Lower Bounds and Impossibility Results for Concurrent Self Composition[*]

Yehuda Lindell

IBM T.J.Watson Research
19 Skyline Drive, Hawthorne
New York 10532, USA
lindell@us.ibm.com

April 4, 2005

Abstract

In the setting of concurrent self composition, a single protocol is executed many times concurrently by a single set of parties. In this paper, we prove lower bounds and impossibility results for secure protocols in this setting. First and foremost, we prove that there exist large classes of functionalities that *cannot* be securely computed under concurrent self composition, by any protocol. We also prove a *communication complexity* lower bound on protocols that securely compute a large class of functionalities in this setting. Specifically, we show that any protocol that computes a functionality from this class and remains secure for m concurrent executions, must have bandwidth of at least m bits. The above results are unconditional and hold for any type of simulation (i.e., even for non-black-box simulation). In addition, we prove a severe lower bound on protocols that are proven secure using black-box simulation. Specifically, we show that any protocol that computes the blind signature or oblivious transfer functionalities and remains secure for m concurrent executions, where security is proven via *black-box simulation*, must have at least m rounds of communication. Our results hold for the plain model, where no trusted setup phase is assumed. While proving our impossibility results, we also show that for many functionalities, security under concurrent *self* composition (where a single secure protocol is run many times) is actually equivalent to the seemingly more stringent requirement of security under concurrent *general* composition (where a secure protocol is run concurrently with other arbitrary protocols). This observation has significance beyond the impossibility results that are derived by it for concurrent self composition.

Keywords: secure computation, protocol composition, self and general composition, impossibility results, lower bounds, non-black-box and black-box simulation.

[*]This paper combines results that appeared in extended abstracts in [25] and [28].

Contents

1 Introduction

In the setting of two-party computation, two parties with respective private inputs x and y, wish to jointly compute a functionality $f(x, y) = (f_1(x, y), f_2(x, y))$, such that the first party receives $f_1(x, y)$ and the second party receives $f_2(x, y)$. This functionality may be probabilistic, in which case $f(x, y)$ is a random variable. Loosely speaking, the security requirements are that nothing is learned from the protocol other than the output (privacy), and that the output is distributed according to the prescribed functionality (correctness). These security requirements must hold in the face of an adversary who controls one of the parties and can arbitrarily deviate from the protocol instructions (i.e., in this work we consider malicious, static adversaries). Powerful feasibility results have been shown for this problem, demonstrating that *any* two-party probabilistic polynomial-time functionality can be securely computed, assuming the existence of trapdoor permutations [36, 19].

Security under concurrent composition. The feasibility results of [36, 19] relate only to the stand-alone setting, where a single pair of parties run a single execution. A more general (and realistic) setting relates to the case that many protocol executions are run concurrently within a network. Unfortunately, the security of a protocol in the stand-alone setting does not necessarily imply its security under concurrent composition. Therefore, it is important to re-establish the feasibility results of the stand-alone setting for the setting of concurrent composition, or alternatively, to demonstrate that this cannot be done.

The notion of protocol composition can be interpreted in many ways. A very important distinction to be made relates to the *context* in which the protocol is executed. This refers to the question of *which protocols* are run together in the network, or in other words, with which protocols should the protocol in question compose. There are two contexts that have been considered, defining two classes of composition:

1. Self composition: A protocol is said to be secure under *self composition* if it remains secure when it alone is executed many times in a network. We stress that in this setting, there is only one protocol that is being run many times. This is the type of composition considered, for example, in the entire body of work on concurrent zero-knowledge (e.g., [11, 34]).

2. General composition: In this type of composition, many different protocols are run together in the network. Furthermore, these protocols may have been designed independently of one another. A protocol is said to maintain security under *general composition* if its security is maintained even when it is run along with other arbitrary protocols. This is the type of composition that was considered, for example, in the framework of universal composability [5].

We stress a crucial difference between self and general composition. In self composition, the protocol designer has control over everything that is being run in the network. However, in general composition, the other protocols being run may even have been designed maliciously after the secure protocol is fixed. This additional adversarial capability has been shown to yield practical attacks against real protocols [23].[1]

Another distinction that we will make relates to the number of times a secure protocol is run. Typically, a protocol is expected to remain secure for any polynomial number of sessions. This is the "default" notion, and we sometimes refer to it as unbounded concurrency. A more restricted notion, first considered by [1], is that of bounded concurrency. In this case, a fixed bound on the

[1] Although the attacks shown in [23] are due to key reuse, they demonstrate the point that the setting of *general* composition poses a real security threat. Specifically, [23] show how the adversary can construct new protocols whose entire aim is to compromise the security of existing protocols.

number of concurrent executions is given, and the protocol only needs to remain secure when the number of concurrent executions does not exceed this bound. (When the bound is m, we call this m-bounded concurrency.) We note that the protocol design may depend on this bound.

Finally, we will distinguish between a setting where parties have fixed roles versus a setting where they may have interchangeable roles. For the sake of this distinction, note that protocols typically involve different roles. In general, in a two-party protocol, one role may be the *protocol initiator* while the other is the *protocol responder*. More notable examples of roles arise in specific cases. For example, in zero-knowledge, there are two different roles: the *prover* role and the *verifier* role. Now, in the setting of composition with fixed roles, each party assumes the same role in all of the executions. In contrast, in the setting of composition with interchangeable roles, parties may assume different roles in different executions. The latter setting, of interchangeable roles, is more general and in many cases is what is needed. However, there are some cases where fixed roles also make sense (for example, when one party is a server and the other a client).

Feasibility of secure computation under composition. The first definition and composition theorem for security under concurrent general composition was provided by [31] who considered the case that a secure protocol is executed *once* concurrently with another arbitrary protocol.[2] The unbounded case, where a secure protocol can be run any polynomial number of times in an arbitrary network, was then considered in the framework of universal composability [5]. Informally speaking, a protocol that is proven secure under the definition of universal composability is guaranteed to remain secure when run any polynomial number of times in the setting of concurrent general composition. This setting realistically models the security requirements in modern networks. Therefore, the construction of protocols that are secure by this definition is of great importance. On the positive side, it has been shown that in the case of an honest majority, essentially any functionality can be securely computed in this framework [5]. Furthermore, even when there is no honest majority, it is possible to securely compute any functionality in the *common reference string* (CRS) model [9]. (In the CRS model, all parties have access to a common string that is chosen according to some distribution. Thus, this assumes some trusted setup phase.) However, it is desirable to obtain protocols in a setting where *no* trusted setup phase is assumed. Unfortunately, in the case of no honest majority and no trusted setup, broad impossibility results for universal composability have been demonstrated [6, 5, 8]. Recently, it was shown in [27] that these impossibility results extend to *any* security definition that guarantees security under concurrent general composition (including the definition of [31]).

Thus, it seems that in order to obtain security without an honest majority or a trusted setup phase, we must turn to *self* composition. Indeed, as a first positive step, it has been shown that any functionality can be securely computed under m-bounded concurrent self composition [25, 30]. Unfortunately, however, these protocols are highly inefficient: The protocol of [25] has many rounds of communication and both the protocols of [25] and [30] have high bandwidth. (That is, in order to obtain security for m executions, the protocol of [25] has more than m rounds and communication complexity of at least mn^2. In contrast, the protocol of [30] has only a constant number of rounds, but still suffers from communication complexity of at least mn^2.) These works still leave open the following important questions:

1. Is it possible to obtain protocols that remain secure under *unbounded* concurrent self composition, and if yes, for which functionalities?

2. Is it possible to obtain *highly efficient* protocols that remain secure under m-bounded con-

[2] An earlier reference to this problem with general ideas about how to define security appeared in [3, Appendix A].

current self composition? (By highly efficient, we mean that the dependence on the bound m is either additive (e.g., $m + \text{poly}(n)$ or $\text{poly}(m) + \text{poly}(n)$), or sublinear (e.g., $m^\epsilon \cdot \text{poly}(n)$ for some small constant $0 < \epsilon < 1$).[3])

As we have mentioned, these questions are open for the case that no trusted setup phase is assumed and when there is no honest majority, as in the important two party case.

Our results. In this paper, we provide negative answers to the above two questions. More precisely, we show that there exist large classes of functionalities that cannot be securely computed under unbounded concurrent self composition, by any protocol. We also prove a communication complexity lower bound for protocols that are secure under m-bounded concurrent self composition (by communication complexity, we mean the bandwidth or total number of bits sent by the parties during the execution). This is the first lower bound of this type, connecting the communication complexity of a protocol with the bound on the number of executions for which it remains secure. We begin with our impossibility result.

Theorem 1.1 (impossibility for unbounded concurrency – informal): *There exist large classes of two-party functionalities that cannot be securely computed under unbounded concurrent self composition, by any protocol.*

In order to prove this result, in Section 3 we show that for many functionalities, obtaining security under unbounded concurrent *self* composition is actually equivalent to obtaining security under concurrent *general* composition (that is, a protocol is secure under one notion if and only if it is secure under the other). This may seem counter-intuitive because in the setting of self composition, the protocol designer has full control over the network. Specifically, the only protocol that is run in the network is the specified secure protocol. In contrast, in the setting of general composition, a protocol must remain secure even when run concurrently with arbitrary other protocols. Furthermore, these protocols may be designed maliciously in order to attack the secure protocol. Despite this apparent difference, we show that equivalence actually holds. We now briefly describe how this is proven.

The above-described equivalence between concurrent self and general composition is proven for all functionalities that "enable bit transmission". Loosely speaking, such a functionality can be used by each party to send any arbitrary bit to the other party. In the setting of interchangeable roles (described above), essentially any functionality that depends on the parties' inputs (and so is non-trivial) enables bit transmission. In the setting of fixed roles, it is also required that both parties receive non-trivial output; see Section 2.4.

Now, many executions of a protocol that securely computes a functionality that enables bit transmission can be used by the parties to send *arbitrary messages* to each other. Essentially, this means that many executions of one secure protocol can be used to emulate the execution of any arbitrary protocol. Thus, the setting of general composition, where a secure protocol runs concurrently with other arbitrary protocols, can be emulated (using the bit transmission property) by many executions of a single secure protocol. We therefore obtain that security under concurrent self composition implies security under concurrent general composition. Since, trivially, security under general composition implies security under self composition, we obtain equivalence between these two notions. We conclude that although general composition considers a very difficult scenario

[3]Notice that a protocol whose complexity has *no* dependence on m can be used to achieve unbounded concurrency by setting $m = n^{\log n}$. Therefore, given that unbounded concurrency cannot be achieved, some dependence on m is necessary.

(in which arbitrary network activity must be considered), for many functionalities it is actually equivalent to the seemingly more restricted setting of self composition. That is, we have the following theorem:

Theorem 1.2 (equivalence of self and general composition – informal): *Let f be any two-party functionality. Then, in the setting of interchangeable roles, f can be securely computed under unbounded concurrent self composition if and only if it can be securely computed under concurrent general composition. If f is a functionality that enables bit transmission, then equivalence also holds in a setting with fixed roles.*

As stated in Theorem 1.2, in the setting of interchangeable roles, we obtain full equivalence between concurrent self and general composition (without any additional requirement regarding bit transmission). This is the case because when interchangeable roles are allowed, all functionalities are either trivial (to the extent that they can be computed without any interaction) or enable bit transmission.

A natural question to ask is whether or not in the setting of fixed roles, equivalence also holds for functionalities that do *not* enable bit transmission. In Section 3.3, we show that in the setting of fixed roles, there exists a functionality that can be securely computed under concurrent self composition but cannot be securely computed under concurrent general composition (specifically, this is the zero-knowledge proof of knowledge functionality). Thus, when there is no bit transmission, it can be "easier" to obtain security under self composition than under general composition.

Returning back to the proof of Theorem 1.1, impossibility is derived by combining the equivalence between concurrent self and general composition as stated in Theorem 1.2 with the impossibility results for concurrent general composition that were demonstrated in [27]. The actual impossibility results obtained are described in Section 4. This answers the first question above, at least in that it demonstrates impossibility for large classes of functionalities. (It is still far, however, from a full characterization of feasibility.) Regarding the second question, in Section 5, we prove the following theorem that rules out the possibility of obtaining "efficient" protocols that remain secure under m-bounded concurrent self composition.

Theorem 1.3 (communication complexity lower bound – informal): *There exists a large class of two-party functionalities with the property that any protocol that securely computes a functionality in this class under m-bounded concurrent self composition, must have communication complexity of at least m.*

Theorem 1.3 is essentially proven by directly combining the proof of Theorem 1.2 with proofs of impossibility from [27] and [8]; see Section 5.

We remark that our definition of security under concurrent self composition is such that honest parties may choose their inputs *adaptively,* based on previously obtained outputs. This is a seemingly harder definition to achieve than one where the inputs to all the executions are fixed ahead of time. We stress that allowing the inputs to be chosen adaptively is *crucial* to the proof of Theorems 1.1 to 1.3. Nevertheless, we believe that this is also the desired definition (since in real settings, outputs from previous executions may indeed influence the inputs of later executions).

Black-box lower bound. The above lower bounds and impossibility results are unconditional. That is, they hold without any complexity assumptions and assume nothing about the simulation; in particular it is not assumed that the simulator is "black-box".[4] In addition to the above, in

[4]A black-box simulator uses only oracle access to the real adversary $\mathcal{A}$; see the paragraph that follows Definition 1.

Section 6, we prove a severe lower bound on the round complexity of protocols that can be proven secure using black-box simulation. This lower bound is proven specifically for the functionalities of blind signatures [10] and 1-out-of-2 oblivious transfer [33, 12].

Theorem 1.4 (black-box lower bound – informal): *Any protocol that securely computes the blind signature or oblivious transfer functionalities under m-bounded concurrent self composition, and can be proven using* black-box simulation*, must have more than m rounds of communication.*

Black-box lower bounds do not imply infeasibility in general. In fact, constant-round protocols for m-bounded concurrent self composition have been demonstrated [30]. Nevertheless, Theorem 1.4 shows that any such protocol must use non-black-box simulation techniques. Note also that all known highly efficient protocols are proven via black-box simulation; therefore, Theorem 1.4 may indicate a certain difficulty in obtaining very efficient protocols in this setting.

The idea behind the proof of Theorem 1.4 is to show that when concurrent self composition is considered, the "rewinding capability" of the simulator is severely limited. In fact, for a protocol of m rounds that is run m times concurrently, there exists a scheduling of messages so that in one of the executions, the simulator is unable to carry out any rewinding of the adversary. However, informally speaking, a black-box simulator must rewind in order to successfully simulate. Therefore, any protocol that remains secure for m concurrent executions must have more than m rounds of communication.

Theorem 1.4 stands in stark contrast with concurrent zero-knowledge, where black-box simulation does suffice for obtaining unbounded concurrent composition [34]. In fact, a logarithmic number of rounds suffice for obtaining security for any polynomial number of executions [32]. Thus, in the "black-box world" and in the setting of concurrent self composition, zero-knowledge is strictly easier to achieve than blind signatures and oblivious transfer.

We remark that Theorems 1.1, 1.3 and 1.4 hold even if at any given time, at most *two* executions are running simultaneously. (Loosely speaking, in such a case the m-bounded concurrency means that m different protocol executions may overlap.) This shows that our lower bounds do not stem from deep protocol nesting (in contrast to [7], for example). Indeed, a nesting depth of at most two is needed.

