Concurrent General 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

February 10, 2005

Abstract

Broad impossibility results have recently been proven regarding the feasibility of obtaining protocols that remain secure under concurrent composition, unless an honest majority or trusted setup phase are assumed. 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). One approach for bypassing these impossibility results is to consider more limited settings of concurrency that still realistically model real-world networks. In this paper, we investigate the feasibility of obtaining secure multiparty protocols in a network where certain *time bounds* are assumed. Specifically, the security of our protocols rely on the very reasonable assumption that local clocks do not "drift" too much (i.e., proceed at approximately the same rate). We show that under this timing assumption, it is possible to securely compute any *multi-party* functionality under concurrent general composition (as long as messages from the arbitrary other protocols are delayed for a specified amount of time).

Keywords: theory of cryptography, secure multi-party computation, concurrent composition, timing assumptions.

*Much of this work was carried out while the authors were all at IBM T.J.Watson Research, New York.

1 Introduction

In modern network settings, protocols must remain secure even when many protocol executions take place concurrently. Recent impossibility results have shown that in the case of *no honest majority* and *no trusted setup*, large classes of functions cannot be securely computed under concurrent composition [10, 9, 11, 29, 30]. These results hold for both concurrent general composition (where a secure protocol is run concurrently with arbitrary other protocols) and concurrent self composition (where a single secure protocol is run many times concurrently). In fact, these two types of composition have been shown to be (almost) equivalent [30]. One way of possibly overcoming these impossibility results is to introduce assumptions on the adversary or network behavior. Needless to say, it is best to not assume any restriction whatsoever. However, as we have mentioned, this is not possible. We therefore aim to construct protocols that remain secure under *reasonable* network assumptions. In this paper, we consider a very reasonable network restriction that holds in real networks.

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 [16] who used them to achieve (efficient) zero-knowledge protocols that remain secure under concurrent self composition. An equivalent formulation of these assumptions was used by [22] and our presentation is more according to this latter formulation. This formulation involves two specific assumptions:

- *Assumption 1 – 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 distance of the parties' local clocks from each other; such an assumption would be far more problematic.
- *Assumption 2 – maximum latency:* It is also 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 will 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. (We note that attacks that significantly slow down a network are actually very easy to carry out.) In contrast, local clocks are usually very accurate, at least with respect to the drift. The only policy that needs to be enforced is one that prevents a machine's local clock from being modified by some external instruction, that may then be issued maliciously by an adversary.

Motivated by this above observation, we relate to these assumptions differently.[1] More specifically, our definition of security for the timing model relies *only* on the first assumption regarding clock drift. Therefore, security holds as long as the clock drifts of the clocks are not too far apart, and *irrespective of the network latency.* In contrast, the latency assumption is only used to ensure

[1]We remark that this distinction also appears in [16], who discuss the ramifications of the timing assumptions on completeness for zero-knowledge proofs.

liveness (or non-triviality of protocols). Namely, we require only that if a protocol execution does not come under attack and the latency is indeed lower than Δ, then the protocol concludes successfully. Finally, we require that if a protocol does not come under attack and fails to conclude only because of the network latency, then the protocol can be re-executed without harming security.[2] We guarantee this by defining a class of functionalities with the property that if the parties halt the execution due to a "time-out," then this occurs independently of the output, and in particular, before the adversary receives its own output. We argue that this property is essential for any realistic use of protocols in the timing model. In particular, given this property it is possible to choose a reasonable Δ that is not too large; if a time-out occurs due to a higher-than-average latence, then no damage is caused and the parties can just restart the protocol and try again.

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 abort the execution" (where f is a function specified by the protocol). A delay command is of the form: "before sending message y, wait until $g(\Delta, \epsilon)$ time units have passed since receiving message x". 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 that if part A of execution i begins after part B of execution j, then part B of execution j concludes 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 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.

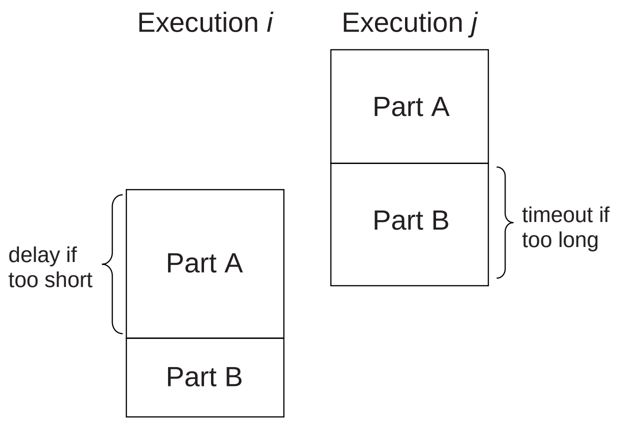

Figure 1: limiting the interleaving (notice that part A must take longer than part B).

Our results. We show that under timing assumptions, for every functionality $\mathcal{F}$ there is a multiparty protocol ρ that securely computes $\mathcal{F}$, such that ρ remains secure under concurrent general composition (with the additional requirement that delay instructions are introduced into the arbitrary protocols that run concurrently to ρ). In order to state our result in more detail, we

[2]This property is non-trivial and does not generally hold for secure protocols. That is, if a protocol is *aborted* due to malicious behavior, then re-executing the protocol from scratch can result in a breach of security. For example, if the 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.

introduce the following notation. Let π be an arbitrary protocol (representing arbitrary protocol activity in the network where our secure protocol is run), and let $\mathsf{delay}(\pi)$ denote a modification in which each message of π is delayed by a specified amount. Then, we show that for every functionality $\mathcal{F}$ there exists a protocol ρ for securely computing $\mathcal{F}$, such that for every π, protocol ρ remains secure when run many times concurrently with $\mathsf{delay}(\pi)$ in the timing model. We call such a protocol secure under concurrent general composition with delays. We stress that the contents of the messages specified by π are untouched; therefore, any protocol in π that was secure (without timing) will remain secure in $\mathsf{delay}(\pi)$. We now informally state our main result:

Theorem 1.1 *Assume that there exist enhanced trapdoor permutations.*[3] *Then, any multiparty functionality $\mathcal{F}$ can be securely computed under concurrent general composition with delays, in the timing model and 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 the most general setting of composition, without relying on a trusted setup phase or an assumed honest majority.

