r_i to all other parties (without decommitting) and, using a 3-round witness indistinguishable proof of knowledge, proves in parallel to every other party P_j that it either knows a preimage for one of v_1^j, v_2^j or that it knows s such that $c_i = C(r_i; s)$. Formally, P_i proves in parallel that it knows a witness for the relation

$$R_3^{i,j} \stackrel{\text{def}}{=} \{((v_1^j, v_2^j, c_i, r_i), (w, s)) \mid v_1^j = f(w) \text{ or } v_2^j = f(w) \text{ or } c_i = C(r_i; s)\}.$$

- Each party P_i defines $R = r_1 \oplus r_2 \oplus \ldots \oplus r_k$, where r_j is the string it received in the previous step from party P_j, and r_i is the string that it broadcasted to all other parties.[23]
- **Output:** Each party outputs (sid, R).

This completes the description of the protocol.

Conventions. If an honest party receives a message that does not have a valid format or if it rejects a proof that it verifies, then the party broadcasts an abort message to all other parties and halts the execution.[24] Any party receiving such an abort message also halts the execution. We also assume that all messages are sent together with the session identifier sid, which is part of the common input. This enables the correct assignment of messages to their intended sessions. We stress that the security of the protocol does not rely on this assignment being correct. Rather, this mechanism just ensures successful termination when honest parties interact.

3.6 Proof of Security

We now show that Protocol ρ securely realizes the $\mathcal{F}_{\text{CRS}}$ functionality in the timing model, even when run many times concurrently with an arbitrary other protocol π, *as long as* all the messages in π are delayed by $\tau\epsilon$ local time units, where $\tau = \alpha(n) \cdot \Delta$.[25] In other words, Protocol ρ securely realizes $\mathcal{F}_{\text{CRS}}$ under concurrent general composition with $\Pi_{\tau\epsilon}$ in the timing model with ϵ. As we have seen in Section 3.2, this (along with the non-triviality condition) suffices for proving Theorem 5.

Theorem 9 *Let Δ and ϵ be fixed constants, such that $1 \le \epsilon < \sqrt[3]{1.5}$, and let $\tau = \alpha(n) \cdot \Delta$. Then, assuming the existence of one-way functions, Protocol ρ securely realizes the $\mathcal{F}_{\text{CRS}}$ functionality under concurrent composition with $\Pi_{\tau\epsilon}$ in the timing model with ϵ, and in the presence of static malicious adversaries. Furthermore, Protocol ρ is non-trivial under timing assumptions (Δ, ϵ).*

[23] Note that since all the r_i's were broadcasts it must be the case that all the honest parties have the same R.

[24] Recall that when a party times-out it behaves differently. Namely, it does not send an abort message, but rather sends a time-out message.

[25] Recall that $\alpha(n)$ is the number of rounds in the WISPOKs of Protocol ρ.

Proof: Let Δ and ϵ be any fixed constants such that $1 \leq \epsilon < \sqrt[3]{1.5}$. Let π be an arbitrary multiparty protocol that may contain ideal calls to the $\mathcal{F}_{\text{CRS}}$ functionality, and let $\pi_{\tau\epsilon}$ be the protocol obtained by delaying all messages in π by $\tau\epsilon$ local time units. Let $\mathcal{A}$ be any static non-uniform probabilistic polynomial-time ϵ-drift preserving adversary that runs protocol $\pi^{\rho}_{\tau\epsilon}$ in the timing model. We begin by describing the hybrid-model simulator $\mathcal{S}$ that runs π in the $\mathcal{F}_{\text{CRS}}$-hybrid model *without timing.*

The simulator $\mathcal{S}$ simulates the real-world adversary $\mathcal{A}$ internally. The aim of $\mathcal{S}$ is to force the output of the coin tossing protocol ρ in any given session to equal the common random string obtained from the $\mathcal{F}_{\text{CRS}}$ functionality. In order to do this, $\mathcal{S}$ deals with each session of the coin-tossing, one at a time.

In order for $\mathcal{S}$ to force a coin-tossing session to output some given random string R_{CRS} it will do the following: for every corrupted party P_j, it will *extract* from $\mathcal{A}$ both a value w_j such that $f(w_j) = v_1^j$ or $f(w_j) = v_2^j$, and a value r_j, which is the decommitment of c_j (sent by P_j in the beginning of Phase 2). These values will be extracted before entering Phase 3 of this session. Then, Phase 3 will be simulated in a "straight-line" manner: $\mathcal{S}$ will simulate each honest party P_i sending a random r_i such that $R_{\text{CRS}} = \oplus_{l=1}^{k} r_l$, and proving to each party P_j that r_i was committed to (even though it was not) using the previously extracted witness w_j.

Thus, the simulation by $\mathcal{S}$ consists of an extraction mode and a straight-line simulation mode. The "rewinding" takes place only when S is in the extraction mode (the rest of the simulation is "straight-line"). In the extraction mode $\mathcal{S}$ rewinds $\mathcal{A}$ internally, without rewinding the simulated protocol. That is, $\mathcal{S}$ pauses the simulation, and internally creates a copy of its simulated world. Then $\mathcal{S}$ rewinds the copy of $\mathcal{A}$. This rewinding is actually carried out in a look-a-head manner. That is, $\mathcal{S}$ (forward) simulates the messages that the honest parties in ρ will send to $\mathcal{A}$ after the paused point, and then rewinds this simulated protocol. The timing restraints ensure that messages from π (that are sent externally by $\mathcal{S}$) never have to be sent while S is in the extraction mode, where rewinding takes place. We now formally describe $\mathcal{S}$.

The simulator $\mathcal{S}$: $\mathcal{S}$ is a "hybrid-world" adversary, that interacts with the parties running protocol π in the $\mathcal{F}_{\text{CRS}}$-hybrid model. The aim of $\mathcal{S}$ is to create the same effect in the $\mathcal{F}_{\text{CRS}}$-hybrid model, as the real-world adversary $\mathcal{A}$ does in a real execution of $\pi^{\rho}_{\tau\epsilon}$.

As was previously mentioned, $\mathcal{S}$'s operations consist of two modes of operation: straight-line simulation mode and extraction mode. $\mathcal{S}$ starts and ends in the straight-line simulation mode, but frequently leaves it and enters the extraction mode. In the straight-line mode, $\mathcal{S}$ interacts with the honest parties (in the $\mathcal{F}_{\text{CRS}}$-hybrid model), while updating internally simulated states of the adversary $\mathcal{A}$ and of the honest parties running the program for the protocol ρ. In the extraction-mode, these simulated states are frozen, while $\mathcal{S}$ applies an extraction subroutine. The output of the extraction subroutine will be needed for continuing the straight-line mode.

We shall denote by $\mathcal{P}_i^{sid}$ the simulated ρ program of an honest party P_i for a session sid, which $\mathcal{S}$ uses in the straight-line simulation mode. The simulation enters the extraction mode every time that $\mathcal{P}_i^{sid}$ is about to take part as a verifier in one of the WISPOKs given by a corrupted player in the protocol. In this extraction mode S calls the extraction subroutine. This subroutine will try to find the witness used by the corrupted prover in that WISPOK. The straight-line simulation continues when the extraction subroutine returns.

We proceed to define the simulator by first describing the straight-line simulation mode and then describing the extraction subroutine.

Straight-line mode: $\mathcal{S}$ internally runs $\mathcal{A}$, and for each honest party, $\mathcal{S}$ simulates the various tapes that $\mathcal{A}$ expects to have access to (namely, the communication tapes and the clock-tape). It also maintains simulated states (i.e., the work-tape) of the ρ programs of the honest parties. As was mentioned above, we denote by $\mathcal{P}_i^{sid}$ the program simulated by $\mathcal{S}$, corresponding to the ρ program of an honest party P_i in session sid. The simulated programs $\mathcal{P}_i^{sid}$ communicate directly with $\mathcal{A}$.

In addition, $\mathcal{S}$ needs to let the π program of the external honest parties communicate with $\mathcal{A}$ (here we mean the real parties with whom $\mathcal{S}$ interacts in the hybrid model). For π messages from $\mathcal{A}$ to a party P_i, this is done simply by sending the messages out to P_i (i.e., they are copied onto P_i's incoming message tape). However, upon receiving a π message from an external honest party P_i, simulator $\mathcal{S}$ needs to simulate the delay of P_i before forwarding it to $\mathcal{A}$ (because in the hybrid world π messages are sent out without any delay, in contrast to the real world). Therefore, $\mathcal{S}$ waits $\tau\epsilon$ time units according to P_i's simulated local clock before sending the received π-message to $\mathcal{A}$. Finally, $\mathcal{S}$ generates the same input-output as $\mathcal{A}$. More formally:

- Whenever a session sid with parties $P_{i_1}, \ldots, P_{i_k}$ is begun, $\mathcal{S}$ sends $(\mathsf{crsgen}, sid, \{P_{i_1}, \ldots, P_{i_k}\})$ to the $\mathcal{F}_{\text{CRS}}$ functionality.[26] We assume that at least one party in $\{P_{i_1}, \ldots, P_{i_k}\}$ is honest, since the case that all parties are corrupted is trivial.

- $\mathcal{S}$ initiates the program $\mathcal{P}_i^{sid}$ corresponding to each honest participant P_i.

- If at any point $\mathcal{P}_i^{sid}$ outputs $(\mathsf{time\text{-}out}, sid)$, $\mathcal{S}$ sends $(\mathsf{time\text{-}out}, sid)$ to the $\mathcal{F}_{\text{CRS}}$ functionality and delivers the message $(\mathsf{time\text{-}out}, sid)$ from the $\mathcal{F}_{\text{CRS}}$ functionality to P_i.

- Whenever $\mathcal{A}$ sends a π message to some party P_i in session sid, $\mathcal{S}$ sends the π message externally to party P_i in session sid.

- Whenever $\mathcal{S}$ (externally) receives a π message from some honest party P_i, it stores the message in an internal delay buffer. Then, after $\tau\epsilon$ time units according to $\mathcal{P}_i^{sid}$'s internally simulated local clock, it forwards the π message to $\mathcal{A}$.

- For all except one honest party, $\mathcal{P}_i^{sid}$ runs exactly the program specified by the protocol ρ. We denote the index of this one chosen honest party by $a(sid)$; when sid is clear from the context we shall write a instead of $a(sid)$.[27] The program $\mathcal{P}_{a(sid)}^{sid}$ is identical to ρ in Phases 1 and 2, and differs from ρ only in Phase 3. We shall describe the differences shortly.

- In Phases 1 and 2, when $\mathcal{P}_a^{sid}$ receives the first message of a WISPOK in which it plays *verifier* and a corrupted party plays *prover*, $\mathcal{S}$ applies the extraction subroutine (to be defined later) to that WISPOK. The output of the extraction subroutine is recorded for later reference. (Note that extraction is only carried out when $\mathcal{P}_a^{sid}$ plays the verifier.)

- At the point that $\mathcal{P}_a^{sid}$ enters Phase 3 of the protocol, $\mathcal{S}$ carries out the following two checks:

  1. $\mathcal{S}$ checks the output of the extraction subroutine applied to each of the Phase 1 and Phase 2 WISPOKs given to $\mathcal{P}_a^{sid}$ by a corrupted party. If in any of them, the extraction subroutine failed to extract a valid witness for the statement of that WISPOK, then $\mathcal{S}$ outputs fail_1 and halts.
  2. Let (v_1^a, v_2^a) be the first message that $\mathcal{P}_a^{sid}$ sends in session sid. Recall that w such that $f(w) \in \{v_1^a, v_2^a\}$ is a valid witness in all of the Phase 2 WISPOKs that $\mathcal{P}_a^{sid}$ verifies. If

[26] Actually, this crsgen message to the functionality may have been sent by one of the honest parties participating in session sid. This is inconsequential.

[27] This honest party can be arbitrarily chosen – say, the one with the "smallest" identity among all honest participants.

the extraction subroutine, applied to any of the Phase 2 WISPOKs given to $\mathcal{P}_a^{sid}$ by a corrupted party outputs such a w as a witness, then $\mathcal{S}$ outputs fail_2 and halts.

Note that if $\mathcal{S}$ did not output fail_1 then for every corrupted party P_j, the extraction subroutine applied to each of the Phase 1 WISPOKs given by P_j must have returned w^j such that $f(w^j) \in \{v_1^j, v_2^j\}$. Furthermore, for every honest party P_i, $\mathcal{S}$ can look up such a w^i from $\mathcal{P}_i^{sid}$ (because $\mathcal{S}$ runs the code of P_i internally).