Extensions to multi-party computation. We note that although Theorems 1.1 and 1.3 are stated for two-party functionalities, they immediately imply impossibility results for multi-party functionalities as well. This is because secure protocols for multi-party functionalities can be used to solve two-party tasks as well. Likewise, by appropriately defining "bit transmission" for multi-party functionalities, it is possible to prove Theorem 1.2 for this setting as well.

A new result for concurrent general composition. While proving Theorem 1.3, we also obtain a new impossibility result for concurrent general composition. Specifically, we show that even if the inputs used by an honest party in a secure protocol are *independent* of the inputs used in the other arbitrary protocols, then impossibility still holds. See Section 5.2 for more details. (Interestingly, as shown in [14] and discussed in Section 6.2, oblivious transfer under concurrent self composition and with fixed roles *can* be achieved in the case that all inputs are independently chosen. This does not contradict the above result for general composition because oblivious transfer does not enable bit transmission. Therefore, the equivalence between self and general composition of Theorem 1.2 does not hold.)

Other related work. The focus of this work is the ability to obtain secure protocols for solving general tasks. However, security under concurrent composition has also been studied for specific tasks of interest. Indeed, the study of security under concurrent composition was initiated in the context of concurrent zero knowledge [13, 11], where a prover runs many copies of a protocol with many verifiers. Thus, concurrent zero-knowledge is cast in the setting of concurrent *self* composition. This problem has received much attention; see [34, 7, 1] for just a few examples. Other specific problems have also been considered, but are not directly related to this paper. One work that requires mentioning is the (black-box) protocol for unbounded concurrent oblivious transfer of [14]. This construction seems to be in direct contradiction to Theorem 1.4. However, in the model of [14], all the inputs in all the executions are assumed to be independent of each other. In contrast, we consider a more standard model where quantification is over all inputs, and in particular, over possibly correlated inputs.

Open questions. As we have mentioned, the constant-round protocol of [30] has very high communication complexity. Specifically, the number of bits sent in the protocol is $\Omega(m(n^2 + |\Pi|))$, where Π is a protocol that remains secure under concurrent self composition when given access to an ideal zero-knowledge functionality. Thus, the factor of m is *multiplicative* in the communication complexity. In contrast, Theorem 1.3 only shows that a linear dependence on m (or an additive factor) is necessary. This gap is very significant because a bandwidth of $m + |\Pi|$ may be acceptable in practice, in contrast to the very high communication complexity of the protocol of [30].

Another interesting question relates to the feasibility of obtaining security under concurrent self composition and with fixed roles, for functionalities that do not enable bit transmission. As we have mentioned, the zero-knowledge functionality does not enable bit transmission and *can* be securely computed under concurrent self composition. However, it is not known which other functionalities can also be securely computed. The oblivious transfer functionality would be of specific interest here; both because of its importance as a cryptographic primitive, and because by Theorem 1.4, it cannot be securely computed using black-box simulation. Thus, the question remains whether or not non-black-box simulation can be used to achieve oblivious transfer under unbounded concurrent self composition.

2 Definitions

In this section, we present definitions of security under concurrent self composition and concurrent general composition. In addition, we define functionalities with "interchangeable roles" and functionalities that "enable bit transmission". Our definitions are presented for the special case of two-party protocols. The extension to the multi-party case is straightforward.

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

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

When X and Y are equivalent distributions, we write $X \equiv Y$.

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

2.1 Concurrent Self Composition of Secure Two-Party Protocols

We begin by presenting the definition for security under concurrent self composition. The basic description and definition of secure computation follows [20, 2, 29, 4].

Two-party computation. A two-party protocol problem is cast by specifying a random process that maps pairs of inputs to pairs of outputs (one for each party).[5] We refer to such a process as a $\mathsf{functionality}$ and denote it $f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* \times \{0,1\}^*$, where $f = (f_1, f_2)$. That is, for every pair of inputs (x, y), the output-pair is a random variable $(f_1(x, y), f_2(x, y))$ ranging over pairs of strings. The first party (with input x) wishes to obtain $f_1(x, y)$ and the second party (with input y) wishes to obtain $f_2(x, y)$. We often denote such a functionality by $(x, y) \mapsto (f_1(x, y), f_2(x, y))$. Thus, for example, the zero-knowledge proof of knowledge functionality for a relation R is denoted by $((x, w), \lambda) \mapsto (\lambda, (x, R(x, w)))$. In the context of concurrent composition, each party actually uses many inputs (one for each execution), and these may be chosen adaptively based on previous outputs. We consider both concurrent self composition (where the number of executions is unbounded) and m-bounded concurrent self composition (where the number of simultaneously concurrent executions is a priori bounded by m and the protocol design can depend on this bound).

Adversarial behavior. In this work we consider a malicious, static adversary that runs in polynomial time (recall that this means that it is polynomial in the security parameter, irrespective of the length of its input). Such an adversary controls one of the parties (who is called corrupted) and may then interact with the honest party while arbitrarily deviating from the specified protocol. Our definition does not guarantee any fairness. That is, the adversary always receives its own output and can then decide when (if at all) the honest party will receive its output. The scheduling of message delivery is decided by the adversary.

Security of protocols (informal). The security of a protocol is analyzed by comparing what an adversary can do in the protocol to what it can do in an ideal scenario that is clearly secure. This is formalized by considering an *ideal* computation involving an incorruptible *trusted third party* to whom the parties send their inputs. The trusted party computes the functionality on the inputs and sends each party its designated output. Unlike in the stand-alone model, here the trusted party computes the functionality many times, each time upon different inputs. Loosely speaking, a protocol is secure if any adversary interacting in the real protocol (where no trusted third party exists) can do no more harm than if it was involved in the above-described ideal computation.

Concurrent executions in the ideal model. An ideal execution with an adversary who controls P_2 proceeds as follows (when the adversary controls P_1 the roles are simply reversed):

Inputs: Party P_1 and P_2's inputs are respectively determined by probabilistic polynomial-time Turing machines M_1 and M_2, and initial inputs x and y to these machines. As we will see below, these Turing machines determine the values that the parties use as inputs in the protocol executions. These input values are computed from the initial input, the current session number and outputs that were obtained from executions that have already concluded. Note that the number of previous outputs ranges from zero (for the case that no previous outputs have yet been obtained) to some polynomial in n that depends on the number of sessions initiated by the adversary.[6]

[5]Thus, our specific definition is for "secure function evaluation" only. However, it can be generalized to reactive functionalities in a straightforward way.

[6]Notice that we place no restriction on the lengths of the input values output by M_1 and M_2. It is known that

Session initiation: The adversary initiates a new session by sending a start-session message to the trusted party. The trusted party then sends (start-session, i) to the honest party, where i is the index of the session (i.e., this is the i^{th} session to be started).

Honest party sends input to trusted party: Upon receiving (start-session, i) from the trusted party, the honest party P_1 applies its input-selecting machine M_1 to its initial input x, the session number i and its previous outputs, and obtains a new input x_i. That is, in the first session, $x_1 = M_1(x, 1)$. In later sessions, $x_i = M_1(x, i, \alpha_{i_1}, \ldots, \alpha_{i_j})$ where j sessions have already concluded and the outputs were $\alpha_{i_1}, \ldots, \alpha_{i_j}$.

The honest party P_1 sends (i, x_i) to the trusted party.

Adversary sends input to the trusted party and receives output: Whenever the adversary wishes, it may send a message (i, y_i) to the trusted party, for any y_i of its choice. Upon sending this pair, it receives back $(i, f_2(x_i, y_i))$ where x_i is the value that P_1 previously sent the trusted party. (If i start-session messages have not yet been sent to the trusted party, then the (i, y_i) message from the adversary is ignored. In addition, once an input indexed by i has already been sent by the adversary, the trusted party ignores any subsequent such messages.)

Adversary instructs trusted party to answer honest party: When the adversary sends a message of the type (send-output, i) to the trusted party, the trusted party sends $(i, f_1(x_i, y_i))$ to the honest party P_1, where x_i and y_i are the respective inputs sent by P_1 and the adversary for this session. (If (i, x_i) and (i, y_i) have not yet been received by the trusted party, then this (send-output, i) message is ignored.)

Outputs: The honest party P_1 always outputs the vector $(f_1(x_{i_1}, y_{i_1}), f_1(x_{i_2}, y_{i_2}), \ldots)$ of outputs that it received from the trusted party. Formally, whenever it receives an output, it writes it to its output-tape. Thus, the outputs do not appear in ascending order according to the session numbers, but rather in the order that they are received. The adversary may output an arbitrary (probabilistic polynomial-time computable) function of its auxiliary input z, the corrupted party P_2's input-selecting machine M_2, initial input y, and the outputs obtained from the trusted party.

Let $f : \{0,1\}^* \times \{0,1\}^* \mapsto \{0,1\}^* \times \{0,1\}^*$ be a functionality, where $f = (f_1, f_2)$, and let $\mathcal{S}$ be a non-uniform probabilistic polynomial-time machine (representing the ideal-model adversary). Then, the ideal execution of f (with security parameter n, input-selecting machines $\overline{M} = (M_1, M_2)$, initial inputs (x, y), and auxiliary input z to $\mathcal{S}$), denoted $\text{IDEAL}_{f,\mathcal{S},\overline{M}}(n, x, y, z)$, is defined as the output pair of the honest party and $\mathcal{S}$ from the above ideal execution.

We note that the definition of the ideal model is the same for unbounded and m-bounded concurrency. This is because this bound is relevant only to the scheduling allowed to the adversary in the real model; see below. However, the fact that a concurrent setting is considered can be seen from the above-described interaction of the adversary with the trusted party. Specifically, the adversary is allowed to obtain outputs in any order that it wishes, and can choose its inputs

secure protocols must, to some extent, reveal information about the lengths of inputs. Therefore, it is impossible to achieve security if the inputs can have arbitrary lengths and the output does not reveal these lengths. This problem can be solved through the definition of the functionality being computed. One possibility is to simply have the functionality output include the lengths of the inputs. Another possibility is to define the functionality so that only "legal" input lengths are allowed. For example, if a functionality f should be computed on equal-length inputs only, then we define $g(x, y)$ so that $g(x, y) = f(x, y)$ when $|x| = |y|$, and $g(x, y) = \perp$ otherwise. From here on we do not relate to this issue, and note that our results do not depend on how it is solved.

adaptively based on previous outputs. This is inevitable in a concurrent setting where the adversary can schedule the order in which all protocol executions take place.

Execution in the real model. We next consider the real model in which a real two-party protocol is executed (and there exists no trusted third party). Formally, a two-party protocol ρ is defined by two sets of instructions ρ_1 and ρ_2 for parties P_1 and P_2, respectively. A protocol is said to be polynomial-time if the running-time of each ρ_i in a *single execution* is bounded by a fixed polynomial in the security parameter n, irrespective of the length of the input.

Let f be as above and let ρ be a probabilistic polynomial-time two-party protocol for computing f. In addition, let $\mathcal{A}$ be a non-uniform probabilistic polynomial-time adversary that controls either P_1 or P_2. Then, the real concurrent execution of ρ (with security parameter n, input-selecting machines $\overline{M} = (M_1, M_2)$, initial inputs (x, y), and auxiliary input z to $\mathcal{A}$), denoted $\text{REAL}_{\rho,\mathcal{A},\overline{M}}(n, x, y, z)$, is defined as the output pair of the honest party and $\mathcal{A}$, resulting from the following process. The parties run concurrent executions of the protocol, where an honest party P_1 follows the instructions of ρ_1 in all of the executions; likewise, an honest P_2 always follows ρ_2. Thus, the parties play the same role in every execution. Now, the i^{th} session is initiated by the adversary by sending a start-session message to the honest party. The honest party then applies its input-selecting machine on its initial input, the session number i and its previously received outputs, and obtains the input for this new session. Upon the conclusion of an execution of ρ, the honest party writes its output from that execution on its output-tape. The scheduling of all messages throughout the executions is controlled by the adversary. That is, the execution proceeds as follows. The adversary sends a message of the form (i, α) to the honest party. The honest party then adds the message α to the view of its i^{th} execution of ρ and replies according to the instructions of ρ and this view.[7] The adversary continues by sending another message (j, β), and so on. We note that there is no restriction on the scheduling allowed by the adversary. (We sometimes refer to this as unbounded concurrency, in order to distinguish it from m-bounded concurrency that is defined next.)

In addition to the above setting where no restriction is placed on the scheduling, we also consider m-bounded concurrency, where the scheduling by the adversary must fulfill the following condition: for every execution i, from the time that the i^{th} execution begins until the time that it ends, messages from at most m different executions can be sent. (Formally, view the schedule as the ordered series of messages of the form (index, message) that are sent by the adversary. Then, in the interval between the beginning and termination of any given execution, the number of different indices viewed can be at most m.) We note that this definition of concurrency covers the case that m executions are run simultaneously. However, it also includes a more general case where many more than m executions take place, but each execution overlaps with at most m other executions. In this setting, the value m is fixed ahead of time, and the protocol design may depend on the choice of m. We denote the output of the adversary and honest party in the setting of m-bounded concurrency by $\text{REAL}^m_{\rho,\mathcal{A},\overline{M}}(n, x, y, z)$.

Security as emulation of a real execution in the ideal model. Having defined the ideal and real models, we can now define security of protocols. Loosely speaking, a protocol is secure if for every real-model adversary $\mathcal{A}$ and pair of input-selecting machines (M_1, M_2), there exists an ideal model adversary $\mathcal{S}$ such that for all initial inputs x, y, the outcome of an ideal execution with $\mathcal{S}$ is computationally indistinguishable from the outcome of a real protocol execution with $\mathcal{A}$. Notice that the order of quantifiers is such that $\mathcal{S}$ comes after M_1 and M_2. Thus, $\mathcal{S}$ knows the *strategy*

[7]Notice that the honest party runs each execution of ρ obliviously to the other executions. Thus, this is stateless composition.

used by the honest parties to choose their inputs. However, $\mathcal{S}$ does not know the initial input of the honest party, nor the random tape used by its input-selecting machine (any "secrets" used by the honest parties are included in the initial input, not the input-selecting machine). Notice also that a special case of this definition is where the inputs are fixed ahead of time. In this case, the initial inputs are vectors where the i^{th} value is the input for the i^{th} session, and in the i^{th} session the input-selecting machines M_1 and M_2 just output the i^{th} value of the input vector. We now present the definition:

Definition 1 (security under concurrent self composition): *Let f and ρ be as above. Protocol ρ is said to* securely compute f under concurrent self composition *if for every real-model non-uniform probabilistic polynomial-time adversary $\mathcal{A}$ controlling party P_i ($i \in \{1,2\}$) and every pair of probabilistic polynomial-time input-selecting machines $\overline{M} = (M_1, M_2)$, there exists an ideal-model non-uniform probabilistic polynomial-time adversary $\mathcal{S}$ controlling P_i, such that*

$$\left\{\text{IDEAL}_{f,\mathcal{S},\overline{M}}(n,x,y,z)\right\}_{n\in\mathsf{N};x,y,z\in\{0,1\}^*} \overset{c}{\equiv} \left\{\text{REAL}_{\rho,\mathcal{A},\overline{M}}(n,x,y,z)\right\}_{n\in\mathsf{N};x,y,z\in\{0,1\}^*}$$

Let $m = m(n)$ be a fixed polynomial. Then, we say that ρ securely computes f under m-bounded concurrent self composition *if*

$$\left\{\text{IDEAL}_{f,\mathcal{S},\overline{M}}(n,x,y,z)\right\}_{n\in\mathsf{N};x,y,z\in\{0,1\}^*} \overset{c}{\equiv} \left\{\text{REAL}^m_{\rho,\mathcal{A},\overline{M}}(n,x,y,z)\right\}_{n\in\mathsf{N};x,y,z\in\{0,1\}^*}$$

Definition 1 requires that the ideal-model simulator/adversary run in strict polynomial-time. However, a more liberal interpretation of "efficient simulation" is often allowed, in which case the simulator can run in *expected* polynomial-time. We note that all of our impossibility results also hold for this more relaxed definition.