We prove Theorem 1.1 by constructing a protocol that securely realizes the *common random string 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 [12]. Combining these together, we obtain that any efficient functionality can be securely computed under concurrent general composition with delays. Of course, since we obtain concurrent general composition, we can also combine different secure protocols, obtaining that any set of functionalities can be securely realized concurrently in a "delayed network".

Inherent drawbacks. As we have mentioned, the timing assumptions are used for introducing time-out and delay instructions in the protocol. Our use of delays is extensive, since we delay all messages of the arbitrary protocol π. This is clearly a drawback of our result. However, in Section 5 we show that some sort of modification of π is *essential* for achieving security. Informally speaking, we say that a protocol π is timing-free if it contains no time-based instructions. We prove the following theorem (again, informally stated here):

Theorem 1.2 *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. We view our result as an initial *feasibility result* that demonstrates the usefulness of timing assumptions even in the very difficult setting of concurrent general composition.

We note an important feature of our protocol relating to efficiency. As we have mentioned, the assumption regarding the maximum latency Δ is used only for obtaining non-trivial termination. Furthermore, if the latency exceeds Δ in any given execution, then the protocol can be restarted with no danger to security. Therefore, it is not necessary to set Δ to be an upper-bound on the latency. Rather, a more optimistic estimate on the latency for Δ can be taken with the price being that some, but not too many, time-outs will be incurred. (There is a tradeoff here between

[3]See [21, Appendix C.1].

choosing a large Δ that slows down all protocol executions and choosing a small Δ that results in more timed-out protocols that must be restarted.)

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 [39, 23, 5, 13]. The study of concurrent composition of protocols was initiated by [18] in the context of witness indistinguishability, and was next considered by [15] in the context of non-malleability. Until recently, the majority of work on concurrent composition was in the context of concurrent zero-knowledge [16, 38]. 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 [36] for the case that a secure protocol is executed once in an arbitrary network.[4] The general case, 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 [9]. It was also shown that any functionality can be securely realized in this setting assuming an honest majority [9], or assuming a trusted setup phase in the form of a common random string [12]. 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 [11, 29, 30].

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 [28, 35, 34] and lower bounds [28, 30] have been demonstrated. In our opinion, the timing model is a far 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 [33] and [37] who both provide "additional power" to the " ideal adversary". In [33], the simulator (or ideal adversary) is allowed to run in quasi-polynomial time, even though the real adversary is still limited to polynomial-time. Thus, the resulting security guarantee is that "whatever can be obtained in polynomial-time in a real execution could be obtained in quasi-polynomial-time in an ideal execution". This definition was used by [33] to construct zero-knowledge protocols that remain secure under different settings of concurrent composition. In a similarly motivated definition, [37] allow the ideal adversary to run in exponential-time and simulate for a polynomial-time real adversary. In this setting, and under somewhat non-standard assumptions, [37] show how to construct protocols that are secure (for their new notion of security) under concurrent general composition. Thus, this latter result achieves security for the broadest setting of composition. However, it does this at the price of obtaining a weaker security guarantee. (In particular, it is not clear what effect a protocol that is secure by the definition of [37] will have on other protocols running in the arbitrary network.)

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 $\mu(n) < 1/p(n)$.

[4] An earlier reference to this problem with general ideas about how to define security appeared in [7, Appendix A].

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 $X \overset{\mathrm{c}}{\equiv} Y$, 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.)[5]

2.2 Security Under General Composition

In this section, we present the definitions of the network model, and security under concurrent general composition. In order to simplify the exposition as much as possible, we do not attempt to present the most general model. Rather, we focus on a model that suffices for our specific result. As we have described in the Introduction, we show how the common random string functionality $\mathcal{F}_{\text{CRS}}$ can be securely computed under concurrent general composition in the timing model. More specifically, we show that it is possible to construct a protocol ρ such that an execution of π with ρ in a real world with clocks (and limitations on their drift) can be simulated in a hybrid world with π and ideal access to the common random string functionality $\mathcal{F}_{\text{CRS}}$ (and without any clocks). Since timing is needed only in the real model (and not in the resulting $\mathcal{F}_{\text{CRS}}$-hybrid world), we introduce it only into the definition of the real model.

We now define what it means for a protocol to be secure under concurrent general composition. 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 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.

Multi-party computation. A multi-party computation problem for a set of parties $P_1, \ldots, P_m$ is cast by specifying a (probabilistic polynomial-time) multi-party ideal functionality machine $\mathcal{F}$ that

[5]We note that modelling polynomial-time like this somewhat restricts generality. For example, encryption of arbitrarily long plaintexts by honest parties cannot be modelled in this way. Nevertheless, this is simpler and suffices for this work.

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 [24, 3, 31, 8], 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, this intuition is formalized by having the parties (and adversary) communicate directly with the ideal functionality. Thus, sending a message to the ideal functionality means that a Turing machine computing $\mathcal{F}$ receives this message on its incoming communication tape. The output generated by this machine is written on its outgoing communication tape to be sent to the party for whom it is designated. (This is equivalent to the trusted party formulation but makes for a less verbose presentation.)

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, in the real model with clocks, the adversary has (some) control over the clocks of the honest parties; this will be described below.

The $\mathcal{F}$-hybrid model. Let π be an arbitrary protocol that utilizes ideal interaction with a trusted party computing the multi-party functionality $\mathcal{F}$ (recall that π actually models arbitrary network activity). This means that π 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 an ideal functionality $\mathcal{F}$, or from an ideal functionality to a participating party (or the adversary). Note that there may be many copies of one functionality, and so these copies are differentiated by unique session identifiers. 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 we have mentioned, 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.)[6]

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 exe-

[6]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.

cution, 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.[7] 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 – without clocks. Let ρ be a multi-party 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 separate probabilistic interactive Turing machine (ITM) that works according to the specification of the protocol ρ.[8] When π instructs a party to send an ideal message α to the 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 marked 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. (Note that 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.