Similarly, if $\mathcal{S}$ also did not output fail_2 then for every corrupt party P_j, the extraction subroutine, applied to each of the Phase 2 WISPOKs given by P_j, must have produced the witness (r_j, s_j) such that $c_j = C(r_j; s_j)$, where c_j is the commitment text sent by P_j in Phase 2 of session *sid* (recall that the only valid witnesses for this WISPOK are either the above mentioned witness (r_j, s_j) or w such that $f(w) \in \{v_1^a, v_2^a\}$, where extraction of the latter witness results with fail_2). In addition, for every honest party P_i, $\mathcal{S}$ can look up r_i from $\mathcal{P}_i^{sid}$ (again, because $\mathcal{S}$ runs $\mathcal{P}_i^{sid}$).

Thus, if the above two checks passed (namely, $\mathcal{S}$ did not output fail_1 or fail_2) then $\mathcal{S}$ has obtained values w^i and r_i, *for all participants* P_i. (As we will see, for all $i \neq a$, the values w^i and r_i will be needed by $\mathcal{S}$ to continue the simulation.)

- If the above two checks passed then $\mathcal{S}$ acts as follows:
  1. $\mathcal{S}$ sends $(sid, \mathsf{compute})$ to the $\mathcal{F}_{\text{CRS}}$ functionality, and receives $(\mathsf{crsgen}, sid, \{P_{i_1}, \ldots, P_{i_k}\}, R_{\text{CRS}})$ in response.[28] Then using the r_i values as given above, $\mathcal{S}$ computes
  $$r = R_{\text{CRS}} \oplus \left(\bigoplus_{i \neq a} r_i \right). \quad (2)$$
  2. $\mathcal{S}$ hands r (from Eq. (2)) and $\{w^i\}_{i \neq a}$ to $\mathcal{P}_a^{sid}$.
- $\mathcal{P}_a^{sid}$ proceeds with the simulation of Phase 3. (Notice that its instructions here differ from the program specified by ρ for the honest parties.)
  1. In the beginning of Phase 3, $\mathcal{P}_a^{sid}$ does not send the value r_a that it committed to in Phase 2, as instructed by protocol ρ. Rather, it sends the value r given to it by $\mathcal{S}$.
  2. After sending r, $\mathcal{P}_a^{sid}$ proves to each party P_j (in the WIPOK of Phase 3) that this "fake" value r is the value that it committed to in Phase 2. This is done using the alternative witness w^j given to it by $\mathcal{S}$.[29]
- If there exists a (corrupted) party P_j that broadcasted $r'_j \neq r_j$ in the beginning of Phase 3 and $\mathcal{P}_a^{sid}$ accepts its Phase 3 WIPOK, then $\mathcal{S}$ outputs fail_3.

[28] At this point of the protocol it is guaranteed that no honest party has or will output $(\mathsf{time\text{-}out}, sid)$, because they must all have sent $\mathsf{Phase2over}$ messages (since $\mathcal{P}_a^{sid}$ entered Phase 3). Hence it is possible for $\mathcal{S}$ to send a $(sid, \mathsf{compute})$ message to $\mathcal{F}_{\text{CRS}}$, thereby receiving back (sid, R_{CRS}).

[29] It's ability to use the "fake" value $r = R_{\text{CRS}} \oplus \left(\bigoplus_{i \neq a} r_i\right)$, rather than the value that it committed to, is exactly what allows the output of this session to equal R_{CRS}. Note that in order to use this "fake" value r it must know all the alternative witnesses $\{w^i\}_{i \neq a}$, which is why $\mathcal{S}$ must apply the extraction subroutine to the WISPOKs of Phase 1. The reason $\mathcal{S}$ must apply the extraction subroutine to the WISPOKs of Phase 2 is in order to obtain all the values $\{r_i\}_{i \neq a}$, which are needed in order to determine the "fake" value $r = R_{\text{CRS}} \oplus \left(\bigoplus_{i \neq a} r_i\right)$.

- For every honest party P_i, if $\mathcal{P}_i^{sid}$ outputs (sid, R) then $\mathcal{S}$ delivers the message (sid, R_{CRS}) from $\mathcal{F}_{\text{CRS}}$ to P_i.[30]

This completes the description of the simulator except for the extraction subroutine.

THE EXTRACTION SUBROUTINE: Recall that in Phases 1 and 2, when $\mathcal{P}_a^{sid}$ receives the first message of a WISPOK in which it plays verifier and a corrupted party P_j plays prover, $\mathcal{S}$ calls the extraction subroutine. (We shall denote such a WISPOK by WISPOK_j^{sid}.) The extraction subroutine will try to extract a witness for the statement of WISPOK_j^{sid} by constructing a stand-alone prover $\mathcal{Q}_j^{sid}$ from $\mathcal{A}$, and then applying the strong proof of knowledge extractor to $\mathcal{Q}_j^{sid}$. The stand-alone prover $\mathcal{Q}_j^{sid}$ is defined as follows.

$\mathcal{Q}_j^{sid}$ is a stand-alone (cheating) prover who proves a single strong proof of knowledge to an external verifier. $\mathcal{Q}_j^{sid}$ works exactly like $\mathcal{S}$, continuing from the point after the extraction subroutine is invoked, except for the following differences:

- In $\mathcal{S}$, the program $\mathcal{P}_a^{sid}$ plays the verifier of the WISPOK: i.e., it receives the WISPOK messages from a prover P_j, and responds to them as a verifier. Instead, in $\mathcal{Q}_j^{sid}$, the program $\mathcal{P}_a^{sid}$ relays out the incoming WISPOK messages from P_j to an external verifier. When it receives a response from the external verifier, it forwards it internally to P_j as its own response.

- Since $\mathcal{Q}_j^{sid}$ is a stand-alone prover, unlike $\mathcal{S}$, it cannot interact with the honest parties running the protocol π in the hybrid world. So all messages generated by $\mathcal{S}$ for these parties are ignored. Furthermore, there are no incoming messages from the π protocol. However the messages that arrived earlier and were stored internally in the delaying buffers of $\mathcal{S}$ will be used just like $\mathcal{S}$ did originally. (As we will see later, this suffices and no "new" π messages are needed.)

  Note also that since $\mathcal{Q}_j^{sid}$ is a stand-alone prover, it cannot interact with the different instances of $\mathcal{F}_{\text{CRS}}$. However, these can all be perfectly simulated internally by $\mathcal{Q}_j^{sid}$.

- $\mathcal{Q}_j^{sid}$ does not invoke the extraction subroutine that $\mathcal{S}$ invokes. Instead, when the extraction subroutine needs to be called, it is just assumed to return $\perp$ (this ensures that $\mathcal{Q}_j^{sid}$ is well-defined).

- $\mathcal{Q}_j^{sid}$ halts as soon as it receives an accept or reject message from the outside verifier. Also, if $\mathcal{P}_a^{sid}$'s local clock reaches a time where the original $\mathcal{P}_a^{sid}$ would have timed-out, then $\mathcal{Q}_j^{sid}$ halts.

The key point to notice is that $\mathcal{Q}_j^{sid}$ is a stand-alone adversary who proves a single strong proof of knowledge to an external verifier. The extraction subroutine applies the strong knowledge extractor K to the prover $\mathcal{Q}_j^{sid}$ (recall that if $\mathcal{Q}_j^{sid}$ convinces an honest verifier V in the proof with probability greater than $\mu(n)$ for some negligible function μ, then K obtains a witness with probability at least $1 - \mu(n)$).

This completes the description of the extraction subroutine. Note that the extraction subroutine is invoked on all the Phase 1 and Phase 2 WISPOKs given to $\mathcal{P}_a^{sid}$ by any corrupted party P_j in any session sid (with at least one honest player). This ensures that if the WISPOKs convince

[30]Notice that if $\mathcal{P}_i^{sid}$ produced an output (sid, R) then it must be the case that $R = R_{\text{CRS}}$. If $R \neq R_{\text{CRS}}$ then there exists a j such that $r_j \neq r'_j$ (follows from the fact that $R = r'_1 \oplus, \ldots, r'_k$ and $R_{\text{CRS}} = r_1 \oplus, \ldots, r_k$). In this case, either $\mathcal{S}$ outputs fail_3 or $\mathcal{P}_a^{sid}$ does not accept the Phase 3 WIPOK of P_j and thus will halt the execution. In both cases $\mathcal{P}_i^{sid}$ would not produce an output.

$\mathcal{P}_a^{sid}$ with non-negligible probability, then the simulator will obtain the corresponding witnesses with overwhelming probability, by applying the extraction subroutine. (Of course, this is the case assuming that $\mathcal{Q}_j^{sid}$ convinces the verifier with essentially the same probability that $\mathcal{P}_a^{sid}$ is convinced. This will be proven below.)

Proof of the simulation. First note that $\mathcal{S}$ runs in strict polynomial-time if $\mathcal{A}$ runs in strict polynomial-time (because the knowledge extractor of a strong proof of knowledge runs in strict polynomial-time, and the only rewinding carried out by $\mathcal{S}$ is in applying the knowledge extractor). We now prove that the output distribution of $\mathcal{S}$ and the honest parties running π in the $\mathcal{F}_{\text{CRS}}$-hybrid model is computationally indistinguishable from the output distribution of an ϵ-drift preserving adversary $\mathcal{A}$ and the honest parties in a real execution of Protocol $\pi^{\rho}_{\tau\epsilon}$ in the timing model. In order to prove this, we first show that $\mathcal{S}$ outputs a fail message with negligible probability. Given this, we then introduce hybrid experiments which bridge the difference between the $\mathcal{F}_{\text{CRS}}$-hybrid execution and the real execution, to prove the claimed indistinguishability.

We now prove that $\mathcal{S}$ outputs a fail message with at most negligible probability. Recall that there are three types of failures: fail_1, fail_2 and fail_3. Intuitively, fail_1 occurs if there exists a WISPOK for which the extractor fails to output a corresponding witness, and yet $\mathcal{P}_a^{sid}$ accepts the WISPOK. fail_2 occurs if the extractor, applied to any of the Phase 2 WISPOKs, outputs the "wrong witness;" i.e., instead of extracting the committed value r_j together with the corresponding randomness s_j (such that $c_j = C(r_j, s_j)$), it somehow extracts a witness w such that $f(w) \in \{v_1^a, v_2^a\}$. fail_3 occurs if there exists a (corrupted) party P_j that in the beginning of Phase 3, sends a value r'_j which is different from the value r_j extracted in the extraction subroutine, and yet $\mathcal{P}_a^{sid}$ accepts the WIPOK in Phase 3.

We show that each of these failures occurs with negligible probability.

$\mathcal{S}$ OUTPUTS fail_1 WITH NEGLIGIBLE PROBABILITY: Recall that $\mathcal{S}$ outputs fail_1 if there exists a session sid such that $\mathcal{P}_a^{sid}$ enters Phase 3, and there exists a corrupted party P_j such that for one of its (Phase 1 or Phase 2) WISPOKs given to $\mathcal{P}_a^{sid}$ in this session, the extractor failed to extract a witness. Note that it must be the case that $\mathcal{P}_a^{sid}$ has accepted this WISPOK, since otherwise it would have never reached Phase 3. Thus, the occurrence of fail_1 implies that there exists a session sid and a corrupted party P_j such that the extractor failed to extract a witness, and yet $\mathcal{P}_a^{sid}$ accepted the WISPOK. In other words, the strong knowledge extractor K failed to obtain a witness from the stand-alone prover $\mathcal{Q}_j^{sid}$, yet later in the simulation, $\mathcal{S}$ accepts that proof from $\mathcal{A}$. Intuitively, this should not happen because K is such that if a prover convinces the honest verifier with non-negligible probability, then it successfully extracts with overwhelming probability. However, this is not immediate because K attempts to extract from the stand-alone adversary $\mathcal{Q}_j^{sid}$, whereas $\mathcal{P}_a^{sid}$ verifies the proof from the original adversary $\mathcal{A}$. Thus, the essence here is to show that $\mathcal{Q}_j^{sid}$ convinces an honest verifier with the same probability that $\mathcal{A}$ convinces $\mathcal{P}_a^{sid}$.