Black-box simulation. We say that a protocol ρ for securely computing f is proven secure using black-box simulation if $\mathcal{S}$ is given only oracle access to $\mathcal{A}$. That is, for every $\mathcal{A}$ and every pair $\overline{M} = (M_1, M_2)$, there exists an ideal-model adversary $\mathcal{S}$ such that

$$\left\{\text{IDEAL}_{f,\mathcal{S}^{\mathcal{A}(z)},\overline{M}}(n,x,y,\lambda)\right\}_{n\in\mathsf{N};x,y,z\in\{0,1\}^*} \overset{c}{\equiv} \left\{\text{REAL}^m_{\rho,\mathcal{A},\overline{M}}(n,x,y,z)\right\}_{n\in\mathsf{N};x,y,z\in\{0,1\}^*}$$

We stress that in this case, $\mathcal{S}$ is *not* given the auxiliary input z, but rather has oracle access to $\mathcal{A}(z)$ only. The proof of our black-box lower bound (Theorem 1.4) relies on the fact that the black-box simulator $\mathcal{S}$ is not given the adversary $\mathcal{A}$'s auxiliary input. Withholding the auxiliary input from $\mathcal{S}$ is standard practice, and is used in an essential way in all known black-box lower bounds for zero-knowledge. Indeed, most known (zero-knowledge) black-box lower bounds are known to *not* hold when the simulator receives the adversary's auxiliary input [1].

Non-trivial protocols. Notice that in the ideal model, the honest party is never guaranteed to receive output. Therefore, the "real" protocol that just hangs and does not provide output to any party is actually secure by definition (and so our impossibility results cannot apply to *all* protocols). We therefore use the notion of non-trivial protocols, as defined in [9]. Such a protocol has the property that if the real-model adversary instructs the corrupted party to act honestly (i.e., follow the protocol specification), then both parties receive output.

Adaptively chosen inputs. Definition 1 allows the honest parties to choose their inputs adaptively, based on previously obtained outputs. This is a potentially more difficult definition to achieve than that presented in [25] where all inputs were fixed ahead of time.[8] It is significant that this ability to adaptively choose inputs is *crucial* to the proof of our non-black-box lower bounds and impossibility results. Nevertheless, we also claim that this is a far more realistic model; in many settings, parties do choose their inputs based on their previous outputs. (Note that our black-box lower bounds in Section 6 hold even if all inputs are fixed ahead of time.)

The complexity of the parties. Our above definition requires that the adversary is bound by a fixed polynomial. In contrast, the complexity of the honest parties depends on the protocol (which is fixed), the number of sessions (which is decided by the adversary), and the complexity of the input-selecting machines. Therefore, there is no single polynomial that bounds an honest party's overall running-time for every scenario. Nevertheless, for every given real-model adversary and input-selecting machine, the running time of the honest party is bounded by a fixed polynomial. We stress that the protocol ρ is required to be polynomial, meaning that the running-time of the honest parties in a single execution is bounded by a fixed polynomial in the security parameter, for every input-selecting machine and initial input.

Generalizations of the model. The above-described model is quite general. However, some generalizations may be considered. First, our definition above considers a rather limited case where one set of parties run all of the executions. An important generalization of this model is a setting where many (possibly different and overlapping) sets of parties run many concurrent executions. Furthermore, instead of restricting the parties to always use the same role (i.e., P_1 always runs ρ_1 and P_2 always runs ρ_2), it is possible to allow P_1 and P_2 to run either ρ_1 or ρ_2 (the specific role taken by each can be negotiated upon initiating the execution). Since the focus of this work is impossibility, we presented the more restricted notion.

Another generalization of the model relates to the running-time of the parties and adversary. In our definition above, the complexity of an honest party's instructions in a protocol must be upper-bound by a fixed polynomial in the security parameter. This suffices for most tasks. However, it does not allow for protocols that take inputs of length that can be any polynomial (say, dependent on the running-time of the adversary). Examples of such protocols are encryption and signatures, where messages of any polynomial length may be considered. In order to capture such applications, it is possible to allow a protocol ρ to run in time that is polynomial in the *length of the input*, instead of in time that is polynomial in the security parameter. Note that since the inputs are selected by machines that are polynomial in the security parameter, we are still guaranteed that polynomial-length inputs are considered (this is important because otherwise the parties may be allowed to run in time that is exponential in the security parameter). A second alternative is to have all the entities (include the input-selecting machines and ρ) run in time that is polynomial in the input-length. However, we must then quantify only over initial inputs that are of length that is polynomial in the security parameter. (That is, we require that for every real adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that for every polynomial $q(\cdot)$ and all sufficiently large n's, the result of an ideal execution with inputs of length $q(n)$ is indistinguishable from the result of a real execution of ρ with inputs of length $q(n)$.) See [5] for more discussion on this issue (these generalizations appear in the new version of [5]).

[8]We note that although the main definition of [25] considers fixed inputs, it is remarked that the security of their protocol holds even when inputs are chosen adaptively. The more restricted definition was used for the sake of simplicity.

2.2 Concurrent General Composition of Secure Two-Party Protocols

Informally speaking, concurrent general composition considers the case that a protocol ρ for securely computing some functionality f, is run concurrently (many times) with arbitrary other protocols π. In other words, the secure protocol ρ is run many times in a network in which arbitrary activity takes place. One important issue that must be dealt with in such a scenario is the way in which the inputs for the secure protocol ρ are chosen. In general, as for self composition, we need to guarantee security for *adaptively* chosen inputs. This means that the honest parties may choose their inputs to ρ as a function of their view in the arbitrary network π *and* their outputs from executions of ρ that have already concluded. Of course, a party's actions in the arbitrary network may also be influenced by its outputs from executions of ρ that have already concluded.[9]

One way to formally define the above would be to take a similar approach as for self composition. Specifically, party P_i's inputs to ρ are determined by an input-selecting machine M_i which is applied to P_i's view in the arbitrary protocol π and the outputs that it has received from executions of ρ that have already concluded. (This suffices for having the inputs to future executions of ρ depend on π and on previous ρ-outputs. However, π itself may also depend on ρ-outputs and thus π should be able to refer to already obtained ρ-outputs.) Then, security is defined by requiring that for every protocol π and all input-selecting machines, an adversary interacting with the parties in a real execution of π with ρ can do no more harm than an adversary that works in a scenario where π is unchanged, yet the executions of ρ are replaced with ideal calls to f.

A simpler and equivalent approach is to model the arbitrary network activity π as a "calling protocol" with respect to the functionality f. That is, π is a protocol that contains, among other things, "ideal calls" to a trusted party that computes a functionality 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 f in protocol π with real executions of protocol ρ. (When we say that an ideal call to 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 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 f, an adversary interacting with the composed 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 f. This therefore means that ρ behaves just like an ideal call to f, even when it is run concurrently with any arbitrary protocol π. (This formulation is equivalent to the previous one. In order to see this, just think of π as incorporating the input-selecting machines M_i. Since quantification is over all protocols π and all input-selecting machines M_i, these can incorporated into one.)

Our formalization of security under concurrent general composition is based on the "modular" composition operation from [29, 4, 5]. In order to define security, we first define the *hybrid model,* where a protocol π can utilize ideal calls to a trusted party, and the *real model,* where protocols are run without any trusted help. Following this, we proceed to define what it means for a protocol to be secure.

[9]This framework is very general in that it allows information to flow from ρ to π and vice versa. We remark that in Section 5.2, we show that impossibility results for concurrent general composition hold even if the inputs to ρ and π are independent and fixed before any protocol execution begins.

The hybrid model. Let π be an arbitrary probabilistic polynomial-time protocol[10] that utilizes ideal interaction with a trusted party computing a two-party functionality f. This means that π contains two types of messages: standard messages and ideal messages: A standard message is one that is sent between the parties that are participating in the execution of π; an ideal message is one that is sent by a participating party to the trusted third party, or from the trusted third party to a participating party. This trusted party computes f on the messages that it receives from the parties, and returns the output ($f_1(x_i, y_i)$ to P_1 and $f_2(x_i, y_i)$ to P_2). 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 standard messages of π are sent directly between the parties, and the trusted party is only used in the ideal calls to f.

The interaction with the trusted party is exactly according to the description of *concurrent executions in the ideal model*, as described in Section 2.1. In contrast, the standard messages are dealt with exactly according to the description of the *real model*, as described in Section 2.1. More formally, computation in the hybrid model proceeds as follows. The computation begins with the adversary receiving the input and random tape of the corrupted party. Throughout the execution, the adversary sends any standard messages that it wishes to the honest party. In addition, it sends the trusted party the adversary's ideal messages as defined in Section 2.1 (these include start-session and send-output messages, along with adversarially generated inputs; as described there, these messages also include unique indices in order to differentiate sessions). The honest party always follows the specification of protocol π. Specifically, upon receiving a message (from the adversary or trusted party), the party reads the message, carries out a local computation as instructed by π, and sends standard and/or ideal messages, as instructed by π. At the end of the computation, the honest party writes the output value prescribed by π on its output tape and the adversary outputs an arbitrary function of its view. Let n be the security parameter, let $\mathcal{S}$ be an adversary for the hybrid model with auxiliary input z, and let $x, y \in \{0,1\}^*$ be the parties' respective inputs to π. Then, the hybrid execution of π with functionality f, denoted $\textsc{hybrid}^{f}_{\pi,\mathcal{S}}(n, x, y, z)$, is defined as the output of the adversary $\mathcal{S}$ and the honest party from the above hybrid execution.

The real model – general composition. Let ρ be a probabilistic polynomial-time two-party protocol for computing the functionality f. Intuitively, the composition of protocol π with ρ is such that ρ takes the place of the interaction with the trusted party that computes f. Formally, each party holds separate probabilistic interactive Turing machines (ITMs) that work according to the specification of protocol ρ for that party (specifically, ρ_1 for P_1 and ρ_2 for P_2). When π instructs a party to send an ideal message α to the trusted party, the party writes α on the input tape of a new ITM for ρ and invokes the machine. Any message that it receives that is marked for an execution of ρ, it forwards to the appropriate ITM; all other messages are answered according to π. (The different executions of ρ are distinguished with indices, as described in Section 2.1. Furthermore, π-messages are distinguished from ρ-messages with a unique index or symbol for π.) Finally, when an execution of ρ concludes and a value β is written on the output tape of the corresponding 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 trusted party. This composition of π with ρ is denoted π^ρ and takes place without any trusted help. Let n be the security parameter, let $\mathcal{A}$ be an adversary for the real model with auxiliary input z, and let $x, y \in \{0,1\}^*$ be the parties' respective inputs to π. Then, the real execution of π with ρ, denoted $\textsc{real}_{\pi^\rho,\mathcal{A}}(n, x, y, z)$, is defined as the output of the adversary

[10]Here too, a protocol is said to be polynomial-time if the running-time of the honest parties in a single execution is polynomial in the security parameter n, irrespective of the length of the input.

$\mathcal{A}$ and the honest party 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 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 f, means that *general composition* is being considered.

Definition 2 (security under concurrent general composition): *Let ρ be a probabilistic polynomial-time two-party protocol and f a two-party functionality. Then, ρ securely realizes f under concurrent general composition if for every probabilistic polynomial-time protocol π that utilizes ideal calls to f and every non-uniform probabilistic polynomial-time real-model adversary $\mathcal{A}$ for π^ρ, there exists a non-uniform probabilistic polynomial-time hybrid-model adversary $\mathcal{S}$ such that*

$$\left\{\text{HYBRID}^f_{\pi,\mathcal{S}}(n,x,y,z)\right\}_{n\in\mathsf{N};x,y,z\in\{0,1\}^*} \stackrel{\text{c}}{\equiv} \left\{\text{REAL}_{\pi^\rho,\mathcal{A}}(n,x,y,z)\right\}_{n\in\mathsf{N};x,y,z\in\{0,1\}^*}$$

Note that non-trivial protocols are also defined for general composition. Once again, the requirement is that if $\mathcal{A}$ instructs the corrupted party to act honestly in the execution of ρ, then the honest party receives its output from ρ.

Remark. The description of the hybrid model execution here differs from that of [27]. Specifically, here the adversary explicitly instructs the honest party to send an input to the trusted party (by sending a start-session message), and explicitly instructs the trusted party to answer the honest party. In contrast, in the definition of [27], the trusted party just receives inputs and computes outputs, and the adversary is in control over the delivery of messages between the parties and the trusted party. As can easily be seen, there is actually no difference between the models because delivering inputs and outputs is equivalent to instructing the parties to send the messages.

Generalizations of the model. The model here for concurrent general composition can be generalized in the same ways as described for concurrent self composition. Specifically, the arbitrary protocol π can be defined for a large set of parties, and the ideal calls to f can be for different (possibly overlapping) subsets of the parties. Furthermore, the model can be generalized so that the protocol ρ can work on inputs of length that is not a priori bounded by any polynomial (as is needed for encryption and signatures). As we have described above, one way to do this is to allow ρ to run in time that is polynomial in the input length, but π (and the adversary) must still run in time that is polynomial in the security parameter. A second alternative is to also allow π and the adversary to run in time that is polynomial in the input length. However, in this case, we must limit x, y and z to be of length that is polynomial in the security parameter n. As before, we require that for every adversary $\mathcal{A}$ and protocol π, there exists a simulator $\mathcal{S}$ such that for every polynomial $q(\cdot)$ and all sufficiently large n's, the result of a hybrid execution of π with inputs of length $q(n)$ is indistinguishable from the result of a real execution of π^ρ with inputs of length $q(n)$.

2.3 Functionalities With Fixed Versus Interchangeable Roles

Our formulation of secure two-party computation under concurrent self and general composition assumes that the same party plays the same role in every execution. Therefore, if the functionality being considered is that of 1-out-of-2 oblivious transfer, defined by $((x_0, x_1), \sigma) \rightarrow (\lambda, x_\sigma)$, then party P_1 always plays the "sender", and P_2 always plays the "receiver". Likewise, for the zero-knowledge functionality, one party always plays the prover while the other party always plays the verifier. This is very limiting, and many real applications would require more flexibility regarding the roles. (Note that in some settings, flexibility is not required. For example, in an asymmetric setting of clients and servers, a client will never play a server role and vice versa.)