The real model – with clocks. Until now, we have defined the standard (timing-free) real and hybrid models for concurrent general composition. However, in this work, we consider the timing model where the parties have clocks. We only introduce timing in the real model, since it suffices

[7]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.

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

for our results. We note that timing can be introduced in the $\mathcal{F}$-hybrid model in a straightforward way if desired.

In order to model parties with 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 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, then this is equivalent to an un-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.[9] In the timing model, as introduced by [16], 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 by modifying the real model in the following way:

1. Before the computation begins, the real adversary $\mathcal{A}$ 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.

2. Every time that the adversary is activated, it is given write access to the clock tapes of all of the parties. This write access is limited in a natural way in that $\mathcal{A}$ 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 $\mathcal{A}$ 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 $\mathcal{A}$ is ϵ-drift preserving if there exist a series of values $\delta_1, \delta_2, \ldots$ so that for every i and every $k = 1, 2, \ldots$

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

[9] 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.

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 $\delta_k\epsilon$.[10]

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 rest of the execution is the same as in the (untimed) real execution described above. Let n be the security parameter, let $\mathcal{A}$ be an ϵ-drift preserving 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 $\text{REAL}^{\epsilon}_{\pi^\rho,\mathcal{A},I}(n,\overline{x},z)$, is defined as the output vector of all the parties and $\mathcal{A}$ from the above real execution.

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. Loosely speaking, the definition asserts that for any context, or calling protocol π, the real execution of π^ρ emulates the hybrid execution of π which utilizes 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. The fact that the above emulation must hold for *every* protocol π that utilizes ideal calls to $\mathcal{F}$, means that *general composition* is being considered (recall that π represents arbitrary network activity).

In addition to considering the notion of general composition where *every* possible protocol π is considered, we also consider the case where π comes from some class of protocols Π. This formalization will be used later when we show security with respect to a generic transformation of any protocol π to be within some class Π; security is then preserved with respect to the transformed protocol in Π. (Jumping ahead, this transformation is just the introduction of delays of a specified length into the protocol representing the arbitrary network activity.)

Definition 1 (security under concurrent general 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 general composition in the timing model with ϵ *if for every polynomial-time protocol π in the $\mathcal{F}$-hybrid model and every ϵ-drift preserving non-uniform polynomial-time real-model adversary $\mathcal{A}$ for π^ρ, there exists a probabilistic non-uniform polynomial-time hybrid-model adversary $\mathcal{S}$ such that for every $I \subseteq [m]$:*

$$\left\{\text{HYBRID}^{\mathcal{F}}_{\pi,\mathcal{S},I}(n,\overline{x},z)\right\}_{n\in\mathsf{N};\overline{x}\in(\{0,1\}^*)^m;z\in\{0,1\}^*} \overset{\text{c}}{\equiv} \left\{\text{REAL}^{\epsilon}_{\pi^\rho,\mathcal{A},I}(n,\overline{x},z)\right\}_{n\in\mathsf{N};\overline{x}\in(\{0,1\}^*)^m;z\in\{0,1\}^*}$$

If the above holds for a specified subset of protocols Π, then we say that ρ securely realizes $\mathcal{F}$ under concurrent composition with Π 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 should rely solely on the more realistic 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 output is generated in executions where the adversary is "well-behaved". More specifically, and 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.

[10]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, $\delta_k \leq \mathsf{clock}_i(a_k) - \mathsf{clock}_i(a_{k-1}) \leq \delta_k\epsilon$.

Definition 2 (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.

Notice that item (2) in Definition 2 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 protocol (for the real model with clocks) utilizes the clocks by introducing delay and time-out instructions. Such instructions are formally modelled 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 $(x, \textit{delay-id}, c, \textit{time})$ 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 on its clock (i.e., that the current contents of its clock is at least $\textit{time} + c$ 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 ITM that it runs as a subprotocol) is instructed to time-out if it doesn't receive a specific message within c time units, then P_i writes the current contents of its clock tape on its work tape. Then, when it receives the specific message, it simply times 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 The Class of Timed Functionalities

As we have discussed above, in the timing model parties may output a special time-out message, indicating that some message was not received within a specified time. Thus, protocols that are secure in the timing model may sometimes conclude with the parties outputting time-out rather than their correct output. This possibility must therefore be included into the definition of the ideal functionality (because a real execution is supposed to look just like an interaction with an ideal functionality in the ideal world). In addition, as we discussed in the Introduction, we require that the parties only output time-out before the functionality generates output. This enables the parties to restart a protocol that timed-out, without any security risk. (This is important because it enables us to take an *average* rather than worst-case bound on Δ, and thereby balance the cost of delays with the cost of restarting timed-out protocols.)

The above leads us to define a *class of functionalities* that captures the above properties of functionalities designed for the timing model. Our definition of this class is actually a transformation, meaning that every functionality $\mathcal{F}$ has an analogous functionality $\mathcal{F}^{timed}$ in the class. The only difference between $\mathcal{F}$ and $\mathcal{F}^{timed}$ is that $\mathcal{F}^{timed}$ waits to receive either a time-out or compute message from the adversary before carrying out any computation. If time-out is received, then $\mathcal{F}^{timed}$ just sends time-out to all the parties and halts. Otherwise, if compute is received, $\mathcal{F}^{timed}$ invokes the original $\mathcal{F}$. (In this case, any messages that $\mathcal{F}^{timed}$ received before the adversary sent a compute message are also forwarded by $\mathcal{F}^{timed}$ to the original $\mathcal{F}$.) We therefore have that unless a time-out is issued, the functionality $\mathcal{F}^{timed}$ is exactly the same as $\mathcal{F}$. Furthermore, if a time-out *is* issued, then it is guaranteed that $\mathcal{F}^{timed}$ did not generate any output, as desired.