Claim 10 *For any corrupted party P_j participating in session sid, let* $\text{WISPOK}_j^{sid,\ell}$ *denote the* ℓ^{th} WISPOK *of P_j in this session. Then, the stand-alone prover $\mathcal{Q}_j^{sid}$, constructed by the extractor in the beginning of* $\text{WISPOK}_j^{sid,\ell}$*, convinces an honest verifier with exactly the same probability as $\mathcal{P}_a^{sid}$ accepts* $\text{WISPOK}_j^{sid,\ell}$ *in the straight-line simulation by $\mathcal{S}$.*

Proof: The main observation involved is that after $\text{WISPOK}_j^{sid,\ell}$ begins, the fact that no further extraction procedures are run and no new π-messages are received, makes no difference in the straight-line mode, *until after the* WISPOK *is finished.* This is ensured by the time-out for the

WISPOK, by the fact that the output of the extractors in a session are not used until the session enters Phase 3, and by the delay introduced to π messages. We elaborate below.

First, we construct a simulator $\mathcal{S}'$ which is the same as $\mathcal{S}$ except that it does not invoke the extraction subroutine after the point at which $\text{WISPOK}_j^{sid,\ell}$ has begun. Thus, if a WISPOK, denoted $\text{WISPOK}_{j'}^{sid',\ell'}$, of Phase 1 or Phase 2 in a session sid' starts after the point at which $\text{WISPOK}_j^{sid,\ell}$ has begun, the simulator $\mathcal{S}'$ will not run the extraction subroutine for $\text{WISPOK}_{j'}^{sid',\ell'}$, whereas $\mathcal{S}$ would. Now, recall that $\mathcal{S}$ does not use the output that this extraction subroutine returns until the session sid' enters Phase 3. We claim that the delay between Phase 2 and Phase 3 in the protocol ensures that $\mathcal{S}$ will enter Phase 3 in session sid' only after $\text{WISPOK}_j^{sid,\ell}$ has already concluded. This follows from the following facts:

1. When $\text{WISPOK}_j^{sid,\ell}$ began, session sid' did not yet finish Phase 2 (because session sid' must still at least run $\text{WISPOK}_{j'}^{sid',\ell'}$).
2. $\text{WISPOK}_j^{sid,\ell}$ is timed-out by $\mathcal{P}_a^{sid}$ if it does not conclude within τ local time units. By the assumption on the bounded clock drifts, this is at most $\tau\epsilon$ local time units according to $\mathcal{P}_a^{sid'}$'s clock.
3. $\mathcal{P}_a^{sid'}$ waits at least $\tau\epsilon$ local time units between Phase 2 and Phase 3.

Thus $\mathcal{S}$ and $\mathcal{S}'$ identically simulate the interaction between $\mathcal{P}_a^{sid}$ and $\mathcal{A}$, until $\text{WISPOK}_j^{sid,\ell}$ concludes. Therefore, the probability that $\text{WISPOK}_j^{sid,\ell}$ is accepted by $\mathcal{P}_a^{sid}$ is equal in both cases.

Next we modify $\mathcal{S}'$ to obtain a stand-alone machine $\mathcal{S}''$ which ignores all communication with the honest parties (in the π protocol) after the point at which $\text{WISPOK}_j^{sid,\ell}$ has begun. Note that if $\mathcal{S}'$ receives a π-message from a party P_i, it will be delivered to $\mathcal{A}$ only after a delay of $\tau\epsilon$ time units according to P_i's local clock. The restriction on the drifts of the clocks ensures that this delay is at least τ time units according to P_a's local clock. So, if $\mathcal{S}'$ received this message *after* $\text{WISPOK}_j^{sid,\ell}$ has begun, it will not be used until $\mathcal{P}_a^{sid}$ concludes $\text{WISPOK}_j^{sid,\ell}$. This is because $\mathcal{P}_a^{sid}$ will conclude the WISPOK (by timing-out if necessary) within τ local time units after $\text{WISPOK}_j^{sid,\ell}$ has begun (which is at most $\tau\epsilon$ on P_i's local clock). Hence the probability that $\mathcal{P}_a^{sid}$ accepts $\text{WISPOK}_j^{sid,\ell}$ in $\mathcal{S}''$ is equal to that in $\mathcal{S}'$.

Finally, we note that the system consisting of the stand-alone prover $\mathcal{Q}_j^{sid}$ interacting with an external honest verifier, is the same system as emulated by the stand-alone machine $\mathcal{S}''$. The role of the external verifier is played honestly by $\mathcal{P}_a^{sid}$ in $\mathcal{S}''$. Thus the probability that $\mathcal{Q}_j^{sid}$ can convince an honest verifier is exactly equal to the probability that $\mathcal{P}_a^{sid}$ will accept $\text{WISPOK}_j^{sid,\ell}$ in the execution of $\mathcal{S}''$ or $\mathcal{S}$. ■

Now, let $\mu(n)$ be the negligible error function of the strong proof of knowledge. That is, if a prover convinces an honest verifier with probability greater than $\mu(n)$, then K successfully extracts with probability greater than $1-\mu(n)$. We define three events: "K-fail" if K fails to extract a witness from $\mathcal{Q}_j^{sid}$, "$\mathcal{S}$-pass" if $\mathcal{P}_a^{sid}$ accepts $\text{WISPOK}_j^{sid,\ell}$, and "$\mathsf{good\text{-}proof}$" if the probability that an honest verifier accepts the proof given by the stand-alone prover $\mathcal{Q}_j^{sid}$ is at least $\mu(n)$. Then, the probability that $\mathcal{S}$ outputs fail_1 corresponding to $\text{WISPOK}_j^{sid,\ell}$ is bounded by

$$\begin{aligned}\Pr[K\text{-}\mathsf{fail} \wedge \mathcal{S}\text{-}\mathsf{pass}] &= \Pr[K\text{-}\mathsf{fail} \wedge \mathcal{S}\text{-}\mathsf{pass} \wedge \mathsf{good\text{-}proof}] + \Pr[K\text{-}\mathsf{fail} \wedge \mathcal{S}\text{-}\mathsf{pass} \wedge \neg\mathsf{good\text{-}proof}] \\ &\leq \Pr[K\text{-}\mathsf{fail}|\mathsf{good\text{-}proof}]\Pr[\mathsf{good\text{-}proof}] + \Pr[\mathcal{S}\text{-}\mathsf{pass}|\neg\mathsf{good\text{-}proof}]\Pr[\neg\mathsf{good\text{-}proof}]\end{aligned}$$

$$\leq \mu(n)\Pr[\textsf{good-proof}] + \mu(n)\Pr[\neg\textsf{good-proof}]$$
$$= \mu(n). \tag{3}$$

<u>$\mathcal{S}$ OUTPUTS $\textsf{fail}_2$ OR $\textsf{fail}_3$ WITH NEGLIGIBLE PROBABILITY:</u> Recall that $\mathcal{S}$ outputs $\textsf{fail}_2$ if the extraction subroutine applied to a Phase 2 WISPOK of some session *sid* outputs w such that $f(w) \in \{v_1^{a(sid)}, v_2^{a(sid)}\}$. It outputs $\textsf{fail}_3$ if in Phase 3 of some session *sid* there exists a corrupted party P_j that does the following: **(a)** it sends a value r'_j different from the value r_j extracted from the extraction subroutine (applied to the Phase 2 WISPOK given by P_j in session *sid*), and **(b)** it succeeds in proving that it either knows w such that $f(w) \in \{v_1^{a(sid)}, v_2^{a(sid)}\}$ or that r'_j is indeed the value it committed to in Phase 2. However, since the second half of **(b)** is false, the soundness of the WIPOK would require that the first half of **(b)** be true, namely that it knows w.

Thus the cause for either of these failures ($\textsf{fail}_2$ or $\textsf{fail}_3$) is essentially that the adversary knows w such that $f(w) \in \{v_1^{a(sid)}, v_2^{a(sid)}\}$. (Note that these $(v_1^{a(sid)}, v_2^{a(sid)})$ values are chosen by an honest party.) Our proof that $\textsf{fail}_2$ or $\textsf{fail}_3$ is unlikely will use the argument that it is unlikely that the adversary can obtain such a w. Intuitively, this is due to the fact that w is only used in proving witness-indistinguishable proofs, which are also witness hiding. However, the actual proof is more complicated due to the fact that the adversary does not have to explicitly guess such a w, but merely succeed in giving a proof of knowledge of w, when concurrently interacting with the honest parties in multiple sessions. In order to prove that this is not feasible, we shall show how to construct a stand-alone machine M which interacts with an external machine $\mathcal{E}$. The machine $\mathcal{E}$ sends a pair (v_1, v_2), like in Phase 1 of our protocol, followed by many WISPOKs to M, to prove that it knows w such that $f(w) \in \{v_1, v_2\}$. Our construction of M will be such that if $\mathcal{S}$ outputs $\textsf{fail}_2$ or $\textsf{fail}_3$ with non-negligible probability, then M can also output w at the end of this interaction with non-negligible probability. Since f is a one-way function and the proofs are witness indistinguishable (and hence witness hiding), this will lead to a contradiction. We note that the formal proof relies heavily on the fact that the scheduling is pairwise disjoint.

M is constructed in two steps. First we describe a modified simulator $\mathcal{T}$, and then, depending on whether it is $\textsf{fail}_2$ or $\textsf{fail}_3$ that occurs with non-negligible probability, we show how to build M from $\mathcal{T}$.

The main feature of $\mathcal{T}$ is that, in a randomly chosen session sid^*, it interacts with the above mentioned external prover $\mathcal{E}$ (instead of with the internally simulated honest protocol program $\mathcal{P}_a^{sid^*}$). We shall ensure that if $\mathcal{S}$ outputs $\textsf{fail}_2$ or $\textsf{fail}_3$ with non-negligible probability, then so does $\mathcal{T}$.

OVERVIEW OF $\mathcal{T}$: $\mathcal{T}$ emulates part of the hybrid system consisting of $\mathcal{F}_{\text{CRS}}$ and $\mathcal{S}$, but with the emulated $\mathcal{S}$ modified as follows: for a randomly chosen session sid^*, the simulated program $\mathcal{P}_a^{sid^*}$ is not entirely run internally; instead part of the Phase 1 protocol is carried out by an external program $\mathcal{E}$, with which $\mathcal{T}$ interacts. The extractors in $\mathcal{S}$ are modified in such a way that they do not use the internal state of $\mathcal{E}$ (and in particular they do not "rewind" $\mathcal{E}$). These modifications will be such that $\mathcal{T}$ outputs $\textsf{fail}_2$ or $\textsf{fail}_3$ with non-negligible probability if $\mathcal{S}$ did so in the original hybrid system. The proof of this fact crucially depends on the way Phase 1 WISPOKs are scheduled; we will use the fact that the scheduling is pairwise disjoint to argue that even without rewinding $\mathcal{E}$, the extraction procedure can still be carried out in $\mathcal{T}$.

OVERVIEW OF M: M will run $\mathcal{T}$ described above, as well as the rest of the hybrid system (namely, the honest parties running the protocol π). M *does not* include $\mathcal{E}$ mentioned above. Instead, it *interacts with* $\mathcal{E}$. Furthermore, M attempts to extract the witness w from $\mathcal{E}$, as mentioned earlier. If $\mathcal{T}$ outputs $\textsf{fail}_2$, the witness should have been extracted by the extractor in $\mathcal{T}$. Thus, M can output this witness. If $\mathcal{T}$ outputs fail_3, then M will construct a stand-alone prover for the Phase 3 WIPOK (corresponding to which $\mathcal{T}$ outputs fail_3) and use an extractor on this prover to obtain w (because, as mentioned earlier, in this case w will be the only valid witness for the WIPOK). In either case M will be able to output w with non-negligible probability.

CONTRADICTION GIVEN M: Notice that M, which interacts with $\mathcal{E}$ as above, can output w with at most negligible probability. This is due to the following two observations:

1. Given the pair (v_1, v_2) which is computed by $\mathcal{E}$ (by choosing w_1, w_2 at random and setting $v_i = f(w_i)$), it is infeasible for M to find w such that $f(w) \in \{v_1, v_2\}$ (this follows from the fact that f is a one-way function).

2. The WISPOKs that $\mathcal{E}$ provides to M are *witness hiding* [23] (this follows from the fact that the proofs are witness indistinguishable with independent witnesses; see [24] for further details), and thus do not give M any non-negligible advantage in guessing w.

Thus, in order to prove that $\mathcal{S}$ has negligible probability of outputting fail_2 or fail_3, it suffices to show that if $\mathcal{S}$ outputs fail_2 or fail_3 with non-negligible probability, then M, which interacts with $\mathcal{E}$ as above, outputs w with non-negligible probability.

It remains only to construct M as claimed, which in turn is built from $\mathcal{T}$.