In order to deal with the case that both parties may play both roles in the computation (e.g., both parties may be provers or verifiers in the zero-knowledge proofs), we define a special class of functionalities that we call "functionalities with interchangeable roles". Such a functionality enables both parties to play both roles, by having part of the input determine who plays which role. In general, let $f = (f_1, f_2)$ be any two-party functionality. Then, we say that $g = (g_1, g_2)$ computes f with interchangeable roles if it computes the following functionality:

$$g((x,\alpha),y) = \begin{cases} (f_1(x,y), f_2(x,y)) & \text{if } \alpha = 1 \\ (f_2(y,x), f_1(y,x)) & \text{if } \alpha = 2 \end{cases}$$

Thus, the index α input by the first party determines which party plays the role of "party 1" and which party plays the role of "party 2". More concretely, in the oblivious transfer functionality, the first party sends its input along with the index $\alpha = 1$ if it is the sender, and $\alpha = 2$ if it is the receiver. In this way, either party can be the sender or receiver, depending on which index is input.

Protocols with interchangeable roles. Our aim in the above definition of functionalities with interchangeable roles is to capture the natural setting where different parties may play different roles in different executions. However, a more natural way to define this would be in the context of the *protocol*, rather than of the *functionality*. Specifically, let ρ be a protocol where ρ_1 and ρ_2 are the instructions for P_1 and P_2, respectively. Then, instead of party P_i always using instruction ρ_i, it is possible to allow both P_1 and P_2 to use instructions ρ_1 and ρ_2, interchangeably. The specific role taken by each party could be mutually negotiated upon initiating the protocol.[11] This is in contrast to our definition of *functionalities* with interchangeable roles. Here, we still consider the case that party P_1 always follows ρ_1 and party P_2 always follows ρ_2. The only difference is that the functionality definition enables the parties to interchange roles with respect to the functionality input and output.

We note that any protocol that is secure under the definition where both P_1 and P_2 can use the instructions ρ_1 and ρ_2 interchangeably, is also secure for our definition of functionalities with interchangeable roles. However, the converse does not hold. (See [25, 30] for examples of protocols that are secure under bounded-concurrent self composition, as long as the same party always plays the same role in the protocol.) In this paper we prove lower bounds and impossibility results. Therefore, considering "weaker" definitions (i.e., ones that are easier to achieve) strengthens our results. We note that the proof of equivalence between self and general composition holds also for the case that both P_2 and P_2 use ρ_1 and ρ_2 interchangeably (and also for other generalizations that have been mentioned).

2.4 Functionalities That Enable Bit Transmission

Informally speaking, a functionality enables bit transmission if it can be used by the parties to send bits to each other. For example, the "less than" functionality enables bit transmission, as follows.

[11]This extension would be *necessary* in the case that many different (and possibly intersecting) pairs of parties run the protocol. This is because it may be the case that two parties who are currently running ρ_1 in other executions wish to begin an execution with each other. In this case, one of them must use the instructions of ρ_2, meaning that it does not always play the same role.

If P_1 wishes to send a bit to P_2, then P_2 fixes its input at a predetermined value (say 5). Now, if P_1 wishes to send P_2 the bit 0, then it uses input 4, and if it wishes to send P_2 the bit 1 then it uses input 6. In this way, P_2 will know which bit P_1 sends based on the output. More generally, P_1 can transmit a bit to P_2 if there exists an input y for P_2 and a pair of inputs x and x' for P_1 such that $f_2(x,y) \neq f_2(x',y)$. This suffices because P_1 and P_2 can decide that $f_2(x,y)$ should be interpreted as bit 0 and $f_2(x',y)$ as bit 1. Likewise, P_2 can transmit a bit to P_1 if the reverse holds. We say that a functionality enables bit transmission if it can be used by P_1 to transmit a bit to P_2 and vice versa. Thus, a functionality can only enable bit transmission if both parties receive output (it is impossible to transmit a bit to a party that receives no output). We now present the formal definition:

Definition 3 (functionalities that enable bit transmission): *A deterministic functionality $f = (f_1, f_2)$* enables bit transmission from P_1 to P_2 *if there exists an input y for P_2 and a pair of inputs x and x' for P_1 such that $f_2(x,y) \neq f_2(x',y)$. Likewise, $f = (f_1, f_2)$* enables bit transmission from P_2 to P_1 *if there exists an input x for P_1 and a pair of inputs y and y' for P_2 such that $f_1(x,y) \neq f_1(x,y')$. We say that a functionality* enables bit transmission *if it enables bit transmission from P_1 to P_2* and *from P_2 to P_1.*

We note that the notion of enabling bit transmission can be generalized to probabilistic functionalities in a straightforward way. Specifically, it is required that there exists an input y, a pair of inputs x and x', and a probabilistic polynomial-time distinguishing algorithm A such that $\Pr[A(f_2(x,y)) = 1] > 1 - \mu(n)$ and $\Pr[A(f_2(x',y)) = 1] < \mu(n)$, where μ is some negligible function. If these conditions hold, then P_2 can input y and use A to determine whether P_1 used input x or x' (and thereby determine whether P_1 intended to send 0 or 1).

Example – zero knowledge. The zero knowledge functionality $((x,w),\lambda) \rightarrow (\lambda, (x, R(x,w)))$ for a relation R enables bit transmission from P_1 to P_2, but *not* from P_2 to P_1. (Actually the functionality only enables bit transmission from P_1 to P_2 when there exists a pair (x,w) such that $R(x,w) = 1$ and another pair (x',w') such that $R(x',w') = 0$. This holds as long as $R \neq \phi$ and $R \neq \{0,1\}^* \times \{0,1\}^*$.)

Functionalities with interchangeable roles. Let $f = (f_1, f_2)$ be a functionality and let g be the functionality that computes f with interchangeable roles. Then, g enables bit transmission if f enables bit transmission from P_1 to P_2 **or** if f enables bit transmission from P_2 to P_1. This follows directly from Definition 3 and from the definition of g.

We note that if f does not enable bit transmission from P_1 to P_2 or from P_2 to P_1, then f can be securely computed by each party through local computation only. Specifically, if f does not enable bit transmission from P_1 to P_2, then for every x, x', y we have $f_2(x,y) = f_2(x',y)$. In other words, f_2 depends only on the value of y and so P_2 can compute its output $f_2(x,y)$ based solely on its own local input y. An analogous claim holds if f does not enable bit transmission from P_2 to P_1. We therefore call such a functionality unilateral, because it can be computed by each party without any interaction with the other.

Definition 4 (unilateral functionalities): *A deterministic two-party functionality $f = (f_1, f_2)$ is* unilateral *if f does not enable bit transmission from P_1 to P_2 or from P_2 to P_1.*

Let f be any functionality that is not unilateral. Then, by the above discussion, we have that the functionality g, that computes f with interchangeable roles, enables bit transmission. That is, we have the following claim:

Claim 2.1 *Let $f = (f_1, f_2)$ be any functionality that is not unilateral. Then, the functionality g that computes f with interchangeable roles enables bit transmission.*

Returning to our example of zero-knowledge, the functionality in which either party can play the role of the prover enables bit transmission (in both directions). This is a natural extension of concurrent zero knowledge and could be called something like "concurrent non-malleable zero knowledge". Thus, as we will see, our results prove that such a zero knowledge functionality *cannot* be securely computed under concurrent self composition; see Section 4.

3 Self Composition Versus General Composition

In this section, we study the relation between concurrent self composition (where a single secure protocol is run many times concurrently, but it is the only protocol being executed) and concurrent general composition (where a secure protocol is run many times concurrently with arbitrary other protocols).

3.1 Equivalence for Functionalities That Enable Bit Transmission

We show that if a functionality f enables bit transmission, then a protocol ρ securely computes f under (unbounded) concurrent self composition if and only if it securely computes f under concurrent general composition. Thus, the difference between self and general composition no longer holds for such functionalities. We stress that there *is* nevertheless a difference between these notions when *bounded* composition is considered. Specifically, security under bounded-concurrency can be achieved for self composition [25, 30], but cannot be achieved for general composition [27]. (By bounded concurrency in the setting of general composition, we mean that the number of executions of the secure protocol is a priori bounded, exactly like in self composition. In contrast, there is no bound on the size of the "calling protocol" π.)

Theorem 5 *Let f be a polynomial-time two-party functionality that enables bit transmission, and let ρ be a polynomial-time protocol. Then, ρ securely computes f under (unbounded) concurrent self composition* if and only if *ρ securely computes f under concurrent general composition.*

Proof: We begin by showing that concurrent general composition implies concurrent self composition. Intuitively, this is obvious because in the setting of general composition, a secure protocol is run many times concurrently to any *other* protocol. If the "other protocol" contains no messages, then we obtain the setting of self composition. We now formally prove the implication. Let ρ be a protocol that securely computes f under concurrent general composition. We show that ρ also securely computes f under concurrent self composition. That is, for every real-model adversary $\mathcal{A}$ controlling a party P_i and every pair of probabilistic polynomial-time input-selecting machines $\overline{M} = (M_1, M_2)$, there exists an ideal-model adversary $\mathcal{S}$ controlling P_i, such that

$$\left\{\text{IDEAL}_{f,\mathcal{S},\overline{M}}(n,x,y,z)\right\}_{n\in\mathsf{N};x,y,z\in\{0,1\}^*} \stackrel{\text{c}}{\equiv} \left\{\text{REAL}_{\rho,\mathcal{A},\overline{M}}(n,x,y,z)\right\}_{n\in\mathsf{N};x,y,z\in\{0,1\}^*}$$

We begin by defining a protocol π that will emulate the setting of concurrent self composition. Let $\mathcal{A}$ be the real model adversary for the setting of concurrent self composition and let $\overline{M} = (M_1, M_2)$ be a pair of probabilistic polynomial-time input-selecting machines. Furthermore, let $t(n)$ be a polynomial upper-bound on the running-time of $\mathcal{A}$. Then, the protocol π for party P_1 with input $x \in \{0,1\}^*$ is defined as follows:

Party P_1 keeps an index counter i that is initialized at 1. Upon receiving a start-session message, P_1 locally computes $x_i = M_1(x, i, \alpha_{i_1}, \ldots, \alpha_{i_j})$, where $\alpha_{i_1}, \ldots, \alpha_{i_j}$ are the outputs already received from the trusted party (the code of M_1 is hardwired into the instructions of π). P_1 then sends x_i to the trusted party computing f. Upon receiving an output $f_1(x_j, y_j)$ from the trusted party, P_1 writes this output on its output-tape. P_1 sends up to $t(n)$ inputs to the trusted party, and then waits until it receives its outputs from the trusted party (recall that $t(n)$ is an upper-bound on the running-time of the adversary $\mathcal{A}$). The instructions for party P_2 are defined analogously. This completes the description of π.

First, observe that π is a polynomial-time protocol. This holds because the honest parties run in time at most $t(n)$ times the complexity of the input-selecting machines, which are polynomial. Second, notice that there exists an adversary $\mathcal{A}'$ for protocol π (in the setting of general composition) such that the output of the honest parties and $\mathcal{A}$ with machines $\overline{M}$ and inputs (x, y, z) for the concurrent self composition of ρ, is identical to the output of the honest parties and $\mathcal{A}'$ upon inputs (x, y, z) in an execution of the composed protocol π^ρ. This follows immediately from the fact that π^ρ does nothing except to emulate many concurrent executions of ρ alone. Now, by the security of ρ under concurrent self composition, we have that there exists a hybrid-model simulator $\mathcal{S}'$ for $\mathcal{A}'$. In a straightforward manner, we can use $\mathcal{S}'$ in order to construct an ideal-model simulator $\mathcal{S}$ for $\mathcal{A}$ in the setting of concurrent self composition. We thus conclude that ρ is secure under concurrent self composition.

We now prove the other (more interesting) direction of the theorem. That is, we show that if ρ computes a functionality f that enables bit transmission and ρ is secure under unbounded concurrent self composition, then ρ is also secure under concurrent general composition. The idea behind the proof is that the parties can use the "bit transmission property" of f in order to emulate an execution of π^ρ, while only running copies of ρ. This can be carried out by sending the messages of π one bit at a time, via ideal calls to f. Thus, it is possible to emulate the setting of concurrent general composition, within the context of concurrent self composition. Before formally proving this, we present some preliminary notations regarding f.

Recall that f enables bit transmission. Therefore, there exist predetermined inputs that the parties can use to transmit bits to each other. Denote by x_0, x_1 and y_{rec} the inputs that are used by P_1 to send a bit to P_2. (That is, in order to send the bit σ, party P_1 inputs x_σ and party P_2 inputs y_{rec}.) Likewise, denote by y_0, y_1 and x_{rec} the inputs that are used by P_2 to send a bit to P_1. We are now ready to prove the theorem.

Let π be a protocol that utilizes ideal calls to f and let $\mathcal{A}_\pi$ be a real-model adversary for π^ρ. Without loss of generality, we assume that in protocol π, each message sent by the parties contains a *single bit* only (this can be achieved by just breaking each message up into individual bits). We also assume that the specified output of π is the entire series of standard and ideal messages received during the execution. We use the fact that ρ is secure under concurrent *self* composition in order to show the existence of a hybrid-model adversary $\mathcal{S}_\pi$ such that

$$\left\{\text{HYBRID}^f_{\pi,\mathcal{S}_\pi}(n, x, y, z)\right\} \stackrel{\text{c}}{\equiv} \left\{\text{REAL}_{\pi^\rho,\mathcal{A}_\pi}(n, x, y, z)\right\}$$

In order to do this, we first construct a real-model adversary $\mathcal{A}_\rho$ and input-selecting machines $\overline{M} = (M_1, M_2)$ for the setting of self composition, such that concurrent executions of ρ with adversary $\mathcal{A}_\rho$, machines $\overline{M}$ and inputs (x, y, z), are identical to an execution of π^ρ with adversary $\mathcal{A}_\pi$ and inputs (x, y, z).

We begin by defining the input-selecting machines $\overline{M} = (M_1, M_2)$. Informally speaking, the machine M_1 contains the specification of protocol π for P_1, with input x. That is, $M_1(x, 1)$

equals the first message sent by P_1 (assuming that P_1 sends the first message in π). Furthermore, $M_1(x, i, \alpha_1, \ldots, \alpha_i)$ computes the next message sent by P_1 in π, when P_1 has input x and its series of incoming (standard and/or ideal) messages equals $\alpha_1, \ldots, \alpha_i$. (Note that although we consider a real-model protocol π^ρ here, the protocol π itself has both standard and ideal messages; recall that in π^ρ the output of ρ is treated by π as if it is an ideal message from the trusted party.) The machine M_2 is exactly the same for P_2; that is, it contains the specification of π for P_2 with input y. More formally, when the specification of π instructs P_1 to send a single bit σ to P_2 (as a standard message), then M_1 outputs x_σ (recall that x_σ is the input to f used by P_1 for transmitting the bit σ). This value x_σ is then used by P_1 in the next execution of ρ. Likewise, when P_1 is supposed to receive a single bit from P_2 (as a standard message), then M_1 outputs x_{rec} (recall that x_{rec} is the input to f used by P_1 for receiving a bit from P_2. Note that after the execution of ρ with the input x_{rec} concludes, the resulting output is interpreted by M_1 to be the bit 0 or 1, according to the specified convention. In addition to the above treatment of standard π-messages, when π instructs P_1 to send an ideal message $\hat{x}$ to the trusted party, then M_1 outputs $\hat{x}$ itself (to be input by P_1 into the next execution of ρ). The definition of M_2 is analogous.