The class of "timed" functionalities. Let $\mathcal{F}$ be a functionality. Then, the timed functionality $\mathcal{F}^{timed}$ consists of an external procedure called the shell, and a main subroutine called the core. The core is the probabilistic polynomial-time algorithm computing $\mathcal{F}$, while the shell is a simple procedure that deals with time-out and compute messages from the adversary. The shell has an external interface and receives messages from the parties and the adversary. Until a time-out or compute message is received, the shell simply stores all messages received in a buffer. Then, if time-out is received, the shell announces time-out to all the parties and halts the execution of $\mathcal{F}^{timed}$. In contrast, if compute is received, the shell hands all the messages received so far to the core (essentially, to the initial $\mathcal{F}$). Furthermore, any later time-out message is ignored, and all other messages are forwarded between the core and the external parties. That is, from the time that compute is received, the shell just acts as a channel sending messages between $\mathcal{F}$ and the external parties. A formal description of $\mathcal{F}^{timed}$ appears in Figure 2.

Functionality $\mathcal{F}^{timed}$

Let n be the security parameter, let $P_1, \ldots, P_k$ be the set of participating parties, and let $\mathcal{S}$ be the adversary. The functionality $\mathcal{F}^{timed}$ is constructed from $\mathcal{F}$ and proceeds as follows:

Compute: Upon receiving a message $(sid, \mathsf{compute})$ from $\mathcal{S}$, invoke the functionality $\mathcal{F}$ and hand it all the previously stored messages (see the "$\mathcal{F}$ messages" item below for how and when these are stored). Furthermore, write any output generated by $\mathcal{F}$ on the outgoing communication tape.

Time-out messages: Upon receiving a message $(sid, \mathsf{time\text{-}out})$ from $\mathcal{S}$, if a compute message has already been received, then do nothing. Otherwise, send the message $(sid, \mathsf{time\text{-}out})$ to all parties $P_1, \ldots, P_k$ and halt.

$\mathcal{F}$ messages: Upon receiving any other message (i.e., any message that is not of the type compute or time-out), if compute has not yet been received, then store the message. Otherwise, forward it directly to $\mathcal{F}$, and write any output generated by $\mathcal{F}$ on the outgoing communication tape. Finally, halt whenever $\mathcal{F}$ halts.

Figure 2: Formal description of the ideal timed functionality $\mathcal{F}^{timed}$

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 [18]. 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 [18]. 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" of the form "I know the preimage of one of v_1 and v_2" [18]. See [20, Section 4.6] for a full treatment of witness indistinguishable and witness hiding proofs.

Strong proofs of knowledge [20]. A proof of knowledge [26, 4] 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 [20, 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 [20, 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 [20] uses a super-logarithmic number of sequential executions of the 3-round zero-knowledge proof for Hamiltonicity [6]. 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 [2].

3 Secure Multiparty Protocols in the Timing Model

In this section, we prove our main result, informally stated in Theorem 1.1. We begin by formally restating Theorem 1.1. Recall that this theorem claims that there exist protocols that remain secure under concurrent general composition *with delays*. In order to formalize this notion of a "delayed network", we define the following class of "delayed protocols":

Definition 3 (delayed protocols): *Let π be any protocol (in the real model or in the $\mathcal{F}$-hybrid model for some $\mathcal{F}$), and let δ be a constant. Then, π_δ is the protocol obtained from π by having every honest party* delay *sending every message by δ local time units. We define the class of functionalities Π_δ to be the set of all protocols π_δ as above.*

We are now ready to state our main result (the limitations on ϵ and δ mentioned in the theorem are needed later in the proof):

Theorem 4 (Theorem 1.1 – restated): *Assume that there exist enhanced trapdoor permutations. Let $\mathcal{F}$ be any probabilistic polynomial-time multiparty functionality, let Δ and ϵ be constants where $1 \leq \epsilon \leq \sqrt[3]{1.5}$, and let $\delta = \omega(1) \cdot \Delta \cdot \epsilon$. Then, there exists a protocol ρ such that ρ securely realizes the functionality $\mathcal{F}^{timed}$ under concurrent general composition with Π_δ 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 majority of the proof of Theorem 4 involves showing how to securely realize the timed "common random string" (or coin-tossing) functionality. We begin in Section 3.1 by formally justifying this fact.

3.1 Reducing the Problem to Realizing the Timed CRS Functionality

In the common random string model, all parties are given the same uniformly distributed string. This is formalized by giving all parties access to a common random string (CRS) functionality, which is actually just a multiparty coin-tossing functionality. This functionality is denoted $\mathcal{F}_{\text{CRS}}$. In our proof of Theorem 4, we rely heavily on the result of [12] that states that any multiparty functionality can be securely realized under concurrent general composition in the $\mathcal{F}_{\text{CRS}}$-hybrid model.[11] In the case of static malicious adversaries (as we consider here), this result relies on the existence of enhanced trapdoor permutations.