CONSTRUCTION OF $\mathcal{T}$: First we present the details of the construction of $\mathcal{T}$, as well as the proof that it outputs fail_2 or fail_3 with non-negligible probability if $\mathcal{S}$ does so. The construction is carried out through a series of modifications to $\mathcal{S}$. The goal is to bring the simulator to a state where it does not need to rewind the Phase 1 WISPOKs of $\mathcal{P}_a^{sid^*}$ (step 6). This will enable us to safely replace this part of $\mathcal{P}_a^{sid^*}$ by the external machine $\mathcal{E}$ (step 7). In order to eliminate the rewinding of the Phase 1 WISPOKs of $\mathcal{P}_a^{sid^*}$, we modify $\mathcal{S}$ so that rather than running the extractor on all Phase 2 WISPOKs, it runs the extractor only on the Phase 2 WISPOKs of session sid^* (step 5). Then we further modify $\mathcal{S}$ so that, rather than running the extractor on all the Phase 1 WISPOKs, it runs the extractor on a single Phase 1 WISPOK in each session; namely, the one which is pairwise disjoint to (i.e., does not overlap with) any of the Phase 1 WISPOKs of $\mathcal{P}_a^{sid^*}$. (This point of the proof is exactly where the pairwise disjointness comes in.)

Formally, the construction of $\mathcal{T}$ is carried out through a series of seven modifications to $\mathcal{S}$. After each modification we show that if the probability of outputting fail_2 or fail_3 is non-negligible in the previous step, it continues to be so in this step too. The simulator in step 7 corresponds to $\mathcal{T}$. We now begin with the modifications:

1. First modify $\mathcal{S}$ so that it never outputs fail_1, and does not check if the fail_1 condition holds. We denote the modified simulator by $\mathcal{S}_1$. Since $\mathcal{S}$ outputs fail_1 with negligible probability, it follows that $\mathcal{S}_1$ and $\mathcal{S}$ are statistically close, and in particular, the probability with which they output fail_2 and fail_3 is the same up to a negligible factor.

2. Modify $\mathcal{S}_1$ to obtain a new simulator $\mathcal{S}_2$ that behaves similarly to $\mathcal{S}_1$ with the following differences: Instead of accessing an external $\mathcal{F}_{\text{CRS}}$ functionality, it internally implements it. (Thus the honest parties obtain their outputs from $\mathcal{F}_{\text{CRS}}$ implemented by $\mathcal{S}_2$.) Furthermore, in Phase 3 of each session sid, instead of first drawing a random R_{CRS} (on behalf of $\mathcal{F}_{\text{CRS}}$) and then defining $r = \bigoplus_{i \neq a} r_i \oplus R_{\text{CRS}}$, it first draws a random r and defines $R_{\text{CRS}} = \bigoplus_{i \neq a} r_i \oplus r$. (See Eq. (2); recall that r_i is the value that party P_i committed to in the beginning of Phase 2, and if P_i is corrupted then r_i is obtained by applying the extraction subroutine to the Phase 2 WISPOK given by P_i.) Note that the output distributions of $\mathcal{S}_1$ and $\mathcal{S}_2$ are identical, and in particular the probability with which $\mathcal{S}_1$ and $\mathcal{S}_2$ output fail_2 and fail_3 is the same.

3. Next we observe that the r_j values extracted from the Phase 2 WISPOKs are used twice by the simulator $\mathcal{S}_2$:

   (a) To check the fail_3 condition.

   (b) To compute R_{CRS}, which is needed when some $\mathcal{P}_i^{sid}$ produces an output (sid, R). In this case, $\mathcal{F}_{\text{CRS}}$ (implemented by the simulator) sends R_{CRS} to P_i.

   We claim that the second usage of the r_j values is not essential. In order to see this, we modify $\mathcal{S}_2$ so that instead of computing $R_{\text{CRS}} = r_1 \oplus, \ldots, \oplus r_k$ and sending it to P_i (thereby using the r_j values), it computes $\mathsf{R}' = r'_1 \oplus, \ldots, \oplus r'_k$ and sends R' to P_i.[31] As was pointed out in footnote 30, if $R_{\text{CRS}} \neq \mathsf{R}'$ then it must be the case that either $\mathcal{S}$ outputs fail_3 or $\mathcal{P}_a^{sid}$ rejects one of the Phase 3 WIPOKs that it verifies, both which result with P_i not receiving any output. Thus if P_i does receive an output it must be the case that $R_{\text{CRS}} = \mathsf{R}'$. Therefore this modification does not change anything in the system, except to make it explicit that the extracted values r_j are used only for determining if fail_3 occurs. We denote the new simulator by $\mathcal{S}_3$.

4. We next define $\mathcal{S}_4$ which behaves identically to $\mathcal{S}_3$ except for the following: $\mathcal{S}_4$ chooses a random session and outputs fail_2 or fail_3 only if it happens in the chosen session. (In other sessions if $\mathcal{S}_3$ would have output fail_2 or fail_3 and halted, $\mathcal{S}_4$ does not even check for the failure condition and so might continue executing.) Note that there are only polynomially sessions possible (as the adversary and the polynomially many parties are all assumed to be strict polynomial time machines). Hence if $\mathcal{S}_3$ outputs fail_2 or fail_3 with non-negligible probability, so does $\mathcal{S}_4$. (The reason that this holds is that with probability 1/poly, the *first* session in which fail_2 or fail_3 occurs will be chosen, and the simulation until that point is identical.)

   We shall denote by $\mathcal{S}_4^{sid^*}$ the resulting simulator when $\mathcal{S}_4$ picks a session with session identifier sid^* as its random choice. All the simulators defined below also choose a random session in the beginning. We use similar notation to denote them.

5. $\mathcal{S}_5^{sid^*}$ is the same as $\mathcal{S}_4^{sid^*}$ with the following difference. $\mathcal{S}_5^{sid^*}$ does not run the Phase 2 extractors for any session except sid^*. (The Phase 1 extractors are run for all sessions.) Note that $\mathcal{S}_4^{sid^*}$ does not use the extracted values from Phase 2 in any other session except sid^*. This is because it neither calculates R_{CRS} nor checks the fail_2 and fail_3 conditions in those sessions. Thus $\mathcal{S}_5^{sid^*}$ and $\mathcal{S}_4^{sid^*}$ behave identically.

6. We would next like to modify $\mathcal{S}_5^{sid^*}$ by having an external machine simulate $\mathcal{P}_a^{sid^*}$ in Phase 1. Namely, rather than having the simulator simulate $\mathcal{P}_a^{sid^*}$ sending a pair $(v_1^{a(sid^*)}, v_2^{a(sid^*)})$ and proving (in the Phase 1 WISPOKs) that it knows a pre-image of one of these values, we would like this to be done by an external machine $\mathcal{E}$. Thus, we would like the simulator to receive a pair $(v_1^{a(sid^*)}, v_2^{a(sid^*)})$ from an external machine $\mathcal{E}$, and have $\mathcal{E}$ provide the corresponding Phase 1 WISPOKs of $\mathcal{P}_a^{sid^*}$. However, we want to avoid rewinding $\mathcal{E}$. (Note that $\mathcal{S}_5^{sid^*}$ runs the extraction subroutine on all the Phase 1 WISPOKs of all sessions. In sessions where the Phase 1 WISPOKs overlap with the Phase 1 WISPOKs of $\mathcal{P}_a^{sid^*}$, the extraction subroutine may need to "rewind" the Phase 1 WISPOKs of $\mathcal{P}_a^{sid^*}$.)

   The natural idea would be to modify $\mathcal{S}_5^{sid^*}$ as follows: For any session sid and for any corrupted party P_j participating in session sid, rather than applying the extraction subroutine to *all* of

[31] Recall that r'_j is the (supposedly committed) value sent by party P_j at the beginning of Phase 3 of this session, and note that (sid, R') is the output of $\mathcal{P}_a^{sid}$ in this session.

the Phase 1 WISPOKs given to $\mathcal{P}_a^{sid}$ by party P_j, apply the extraction subroutine only to one of these WISPOKs: specifically, the one which does not overlap, according to $\mathcal{P}_a^{sid}$ and $\mathcal{P}_a^{sid^*}$, with any of the Phase 1 WISPOKs given by $\mathcal{P}_a^{sid^*}$ in session sid^*.[32] Notice that the existence of such a WISPOK follows from the fact that the scheduling of the Phase 1 WISPOKs is *pairwise disjoint*. Unfortunately, there is no guarantee that it is easy to find this "disjoint" WISPOK.

So, instead we apply the extraction subroutine to all of the Phase 1 WISPOKs. We avoid the "rewinding" of the Phase 1 WISPOKs of $\mathcal{P}_a^{sid^*}$ by modifying the stand-alone provers as follows: rather than using the usual stand-alone prover $\mathcal{Q}_j^{sid}$, we use an alternate stand-alone prover $\mathcal{Q}_j^{sid,sid^*}$, which is the same as $\mathcal{Q}_j^{sid}$, except that $\mathcal{P}_a^{sid^*}$ is modified so that it does not take part in any Phase 1 WISPOK as a prover. We shall denote this new simulator by $\mathcal{S}_6^{sid^*}$.

First, notice that the internal state of $\mathcal{E}$ is not needed to construct $\mathcal{Q}_j^{sid,sid^*}$, since the Phase 1 WISPOKs of $\mathcal{P}_a^{sid^*}$ (given externally by $\mathcal{E}$) are not needed in order to construct $\mathcal{Q}_j^{sid,sid^*}$. Second, the pairwise disjointness property of the scheduling ensures that for every corrupted party P_j there exists at least one Phase 1 WISPOK proven by P_j which does not overlap, according to $\mathcal{P}_a^{sid}$ and $\mathcal{P}_a^{sid^*}$, with any of the Phase 1 WISPOKs proven by $\mathcal{P}_a^{sid^*}$. Recall that the first and the last messages of these WISPOKs are sent by the verifier. This implies that for every corrupted party P_j participating in session sid, there exists a Phase 1 WISPOK (verified by $\mathcal{P}_a^{sid}$) such that $\mathcal{P}_a^{sid^*}$ does not send any prover message between the time that the first and the last messages of this "disjoint" WISPOK were sent. This in turn implies that the stand-alone prover $\mathcal{Q}_j^{sid,sid^*}$, corresponding to the "disjoint" WISPOK, is identical to $\mathcal{Q}_j^{sid}$. Thus, if this "disjoint" WISPOK is accepted with non negligible probability, then its witness will be extracted with overwhelming probability *without rewinding* $\mathcal{E}$.[33] We conclude that throughout its simulation, $\mathcal{S}_6^{sid^*}$ *never* rewinds the Phase 1 WISPOKs proven by $\mathcal{P}_a^{sid^*}$, and if a session sid reaches Phase 3 then with overwhelming probability, $\mathcal{S}_6^{sid^*}$ obtains from the extractor witnesses w^j (such that $f(w^j) \in \{v_1^j, v_2^j\}$) for every corrupted participant P_j.

Note that the only difference between $\mathcal{S}_5^{sid^*}$ and $\mathcal{S}_6^{sid^*}$ is in the way the Phase 1 witnesses of corrupted parties are extracted. Since both $\mathcal{S}_5^{sid^*}$ and $\mathcal{S}_6^{sid^*}$ succeed in extracting these witnesses (with overwhelming probability), for every session that reaches Phase 3, and since these witnesses are used only in Phase 3, we would like to conclude and say that the output distributions of $\mathcal{S}_5^{sid^*}$ and $\mathcal{S}_6^{sid^*}$ are statistically indistinguishable. However, there is a subtle point here: The witnesses w^j obtained by $\mathcal{S}_5^{sid^*}$ and $\mathcal{S}_6^{sid^*}$ may be distributed differently. But, since these witnesses are used only in WIPOKs we conclude that the output distributions of $\mathcal{S}_5^{sid^*}$ and $\mathcal{S}_6^{sid^*}$ are computationally indistinguishable, which in particular implies that the probability with which they output fail_2 and fail_3 is the same (up to a negligible factor). We note that the formal reduction (reducing any algorithm that distinguishes between the outputs of $\mathcal{S}_5^{sid^*}$ and $\mathcal{S}_6^{sid^*}$ to an algorithm that breaks the witness indistinguishability property of the WIPOK) is straightforward, and therefore omitted.

[32] Recall that pairwise disjointness between two sessions sid and sid^* is with respect to two honest parties, one from each session. Here we take $\mathcal{P}_a^{sid}$ to be the honest party in sid and $\mathcal{P}_a^{sid^*}$ to be the honest party in sid^*. This means that $\mathcal{P}_a^{sid}$ does not send any message (as a verifier) in the WISPOK proven by P_j during the WISPOKs proven by $\mathcal{P}_a^{sid^*}$, at least as far as $\mathcal{P}_a^{sid}$ is concerned. Therefore, the verification by $\mathcal{P}_a^{sid}$ of P_j's WISPOK is disjoint from all the proofs of $\mathcal{P}_a^{sid^*}$. It is therefore possible to extract from P_j's WISPOK without rewinding any of $\mathcal{P}_a^{sid^*}$'s proofs.