Next, we construct an adversary $\mathcal{A}_\rho$ from $\mathcal{A}_\pi$. Recall that $\mathcal{A}_\rho$ works in a setting of unbounded concurrent self composition of ρ, whereas $\mathcal{A}_\pi$ runs π^ρ; we distinguish between π-messages (belonging to protocol π) and ρ-messages. Upon auxiliary input z, $\mathcal{A}_\rho$ internally invokes $\mathcal{A}_\pi$ with z, and emulates an execution of π^ρ for $\mathcal{A}_\pi$, while actually running many copies of ρ only. It does this as follows: when $\mathcal{A}_\pi$ attempts to send a π-message $\sigma \in \{0,1\}$ to the honest party, $\mathcal{A}_\rho$ sends the same message by running an execution of ρ and inputting v_σ (where v_σ is the input to f used by the corrupted party to send the bit σ to the honest party; note that v_σ equals either x_σ or y_σ, depending on which party is corrupted). Likewise, when $\mathcal{A}_\rho$ receives a bit σ from the honest party through an execution of ρ where it input v_{rec} (as described above, σ is derived by convention from the output received from this execution of ρ), then it internally forwards σ to $\mathcal{A}_\pi$ as if it was received directly from the honest party. When $\mathcal{A}_\pi$ wishes to run an execution of ρ with the honest party, then $\mathcal{A}_\rho$ just forwards all messages between the parties directly. At the conclusion, $\mathcal{A}_\rho$ outputs whatever $\mathcal{A}_\pi$ does. We now claim that,

$$\left\{\text{REAL}_{\rho,\mathcal{A}_\rho,\overline{M}}(n,x,y,z)\right\}_{n\in\mathsf{N};x,y,z\in\{0,1\}^*} \equiv \left\{\text{REAL}_{\pi^\rho,\mathcal{A}_\pi}(n,x,y,z)\right\}_{n\in\mathsf{N};x,y,z\in\{0,1\}^*} \tag{1}$$

In order to see that Eq. (1) holds, notice first that when an honest party runs concurrent executions of ρ using the above-defined input-selecting machine, the result is exactly the same as when the honest party runs π^ρ itself. This is due to the fact that the input-selecting machine just translates π-messages to inputs for ρ and vice versa. Next, notice that $\mathcal{A}_\rho$ also just translates π-messages from $\mathcal{A}_\pi$ to inputs for ρ and vice versa. Therefore, the views of $\mathcal{A}_\pi$ and the honest party in a real execution of π^ρ are *identical* to their views when $\mathcal{A}_\rho$ is the adversary and many executions of ρ take place.

Now, since ρ securely computes f under concurrent self composition, we have that there exists an ideal-model simulator $\mathcal{S}_\rho$ such that,

$$\left\{\text{IDEAL}_{f,\mathcal{S}_\rho,\overline{M}}(n,x,y,z)\right\}_{n\in\mathsf{N};x,y,z\in\{0,1\}^*} \overset{\text{c}}{\equiv} \left\{\text{REAL}_{\rho,\mathcal{A}_\rho,\overline{M}}(n,x,y,z)\right\}_{n\in\mathsf{N};x,y,z\in\{0,1\}^*} \tag{2}$$

It remains to show how to construct a hybrid-model simulator $\mathcal{S}_\pi$ for π with ideal calls to f, from the ideal-model simulator $\mathcal{S}_\rho$ who has many ideal calls to f (and nothing else).

Upon input z, simulator $\mathcal{S}_\pi$ internally invokes $\mathcal{S}_\rho$ with input z, and emulates the setting of many ideal calls to f. In this emulation, $\mathcal{S}_\pi$ plays the role of the trusted party for all of the calls to f that are used for the purpose of sending π-messages. In contrast, other calls to f are sent to the real trusted party. Specifically, when $\mathcal{S}_\rho$ sends a message (i, v_i) to the trusted party computing f, simulator $\mathcal{S}_\pi$ works as follows:

1. If π specifies that the i^{th} message is an ideal one (i.e., v_i is an input to be sent to the trusted party in the hybrid execution with π), then $\mathcal{S}_\pi$ just forwards this to the trusted party.

2. If π specifies that the i^{th} message is a standard bit that is sent by the corrupted party to the honest party (i.e., v_i is an input that is used for bit transmission from the corrupted party to the honest party), then $\mathcal{S}_\pi$ computes the functionality f on the input v_i sent by $\mathcal{S}_\rho$ and the prescribed "receive transmission value" for the honest party (this value is either x_{rec} or y_{rec} depending on which party is corrupted). If the output of f is one of the two prescribed values for bit transmission (i.e., either the value that is pre-determined to denote 0 or the value that is predetermined to denote 1), then $\mathcal{S}_\pi$ sends the appropriate bit to the honest party as a standard message. Otherwise, $\mathcal{S}_\pi$ sends the honest party an invalid message, causing an abort. In addition, $\mathcal{S}_\pi$ simulates the message that $\mathcal{S}_\rho$ expects to receive from the trusted party.

3. If π specifies that the i^{th} message is a standard bit that is sent by the honest party to the corrupted party, then $\mathcal{S}_\pi$ waits to receive a standard π-message σ from the honest party. When it does, it internally emulates the trusted party for an execution with $\mathcal{S}_\rho$ where the honest party inputs its prescribed "send transmission value" in order to send the corrupted party the bit σ (this "transmission value" is either x_σ or y_σ depending on which party is corrupted).

The above instruction are followed until the conclusion of π. $\mathcal{S}_\pi$ then just outputs whatever $\mathcal{S}_\rho$ does and halts.

It follows from the above description that the output of $\mathcal{S}_\pi$ and the honest party in this hybrid execution of π, is identical to the output of $\mathcal{S}_\rho$ and the honest party in many ideal executions of f. That is,

$$\left\{\text{HYBRID}^{f}_{\pi,\mathcal{S}_\pi}(n,x,y,z)\right\}_{n\in\mathsf{N};x,y,z\in\{0,1\}^*} \equiv \left\{\text{IDEAL}_{f,\mathcal{S}_\rho,\overline{M}}(n,x,y,z)\right\}_{n\in\mathsf{N};x,y,z\in\{0,1\}^*} \tag{3}$$

As in the proof of Eq. (1), the simulator $\mathcal{S}_\pi$ perfectly emulates the setting for $\mathcal{S}_\rho$ by just translating messages into the correct format. Therefore, Eq. (3) holds.

By combining Equations (1) to (3), we obtain that ρ is secure under concurrent general composition, as desired. ■

Extension to multi-party functionalities. Theorem 5 can be extended to multi-party functionalities in a straightforward way. Thus, we actually obtain equivalence between concurrent self composition and concurrent general composition for any multi-party functionality that enables bit transmission. (Note that in the multi-party case, we will say that a functionality enables bit transmission if it enables bit transmission between all pairs of parties.)

3.2 Equivalence For Functionalities with Interchangeable Roles

Recall that by Claim 2.1, if a functionality g computes a non-unilateral functionality f with interchangeable roles, then g enables bit transmission. (Recall that a functionality is unilateral if it can be computed without any interaction between the parties; that is, a party's output is completely independent of the other party's input.) Now, if a functionality f is unilateral, then it can be securely computed under both concurrent general composition and concurrent self composition, because only local computation is required. On the other hand, if g computes a functionality f that is *not* unilateral, then it enables bit transmission and so by Theorem 5, it can be computed for concurrent general composition if and only if it can be computed for concurrent self composition. It therefore follows that when interchangeable roles are considered, concurrent self composition and concurrent general composition are equivalent for every functionality f. That is, we have the following corollary:

Corollary 6 *Let $f = (f_1, f_2)$ be any deterministic polynomial-time two-party functionality and let g compute f with interchangeable roles. Then, a polynomial-time protocol ρ securely computes g under concurrent self composition if and only if it securely computes g under concurrent general composition.*

Corollary 6 has important ramifications to a general setting of composition where many different (and possibly intersecting) pairs of parties run a secure protocol ρ. In this natural setting, there are no fixed roles for the parties. We therefore have that obtaining security under self composition is equivalent to obtaining security under general composition, for every functionality.

3.3 Separation for Other Functionalities

In this section, we consider functionalities that do not enable bit transmission and do not compute other functionalities with interchangeable roles. (In other words, we consider functionalities that enable bit transmission in one direction only.) We show that there exists a functionality of this type that *cannot* be securely computed under concurrent general composition, but *can* be securely computed under concurrent self composition. Thus, there do exist examples where it is "easier" to obtain concurrent self composition than concurrent general composition. We prove this separation for the important zero-knowledge proof of knowledge functionality

$$((x, w), \lambda) \to (\lambda, (x, R(x, w)))$$

where R is an NP-relation. Note that this functionality does enable bit transmission from P_1 to P_2, but does not enable bit transmission from P_2 to P_1. (Since bit transmission is enabled from P_1 to P_2, this functionality is not unilateral. Therefore, by Corollary 6, in the setting of interchangeable roles, the functionality can be securely computed under concurrent self composition if and only if it can be securely computed under concurrent general composition. Here, we consider the case that one party *always* plays the prover and the other party *always* plays the verifier. Therefore, Corollary 6 does not apply.)

Proposition 7 *Assume that (weak) one-way functions exist. Then, the zero-knowledge proof of knowledge functionality for an NP-complete relation R cannot be securely computed under concurrent general composition, but can be securely computed under concurrent self composition.*

Proof Sketch: The fact that the zero-knowledge proof of knowledge functionality for an NP-complete relation *cannot* be securely computed under concurrent general composition was shown in [27].[12]

[12]To be exact, in [27] impossibility was shown for the functionality $(x, \lambda) \mapsto (\lambda, f(x))$, where f is weakly one-way. Now, let f be a weak one-way function, and define the relation $R_f = \{(f(x), x)\}$. Then, the zero-knowledge proof of knowledge functionality for R_f is exactly $((f(x), x), \lambda) \mapsto (\lambda, (f(x), 1))$. Since $f(x)$ can be efficiently computed from x, this functionality is equivalent to $(x, \lambda) \mapsto (\lambda, f(x))$, for which impossibility has been demonstrated. Impossibility for the NP-relation R_f, and therefore any NP-complete relation R, follows.

We now sketch a proof that the zero-knowledge proof of knowledge functionality *can* be securely computed under concurrent self composition, for *any* NP-relation R. In this sketch, we assume that the reader is familiar with the literature on both zero knowledge proofs and proofs of knowledge. We show that in order to securely compute the zero knowledge proof of knowledge functionality defined above, one can use *any* concurrent zero-knowledge proof system (e.g., [34]), that is also a stand-alone *proof of knowledge*. (Such a proof system exists by taking the construction of [34] and using a witness indistinguishable proof of knowledge in the last stage.) We also assume that the protocol is such that the honest prover always first checks that its input is correct; i.e., for input (x, w) it first checks that $R(x, w) = 1$). If not, then the honest prover just sends $(x, 0)$ to the verifier (otherwise, it runs the prescribed protocol).

Consider first the case that the verifier is corrupted. In this case, the concurrent zero-knowledge simulator can be used to simulate the view of the verifier. Specifically, the ideal-model adversary can just obtain all of the outputs from the trusted party $(x_1, b_1), \ldots, (x_N, b_N)$, where $b_i = R(x_i, w_i)$, and then run the zero-knowledge simulator on these inputs. More precisely, if $b_i = 0$, then the ideal-model adversary need only simulate the prover sending $(x_i, 0)$ to the verifier (because this is what the honest prover does). In contrast, for all x_i's such that $b_i = 1$, the standard concurrent zero-knowledge simulator can be used. An important point to note here is that since the prover never receives any output, its inputs are effectively fixed at the onset (i.e., $x_i = M(x, i)$ for all i). Therefore, it suffices to use the standard formulation of concurrent zero knowledge where inputs are not chosen adaptively [11].

Next, consider the case that the prover P^* is corrupted. In this case, the ideal-model adversary must extract the witnesses w_i from the prover in order to send them to the trusted party. It does this as follows. First, it chooses random coins $r_1, \ldots, r_N$, so that each r_i is the length of the random string used by the honest verifier in an execution of the protocol, and N is an upper bound on the number of executions. Then, the ideal-model adversary invokes P^* and runs the honest verifier strategy for all executions, using random coins $r_1, \ldots, r_N$. (Note that the prover receives no output from the functionality and so the ideal-model adversary does not yet interact with the trusted party.) Now, let I be the set of indices so that $i \in I$ if the i^{th} session of the proof was accepting. In the next step, the ideal-model adversary will obtain the witnesses for all of these sessions. (For the non-accepting sessions, the ideal-model adversary can just send $(x_j, \perp)$ to the trusted party.)

In order to extract the witnesses, for every $i \in I$, the ideal-model adversary defines a prover P_i^* who invokes P^* and internally runs sessions $1, \ldots, i-1, i+1, \ldots, N$ by running the honest verifier with coins $r_1, \ldots, r_{i-1}, r_{i+1}, \ldots, r_N$. In contrast, session i is run externally. This P_i^* is a stand-alone prover for a proof of knowledge. Therefore, the given knowledge extractor can be used to extract the witness w_i. In this way, all witnesses $\{w_i \mid i \in I\}$ can be obtained, and the ideal-model adversary can send all the pairs (x_i, w_i) to the trusted party. We note that P_i^* will prove the same statement x_i that was proven by P^* in the execution where the ideal-model adversary played the honest verifier in all sessions. This is because the same r_j's are used for all $j \neq i$, and so the view of P^* is the same before the i^{th} session begins. (It is crucial that the same statements x_i are proven because otherwise the distribution over the inputs may be skewed. Note that, unlike the honest prover, the cheating prover may choose the statements to be proven as a function of its view in a real execution.)

It remains to prove that this extraction procedure runs in polynomial time. We show this for the simplified (but incorrect) case that the knowledge extractor for P_i^* runs in expected-time that is exactly $\text{poly}(n)/p_i$, where p_i is the probability that the i^{th} proof from P^* is convincing. (Alternatively, we could use strong proofs of knowledge, see [15].) Now, the probability that the extractor is run at all for the i^{th} session is p_i (because with probability $1-p_i$ the proof of this session was not accepting in the first execution). Furthermore, in this simplified case, the expected number of steps of the extractor for this session is $\text{poly}(n)/p_i$. Therefore, the overall expected time is $\text{poly}(n)$. Since this is repeated at most $N = \text{poly}(n)$ times, we have that the expected overall running time of the extractor, and thus the ideal-model adversary, is expected polynomial-time. We stress that the proof without this simplifying assumption is significantly more difficult and uses techniques from [18].

We conclude that there exists a protocol that securely computes the zero-knowledge proof of knowledge functionality under concurrent self composition. ■

4 Impossibility Results for Concurrent Self Composition

An immediate and important ramification of Theorem 5 and Corollary 6 is that known impossibility results for concurrent *general* composition also apply to unbounded concurrent *self* composition, as long as the functionality in question enables bit transmission (or we consider interchangeable roles). As we will see, this rules out the possibility of obtaining security under concurrent self composition for large classes of two-party functionalities. We stress that the impossibility results are *unconditional*. That is, they hold without any complexity assumptions and for any type of simulation (in particular they are not limited to "black-box" simulation).