From the above, it follows 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. However, we cannot securely realize the $\mathcal{F}_{\text{CRS}}$ functionality under concurrent general composition. Rather, we securely realize the *timed functionality* $\mathcal{F}_{\text{CRS}}^{timed}$ under concurrent general composition *with delays* (i.e., with the class of protocols Π_δ for some δ). This suffices due to the following two claims (note that the first claim refers to a model without time):

Claim 3.1 *Let $\mathcal{F}$ be a functionality and let ρ be a protocol that securely realizes $\mathcal{F}$ under concurrent general composition in the $\mathcal{F}_{\text{CRS}}$-hybrid model. Then, there exists a protocol ρ' that securely realizes $\mathcal{F}^{timed}$ under concurrent general composition in the $\mathcal{F}_{\text{CRS}}^{timed}$-hybrid model.*

Proof Sketch: First, recall that the transformation of a functionality $\mathcal{F}$ to $\mathcal{F}^{timed}$ is such that either time-out is received before any computation is carried out by $\mathcal{F}$, or $\mathcal{F}^{timed}$ behaves in exactly the same way as $\mathcal{F}$. This leads us to the following way of securely realizing $\mathcal{F}^{timed}$. Let ρ be a protocol that securely realizes $\mathcal{F}$ under concurrent general composition in the $\mathcal{F}_{\text{CRS}}$-hybrid model. Then, define ρ' to be the protocol that begins by invoking $\mathcal{F}_{\text{CRS}}^{timed}$. Next, if the output of $\mathcal{F}_{\text{CRS}}^{timed}$ is

[11] Actually, in [12] a class of "well-formed" functionalities is defined. However, in the case of static adversaries, this only limits the functionalities to those that are "unaware" of which parties are corrupted and which 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.

not time-out, ρ' proceeds exactly like ρ, except that the output of $\mathcal{F}_{\text{CRS}}^{timed}$ is used any time that ρ would call $\mathcal{F}_{\text{CRS}}$. (If the output of $\mathcal{F}_{\text{CRS}}^{timed}$ *is* time-out, the the output of ρ' is also time-out.) Since $\mathcal{F}_{\text{CRS}}^{timed}$ is invoked first, we have the following two cases:

1. *Case 1 – $\mathcal{F}_{\text{CRS}}^{timed}$ is not timed-out:* In this case, once a compute message is sent to $\mathcal{F}_{\text{CRS}}^{timed}$, we have that $\mathcal{F}_{\text{CRS}}^{timed}$ behaves exactly like $\mathcal{F}_{\text{CRS}}$. Therefore, ρ' will behave exactly like ρ does in the $\mathcal{F}_{\text{CRS}}$-hybrid model. In particular, the simulator for ρ' sends a compute message to $\mathcal{F}^{timed}$ and then proceeds just like the simulator for ρ.

2. *Case 2 – $\mathcal{F}_{\text{CRS}}^{timed}$ is timed-out:* In this case the simulator for ρ' simply sends time-out to $\mathcal{F}^{timed}$. (Note that since the simulator has not sent a compute message to $\mathcal{F}^{timed}$, the functionality will send time-out to all parties after receiving this message.)

This completes the proof sketch. ■

So far, we have demonstrated that ρ' securely realizes $\mathcal{F}^{timed}$ in the $\mathcal{F}_{\text{CRS}}^{timed}$-hybrid model. It still remains to show that we can replace $\mathcal{F}_{\text{CRS}}^{timed}$ with a real protocol that securely realizes $\mathcal{F}_{\text{CRS}}^{timed}$ under concurrent general composition with Π_δ, *in the timing model.*

Claim 3.2 *Let ρ' be a protocol that securely realizes a functionality $\mathcal{F}^{timed}$ under concurrent general composition in the $\mathcal{F}_{\text{CRS}}^{timed}$-hybrid model (in a model without time), and let σ be a real protocol that securely realizes $\mathcal{F}_{\text{CRS}}^{timed}$ under concurrent general composition with Π_δ, in the timing model. Then the protocol $(\rho'_\delta)^\sigma$ securely realizes $\mathcal{F}^{timed}$ under concurrent general composition with Π_δ, in the timing model.*[12]

Proof Sketch: Let σ be a protocol that securely realizes $\mathcal{F}_{\text{CRS}}^{timed}$ under concurrent general composition with Π_δ in the timing model, and let $\pi^{\rho'}$ be any protocol (i.e., π is an arbitrary protocol that may contain subroutine calls to ρ'). Then, by Definition 1 it follows that for every real adversary running $((\pi^{\rho'})_\delta)^\sigma$ in the timing model, there exists an adversary running $\pi^{\rho'}$ in the $\mathcal{F}_{\text{CRS}}^{timed}$-hybrid model (without any time), such that the output distributions are computationally indistinguishable.[13] Applying the security of ρ' under concurrent general composition (without timing and in the $\mathcal{F}_{\text{CRS}}^{timed}$-hybrid model), we have that there exists an adversary running π in the $\mathcal{F}^{timed}$-hybrid model such that the output is computationally indistinguishable from $\pi^{\rho'}$ in the $\mathcal{F}_{\text{CRS}}^{timed}$-hybrid model, and so also from $((\pi^{\rho'})_\delta)^\sigma$ in the real model with timing. We conclude that $(\rho'_\delta)^\sigma$ securely realizes $\mathcal{F}^{timed}$ under concurrent general composition with Π_δ, in the timing model. ■

Combining the result of [12] with Claim 3.1, we have that for every efficient functionality $\mathcal{F}$, there exists a protocol ρ' that securely realizes $\mathcal{F}^{timed}$ in the $\mathcal{F}_{\text{CRS}}^{timed}$-hybrid model under concurrent general composition. The proof of Theorem 4 is therefore derived from Claim 3.2 and a proof of the existence of a protocol σ that securely realizes $\mathcal{F}_{\text{CRS}}^{timed}$ under concurrent general composition with Π_δ in the timing model. The rest of this section is devoted to this task of realizing $\mathcal{F}_{\text{CRS}}^{timed}$.

3.2 The Common Random String 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 crs request. Once the functionality receives such a request, it generates a random

[12] Recall that ρ'_δ is obtained from ρ' by just delaying all messages by δ time units.

[13] The protocol $((\pi^{\rho'})_\delta)^\sigma$ is obtained by delaying all messages of $\pi^{\rho'}$ by δ time units, and then replacing calls to $\mathcal{F}_{\text{CRS}}^{timed}$ by executions of σ.

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). A formal description of $\mathcal{F}_{\text{CRS}}$ appears in Figure 3.