[33] Notice that a witness to the Phase 2 WISPOK is also extracted without rewinding $\mathcal{E}$. This is due to the fact that the external machine $\mathcal{E}$ is used only to replace Phase 1 of session sid^*, and $\mathcal{S}_6^{sid^*}$ only extracts from Phase 2 in session sid^* (so this extraction comes strictly after the external machine $\mathcal{E}$ terminated).

7. Finally we define $\mathcal{S}_7^{sid^*}$ which replaces the internal simulation of the first message (namely, $(v_1^{a(sid^*)}, v_2^{a(sid^*)})$ and the WISPOKs given by $P_{a(sid^*)}$ in session sid^* by externally received messages. That is, $\mathcal{S}_7^{sid^*}$ interacts with an external machine $\mathcal{E}$ that picks (w_1, w_2), sets $v_i = f(w_i)$, sends them to $\mathcal{S}_7^{sid^*}$, and then engages in multiple WISPOKs to prove knowledge of w such that $f(w) \in \{v_1, v_2\}$. Internally, $\mathcal{S}_7^{sid^*}$ uses this to replace (part of) the computation carried out by $\mathcal{P}_a^{sid^*}$. In other words, the program of $\mathcal{P}_a^{sid^*}$ will be considered split into an external machine $\mathcal{E}$ (which sends $(v_1^{a(sid^*)}, v_2^{a(sid^*)})$ and carries out the proofs of Phase 1) and an internal machine (which carries out the rest of the protocol execution). The extractors will not have access to the state of the external machine. As was mentioned above, both $\mathcal{S}_6^{sid^*}$ and $\mathcal{S}_7^{sid^*}$ do not use the "external" part of the program $\mathcal{P}_a^{sid^*}$. Therefore, the probability that $\mathcal{S}_7^{sid^*}$ outputs fail_2 or fail_3 is the same as the probability that $\mathcal{S}_6^{sid^*}$ does so.

$\mathcal{T}$ is the same as $\mathcal{S}_7^{sid^*}$, with sid^* chosen randomly. The above series of steps shows that if $\mathcal{S}$ outputs fail_2 or fail_3 with non-negligible probability, then $\mathcal{T}$ also outputs fail_2 or fail_3 with non-negligible probability.

Construction of M: We seek to construct a machine M such that, if $\mathcal{T}$ has a non-negligible probability of outputting fail_2 or fail_3 in its interaction with $\mathcal{E}$, then with non-negligible probability, when M interacts with $\mathcal{E}$ it will succeed in extracting w such that $f(w) \in \{v_1, v_2\}$, where (v_1, v_2) is the pair sent to it by $\mathcal{E}$.

The machine M emulates the entire system of honest parties running π and the simulator $\mathcal{T}$. However, it does not simulate $\mathcal{E}$. Instead M itself interacts with $\mathcal{E}$. We construct M separately for the following two cases.

$\mathcal{T}$ outputs fail_2 with non-negligible probability: While emulating the system, if $\mathcal{T}$ outputs fail_2, then M can output the witness w that caused $\mathcal{T}$ to fail. This witness, by definition of fail_2, equals w such that $f(w) \in \{v_1, v_2\}$, where (v_1, v_2) is the first message sent by $\mathcal{E}$. This is in contradiction to the fact that the WISPOKs are witness hiding.

$\mathcal{T}$ outputs fail_3 with non-negligible probability: Recall that $\mathcal{T}$ outputs fail_3 if some P_j sent in the beginning of Phase 3 a value $r'_j \neq r_j$, where r_j was the value extracted in Phase 2. Let sid^* be the random session chosen by $\mathcal{T}$ (i.e., $\mathcal{T}$ is identical to $\mathcal{S}_7^{sid^*}$). In $\mathcal{T}$, when $\mathcal{P}_a^{sid^*}$ enters Phase 3, M will randomly pick a corrupt party P_j and construct a stand-alone prover corresponding to P_j's Phase 3 WIPOK to $\mathcal{P}_a^{sid^*}$. The stand-alone prover is constructed by modifying $\mathcal{P}_a^{sid^*}$ to simply relay messages between P_j and an external verifier. This construction is similar to, but simpler than that of $\mathcal{Q}_j^{sid^*}$ described earlier. Recall that there $\mathcal{Q}_j^{sid^*}$ worked exactly like the simulator, continuing from the point where the extractor was invoked, *except* that $\mathcal{Q}_j^{sid^*}$ (unlike the simulator) did not interact with the honest parties running π and did not invoke the extractors that $\mathcal{S}$ invokes. But now, the stand-alone prover *includes* the honest parties running the π protocol, and also runs all extractors. Note that by running the extractors there is no danger in rewinding $\mathcal{E}$ since $\mathcal{E}$ is not active any more when M reaches Phase 3 of the session sid^*.

Now, M applies a knowledge extractor to this stand alone prover, and if it extracts a witness w such that $f(w) \in \{v_1, v_2\}$, then M outputs w. Note that $\mathcal{T}$ outputs fail_3 when for some party $P_{j'}$, the value $r_{j'}$ extracted in Phase 2 is different from the value $r'_{j'}$ that it sent out in Phase 3, and yet its Phase 3 WIPOK is accepted. Note that the only valid witnesses for this WIPOK are values w such that $f(w) \in \{v_1, v_2\}$. Now, since the probability of $\mathcal{T}$ outputting fail_3 is non-negligible, and since there are only polynomially many (corrupt) parties from which M picked P_j, with non-negligible probability M picked party $P_{j'}$, and thus convinces the external verifier of a statement with the only witnesses being w such that $f(w) \in \{v_1, v_2\}$. This implies that the knowledge extractor when run on

M, will succeed in outputting such a w with non-negligible probability. (This knowledge extractor runs in expected, and not strict, polynomial-time. Nevertheless, using standard arguments, we can obtain a strict polynomial-time machine that obtains w with non-negligible probability.)

Completing the proof for fail_2 and fail_3. This completes the construction of M, and also the proof that $\mathcal{S}$ outputs fail_2 or fail_3 with only negligible probability.

THE HYBRIDS: Above we have shown that $\mathcal{S}$ outputs fail_1, fail_2 or fail_3 only with negligible probability. We now prove that the output distributions of $\mathcal{S}$ and the honest parties running π in the $\mathcal{F}_{\text{CRS}}$-hybrid model are indistinguishable from that of $\mathcal{A}$ and the honest parties running $\pi^{\rho}_{\tau\epsilon}$ in the real world with timing. For this we note that $\mathcal{S}$ in the hybrid world almost perfectly emulates the real world interaction, but with a few differences. The main difference is that in the simulated world in every session sid there is one party $\mathcal{P}^{sid}_a$ that deviates from the protocol. This is the case since the simulator gets a random string R_{CRS} from the functionality and needs to simulate the protocol so that its output will be equal to R_{CRS}.

We shall build some hybrid simulators to bridge the gap between the real and hybrid worlds.

- HYBRID SIMULATOR $\mathcal{H}_1$: This is similar to $\mathcal{S}_2$ as defined earlier: It implements $\mathcal{F}_{\text{CRS}}$ internally and defines R_{CRS} by randomly picking r and setting $R_{\text{CRS}} = \bigoplus_{i\neq a} r_i \oplus r$ (however it outputs fail_1 just like $\mathcal{S}$ does). As argued above, this does not change anything in the system, and in particular the output distributions remain unchanged.

- HYBRID SIMULATOR $\mathcal{H}_2$: Recall that when $\mathcal{P}^{sid}_i$ produces an output (sid, R), $\mathcal{H}_1$ delivers the output (sid, R_{CRS}) from $\mathcal{F}_{\text{CRS}}$ to P_i (after $\mathcal{P}^{sid}_a$ produces an output). In contrast, the simulator $\mathcal{H}_2$ will hand P_i the output R generated by $\mathcal{P}^{sid}_i$ in the simulation. Note that if $\mathcal{P}^{sid}_i$ outputs (sid, R) and $\mathcal{H}_1$ did not output fail_3 (and $\mathcal{P}^{sid}_a$ produced an output) then it must be the case that with overwhelming probability $R = R_{\text{CRS}}$, since the fact that $\mathcal{H}_1$ did not output fail_3 and that $\mathcal{P}^{sid}_a$ produced an output implies that all parties must have sent in Phase 3 the decommitment value which was extracted by the extracted subroutine. Therefore, the output distributions of $\mathcal{H}_1$ and $\mathcal{H}_2$ are statistically close.

- HYBRID SIMULATOR $\mathcal{H}_3$: $\mathcal{H}_3$ is defined exactly as $\mathcal{H}_2$ is, except with the following difference: instead of running $\mathcal{P}^{sid}_a$ in every session sid (with at least one honest player a), $\mathcal{H}_3$ runs another program $\mathcal{P}'^{sid}_a$. This program is exactly like $\mathcal{P}^{sid}_a$, except that in Phase 3, instead of sending r received from $\mathcal{S}$, it sends out r_a as instructed by the honest program of ρ.

  The hiding property of the Phase 2 commitment scheme, the hiding property of the Phase 2 WISPOK, and the fact that r and the committed value r_a are identically distributed (both are uniformly distributed) imply that the output distributions of $\mathcal{H}_2$ and $\mathcal{H}_3$ are computationally indistinguishable.

- HYBRID SIMULATOR $\mathcal{H}_4$: $\mathcal{H}_4$ uses exactly the program specified by ρ for $\mathcal{P}^{sid}_a$. Note that the only difference between $\mathcal{P}'^{sid}_a$ used by $\mathcal{H}_3$ and the program specified by ρ is that while giving Phase 3 WIPOK to a party P_j, $\mathcal{P}'^{sid}_a$ uses the alternate witness provided by $\mathcal{S}$ (namely w^j such that $f(w^j) \in \{v^j_1, v^j_2\}$) instead of what is specified by the protocol ρ. The witness indistinguishable property of this WIPOK implies that the output distributions of $\mathcal{H}_3$ and $\mathcal{H}_4$ are computationally indistinguishable.

Now note that the system run by $\mathcal{H}_4$ and the real world system are identical, except that $\mathcal{H}_4$ also runs the extractors and might output fail depending on the extractor's outputs. Other than that, the extractors are not used in the system (because we replaced the $\mathcal{P}^{sid}_a$ programs by the original programs specified by ρ). Now since $\mathcal{S}$ outputs fail with negligible probability and the output of $\mathcal{H}_4$ is indistinguishable from that of $\mathcal{S}$, we see that $\mathcal{H}_4$ also outputs fail only with negligible probability. Thus, it follows that the output of the system with $\mathcal{H}_4$ is indistinguishable from that of the real world system. From the line of reasoning above, we conclude that the distribution of the output of the system consisting of $\mathcal{S}$ and the honest parties running π in the $\mathcal{F}_{\text{CRS}}$-hybrid world is indistinguishable from the output of the system consisting of $\mathcal{A}$ and the honest parties running $\pi^{\rho}_{\tau\epsilon}$ in the real world (with time).