Impossibility for concurrent general composition. The following impossibility results for concurrent general composition were shown in [27]:

1. Let $f : \{0,1\}^* \to \{0,1\}^*$ be a deterministic polynomial-time function that is (weakly) one-way. Then, the functionality $(x, \lambda) \to (\lambda, f(x))$ cannot be securely computed under concurrent general composition by any non-trivial protocol. (Recall that a protocol is non-trivial if it generates output when both parties are honest.)

2. Let $f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ be a deterministic polynomial-time functionality. If f depends on both parties' inputs,[13] then the functionality $(x, y) \to (f(x, y), f(x, y))$ cannot be securely computed under concurrent general composition by any non-trivial protocol.

3. Let $f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* \times \{0,1\}^*$ be a deterministic polynomial-time functionality and denote $f = (f_1, f_2)$. If f is *not completely revealing*,[14] then the functionality $(x, y) \to (f_1(x, y), f_2(x, y))$ cannot be securely computed under concurrent general composition by any non-trivial protocol.

Impossibility results for concurrent self composition. Let Φ be the set of functionalities described above that cannot be securely realized under concurrent general composition, and let Ψ be the set of all two-party functionalities that enable message transmission. Applying Theorem 5 to the results of [27], we obtain the following corollary:

Corollary 8 *Let f be a functionality in $\Phi \cap \Psi$. Then, f cannot be securely computed under unbounded concurrent self composition by any non-trivial protocol.*

[13]Formally, a functionality f depends on both inputs if there exist x_1, x_2, y and x, y_1, y_2 such that $f(x_1, y) \neq f(x_2, y)$ and $f(x, y_1) \neq f(x, y_2)$.

[14]The definition of completely revealing functionalities (Definition 10) can be found in Section 5. Intuitively, a functionality is completely revealing if one party can choose an input so that the output of the functionality will then fully reveal the other party's input.

The set of functionalities $\Phi \cap \Psi$ contains all the functionalities ruled out in [27] that also enable bit transmission. For example, Yao's famous millionaires' problem (i.e., the computation of the "less than" functionality), where both parties receive output, is included in this set. Applying Corollary 6, we also obtain the following result:

Corollary 9 *Let f be any functionality in Φ. Then, the functionality g that computes f with interchangeable roles cannot be securely computed under unbounded concurrent self composition by any non-trivial protocol.*

As we mentioned in the proof of Proposition 7, the zero-knowledge proof of knowledge functionality for an NP-complete relation cannot be securely computed under concurrent general composition. Therefore, Corollary 9 shows that in a concurrent setting where both parties prove and verify statements (sometimes called non-malleable concurrent zero-knowledge), it is impossible to securely compute the zero-knowledge proof of knowledge functionality.

5 Communication Complexity Lower Bound

In this section we prove that for a class of functionalities $\mathcal{F}$, if a protocol ρ securely computes a functionality $f \in \mathcal{F}$ under m-bounded concurrent composition, and f enables bit transmission, then ρ must have bandwidth of at least m bits. We prove this for one class of functionalities $\mathcal{F}$, although the proof can be extended to other classes of functionalities that suffer from the impossibility results stated in Section 4.

In order to prove this lower bound, we "unravel" a number of different theorems that are used in order to obtain the impossibility result stated in Corollary 8. The chain of theorems leading to this corollary is as follows:

1. Theorem 5 (in this paper) shows that security under concurrent *self* composition implies security under concurrent *general* composition.
2. In [27], it was shown that security under concurrent *general* composition implies a relaxed variant of universal composability (called *specialized-simulator universal composability*).
3. In [8], impossibility results for *specialized-simulator universal composability* were demonstrated.

Thus, impossibility for concurrent self composition is obtained by combining all of the three above steps. Here, we combine all of these steps into one proof. This is needed for proving the lower bound on the communication complexity of protocols that are secure under m-bounded concurrent self composition. In addition, this enables us to provide a self-contained impossibility result.

5.1 The Lower Bound

Functionalities that are completely revealing. We prove the lower bound for one class of functionalities: those that do not "completely reveal P_1 or P_2's input", and enable bit transmission. In order to state this, we need to formally define what it means for a functionality to be "completely revealing". Loosely speaking, a functionality is completely revealing for party P_1, if party P_2 can choose an input so that the output of the functionality fully reveals P_1's input (for *all* possible choices of that input). That is, a functionality is completely revealing for P_1 if there exists an input y for P_2 so that for every x, it is possible to derive x from $f(x, y)$. For example, let us take the maximum function for a given range, say $\{0,\ldots,n\}$. Then, party P_2 can input $y=0$ and the result is that it will always learn P_1's exact input. In contrast, the less-than function is *not* completely revealing because for any input used by P_2, there will always be uncertainty about P_1's input (unless P_1's input is the smallest or largest in the range).

For our lower bound here, we will consider functionalities over finite domains only. This significantly simplifies the definition of "completely revealing". However, our proof holds for the general case as well; see the full version of [8] for a complete definition.

We begin by defining what it means for two inputs to be "equivalent": Let $f : X \times Y \to \{0,1\}^* \times \{0,1\}^*$ be a two-party functionality and denote $f=(f_1,f_2)$. Let $x_1,x_2 \in X$. We say that x_1 and x_2 are equivalent with respect to f_2 if for every $y \in Y$ it holds that $f_2(x_1,y)=f_2(x_2,y)$. Notice that if x_1 and x_2 are equivalent with respect to f_2, then x_1 can always be used instead of x_2 (at least without affecting P_2's output). We now define completely revealing functionalities:

Definition 10 (completely revealing functionalities over finite domains): *Let $f : X \times Y \to \{0,1\}^* \times \{0,1\}^*$ be a deterministic two-party functionality such that the domain $X \times Y$ is finite, and denote $f=(f_1,f_2)$. We say that the function f_2* completely reveals P_1's input *if there exists a single input $y \in Y$ for P_2, such that for every two distinct inputs x_1 and x_2 for P_1 that are not equivalent with respect to f_2, it holds that $f_2(x_1,y) \neq f_2(x_2,y)$. Complete revealing for P_2's input is defined analogously. We say that a functionality f is* completely revealing *if f_1 completely reveals P_2's input* and *f_2 completely reveals P_1's input.*

If a functionality is completely revealing for P_1, then party P_2 can set its own input to be the "special value" y from the definition, and then P_2 will always obtain the exact input used by P_1. Specifically, given $v=f_2(x,y)$, party P_2 can traverse over all X and find the unique x for which it holds that $f_2(x,y)=v$ (where uniqueness here is modulo equivalent inputs x and x'). It then follows that x must be P_1's input (or at least is equivalent to it). Thus we see that P_1's input is completely revealed by f_2. In contrast, if f_2 is *not* completely revealing for P_1, then there does not exist such an input for P_2 that enables it to completely determine P_1's input. This is because for every y that is input by P_2, there exist two non-equivalent inputs x_1 and x_2 such that $f_2(x_1,y)=f_2(x_2,y)$. Therefore, if P_1's input happens to be x_1 or x_2, it follows that P_2 is unable to determine which of these inputs were used by P_1. Notice that if a functionality is not completely revealing, P_2 may still learn much of P_1's input (or even the exact input "most of the time"). However, there is a *possibility* that P_2 will not fully obtain P_1's input. As we will see, the existence of this "possibility" suffices for proving our lower bound.

Note that we require that x_1 and x_2 be non-equivalent because otherwise, x_1 and x_2 are really the same input and so, essentially, both x_1 and x_2 are P_1's input. Technically, if we do not require this, then a functionality may not be completely revealing simply due to the fact that no y can have the property that $f_2(x_1,y) \neq f_2(x_2,y)$ when x_1 and x_2 are equivalent. This would therefore not capture the desired intuition.

As we have mentioned above, the "less than" function (otherwise known as Yao's millionaires' problem) is not completely revealing, as long as the range of inputs is larger than 2. This can easily be demonstrated.

Bandwidth. The statement of the theorem below refers to the bandwidth of a protocol ρ. This is defined to be the *total number of bits* sent by *both* parties in a protocol execution.

We are now ready to state the lower bound:

Theorem 11 *Let $f = (f_1, f_2)$ be a deterministic polynomial-time two-party functionality over a finite domain that is not completely revealing and enables bit transmission. If a non-trivial polynomial-time protocol ρ securely computes f under m-bounded concurrent self composition, then the* bandwidth *of ρ is greater than or equal to m.*

Proof: As a first step, we note that the proof of Theorem 5 actually proves something stronger than the theorem statement. Before showing this, we first define the bandwidth of a hybrid-model protocol π that utilizes ideal calls to f: the bandwidth of such a protocol equals the total number of bits sent by the parties to *each other*, plus a *single bit* for each call to f.[15] Now, let π be a hybrid-model protocol that utilizes ideal calls to f, and has bandwidth at most m. Then, in the proof of Theorem 5, we actually showed that if f enables bit transmission, then m invocations of ρ suffice for perfectly emulating π^ρ (one invocation for each bit of π and one invocation for replacing each ideal call to f). In other words, for *any protocol π of bandwidth at most m*, an execution of π^ρ can be emulated using m concurrent executions of ρ. Furthermore, this yields a simulator for the hybrid-model execution of π with f. Thus, security under m-bounded concurrent self composition implies security under concurrent general composition for protocols π of bandwidth at most m. We conclude that the following claim holds:

Claim 5.1 *Let f be a polynomial-time two-party functionality that enables bit transmission, and let ρ be a polynomial-time protocol. If ρ securely computes f under m-bounded concurrent self composition, then for every hybrid-model polynomial-time protocol π of bandwidth at most m that utilizes ideal calls to f and for every non-uniform probabilistic polynomial-time real-model adversary $\mathcal{A}$ for π^ρ, there exists a non-uniform probabilistic polynomial-time hybrid-model adversary $\mathcal{S}$ such that*

$$\left\{\text{HYBRID}^f_{\pi,\mathcal{S}}(n,x,y,z)\right\}_{n\in\mathsf{N};x,y,z\in\{0,1\}^*} \stackrel{\text{c}}{\equiv} \left\{\text{REAL}_{\pi^\rho,\mathcal{A}}(n,x,y,z)\right\}_{n\in\mathsf{N};x,y,z\in\{0,1\}^*} \tag{4}$$

We now proceed with the actual proof of Theorem 11. Let $f = (f_1, f_2)$ be a deterministic polynomial-time two-party functionality over a finite domain, such that f enables bit transmission and is also not completely revealing. We prove the theorem for the case that f_2 does not completely reveal P_1's input; the other case is analogously proven. Assume, by contradiction, that there exists a protocol ρ that securely computes f under m-bounded concurrent self composition, and has bandwidth *less than m*. We then show that in such a case, it is possible to construct a protocol π that utilizes ideal calls to f and has bandwidth at most m, such that π has the following property: There exists a real-model adversary $\mathcal{A}$ for π^ρ such that no hybrid-model adversary/simulator $\mathcal{S}$ can cause Eq. (4) of Claim 5.1 to hold. This thereby contradicts Claim 5.1, and we conclude that if ρ securely computes f under m-bounded concurrent self composition, then it must have bandwidth of at least m.

Protocol π of bandwidth m: Protocol π works as follows. Party P_2 receives for input two uniformly chosen values $x \in_R X$ and $y \in_R Y$. (Note that since security must hold for all inputs, it must also hold for uniformly chosen inputs.) Then, P_2 sends the input y to the trusted party for an ideal call to f. In addition, P_2 runs the instructions of P_1 in ρ with input x. At the conclusion, P_2 outputs 1 if and only if the output that it receives from the ideal call to the trusted party is $f_2(x,y)$. This completes the instructions for P_2. Regarding the instructions for Party P_1, it actually makes no difference because this party will always be corrupted in π. Nevertheless, for the sake of

[15]This may seem to be a strange way to count the bandwidth of a hybrid-model protocol. However, what we are really interested in is the bandwidth of a *real* protocol; this is just a tool to reach that aim and defining it in this way simplifies things.

completeness of π, one can define P_1 in an analogous way to P_2. This completes the description of π. Note that by the assumption that ρ has bandwidth of less than m, the protocol π has bandwidth less than or equal to m. (Protocol π consists of one reversed execution of ρ plus one ideal call. Therefore, if ρ has bandwidth $m-1$, then π has bandwidth m; the additional bit being due to the single ideal call to f.)

We stress that P_2's instructions in protocol π are *not* equivalent to its instructions in ρ. This is because in π, party P_2 follows the instructions of P_1 in ρ. However, such behaviour may not be in accordance with ρ, because P_1's instructions in ρ may not be symmetric with P_2's instructions (e.g., see the protocols of [26, 30] that use asymmetrical instructions in an inherent way). Nevertheless, by Claim 5.1, protocol ρ must remain secure for *all* protocols π of bandwidth at most m, and in particular, for the protocol π above.

Real-model adversary $\mathcal{A}$ for π^ρ: Let $\mathcal{A}$ be an adversary who controls the corrupted party P_1. Before describing $\mathcal{A}$, notice that the composed protocol π^ρ essentially consists of two executions of ρ: in one of the executions, each party plays its designated role (these are the ρ-messages) and in the other, the parties play reversed roles (these are the π-messages). Adversary $\mathcal{A}$ works as follows. When P_2 sends the first ρ-message to P_1 (we assume without loss of generality that the first message in ρ is sent by P_2), adversary $\mathcal{A}$ forwards this same message back to P_2 as if it is P_1's first standard π-message to P_2. Then, when P_2 answers this standard π-message (according to P_1's instructions in ρ and with input x), $\mathcal{A}$ forwards it back to P_2 as if it is a ρ-message from P_1.

Since party P_2 runs the ρ-instructions of P_1 in π, the execution of π^ρ with adversary $\mathcal{A}$ amounts to P_2 playing both roles in a single execution of ρ, where input x is used for P_1's role and input y is used for P_2's role. Furthermore, P_2 plays both roles honestly and according to the respective instructions of P_1 and P_2. Therefore, the transcript is identical to the case that two honest parties P_1 and P_2 run ρ with respective inputs x and y. By the security of ρ and the fact that it is a non-trivial protocol, we have that except with negligible probability, P_2 receives the P_2-output from this execution of ρ, and this output must equal $f_2(x, y)$. (This follows from the guaranteed behaviour of a non-trivial protocol when two honest parties participate.) Now, since P_2 outputs 1 in π if and only if it receives $f_2(x, y)$ from the trusted party, we have that it outputs 1 in the π^ρ execution with $\mathcal{A}$, except with negligible probability (recall that in π^ρ, the output from ρ is treated by P_2 as if it was received from the trusted party).

Hybrid-model adversary $\mathcal{S}$ for π: By the assumption that ρ is secure under m-bounded concurrent self composition and from Claim 5.1, we have that there exists a hybrid-model adversary $\mathcal{S}$ such that:

$$\left\{\text{HYBRID}^f_{\pi,\mathcal{S}}(n, \lambda, (x, y), \lambda)\right\} \overset{c}{\equiv} \left\{\text{REAL}_{\pi^\rho,\mathcal{A}}(n, \lambda, (x, y), \lambda)\right\} \tag{5}$$

Notice here that P_2's input is (x, y) as described above and we can assume that P_1's input and the adversary's auxiliary input are empty strings.