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

Let n be the security parameter and let $p(\cdot)$ be a fixed polynomial.[14] 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 $(\mathsf{crs}, sid, \{P_{i_1}, \ldots, P_{i_k}\})$, choose a uniformly distributed string $R_{\text{CRS}} \in_R \{0,1\}^{p(n)}$, send $(\mathsf{crs}, sid, \{P_{i_1}, \ldots, P_{i_k}\}, R_{\text{CRS}})$ to $\mathcal{S}$ and to all parties $P_{i_1}, \ldots, P_{i_k}$, and halt.

Figure 3: The ideal multi-party common random string 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.

From here on, we will refer only to the timed functionality $\mathcal{F}_{\text{CRS}}^{timed}$. As we have discussed, this is identical to $\mathcal{F}_{\text{CRS}}$ except that first either a time-out or compute message is received. If time-out is received, then $\mathcal{F}_{\text{CRS}}^{timed}$ concludes by sending time-out to all specified parties. Otherwise, if compute is received, then $\mathcal{F}_{\text{CRS}}^{timed}$ proceeds exactly as $\mathcal{F}_{\text{CRS}}$ (i.e., by sending a random R_{CRS} to all specified parties).

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

Before proceeding to describe the actual protocol for securely realizing the $\mathcal{F}_{\text{CRS}}^{timed}$ 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 [27] (which is in turn an extension of Blum's protocol [6]). 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. In contrast, the simulator may extract all of the committed values (via the proofs of knowledge of the first phase) and then have one of the "honest" parties reveal any value that it wishes in the second phase. Of course, this revealed value would then not be the one really committed to. However, the simulator can run the zero-knowledge simulator (and so is not bound to the actually committed values). The effect of this is that the simulator can force the output to be any value; in particular, it can force it to equal the string that it received from the $\mathcal{F}_{\text{CRS}}^{timed}$ functionality, as required.

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

[14] $\mathcal{F}_{\text{CRS}}$ 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).

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 crucial step because *time-out* instructions are crucial for enabling "proper rewinding," and as we have discussed above (Subsection 2.3), 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 [17]. 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.

More precisely, 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 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 [27]. The coin-tossing protocol is then completed in Phase 3 where each player reveals the value that it committed to in Phase 2, 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.[15]

Until now, we have focused on how we 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 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

[15]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.

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 (delayed) protocol π. We separately explain, at an intuitive level, the security of the WIPOKs under these two types of composition.[16]

Composition with arbitrary (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 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. This is achieved by delaying the messages of the external protocol π by the amount of time that it takes to complete a WIPOK. Since the rewinding spans only over this amount of time, this implies that new π messages need not be dealt with during rewinding.

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 [14], with modifications necessary due to the fact that we work in the concurrent setting with timing. 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.[17] 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.[18] 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

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

[17]We remark that the Chor-Rabin scheduling was also used by [15] 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").

[18]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.

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 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 $\mathcal{F}_{\text{CRS}}^{timed}$ functionality in the timing model, in a general multi-party 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 [14], who introduce a method for concurrently alternating and interleaving protocol executions while preserving independence. Loosely speaking, [14] 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, [14] 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. 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 note that the construction of [14] was for the stand-alone and synchronous setting. 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, and show that such a scheduling can be achieved in the timing model. Then, in Section 3.5, we show how such a scheduling can be used to design a protocol that securely realizes the $\mathcal{F}_{\text{CRS}}^{timed}$ functionality under concurrent general composition with delays, in the timing model. In Section 4 we show how to construct a pairwise disjoint scheduling 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 other executions of σ. 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 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.

We begin by formally defining 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 5 (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 2).*

Convention. For simplicity, we assume (without loss of generality) that for every protocol σ there exists one party that sends an initial "start" message and a concluding "end" message to all participating parties.

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 received its first σ_2 message, but *before* P_2 receives 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 6 (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 10 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 $\mathcal{F}_{\rm CRS}^{timed}$ functionality.

Theorem 7 *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 $sid \in \{0,1\}^{{\rm poly}(n)}$.*

Before proceeding, we explain in more detail why *pairwise* disjointness suffices. In our protocol, we use a pairwise disjoint scheduling of WIPOK proofs. Then, at some stage in our proof of the 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 pair of sessions sid and 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 $\mathcal{F}_{\rm CRS}^{timed}$

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 [32] where the receiver first sends a single (random) message and then the committer sends its commitment. Importantly, the scheme of [32] assumes only the existence of one-way functions. Our protocol also uses a broadcast primitive. However, as shown in [25], 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 [27] to the multi-party 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 [17] that is constructed from two witness-indistinguishable proofs of knowledge.[19] 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.

Recall, that in order to extract a witness from these proofs of knowledge we need to be able to rewind, a task which is problematic in a concurrent network. To enable these extractions, we use pairwise disjoint scheduling and delay messages in the protocol π that is running concurrently to our protocol. The motivation for the pairwise disjoint scheduling is to ensure some level of independence between the proofs of knowledge. As our proof will show, essentially we need to ensure that in every session there exists *one* honest party P (previously referred to as the "external party") whose proofs of knowledge are "independent" of the proofs of knowledge of the corrupted parties *in all sessions*. This property follows from the fact that pairwise disjoint scheduling is used. The motivation for the delaying of messages in the protocol π is to ensure that external π-messages need not be dealt with while rewinding takes place. Specifically, the delay placed upon π messages is longer than the time needed to rewind (this is enforced via time-out instructions in the proofs of knowledge, which do not allow an execution, and thus the rewinding, to take too long). Therefore, if a new π-message is received while rewinding, it can be delayed until after this rewinding part of the simulation is finished.

We now present the protocol.

Protocol $\boldsymbol{\rho}$ (protocol for realizing the $\mathcal{F}_{\text{CRS}}^{timed}$ functionality in a general multi-party 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 $v_1^i = f(w_1^i), v_2^i = f(w_2^i)$.
    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

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

       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.[20] 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 $\alpha(\cdot)$.[21] (Henceforth, we denote this proof by WISPOK, for short).

[19]We note that looking at our protocol it is not clear that we use the zero-knowledge proof of knowledge of [17], 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 [17] though the concept is similar.