It remains to show that ρ is a non-trivial protocol. Notice that in ρ an honest party will output a $\mathsf{time\text{-}out}$ message only if a WISPOK takes more than $\tau = \alpha(n)\Delta$ local time units or if the (pairwise-disjoint) schedule instructs it to time-out. Since the WISPOKs consist of $\alpha(n)$ rounds, if the latency of the network is at most Δ (according to *all* local clocks) then each WISPOK will conclude within at most $\tau = \alpha(n)\Delta$ local time units (recall that we assume that local computation is instantaneous). This together with the fact that the schedule used in ρ is non-trivial, implies that ρ is non-trivial. ■

4 Pairwise-Disjoint Scheduling

In this section, we construct a pairwise-disjoint scheduling algorithm, thereby proving Theorem 8 of Section 3.4. On a very high level, the idea is that for each session $sid \in \{0,1\}^m$, the schedule output by $S(\sigma, sid, \Delta, \epsilon)$ is such that protocol σ is executed $m+2$ times, with delays between each execution (here we make use of the timing model). The crux of the idea is that the delays depend on the bits of sid, so that for any $sid \neq sid'$ the executions of $S(\sigma, sid, \Delta, \epsilon)$ and $S(\sigma, sid', \Delta, \epsilon)$ will not be aligned. The schedule is enforced by requiring the parties to "time-out" if the execution is too long, say if it takes more that τ local time units, where τ is a function of σ and Δ (and the delays depend on this parameter τ). In our specific protocol, σ is a strong proof-of-knowledge with $\alpha(n)$ rounds, and we set $\boldsymbol{\tau \stackrel{\text{def}}{=} \alpha(n)\cdot \Delta}$.[34]

Motivation to the schedule. Due to the technical nature of the schedule and its proof, we first provide a lengthy discussion explaining the idea behind the construction. Recall that our aim is to obtain pairwise disjointness, meaning that for every two sessions sid and sid', there exists at least *one* execution of σ in sid that does not overlap with *any* execution of σ in sid'. As a first try, suppose that the schedule consists of running σ twice, with a delay between each execution that is "large" and directly proportionate to the session ID sid. For example, interpret the value $sid \in \{0,1\}^m$ as an integer in the range $[1, \ldots, 2^m]$ and delay $2sid \cdot \tau$ time units between the executions, where τ is an upper bound on how long σ should run. Furthermore, time-out an execution of σ if it runs longer than τ time units. Now, let $sid' \neq sid$ be two different sessions. Denote by σ_1, σ_2 the two executions of σ in session sid, and denote by σ'_1, σ'_2 the two executions of σ in session sid'. Without taking the clock drift ϵ into account for now, we have the following cases:

1. *Execution σ_1 overlaps with execution σ'_1:* Notice that σ_2 is delayed by $2sid \cdot \tau$ time units, whereas σ'_2 is delayed by $2sid' \cdot \tau$ time units. Since $sid' \neq sid$, there is a difference of at least 2τ time units between the delay before σ_2 and the delay before σ'_2. The fact that each execution of σ takes at most τ time units ensures that the σ'_2 execution does not overlap with σ_2. Also, the fact that the delay before σ'_2 is longer than τ time units implies that σ'_2 does not overlap with σ_1.

[34] Note that since σ consists of $\alpha(n)$ rounds and Δ is an upper bound on the latency according to *all* clocks, we have that $\tau \stackrel{\text{def}}{=} \alpha(n)\cdot \Delta$ is an upper bound on σ's overall running time, assuming that all messages are delivered with Δ time units (and assuming local computation is instantaneous).

2. *Execution σ_2 overlaps with execution σ'_2:* The same analysis as above yields that σ'_1 does not overlap with σ_1 or σ_2.

3. *Execution σ_1 overlaps with execution σ'_2:* In this case, it follows immediately that σ'_1 concluded before σ_1 began (because there is a delay of more than τ time units between σ'_1 and σ'_2). Thus, σ'_1 does not overlap with σ_1 or σ_2.

4. *Execution σ_2 overlaps with execution σ'_1:* As above, it follows that σ'_2 does not overlap with σ_1 or σ_2.

We therefore obtain that the above is a pairwise-disjoint schedule. However, this schedule is problematic because the length of the delays are *exponential* in the length of *sid*. Thus, unless there is an *a priori* polynomial bound on the number of sessions (in which case, *sid* can be of length $O(\log n)$), we obtain that the schedule is not polynomial in the security parameter.

We solve this problem by using a more involved scheduling strategy, adapted from the strategy of Chor and Rabin [16]. We now recall this strategy (already described in Section 3.4). It was observed in [16] that if the identifiers *sid* and *sid*$'$ are encoded (one-to-one) into $2m$-bit strings containing m zeros and m ones, then for any two different identifiers $sid \neq sid'$, there is at least one bit position where the encoding of *sid* has a zero and that of *sid*$'$ has a one. Suppose now that the time is divided into $2m$ distinct slots (each slot corresponding to a bit of the encoding of the identifier), and executions of σ in the session *sid* are run only in the slots where the encoding of *sid* has a one in that slot. Then there is a slot in which an execution of σ is run in *sid*$'$, but not in *sid*. The improvement over the previous scheme is that this encoding is compact (i.e., linear), rather than exponential, in the length of *sid*.

However, there are numerous complications in adapting this strategy to our setting. Firstly, unlike the setting considered in [16], we consider executions of ρ occurring in different sessions at different times. Therefore, two encodings which are different may be shifted with respect to each other in a way that all the positions with ones align with each other (e.g. the ones in 0110 and 1100 can be aligned with each other by shifting one of the two strings by one position). This problem is solved simply by prepending a one to the encoding (for convenience in later analysis, we shall actually add a one to both ends of the encoding). We therefore have that the above encodings become 101101 and 111001, respectively, and shifting in either direction will result in independence.

Another problem that arises is due to the fact that in our setting, it is not possible to define distinct time-slots (because the parties' clocks are not synchronized). Therefore, one execution of σ in session *sid* can partially overlap with two executions of σ in session *sid*$'$. We solve this by introducing delays between the time slots in each session. We note that it suffices to delay for at least the maximum time that it takes to conclude an execution of σ. (It is possible to limit the maximum time for any execution of σ by using a time-out instruction.) We thereby obtain that any execution of σ in session *sid* can overlap with *at most one* execution of σ in session *sid*$'$.

The final complication that arises is due to the fact that the parties' local clocks do not proceed at exactly the same rate, but rather can drift. Since the rates at which the local clocks of the different parties proceed may vary adversarially (up to a factor ϵ), it is possible that two different schedules from different sessions may perfectly overlap. For example, suppose that the schedule for session *sid* is $10^i10^j1^k$ and the schedule in *sid*$'$ is $10^j10^i1^k$ (with say $i > j$). Furthermore, suppose that an honest party P is participating in session *sid*, and another honest party P' is participating in session *sid*$'$. Then, the adversary can cause the executions of P and P' to overlap by first running the clock of P faster than that of P' by a factor of i/j (starting after the first execution of σ, up to the second execution of σ), and then running it slower by a factor of j/i (after finishing the second execution of σ and until reaching the third execution of σ).[35] Now, note that although P and P' use the prescribed distinct schedules, the adversary can make every execution of σ in sid fully coincide with every execution of σ in sid'. However, for this to work, it must hold that i/j is less than ϵ. Thus, if we make sure that there are no long runs of zeros in the encoding used, we can use our scheduling for values of ϵ that are reasonably larger than one (but not too large). This explains the somewhat strange looking requirement that ϵ must be less than $\sqrt[3]{1.5}$. The particular encoding we use (which is sometimes called the "Manchester encoding") ensures that there will be at most two consecutive zeros. Our complete description of the schedule, and the formal proof, take all of the above discussed factors into account.

Convention. We assume for simplicity (and without loss of generality) that in protocol σ there exists one party that sends the first message which is of the form "start" and the last message which is of the form "end" to all of the parties that participate in the protocol. This ensures that (when the adversary does not corrupt parties and delivers all messages within time Δ) the duration of the protocol is roughly the same for all parties participating in σ.

The construction. We now present our construction of a pairwise disjoint scheduling. We associate with each session sid a unique session identifier u^{sid} which is a vector of zeros and ones, so that the number of ones is the same for each identifier. Loosely speaking, each 1 entry will correspond to an execution of σ.

Formally, our scheduling algorithm, on input a protocol σ, a session identifier sid, and time-bounds Δ and ϵ, operates as follows. We specify the delay and time-out mechanisms in terms of some parameters d, τ, $\tau_{\text{MIN}}(\cdot)$ and $\tau_{\text{MAX}}(\cdot)$. We shall fix these parameters later, as functions of Δ and ϵ.

1. Associate with session $sid \in \{0,1\}^m$ a vector $u^{sid} = (u_1^{sid}, \ldots, u_{2m+2}^{sid}) \in \{0,1\}^{2m+2}$, defined as follows:

   (a) $u_1^{sid} = 1$ and $u_{2m+2}^{sid} = 1$.

   (b) For every $j \in \{1, \ldots, m\}$, if $sid_j = 1$ then $(u_{2j}^{sid}, u_{2j+1}^{sid}) = (1, 0)$, and if $sid_j = 0$ then $(u_{2j}^{sid}, u_{2j+1}^{sid}) = (0, 1)$.

   Notice that u^{sid} has exactly $m+2$ ones and m zeros. Moreover, it has at most two consecutive zeros. $S(\sigma, sid, \Delta, \epsilon)$ will consist of $m + 2$ executions of σ, one execution corresponding to each 1 entry of the u^{sid} vector.

2. Carry out $m + 2$ executions of σ according to the following scheduling.

   (a) Set $j = 1$.

   (b) If $u_j^{sid} = 1$ then carry out an execution of σ and then continue to step 2c. Otherwise, continue immediately to step 2c

   (c) Wait d local time units (d will be specified later).

   (d) Set $j \stackrel{\text{def}}{=} j + 1$.

   (e) If $j \leq 2m + 2$ then goto step 2b.

[35] We ignore the "delaying slots" between the time slots for this discussion.

3. Time-Out Mechanism: In each of the above executions of σ, each participant checks that no more than τ local time units passed from the time that it received its first message of the execution ("start"), to the time that it received its last message of the execution ("end"). If more time passes before the execution is over, then it outputs $(\mathsf{time\text{-}out}, sid)$ on its output tape and halts the execution.

4. Delay Mechanism: For any $x \in \{0,1,2\}$ and for any two consecutive executions of $S(\sigma, sid)$ that correspond to two 1's with x zeros in between, each party P participating in session sid, checks that δ, denoting the delay (according to P's local clock) between these two executions (i.e., the time between receiving its last message in one execution and receiving its first message in the next execution), is between $\tau_{\mathrm{MIN}}(x)$ and $\tau_{\mathrm{MAX}}(x)$. Here $\tau_{\mathrm{MIN}}(\cdot)$ and $\tau_{\mathrm{MAX}}(\cdot)$ are increasing functions, to be specified later. For each honest participant, if the delay is too short or too long then it outputs $(\mathsf{time\text{-}out}, sid)$ on its output tape and halts the execution.

Theorem 11 *Assume that $1 \leq \epsilon < \sqrt[3]{1.5}$. Then the above scheduling is a non-trivial pairwise-disjoint scheduling, for the following parameters:*

$$\begin{aligned} \tau &\geq \alpha(n) \cdot \Delta \\ d &> (2\tau\epsilon^2 + \Delta(1+\epsilon)\epsilon)/(3 - 2\epsilon^3) \\ \tau_{\mathrm{MIN}}(x) &= (x+1)d/\epsilon - \Delta \\ \tau_{\mathrm{MAX}}(x) &= (x+1)d\epsilon + \Delta \end{aligned}$$

Note that the efficiency of the scheduling depends on ϵ. The closer ϵ is to $\sqrt[3]{1.5}$, the greater the delay is, and the less efficient the scheduling is. (This is due to the $(3 - 2\epsilon^3)$ factor in the denominator of d.)

Proof: First, we collect a few inequalities, which we shall refer to throughout the proof.

$$\tau_{\mathrm{MIN}}(0) > \tau\epsilon \tag{4}$$

$$\tau_{\mathrm{MIN}}(1) > (2\tau + \tau_{\mathrm{MAX}}(0))\epsilon \tag{5}$$

$$\tau_{\mathrm{MIN}}(2) > (2\tau + \tau_{\mathrm{MAX}}(0))\epsilon \tag{6}$$

$$\tau_{\mathrm{MIN}}(2) > (2\tau + \tau_{\mathrm{MAX}}(1))\epsilon \tag{7}$$

We note that these inequalities easily follow from the inequalities listed in the hypothesis.[36]

Assume for the sake of contradiction that S is not a pairwise-disjoint scheduling for some protocol σ, and timing parameters (Δ, ϵ) such that $1 \leq \epsilon < \sqrt[3]{1.5}$. Thus, there exists a concurrent network (in the timing-model), an ϵ-drift preserving adversary, and two distinct sessions sid and sid', such that the following holds. There exist honest parties P and P' participating in sessions sid and sid' respectively, such that according to P and P', every execution of $S(\sigma, sid', \Delta, \epsilon)$ overlaps with at least one of the executions of $S(\sigma, sid, \Delta, \epsilon)$. For simplicity of notation, throughout this proof we denote $S(\sigma, sid, \Delta, \epsilon)$ by Σ, and $S(\sigma, sid', \Delta, \epsilon)$ by Σ'. Further, we shall use "overlaps" as a short hand for "overlaps according to P and P'".