We now make an important observation about the hybrid-model simulator $\mathcal{S}$ from Eq. (5). In the ideal execution, with overwhelming probability, $\mathcal{S}$ must send the trusted party an input $\tilde{x} \in X$ such that for every $\tilde{y} \in Y$, $f_2(\tilde{x}, \tilde{y}) = f_2(x, \tilde{y})$, where x is from P_2's input to π. In other words, $\mathcal{S}$ must send the trusted party a value $\tilde{x}$ that is equivalent to P_2's input x. Otherwise, P_2's output from the hybrid and real executions will be distinguishable. In order to see this, recall that in a real execution with $\mathcal{A}$, party P_2 outputs 1 except with negligible probability. Therefore, the same must be true in the hybrid execution. However, if $\mathcal{S}$ sends an input $\tilde{x}$ for which there *exists* at least one $\tilde{y}$ so that $f_2(\tilde{x}, \tilde{y}) \neq f_2(x, \tilde{y})$, then with probability at least $1/|Y|$ party P_2 will output 0, specifically when P_2's input y equals this $\tilde{y}$. (Note that since Y is finite, this is a constant probability. Also, recall that P_2 only outputs 1 if it receives $f_2(x, y)$ from the ideal call to the trusted party.) This argument works because P_2 does not use y in any messages sent to $\mathcal{S}$ in the hybrid-model execution of π. Thus, $\mathcal{S}$ works independently of the choice of y.

Until now, we have shown that the hybrid-model adversary $\mathcal{S}$ can "extract" an input $\tilde{x}$ that is equivalent to x. However, notice that $\mathcal{S}$ does this while essentially running an on-line execution of ρ with party P_1. (Of course, the interaction is actually of π-messages with P_2. Nevertheless, P_2 just plays P_1's role in ρ for this interaction, so this makes no difference.) This means that $\mathcal{S}$ could actually be used by an adversary who has corrupted P_2 and wishes to extract the honest P_1's input, or one equivalent to it. Since f is not completely revealing, this is a contradiction to the security of ρ. We proceed to formally prove this.

A different scenario: We now change scenarios and consider a *single* execution of ρ with an honest party P_1 who has input $x \in_R X$, and a real-model adversary $\mathcal{A}'$ who controls a corrupted P_2. The strategy of $\mathcal{A}'$ is to internally invoke the hybrid-model adversary $\mathcal{S}$, and perfectly emulate for it the hybrid-model execution of π with ideal calls to f. Adversary $\mathcal{A}'$ needs to emulate the trusted party for the ideal call to f that is made by $\mathcal{S}$, as well as the π-messages that $\mathcal{S}$ expects to receive. Notice that in the setting of a hybrid-model execution of π, these π-messages are sent by P_2. However, they are exactly the messages that an honest P_1 would send in a single real-model execution of ρ, with input x. Therefore, $\mathcal{A}'$ forwards $\mathcal{S}$ the messages that it receives from P_1 in its real execution of ρ, as if $\mathcal{S}$ received them from P_2 in a hybrid-model execution of π. Likewise, standard π-messages from $\mathcal{S}$ are sent externally to P_1. At some stage of the emulation, $\mathcal{S}$ must send a value $\tilde{x}$ to the trusted party. $\mathcal{A}'$ obtains this $\tilde{x}$, outputs it and halts.

The view of $\mathcal{S}$ in this emulation by $\mathcal{A}'$ (until $\mathcal{A}'$ halts) is *identical* to its view in a hybrid-model execution of π. Therefore, by the above observation regarding $\mathcal{S}$, it holds that $\tilde{x}$ must be such that for every $\tilde{y} \in Y$, $f_2(\tilde{x}, \tilde{y}) = f_2(x, \tilde{y})$, except with negligible probability. That is, in a single real execution of ρ between an honest P_1 and an adversary $\mathcal{A}'$ controlling P_2, we have that $\mathcal{A}'$ outputs a value $\tilde{x}$ that is equivalent to P_1's input x (except with negligible probability).

It remains to show that in an ideal execution of f, for every ideal-model simulator $\mathcal{S}'$ controlling P_2, the probability that $\mathcal{S}'$ outputs a value $\tilde{x}$ that is equivalent to x, is less than $1 - 1/p(n)$, for some polynomial $p(\cdot)$. This suffices because the real-model adversary $\mathcal{A}'$ does output such an $\tilde{x}$; this therefore proves that there does not exist a simulator for $\mathcal{A}'$, in contradiction to the (stand-alone) security of ρ. Now, in an ideal execution, $\mathcal{S}'$ sends some input $\tilde{y}$ to the trusted party and receives back $f_2(x, \tilde{y})$. Furthermore, $\mathcal{S}'$ sends $\tilde{y}$ before receiving any information about x. Therefore, we can view the ideal execution as one where $\mathcal{S}'$ first sends some $\tilde{y}$ to the trusted party and then P_1's input x is chosen uniformly from X. Now, since f_2 is not completely revealing, we have that for every $\tilde{y} \in Y$, there exist two non-equivalent inputs $x_1, x_2 \in X$ such that $f_2(x_1, \tilde{y}) = f_2(x_2, \tilde{y})$. Since $x \in_R X$, we have that with probability $2/|X|$, party P_1's input x is in the set $\{x_1, x_2\}$. Thus, with probability $2/|X|$, party P_2's output (and so the value received by $\mathcal{S}'$) is $f_2(x_1, \tilde{y}) = f_2(x_2, \tilde{y})$. Given that this event occurred, $\mathcal{S}$ can output a value that is equivalent to x with probability at most $1/2$. (Recall that x_1 and x_2 are not equivalent. Therefore, $\mathcal{S}'$ cannot output a value that is equivalent to both x_1 and x_2. Furthermore, the probability that $x = x_1$ equals the probability that $x = x_2$. In other words, $\mathcal{S}'$ must fail with probability $1/2$ in this case.) We conclude that in the ideal execution, $\mathcal{S}'$ outputs a value that is not equivalent to P_1's input with probability at least $1/|X|$. Thus, the REAL and IDEAL executions can be distinguished with advantage that is at most negligibly smaller than $1/|X|$. Since X is finite, $1/|X|$ is a constant probability and so this contradicts the security of ρ, completing the proof. ∎

Functionalities with interchangeable roles. Recall that by Claim 2.1, if a functionality g

computes a non-unilateral functionality f with interchangeable roles, then g enables bit transmission. (Recall that a functionality is not unilateral if it cannot be computed without any interaction between the parties; that is, there is some dependence on the inputs.) Now, observe that if a functionality is not completely revealing, then it cannot be unilateral. This holds because if a functionality f is unilateral, then all inputs x_1 and x'_1 are equivalent with respect to f_2 and all inputs x_2 and x'_2 are equivalent with respect to f_1. Therefore, f is completely revealing (in a vacuous sense). Combining the above together, we have the following corollary to Theorem 11:

Corollary 12 *Let $f = (f_1, f_2)$ be any deterministic two-party functionality over a finite domain that is not completely revealing and let g compute f with interchangeable roles. If a non-trivial protocol ρ securely computes g under m-bounded concurrent self composition, then the* bandwidth *of ρ is greater than or equal to m.*

Bounded simultaneity. We note that Theorem 11 holds even if at any given time, at most *two* sessions of ρ are running simultaneously. Thus, the lower bound holds even if the number of sessions that are open at any given time is severely limited.

5.2 Concurrent General Composition With Independent Inputs

The proof of Theorem 11 actually yields an interesting new result with respect to concurrent general composition. Notice first that the proof relates almost exclusively to concurrent general composition, with the connection to concurrent self composition stated in Claim 5.1. Essentially, we show that π^ρ cannot be simulated in the hybrid model, by any simulator $\mathcal{S}$. An important observation is that the honest party P_2's input to π is $y \in_R Y$ and its input to ρ is $x \in_R X$. Furthermore, these inputs are fixed before any protocol executions take place. That is, a contradiction is achieved even in a scenario where two protocols with *independent inputs* are run concurrently. (Recall that the honest party's inputs are independently chosen in each execution and that, as far as its concerned, it runs π and ρ without any interaction between them.) This extends previous lower bounds for concurrent general composition that were demonstrated in [27].

We note that this has serious implications. Specifically, one may have thought that as long as independently chosen inputs are used, there is no danger posed by concurrent general composition. This may actually be the case; however, it *cannot be proven* using the simulation (i.e., real/ideal model) based definition adopted here.

6 Black-Box Lower Bounds on Round Complexity

In this section we prove *black-box* lower bounds for m-bounded concurrent secure two-party computation. Specifically, we show that any protocol that securely computes the blind signature and oblivious transfer functionalities under m-bounded concurrent self composition, and can be proven using *black-box* simulation, must have more than m rounds of communication. (In order to simplify the exposition, we count a round of communication to be a pair of messages sent between the parties. Thus, in more common terminology, we actually show that more than $2m$ rounds are required for obtaining m-bounded concurrent self composition.)

Before proceeding, we compare these lower bounds to those presented in Section 5. First, and most significantly, the lower bounds here are only for protocols that can be proven secure via *black-box simulation.* Thus, they do not constitute absolute lower bounds. In fact, it has been shown that any efficient functionality *can* be securely computed under m-bounded concurrent self composition, with a *constant-round* protocol [30]. Nevertheless, the lower bounds do demonstrate that *non-black-box techniques* are essential for achieving round complexity that is lower than m. This has important ramifications since all known (highly) efficient protocols, that use specific number-theoretic hardness assumptions, are proven using black-box simulation. Another important difference between the lower bounds is with respect to the functionalities for which the lower bounds hold. Our non-black-box lower bounds hold for a large class of functionalities (and, as mentioned, this can be extended further). In contrast, we only know how to prove black-box lower bounds for the two specific functionalities of blind signatures and oblivious transfer. Of course, on the other hand, our black-box lower bounds are far more severe than our non-black-box lower bounds (high round complexity is more problematic than high communication complexity). Another "advantage" of our black-box lower bounds is that they hold even if the inputs in all the concurrent executions are fixed *ahead of time,* rather than being chosen adaptively (recall that an adaptive choice of inputs is essential for our non-black-box lower bounds). Finally, we note that the blind signature and oblivious transfer functionalities do *not* enable bit transmission (unless interchangeable roles are considered). Therefore, our non-black-box lower bounds do not hold for these functionalities.

6.1 The Main Result

As we have seen, two-party protocols are proven secure by demonstrating that for every real model adversary $\mathcal{A}$ controlling one of the parties, there exists an ideal model adversary/simulator $\mathcal{S}$ who can simulate a real execution for $\mathcal{A}$. The adversary $\mathcal{S}$ interacts with the trusted third party to whom it sends input and receives output. Typically, $\mathcal{S}$ extracts an input used by $\mathcal{A}$ and sends this to the trusted party. The trusted party then computes the output (based on the input sent by $\mathcal{S}$ and the input of the honest party) and returns the output to $\mathcal{S}$. Having received this output, $\mathcal{S}$ completes the simulation, ensuring that $\mathcal{A}$'s view is consistent with the output received from the trusted party. Since the output of $\mathcal{A}$ depends on the input of the honest party, $\mathcal{S}$ must query the trusted party in order to complete the simulation. The fact that $\mathcal{S}$ needs to *interact* with the trusted party is crucial to the lower bound. This is fundamentally different to the case of zero-knowledge where the verifier's output is always the same (i.e., a bit saying accept). Thus a simulator for zero-knowledge need not have any interaction with an external trusted third party. This difference explains why our lower bound does not apply to concurrent zero-knowledge.

Continuing the above idea, it follows that for $\mathcal{S}$ to simulate m executions of a secure protocol, it must query the trusted party m times. Another key observation is that once the input of an execution has been sent to the trusted party, further rewinding of $\mathcal{A}$ is problematic. This is because $\mathcal{A}$ may choose its inputs depending on the messages it has seen. Therefore, rewinding $\mathcal{A}$ may result in $\mathcal{A}$ modifying its input. Since this modified input would also need to be sent to the trusted party and only one input can be sent to the trusted party in each execution, it is not possible to rewind $\mathcal{A}$ in an effective way.

Finally, consider the following scheduling of messages in a concurrent execution: Between every round of messages in the first execution, place a complete protocol execution. We remark that this scheduling is possible if the protocol has m rounds and remains secure for m concurrent executions. (Also, notice that at any given time, at most two different protocol executions are actually running simultaneously.) Then, by the above discussion, $\mathcal{S}$ must send an input to the trusted party between every round of the first execution (otherwise $\mathcal{S}$ can simulate a complete execution without conversing with the trusted party). Since $\mathcal{S}$ cannot rewind $\mathcal{A}$ behind the points at which input is sent to the trusted party, this implies that $\mathcal{S}$ cannot rewind $\mathcal{A}$ at all in the first execution. However, $\mathcal{S}$ is a black-box simulator and it must be able to rewind in order to successfully simulate. We conclude that one cannot obtain a secure protocol that has m rounds and remains secure for m concurrent executions. Before proceeding to the formal proof, we define what it means for parties to use fixed inputs.

Fixed inputs. We say that an input-selecting machine M generates fixed inputs if its input is a vector $\overline{v} = (v_1, v_2, \ldots)$ and for every i and every α, $M(\overline{v}, i, \alpha) = v_i$. Thus, M outputs the i^{th} element of its input vector, irrespective of the sequence α of previously obtained outputs. In this sense, the sequence of inputs that M generates is fixed *before* any executions take place.

Theorem 13 *There exists a probabilistic polynomial-time two-party functionality f such that every non-trivial protocol that securely computes f under m-bounded concurrent self composition and can be proven using* black-box simulation*, must have more than m rounds of communication. This holds even if the input-selecting machines M_1 and M_2 generate fixed inputs only.*

Proof: We prove this theorem under the assumption that one-way functions exist. This suffices because it has been shown that secure coin tossing (even for a single execution) implies the existence of one-way functions [22]. Therefore, if one-way functions do not exist, the coin tossing functionality already fulfills the requirement of the theorem statement.

Our proof is based on the motivating discussion above and is according to the following outline. We begin by defining a functionality f for which the lower bound holds, and assume by contradiction that there exists an m-round protocol Π that securely computes f under m-bounded concurrent self composition. Next, a specific real-model adversary $\mathcal{A}$ is constructed who controls P_2 and interacts with P_1 in an execution of Π. By the security of Π, there exists an ideal-model simulator $\mathcal{S}$ for $\mathcal{A}$. The adversary $\mathcal{A}$ is constructed so that we can claim certain properties of the simulator $\mathcal{S}$. One important claim about $\mathcal{S}$ will be that in one of the executions, it is essentially unable to rewind $\mathcal{A}$. Furthermore, in this execution, $\mathcal{A}$ basically plays the role of an honest P_2. Therefore, we have that $\mathcal{S}$ can actually "simulate" while interacting with an honest P_2, whom it cannot rewind. This observation is used to construct a new adversary $\mathcal{A}'$ who controls P_1 and attacks P_2. In this attack, $\mathcal{A}'$ invokes $\mathcal{S}$ who proceeds to "simulate" for P_2. The proof is then concluded by showing that this enables $\mathcal{A}'$ to obtain information that cannot be obtained in the ideal-model, contradicting the security of Π.