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

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

(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 7).
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 $(sid, \text{time-out})$ and halts. Any party receiving such a time-out message also outputs $(sid, \text{time-out})$ 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

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

     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 $\boldsymbol{\tau \stackrel{\text{def}}{=} \alpha(n) \cdot \Delta}$ 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 $(sid, \text{time-out})$ and halts the execution. Any party receiving such a time-out message also outputs $(sid, \text{time-out})$ 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 $(sid, \text{time-out})$.
- 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.[22]
- **Output:** Each party outputs (sid, R).

This completes the description of the protocol.

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

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.[23] 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 $\mathcal{F}_{\text{CRS}}^{timed}$ 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 $\tau = \alpha(n) \cdot \Delta$.[24] In other words, Protocol ρ securely realizes $\mathcal{F}_{\text{CRS}}^{timed}$ under concurrent general composition with $\Pi_{\tau\epsilon}$ in the timing model with ϵ. As we have seen in Section 3.1, this (along with the non-triviality condition) suffices for proving Theorem 4.

Theorem 8 *Let Δ and ϵ be fixed constants, such that $1 \leq \epsilon < \sqrt[3]{1.5}$, and let $\tau = \alpha(n) \cdot \Delta$. Then, assuming the existence of one-way functions, Protocol ρ securely realizes the $\mathcal{F}_{\text{CRS}}^{timed}$ 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).*

Proof: Let Δ and ϵ be any fixed constants such that $1 \leq \epsilon < \sqrt[3]{1.5}$. Let π be an arbitrary multi-party protocol that may contain ideal calls to the $\mathcal{F}_{\text{CRS}}^{timed}$ 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 $\pi_{\tau\epsilon}^{\rho}$ in the timing model. We begin by describing the hybrid-model simulator $\mathcal{S}$ that runs π in the $\mathcal{F}_{\text{CRS}}^{timed}$-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 $\mathcal{F}_{\text{CRS}}^{timed}$ 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

[23]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.

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

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 $\mathcal{F}_{\text{CRS}}^{timed}$-hybrid model. The aim of $\mathcal{S}$ is to create the same effect in the $\mathcal{F}_{\text{CRS}}^{timed}$-hybrid model, as the real-world adversary $\mathcal{A}$ does in a real execution of $\pi_{\tau\epsilon}^{\rho}$.

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 $\mathcal{F}_{\text{CRS}}^{timed}$-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 $(\mathsf{crs}, sid, \{P_{i_1}, \ldots, P_{i_k}\})$ to the $\mathcal{F}_{\text{CRS}}^{timed}$ functionality.[25] 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 $(sid, \mathsf{time\text{-}out})$, $\mathcal{S}$ sends $(sid, \mathsf{time\text{-}out})$ to the $\mathcal{F}_{\text{CRS}}^{timed}$ functionality and delivers the message $(sid, \mathsf{time\text{-}out})$ from $\mathcal{F}_{\text{CRS}}^{timed}$ 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.

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

- 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 $a(sid)$.[26] 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 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 $\mathcal{F}_{\text{CRS}}^{timed}$, and receives $(\mathsf{crs}, sid, \{P_{i_1}, \ldots, P_{i_k}\}, R_{\text{CRS}})$ in response.[27]

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

[27] At this point of the protocol it is guaranteed that no honest party has or will output $(sid, \mathsf{time\text{-}out})$, 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 $\mathcal{F}_{\text{CRS}}^{timed}$, thereby receiving back (sid, R_{CRS}).

Then using the r_i values as given above, $\mathcal{S}$ computes

$$r = R_{\text{CRS}} \oplus \left(\bigoplus_{i \neq a} r_i \right) . \tag{2}$$

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 $\mathcal{S}$.[28]
- 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.
- For every honest party P_i, if $\mathcal{P}_i^{sid}$ outputs (sid, R) then $\mathcal{S}$ delivers the message (sid, R_{CRS}) from $\mathcal{F}_{\text{CRS}}^{timed}$ to P_i.[29]

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.

[28]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 $r = R_{\text{CRS}} \oplus \left(\bigoplus_{i\neq a} r_i\right)$.

[29]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.

- 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 $\mathcal{F}_{\text{CRS}}^{timed}$. 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 $\mathcal{P}_a^{sid}$ 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 $\mathcal{F}_{\text{CRS}}^{timed}$-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 $\pi_{\tau\epsilon}^{\rho}$ 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 $\mathcal{F}_{\text{CRS}}^{timed}$-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 9 *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 at 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 "good-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

$$\begin{aligned}
\Pr[K\text{-}\mathsf{fail} \wedge \mathcal{S}\text{-}\mathsf{pass}] &= \Pr[K\text{-}\mathsf{fail} \wedge \mathcal{S}\text{-}\mathsf{pass} \wedge \mathsf{good\text{-}proof}] + \Pr[K\text{-}\mathsf{fail} \wedge \mathcal{S}\text{-}\mathsf{pass} \wedge \neg\mathsf{good\text{-}proof}] \\
&\leq \Pr[K\text{-}\mathsf{fail}|\mathsf{good\text{-}proof}]\Pr[\mathsf{good\text{-}proof}] + \Pr[\mathcal{S}\text{-}\mathsf{pass}|\neg\mathsf{good\text{-}proof}]\Pr[\neg\mathsf{good\text{-}proof}] \\
&\leq \mu(n)\Pr[\mathsf{good\text{-}proof}] + \mu(n)\Pr[\neg\mathsf{good\text{-}proof}] \\
&= \mu(n). \qquad (3)
\end{aligned}$$

<u>$\mathcal{S}$ OUTPUTS fail_2 OR fail_3 WITH NEGLIGIBLE PROBABILITY:</u> Recall that $\mathcal{S}$ outputs 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 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 (fail_2 or 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 fail_2 or 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 fail_2 or 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 fail_2 or 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 fail_2 or 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 $\mathcal{F}_{\text{CRS}}^{timed}$ 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 fail_2 or 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 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* [18] (this follows from the fact that the proofs are witness indistinguishable with independent witnesses; see [20] 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 $\mathcal{F}_{\text{CRS}}^{timed}$ functionality, it internally implements it. (Thus the honest parties obtain their outputs from $\mathcal{F}_{\text{CRS}}^{timed}$ implemented by $\mathcal{S}_2$.) Furthermore, in Phase 3 of each session sid, instead of first drawing a random R_{CRS} (on behalf of $\mathcal{F}_{\text{CRS}}^{timed}$) 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, $\mathcal{F}_{\text{CRS}}^{timed}$ (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.[30] As was pointed out in footnote 29, 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

[30] 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.