[36]This can be seen as follows. The denominator of the delay d is at most 1 (assuming $1 \leq \epsilon < \sqrt[3]{1.5}$), which implies that $d > 2\tau\epsilon^2 + \Delta(1+\epsilon)\epsilon$. Thus, $\tau_{\mathrm{MIN}}(0) = d/\epsilon - \Delta > (2\tau\epsilon + \Delta(1+\epsilon)) - \Delta = 2\tau\epsilon + \Delta\epsilon > \tau\epsilon$, implying Eq. (4). Next, in order to prove Eq. (5) and Eq. (7) it suffices to prove that $\tau_{\mathrm{MIN}}(x) - \tau_{\mathrm{MAX}}(x-1)\epsilon > 2\tau\epsilon$ (this can be seen by simply manipulating the equations). In order to prove that $\tau_{\mathrm{MIN}}(x) - \tau_{\mathrm{MAX}}(x-1)\epsilon > 2\tau\epsilon$, note that $\tau_{\mathrm{MIN}}(x) - \tau_{\mathrm{MAX}}(x-1)\epsilon = (x+1)d/\epsilon - \Delta - (xd\epsilon + \Delta)\epsilon = d(x/\epsilon - x\epsilon^2 + 1/\epsilon) - \Delta(1+\epsilon)$. Since $1/\epsilon - \epsilon^2 \leq 0$, the latter equality is smallest when $x = 2$. Thus, $\tau_{\mathrm{MIN}}(x) - \tau_{\mathrm{MAX}}(x-1)\epsilon \geq d(2/\epsilon - 2\epsilon^2 + 1/\epsilon) - \Delta(1+\epsilon) = d(3-2\epsilon^3)/\epsilon - \Delta(1+\epsilon) > (2\tau\epsilon + \Delta(1+\epsilon) - \Delta(1+\epsilon) = 2\tau\epsilon$, as desired. Finally, note that Eq. (6) follows immediately from Eq. (7).

We first show that any execution of Σ can overlap with at most one execution of Σ'. This is due to the delay inserted between each execution. More specifically, assume that there is one execution σ in Σ which overlaps with two executions σ'_1 and σ'_2 in Σ'. Then there are two messages of σ that were sent by P such that one was sent out when P' was in the middle of execution of σ'_1 and the other when P' was in the middle of execution of σ'_2. Let the time between sending these two messages be δ as measured by the clock of P, and δ' as measured by the clock of P'. Since the clock drift factor is at most ϵ, we have $\delta' \leq \delta\epsilon$. Note that the executions σ'_1 and σ'_2 are separated by at least $\tau_{\text{MIN}}(0)$ local time units, according to P''s clock. This is the case since otherwise P' would timeout the execution before σ'_2 really started, which would imply that σ does not overlap σ'_2 according to P and P', contradicting our assumption. Thus, the above mentioned messages sent by P must also be separated by at least that much time, i.e., $\delta' \geq \tau_{\text{MIN}}(0)$. Finally, we note that since both the messages were sent out by the honest party P in the same execution, $\delta \leq \tau$. Combining the above relations we get $\tau_{\text{MIN}}(0) \leq \delta' \leq \delta\epsilon \leq \tau\epsilon$. This contradicts Eq. (4).

We thus have that any execution of Σ can overlap with at most one execution of Σ'. Since both schedules carry out exactly $m+2$ executions, every execution of Σ overlaps with exactly one execution of Σ'. Moreover, it must be the case that for every $l \in [m+2]$, the l^{th} execution of Σ overlaps only with the l^{th} execution of Σ'.

Fix any $l \in [m+1]$. Let x' be the number of zeros between the l^{th} one and the $l+1$'st one in $u^{sid'}$. Note that the encoding guarantees that $x' \in \{0,1,2\}$. We prove that the number of zeros between the l^{th} one and the $l+1$'st one in u^{sid} is also x'. This will imply that $u^{sid} = u^{sid'}$, which in turn will imply that $sid = sid'$, contradicting our assumption that sid and sid' are distinct.

Suppose that two (consecutive) executions σ_1 and σ_2 in Σ overlap with two consecutive executions σ'_1 and σ'_2 in Σ' respectively. Let x be the number of zeros between the ones corresponding to σ_1 and σ_2 in u^{sid}. Similarly let x' be the number of zeros between the ones corresponding to σ'_1 and σ'_2 in $u^{sid'}$. We need to show that $x = x'$.

Since σ_1 overlaps with σ'_1, party P must have sent a message in σ_1 while P' was in the middle of σ'_1. Call this the "first event". The "second event" is defined analogously as party P sending a message in σ_2 while P' was in the middle of σ'_2. Let δ denote the duration between these two events according to the clock of P, and δ' the duration between them according to the clock of P'. Then,

$$\delta/\epsilon \leq \delta' \leq \delta\epsilon.$$

Now, since σ_1 and σ_2 are separated by x zeros, and P is an honest party, we are assured that

$$\tau_{\text{MIN}}(x) \leq \delta \leq \tau_{\text{MAX}}(x) + 2\tau.$$

Consider σ'_1 and σ'_2. Recall that P', being honest, checks that each of these executions run for at most τ time units. It also checks that the delay between the last message of σ'_1 and the first message of σ'_2 is in the range $[\tau_{\text{MIN}}(x'), \tau_{\text{MAX}}(x')]$. Note that these checks must be satisfied since otherwise P' would timeout, and thus would not participate in σ'_2. Therefore, σ_2 and σ'_2 would not overlap according to P and P', contradicting our assumption. Since the first and second events occur in the middle of σ'_1 and σ'_2 respectively, we are assured that

$$\tau_{\text{MIN}}(x') \leq \delta' \leq \tau_{\text{MAX}}(x') + 2\tau.$$

The above three displayed inequalities imply

$$\tau_{\text{MIN}}(x) \leq \delta \leq \delta'\epsilon \leq (2\tau + \tau_{\text{MAX}}(x'))\epsilon$$
$$\tau_{\text{MIN}}(x') \leq \delta' \leq \delta\epsilon \leq (2\tau + \tau_{\text{MAX}}(x))\epsilon$$

From these two inequalities we can easily derive contradictions for all the combinations $(x, x') = (1, 0)$, $(x, x') = (2, 0)$, $(x, x') = (0, 1)$, $(x, x') = (2, 1)$, $(x, x') = (0, 2)$ and $(x, x') = (1, 2)$. For instance, setting $(x, x') = (1, 0)$ or $(x, x') = (0, 1)$, we obtain

$$2\tau + \tau_{\text{MAX}}(0) \geq \delta' \geq \delta/\epsilon \geq \tau_{\text{MIN}}(1)/\epsilon$$

which contradicts Eq. (5). Similarly, setting $(x, x') = (2, 0)$ or $(x, x') = (0, 2)$ contradicts Eq. (6), and setting $(x, x') = (2, 1)$ or $(x, x') = (1, 2)$ contradicts Eq. (7). Hence we conclude that $x' = x$, as required.

This shows that the scheduling is indeed pairwise disjoint. It remains to show that it is non-trivial. For this, consider a scheduling Σ being executed in the presence of an adversary who does not corrupt any party and delivers all messages within time Δ by the clocks of *all* the parties. Firstly, since the protocol has $\alpha(n)$ rounds, setting the time-out for an individual execution to be $\tau = \alpha(n) \cdot \Delta$ ensures that no party times-out an execution. We need to also ensure that for every party, the checks on the delays between the executions are also satisfied. Recall our convention that a designated party sends out "start" and "end" messages to every party in the protocol; call this party P. For any two executions σ_1 and σ_2, corresponding to two ones with x zeros in between, party P delays $\delta \stackrel{\text{def}}{=} (x + 1)d$ local time units between the "end" message of σ_1 and the "start" message of σ_2. By the clock of another party P' this duration will be measured as δ', where $\delta/\epsilon \leq \delta' \leq \delta\epsilon$. However P' considers the time at which these two messages reach it (rather than when they were sent). At one extreme, the "end" message may be delivered instantaneously and the subsequent "start" message delivered with a delay of Δ (according to P''s clock), in which case the time between the arrival of the two messages will be $\delta' + \Delta$. At the other extreme, "end" is delayed by Δ, while "start" reaches instantaneously, making the time between the two arrivals $\delta' - \Delta$. Thus, the delay between the two messages will be in the range $[\delta' - \Delta, \delta' + \Delta]$ which is in turn in the range $[\delta/\epsilon - \Delta, \delta\epsilon + \Delta]$. Since $\delta = (x+1)d$, this range is the same as $[\tau_{\text{MIN}}(x), \tau_{\text{MAX}}(x)]$. Thus no party will time-out in the schedule. Also note that m and $O(\alpha(n))$ are bounded by a polynomial. Hence the schedule will be completed in polynomial number of steps and within polynomial number of time units according to any party. Thus the scheduling algorithm is polynomial and non-trivial. ∎

5 Impossibility for Non-Delayed General Composition

In this section, we prove that introducing some element of time into the protocol π (as we did in modifying π into $\pi_{\tau\epsilon}$) is essential for obtaining secure composition. In order to state this result, we first define the notion of a timing-free protocol. Intuitively, such a protocol does not use timing in its instructions. Formally, in our model, a timing-free protocol does not read the clock tape. (The "plain model" in the theorem refers to the model as defined in this paper, without for example, any trusted setup phase.)

Theorem 12 *In the plain model and without an assumed honest majority, there exist probabilistic polynomial-time functionalities that cannot be securely computed (by a non-trivial protocol) under concurrent general composition with timing-free protocols, even in the (Δ, ϵ)-timing model, for* any *Δ and* any *$\epsilon \geq 1$.*

We prove this theorem by showing that for every protocol ρ in the timing model, if ρ is secure under concurrent general composition with timing-free protocols, then it can be modified to become secure under *1-bounded parallel general composition* in a model with no timing. (In the setting of 1-bounded parallel general composition, a secure protocol ρ is executed *once* in parallel with an arbitrary protocol π.) This suffices for proving Theorem 12 because impossibility of this case is proven explicitly in [34]. As in [34], we also limit ourselves to 2-party protocols.

The intuition behind the proof of Theorem 12 is as follows. If a secure protocol ρ is run together with a timing-free protocol π, then this means that the adversary has *full control* over the scheduling of the messages of π. Now, consider a single execution of ρ together with π. Since the adversary can schedule π-messages as it wishes, it can force π to run perfectly in parallel with ρ. Notice that this holds irrespective of the timing instructions used in ρ. We conclude that ρ must remain secure when run in parallel with an arbitrary protocol π, in contradiction to the impossibility results of [34]. We now proceed to the formal proof.

Proof: Let $\Delta \geq 1$ and $\epsilon \geq 1$ be any values, and let ρ be a 2-party protocol that securely realizes a functionality $\mathcal{F}$ under concurrent general composition with timing-free protocols, in the (Δ, ϵ)-timing model.[37] Denote the participating parties by P_1 and P_2.

We now construct a modified protocol ρ' that is timing-free. In ρ', instead of using the clock, the parties simulate the clock themselves by incrementing a counter *on each activation* (this counter is initialized to 0). This simulated clock is then made available to Protocol ρ (or more precisely, to the computation specified by Protocol ρ). Note that ρ' consists of two components: a clock simulation protocol and the original protocol ρ in the timing-model.

We now show that if ρ is non-trivial and secure under concurrent general composition in the timing model, then ρ' is non-trivial and secure under 1-bounded *parallel* general composition (in the timing-free model).[38] We note that if the adversary in the timing-free model activates the same party multiple times before activating the other party, then in ρ' the simulated clocks would have an unavoidable drift. This is problematic because in this case ρ does not give any guarantee of security. However, we consider *parallel* general composition for ρ'. In this setting, the adversary strictly alternates between activating P_1 and P_2. Furthermore, in the $i+1^{\text{th}}$ activation of a party, the adversary delivers it the i^{th}-round message from ρ' and the i^{th}-round message from π (where π is the arbitrary protocol running concurrently with ρ'). We call such an adversary for the parallel setting a *round-robin adversary*. The formal arguments are given in the proof of the following claim.

Claim 13 *Let ρ be a two-party protocol and let $\Delta, \epsilon \geq 1$ be any values. If ρ is non-trivial and securely realizes a functionality $\mathcal{F}$ under concurrent general composition in the (Δ, ϵ)-timing model (even when run concurrently with timing-free protocols), then ρ' as described above is a non-trivial protocol that securely realizes $\mathcal{F}$ under 1-bounded parallel general composition in the timing-free model.*

Proof: Let π be an arbitrary timing-free two-party protocol. In order to prove the security claim on ρ', we need to show that for any given *round-robin adversary*, there exists a simulator $\mathcal{S}$ such that the output distribution of $\mathcal{A}$ and the honest parties running π and ρ' in the real model is computationally indistinguishable from the output distribution of $\mathcal{S}$ and the honest parties running π with ideal access to $\mathcal{F}$ in the $\mathcal{F}$-hybrid model. In order to construct $\mathcal{S}$, we first we show an intermediate adversary $\mathcal{H}$ (who interacts with the parties running ρ in the timing model) such

[37] We stress that a contradiction will be derived for *any* choice of $\Delta, \epsilon \geq 1$. Note that $\epsilon \geq 1$ by definition, and that $\Delta \geq 1$ is the smallest increment possible.

[38] The notion of non-trivial protocols has also been considered in the timing-free model since the trivial protocol that just hangs and never generates output securely realizes all functionalities. Therefore, as in the timing model, only non-trivial protocols are of interest. In the timing-free model, a protocol is called non-trivial if output is guaranteed in the event that the adversary corrupts no parties and (eventually) delivers all messages. As expected, the impossibility results of [34] for parallel general composition hold only for non-trivial protocols.

that the output distributions of the adversary and honest parties in the following two scenarios are identical:

- *Scenario A:* The honest parties and the adversary $\mathcal{A}$ run π and ρ' in the timing-free (real) model.
- *Scenario B:* The honest parties and the adversary $\mathcal{H}$ run π and ρ in the real model with time.

We now describe the adversary $\mathcal{H}$ in the timing model. $\mathcal{H}$ internally invokes $\mathcal{A}$ and perfectly emulates all of $\mathcal{A}$'s actions. This means that $\mathcal{H}$ delivers messages whenever $\mathcal{A}$ does (thereby activating the recipients) and passes $\mathcal{A}$ the messages that it receives. In addition to this emulation, $\mathcal{H}$ needs to increment the clocks of the honest parties (because $\mathcal{H}$ works in the timing model, unlike $\mathcal{A}$). This is carried out simply by having $\mathcal{H}$ increment the clocks of all honest parties by 1 at the beginning of each round-robin round.

Before proceeding, we show that the outputs of the honest parties and adversaries are identical in scenarios A and B, described above. This follows from the fact that in both scenarios, the clock of each party is incremented by 1 between every activation. Furthermore, $\mathcal{H}$ carries out exactly the same actions as $\mathcal{A}$. (The only difference is that in scenario A, the clocks are updated in sequence upon each activation, whereas in scenario B, they are all updated together. However, since parties only read their clocks upon activation, this is exactly the same.) We therefore have that for every round-robin adversary $\mathcal{A}$ in the timing-free real model with π and ρ' there exists an adversary $\mathcal{H}$ in the timing model with π and ρ such that the output distributions in both cases are identical.

Next, notice that as long as $\mathcal{H}$ is ϵ-drift preserving, the assumed security of ρ implies that there exists a simulator $\mathcal{S}$ such that the output distribution of an execution with $\mathcal{S}$ and the honest parties running π in the $\mathcal{F}$-hybrid model is indistinguishable from an execution with $\mathcal{H}$ and the honest parties running π and ρ in the real timing model. This suffices because $\mathcal{H}$ satisfies the drift condition for any ϵ (notice that the clocks of all the honest parties are always the same). Combining the above two steps, we obtain that ρ' securely realizes $\mathcal{F}$ under one-bounded parallel general composition.

To complete the proof of the claim, we shall show that if ρ is non-trivial then so is ρ'. Recall that ρ' is non-trivial (in the timing-free model) if in the case that $\mathcal{A}$ corrupts no parties and delivers all messages, then all parties receive output. In order to see that this holds, first recall that $\mathcal{H}$ essentially just emulates $\mathcal{A}$. Therefore, if $\mathcal{A}$ corrupts no parties, then so does $\mathcal{H}$. Furthermore, by the assumption that $\mathcal{A}$ is a round-robin adversary, we know that it *always* delivers all messages immediately (i.e., all round i messages are received in round $i+1$). Therefore, $\mathcal{H}$ delivers all messages within time $\Delta = 1$. Finally, as we have shown above, $\mathcal{H}$ is always ϵ-drift preserving (for any $\epsilon \geq 1$). We conclude that in an execution of ρ' in which $\mathcal{A}$ does not corrupt any parties, the analogous execution of ρ with $\mathcal{H}$ is such that $\mathcal{H}$ corrupts no parties, is ϵ-drift preserving and delivers all messages within time $\Delta = 1$. Therefore, by Definition 3 and the assumption that ρ is non-trivial, we have that in this execution of ρ with $\mathcal{H}$, the honest parties all obtain their output (and this output does not equal time-out). By the equivalence between scenarios A and B above, we obtain that in the execution of ρ' with $\mathcal{A}$, the parties also all receive output. That is, ρ' is non-trivial. This completes the proof of non-triviality and of the claim. ■

As we have mentioned above, the proof of the theorem follows immediately from the above claim and the impossibility results for 1-bounded parallel general composition (in the timing-free model) as proven in [34]. ■

Remark. Theorem 12 states that there *exist* functionalities that cannot be securely computed under concurrent general composition with timing-free protocols. However, the proof actually shows that this setting inherits all of the impossibility results of [34], which are in turn inherited from [13]. Thus, we actually obtain very broad impossibility results that hold for large classes of functionalities.

Acknowledgements

We would like to thank Ran Canetti, Oded Goldreich, Shafi Goldwasser and Shai Halevi for helpful discussions and comments.

References

[1] B. Barak. How to Go Beyond the Black-Box Simulation Barrier. In 42*nd FOCS*, pages 106–115, 2001.

[2] B. Barak, R. Canetti, J. B. Nielsen, R. Pass. Universally Composable Protocols with Relaxed Set-Up Assumptions. In 45*th FOCS*, pages 186-195, 2004.

[3] B. Barak, Y. Lindell and S. Vadhan. Lower Bounds for Non-Black-Box Zero-Knowledge. In 44*th FOCS,* pages 384–393, 2003.

[4] D. Beaver. Foundations of Secure Interactive Computing. In *CRYPTO'91*, Springer-Verlag (LNCS 576), pages 377–391, 1991.

[5] M. Bellare and O. Goldreich. On Defining Proofs of Knowledge. In *CRYPTO'92*, Springer-Verlag (LNCS 740), pages 390–420, 1992.

[6] M. Ben-Or, S. Goldwasser and A. Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. In 20*th STOC,* pages 1–10, 1988.

[7] M. Blum. Coin Flipping by Phone. *IEEE Spring COMPCOM*, pages 133–137, 1982.

[8] M. Blum. How to Prove a Theorem So No One Else Can Claim It. *Proceedings of the International Congress of Mathematicians*, pages 1444–1451, USA, 1987.

[9] R. Canetti. Security and Composition of Multiparty Cryptographic Protocols. *Theory of Cryptography Library*, Record 98-18, version of June 4th, 1998 (later versions do not contain the referenced material).

[10] R. Canetti. Security and Composition of Multiparty Cryptographic Protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

[11] R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In 42*nd FOCS*, pages 136–145, 2001.

[12] R. Canetti and M. Fischlin. Universally Composable Commitments. In *CRYPTO 2001*, Springer-Verlag (LNCS 2139), pages 19–40, 2001.

[13] R. Canetti, E. Kushilevitz and Y. Lindell. On the Limitations of Universal Composable Two-Party Computation Without Set-Up Assumptions. In *EUROCRYPT'03,* Springer-Verlag (LNCS 2656), pages 68–86, 2003.

[14] R. Canetti, Y. Lindell, R. Ostrovsky and A. Sahai. Universally Composable Two-Party and multiparty Computation. In 34*th STOC*, pages 494–503, 2002.

[15] D. Chaum, C. Crepeau and I. Damgard. multiparty Unconditionally Secure Protocols. In 20*th STOC*, pages 11–19, 1988.

[16] B. Chor and M. Rabin. Achieving Independence in Logarithmic Number of Rounds. In 6*th PODC*, pages 260–268, 1987.

[17] Y. Dodis and S. Micali. Parallel Reducibility for Information-Theoretically Secure Computation. In *CRYPTO 2000*, Springer-Verlag (LNCS 1880), pages 74–92, 2000.

[18] D. Dolev, C. Dwork and M. Naor. Non-Malleable Cryptography. *SIAM Journal on Computing,* 30(2):391–437, 2000.

[19] C. Dwork, M. Naor, and A. Sahai. Concurrent Zero-Knowledge. *Journal of the ACM*, 51(6):851–898, 2004.

[20] C. Dwork and M. Naor. Zaps and Their Applications. In *41st FOCS*, pages 283–293, 2000.

[21] C. Dwork and A. Sahai. Concurrent Zero-Knowledge: Reducing the Need for Timing Constraints. In *CRYPTO'98*, Springer-Verlag (LNCS 1462), pages 442–457, 1998.

[22] U. Feige and A. Shamir. Zero-Knowledge Proofs of Knowledge in Two Rounds. In *CRYPTO'89*, Springer-Verlag (LNCS 435), pages 526–544, 1989.

[23] U. Feige and A. Shamir. Witness Indistinguishability and Witness Hiding Protocols. In *22nd STOC*, pages 416–426, 1990.

[24] O. Goldreich. *Foundations of Cryptography: Volume 1 – Basic Tools.* Cambridge University Press, 2001.

[25] O. Goldreich. *Foundations of Cryptography: Volume 2 – Basic Applications.* Cambridge University Press, 2004.

[26] O. Goldreich. Concurrent Zero-Knowledge With Timing Revisited. In 34*th STOC*, pages 332–340, 2002.

[27] O. Goldreich, S. Micali and A. Wigderson. How to Play any Mental Game – A Completeness Theorem for Protocols with Honest Majority. In 19*th STOC,* pages 218–229, 1987.

[28] S. Goldwasser and L. Levin. Fair Computation of General Functions in Presence of Immoral Majority. In *CRYPTO'90,* Springer-Verlag (LNCS 537), pages 77–93, 1990.

[29] S. Goldwasser and Y. Lindell. Secure Computation Without Agreement. *Journal of Cryptology,* 18(3):247–287, 2005.

[30] S. Goldwasser, S. Micali and C. Rackoff The Knowledge Complexity of Interactive Proof Systems. *SIAM Journal on Computing,* 18(1):186–208, 1989.

[31] J. Katz. Efficient and Non-malleable Proofs of Plaintext Knowledge and Applications. In *EUROCRYPT 2003*, Springer-Verlag (LNCS 2656), pages 211–228, 2003.

[32] Y. Lindell. Parallel Coin-Tossing and Constant-Round Secure Two-Party Computation. *Journal of Cryptology*, 16(3):143–184, 2003.

[33] Y. Lindell. Bounded-Concurrent Secure Two-Party Computation Without Setup Assumptions. In 35*th STOC*, pages 683–692, 2003.

[34] Y. Lindell. General Composition and Universal Composability in Secure multiparty Computation. In 44*th FOCS*, pages 394–403, 2003.

[35] Y. Lindell. Lower Bounds for Concurrent Self Composition. In the *1st Theory of Cryptography Conference* (TCC), Springer-Verlag (LNCS 2951), pages 203–222, 2004.

[36] S. Micali and P. Rogaway. Secure Computation. Unpublished manuscript, 1992. Preliminary version in *CRYPTO'91*, Springer-Verlag (LNCS 576), pages 392–404, 1991.

[37] M. Naor. Bit Commitment using Pseudorandom Generators. *Journal of Cryptology*, 4(2):151–158, 1991.

[38] R. Pass. Simulation in Quasi-Polynomial Time, and Its Application to Protocol Composition. In *Eurocrypt 2003*, Springer-Verlag (LNCS 2656), pages 160–176, 2003.

[39] R. Pass. Bounded-Concurrent Secure Multiparty Computation with a Dishonest Majority. In the 36*th STOC,* pages 232–241, 2004.

[40] R. Pass and A. Rosen Bounded-Concurrent Secure Two-Party Computation in a Constant Number of Rounds. In 44*th FOCS*, pages 404–413, 2003.

[41] B. Pfitzmann and M. Waidner. Composition and Integrity Preservation of Secure Reactive Systems. In *7th ACM Conference on Computer and Communication Security*, pages 245–254, 2000.

[42] M. Prabhakaran and A. Sahai. New Notions of Security: Universal Composability Without Trusted Setup. In 36*th STOC,* pages 242–251, 2004.

[43] R. Richardson and J. Kilian. On the Concurrent Composition of Zero-Knowledge Proofs. In *EUROCRYPT'99*, Springer-Verlag (LNCS 1592), pages 415–431, 1999.

[44] A. Yao. How to Generate and Exchange Secrets. In 27*th FOCS*, pages 162–167, 1986.