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 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^*.[31] 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

[31]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.

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* $\mathcal{E}$.[32] 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.

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 $\mathcal{E}$.

[32]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).

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 $\mathcal{F}_{\text{CRS}}^{timed}$-hybrid model are indistinguishable from that of $\mathcal{A}$ and the honest parties running $\pi^{\rho}_{\mathcal{T}\epsilon}$ 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 $\mathcal{P}_a^{sid}$ 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 $\mathcal{F}_{\text{CRS}}^{timed}$ 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 $\mathcal{P}_i^{sid}$ produces an output (sid, R), $\mathcal{H}_1$ delivers the output (sid, R_{CRS}) from $\mathcal{F}_{\text{CRS}}^{timed}$ to P_i (after $\mathcal{P}_a^{sid}$ produces an output). In contrast, the simulator $\mathcal{H}_2$ will hand P_i the output R generated by $\mathcal{P}_i^{sid}$ in the simulation. Note that if $\mathcal{P}_i^{sid}$ outputs (sid, R) and $\mathcal{H}_1$ did not output fail_3 (and $\mathcal{P}_a^{sid}$ 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 $\mathcal{P}_a^{sid}$ 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 $\mathcal{P}_a^{sid}$ in every session sid (with at least one honest player a), $\mathcal{H}_3$ runs another program $\mathcal{P'}_a^{sid}$. This program is exactly like $\mathcal{P}_a^{sid}$, 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 $\mathcal{P}_a^{sid}$. Note that the only difference between $\mathcal{P'}_a^{sid}$ used by $\mathcal{H}_3$ and the program specified by ρ is that while giving Phase 3 WIPOK to a party P_j, $\mathcal{P'}_a^{sid}$ uses the alternate witness provided by $\mathcal{S}$ (namely w^j such that $f(w^j) \in \{v_1^j, v_2^j\}$) 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 $\mathcal{P}_a^{sid}$ 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 $\mathcal{F}_{\text{CRS}}^{timed}$-hybrid world is indistinguishable from the output of the system consisting of $\mathcal{A}$ and the honest parties running $\pi_{\tau\epsilon}^{\rho}$ 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 7 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 $\boldsymbol{\tau \stackrel{\text{def}}{=} \alpha(n) \cdot \Delta}$.[33]

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 then 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.

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 [14]. We now recall this strategy (already described in Section 3.4). It was observed in [14] 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

[33]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).

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 the *sid*.

However, there are numerous complications in adapting this strategy to our setting. Firstly, unlike the setting considered in [14], 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).[34] 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 can be 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,

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

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.

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 $(sid, \textsf{time-out})$ 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 $\tau_{\text{MIN}}(x)$ and $\tau_{\text{MAX}}(x)$. Here $\tau_{\text{MIN}}(\cdot)$ and $\tau_{\text{MAX}}(\cdot)$ are increasing functions, to be specified later. For each honest participant, if the delay is too short or too long then it outputs $(sid, \textsf{time-out})$ on its output tape and halts the execution.

Theorem 10 *Assume that $1 \leq \epsilon < \sqrt[3]{1.5}$. Then the above scheduling is a non-trivial pairwise-disjoint scheduling, for the following parameters:*

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

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.

$$\tau_{\text{MIN}}(0) > \tau\epsilon \tag{4}$$
$$\tau_{\text{MIN}}(1) > (2\tau + \tau_{\text{MAX}}(0))\epsilon \tag{5}$$
$$\tau_{\text{MIN}}(2) > (2\tau + \tau_{\text{MAX}}(0))\epsilon \tag{6}$$
$$\tau_{\text{MIN}}(2) > (2\tau + \tau_{\text{MAX}}(1))\epsilon \tag{7}$$

We note that these inequalities easily follow from the inequalities listed in the hypothesis.[35]

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'".

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.

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

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

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

Now 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

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

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 $\delta \overset{\text{def}}{=} (x + 1)d$ 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 (by 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 11 *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 $\epsilon \geq 1$.[36]

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 11 because impossibility of this case is proven explicitly in [29]. As in [29], we also limit ourselves to 2-party protocols.

The intuition behind the proof of Theorem 11 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 [29]. 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 computes 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.

[36]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 [29] for parallel general composition hold only for non-trivial protocols.

[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.

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). 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 12 *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 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 2 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 [29]. ■

Remark. Theorem 11 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 [29], which are in turn inherited from [11]. Thus, we actually obtain very broad impossibility results that hold for large classes of functionalities.

Acknowledgements

We would like to thank Oded Goldreich and Ran Canetti for many many helpful comments. We would also like to thank Shai Halevi for helpful discussions.

References

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

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

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

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

[5] 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.

[6] 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.

[7] 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).

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

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

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

[11] 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.

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

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

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

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

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

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

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

[19] J. Garay and P. Mackenzie. Concurrent Oblivious Transfer. 41*st FOCS*, pp. 314–324, 2000.

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

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

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

[23] 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.

[24] 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.

[25] S. Goldwasser and Y. Lindell. Secure Computation Without Agreement. In 16*th DISC,* Springer-Verlag (LNCS 2508), pages 17–32 2002.

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

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

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

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

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

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

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

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

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

[35] 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.

[36] 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.

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

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

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