As mentioned in the above outline, we begin by defining a functionality f for which the lower bound holds. The functionality definition refers to a secure signature scheme [21] which can be constructed from any one-way function [35]. It actually suffices for us to use a *one-time* signature scheme. Before defining the functionality, we present an informal definition of secure one-time signature schemes. A signature scheme is a triplet of algorithms $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$, where Gen is a probabilistic generator that outputs a pair of signing and verification keys (sk, vk), Sign is a signing algorithm and Verify is a verification algorithm. Without loss of generality, we assume that the signing key sk explicitly contains the verification key vk. The validity requirement for signature schemes states that except with negligible probability, honestly generated signatures are almost always accepted; i.e., for almost every $(vk, sk) \leftarrow \mathsf{Gen}(1^n)$ and for every message x, $\mathsf{Verify}_{vk}(x, \mathsf{Sign}_{sk}(x)) = 1$. The security requirement of a signature scheme states that the probability that an efficient forging algorithm M can succeed in generating a valid forgery is negligible. This should hold even when M is given a signature to any single message of its choice. That is, in order for M to succeed, it must generate a valid signature on any message other than the one for which it received a signature. More formally, the following experiment is defined: The generator Gen is run, outputting a key-pair (vk, sk). Next, M is given vk and outputs a message x. Finally, M is given a signature $\sigma = \mathsf{Sign}_{sk}(x)$ and outputs a pair (x^*, σ^*). Then, we say that M succeeded if $\mathsf{Verify}_{vk}(x^*, \sigma^*) = 1$

and $x^* \neq x$. (That is, M output a valid signature on a message other than x.) We say that a one-time signature scheme is secure if for every non-uniform polynomial-time M, the probability that M succeeds is negligible.

We are now ready to define the functionality f, which can be seen as a "blind signature" type functionality [10]:

The blind signature functionality f:

- Inputs:
  - P_1 has a signing-key sk (for a secure one-time signature scheme).
  - P_2 has a verification-key vk and an input $\alpha \in \{0,1\}^n$.
- Output:
  - P_1 receives no output and P_2 receives $\sigma = \mathsf{Sign}_{sk}(\alpha)$.

That is, the functionality can be defined as follows:

$$(sk, (vk, \alpha)) \mapsto (\lambda, \mathsf{Sign}_{sk}(\alpha))$$

There are three important features of this functionality. First, in the ideal model, in order to compute P_2's output, the trusted third party must be queried. (Otherwise, a signature would be generated without access to sk and this would constitute a forgery.) Second, if the verification-key vk given to P_2 is the key associated with sk, then party P_2 can check by itself that it received the correct output. Combined with the first observation, this means that P_2 can essentially check whether or not the trusted party was queried in an ideal execution. Third, P_1 learns nothing about P_2's input. These features are central in the proof of the lower bound.

We note that secure computation requirements are for all inputs and all sufficiently large n's. Therefore, the security of a protocol must also hold when P_1 and P_2 are given randomly generated associated signing and verification keys (i.e., vk and sk such that $(vk, sk) \leftarrow \mathsf{Gen}(1^n)$). From here on, we assume that the inputs of the parties P_1 and P_2 are always such that they have associated keys. That is, in every execution the fixed-input vectors $\overline{x}$ and $\overline{y}$ are such that for every i, x_i and y_i contain associated keys. We also assume that P_2's inputs α and $\mathcal{A}$'s auxiliary input z are uniformly distributed in $\{0,1\}^n$. Once again, since security holds for all inputs, it also holds for uniformly distributed inputs.

Proving the lower bound. In the remainder of the proof, we show that any m-bounded concurrent secure two-party protocol for computing f must have more than m rounds. By contradiction, assume that there exists such a protocol Π with exactly m rounds (if it has less, then empty messages can just be added). Without loss of generality, assume also that the first message of Π is sent by P_2.

We consider a scenario where exactly $m = m(n)$ executions are run; denote these m executions of Π by $\Pi_1, \ldots, \Pi_m$ and denote the ith round of execution Π_j by $\Pi_j(i)$. Also, denote the *messages* sent by P_1 and P_2 in $\Pi_j(i)$ by $\Pi_j(i)_1$ and $\Pi_j(i)_2$, respectively. Finally, denote the inputs of P_1 and P_2 in the ith execution by sk_i and (vk_i, α_i), respectively, where each (vk_i, sk_i) is independently generated of all other pairs (vk_j, sk_j).[16] That is, the fixed-input vectors used by the parties are of the form $\overline{x} = (sk_1, \ldots, sk_m)$ and $\overline{y} = ((vk_1, \alpha_1), \ldots, (vk_m, \alpha_m))$ where $\alpha_i \in_R \{0,1\}^n$ and $(vk_i, sk_i) \leftarrow \mathsf{Gen}(1^n)$ are all independently chosen. We now define an adversary $\mathcal{A}$ for Π.

[16]We could have used the same pair (sk, vk) in all m executions and define the functionality f using a many-time signature scheme.

The adversary $\mathcal{A}$: Adversary $\mathcal{A}$ corrupts party P_2 and schedules the m executions as follows. $\mathcal{A}$ plays $\Pi_1(1)$ (i.e., the first round of Π_1) and then runs the entire execution of Π_2. Next, $\Pi_1(2)$ is run and then the entire execution of Π_3. In general, the $(i-1)^{\text{th}}$ round of Π_1 is run followed by the entire execution of Π_i. Since there are m rounds in Π_1 and m concurrent executions, we have that between every two rounds $\Pi_1(i-1)$ and $\Pi_1(i)$ of Π_1, there is a complete execution of Π_i. See Figure 1 for a diagram of the schedule.

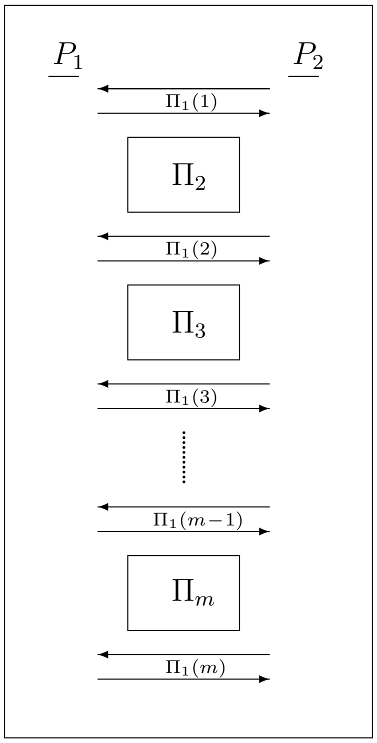

Figure 1: The schedule of the adversary. The arrows refer to messages from execution Π_1.

We now describe $\mathcal{A}$'s strategy within each execution. For the first execution Π_1, adversary $\mathcal{A}$ replaces its input (vk_1, α_1) with (vk_1, z); recall that z is $\mathcal{A}$'s auxiliary input and that it is uniformly distributed in $\{0,1\}^n$. Then, $\mathcal{A}$ runs execution Π_1 on this input, following the protocol instructions exactly. So, apart from replacing part of its input, $\mathcal{A}$ plays exactly like the honest P_2 in execution Π_1.

For all other executions Π_i, adversary $\mathcal{A}$ behaves as follows. Recall that execution Π_i follows immediately after round $\Pi_1(i-1)$ (i.e., immediately after $\mathcal{A}$ receives message $\Pi_1(i-1)_1$). Now, $\mathcal{A}$ takes the input (vk_i, α_i) of P_2 and replaces α_i with $\Pi_1(i-1)_1$. Actually, the input α_i to f must be of length n. Therefore, $\mathcal{A}$ uses a pseudorandom function F [17] with a random key k, and replaces α_i with $F_k(\Pi_1(i-1)_1)$. Formally, the key k must be randomly chosen and included in $\mathcal{A}$'s auxiliary input; we omit this in order to simplify notation. (Recall that pseudorandom functions can be constructed from any one-way function, as needed.) After replacing α_i with $F_k(\Pi_1(i-1)_1)$, adversary $\mathcal{A}$ honestly follows the protocol instructions of P_2 in Π_i. By the definition of f, $\mathcal{A}$'s output from Π_i should be a value σ_i that constitutes a valid signature on $F_k(\Pi_1(i-1)_1)$ using signing-key sk_i (i.e., σ_i is such that $\mathsf{Verify}_{vk_i}(\sigma_i, F_k(\Pi_1(i-1)_1)) = 1$). Now, $\mathcal{A}$ checks that this holds for σ_i. If yes, $\mathcal{A}$ proceeds as in the above-described schedule. Otherwise,

$\mathcal{A}$ sends $\perp$, outputs the output values from already completed executions (i.e., $\sigma_2, \ldots, \sigma_{i-1}$) and aborts. At the conclusion, $\mathcal{A}$ outputs all of its inputs and outputs from the protocol executions.

(Before proceeding with the proof, we provide some motivation to this construction of the adversary $\mathcal{A}$. The basic idea is to prevent any simulator $\mathcal{S}$ from effectively "rewinding" $\mathcal{A}$ in the execution of Π_1. This is achieved as follows. First, $\mathcal{A}$ aborts unless its output from Π_i is a valid signature on $F_k(\Pi_1(i-1)_1)$, where validity is with respect to vk_i. Second, in an ideal-model execution, $\mathcal{S}$ can obtain at most one signature that is valid with respect to the verification key vk_i. (This is because $\mathcal{S}$ does not know the signing key sk_i. Therefore, unless $\mathcal{S}$ can forge signatures, it can only obtain a signature by querying the trusted third party. However, the trusted party can only be queried once in each execution, and so only provides a single signature with key sk_i.) Third, by the collision resistance of pseudorandom functions,[17] $\mathcal{S}$ cannot generate a message $\Pi_1(i-1)'_1$ such that $F_k(\Pi_1(i-1)'_1) = F_k(\Pi_1(i-1)_1)$. Putting this together, we have that there is at most one $\Pi_1(i-1)_1$ message seen by $\mathcal{A}$ for which it does not abort. That is, $\mathcal{S}$ is unable to see $\mathcal{A}$'s response on different $\Pi_1(i-1)_1$ messages, or in other words, $\mathcal{S}$ is unable to rewind $\mathcal{A}$ in Π_1. We note that this heavily utilizes the first two features of the functionality f as described above.

The above explains why $\mathcal{S}$ cannot rewind $\mathcal{A}$ in execution Π_1. This, in itself, does not lead to any contradiction. However, recall that in order for $\mathcal{S}$ to simulate $\mathcal{A}$, it must send $\mathcal{A}$'s input to the trusted third party. This means that $\mathcal{S}$ must successfully extract $\mathcal{A}$'s input from all executions, and in particular from Π_1. We conclude that $\mathcal{S}$ can extract $\mathcal{A}$'s input in Π_1 without rewinding $\mathcal{A}$. Such an $\mathcal{S}$ can therefore be used by an adversary who controls P_1 and attacks P_2, in order to extract P_2's input (P_2 cannot be rewound, but $\mathcal{S}$ does not need to rewind it). Since P_2's input remains secret in f, this contradicts the security of the protocol. This last point utilizes the third feature of f described above; i.e., the fact that the signature is "blind".)

Ideal-model simulation of $\mathcal{A}$. By the assumption that Π is black-box secure, we have that there exists a black-box simulator $\mathcal{S}$ such that the result of an ideal execution with $\mathcal{S}$ is indistinguishable from the result of a real execution of $\mathcal{A}$ running $\Pi_1, \ldots, \Pi_m$. We now make some claims regarding the behavior of $\mathcal{S}$. Recall that $\mathcal{S}$ is given oracle access to the adversary $\mathcal{A}$. Thus, the oracle given to $\mathcal{S}$ is a next message function that receives a series of messages and computes $\mathcal{A}$'s response in the case that $\mathcal{A}$ received these messages in an execution. We note that if $\mathcal{A}$ would abort after receiving only a prefix of the messages in an oracle query, then its reply to the query is $\perp$. We now prove a special property of all oracle calls in which $\mathcal{A}$ does not abort.

Claim 6.1 *For every i, let Q_i be the set of all oracle queries sent by $\mathcal{S}$ to $\mathcal{A}$ during the ideal-model execution which include all messages from the Π_i execution and where $\mathcal{A}$ does not abort.*[18] *Then, except with negligible probability, the same message $\Pi_1(i-1)_1$ appears in every $q \in Q_i$.*

Proof: The proof of this claim is based on the security of the signature scheme and the collision resistant property of random (and thus pseudorandom) functions. First, we claim that except with negligible probability, $\mathcal{S}$ does not produce two oracle queries q and q' containing $\Pi_1(i-1)_1 \neq \Pi_1(i-1)'_1$, such that $F_k(\Pi_1(i-1)_1) = F_k(\Pi_1(i-1)'_1)$. Otherwise, we can construct a machine M that distinguishes F_k from a random function. Machine M works by emulating the entire experiment with $\mathcal{A}$ and $\mathcal{S}$, including choosing all inputs. The only difference is that instead of $\mathcal{A}$

[17]We use pseudorandom functions and not collision resistant hash functions so that we can rely on the existence of one-way functions only.

[18]That is, we consider all of the oracle queries made by $\mathcal{S}$ to $\mathcal{A}$ throughout the simulation and take the subset of those queries which include all the messages of Π_i. By the scheduling described in Figure 1, such a query defines the i^{th} message of Π_1 that is sent by $\mathcal{A}$ (i.e., $\Pi_1(i)_2$).

computing $\alpha_i = F_k(\Pi_1(i-1)_1)$, machine M queries its oracle with $\Pi_1(i-1)_1$ and sets α_i to equal its oracle response. (Recall that M has oracle access to a function that is either truly random or equals $F_k(\cdot)$, for a random key k.) Now, on the one hand, if M has oracle access to $F_k(\cdot)$, then M's emulation for $\mathcal{S}$ is perfect. Therefore, with non-negligible probability M obtains from $\mathcal{S}$ two messages $\Pi_1(i-1)_1 \neq \Pi_1(i-1)'_1$ such that its oracle response is the same. On the other hand, if M has oracle access to a random function, then it will see such a collision with at most negligible probability. Therefore, with non-negligible probability, M distinguishes a pseudorandom function from a random one.

We now prove the claim under the assumption that such oracle queries q and q' are never produced by $\mathcal{S}$. Intuitively, in this case, the claim follows from the security of the signature scheme. That is, $\mathcal{S}$ is allowed to query the trusted party for the i^{th} execution only once, and this query provides $\mathcal{S}$ with a signature such that if the signature is on the message x where $x = F_k(\Pi_1(i-1)_1)$, then $\mathcal{A}$ does not abort. Now, if there are two oracle queries with different $\Pi_1(i-1)_1$ messages (and thus with different $F_k(\Pi_1(i-1)_1)$ values) such that $\mathcal{A}$ does not abort in either of them, then it must be that $\mathcal{S}$ provided $\mathcal{A}$ with valid signatures on both $F_k(\Pi_1(i-1)_1)$ values. Since only one signature could have been obtained from the trusted party, $\mathcal{S}$ must have generated a forgery.

More formally, we construct a signature-forging algorithm M that is given vk and simulates $\mathcal{A}$ and the trusted party for $\mathcal{S}$. That is, M chooses the inputs for both parties in every execution Π_j for $j \neq i$. This means that M knows the signing-key sk_j for all of these executions. In contrast, the verification-key vk_i of the i^{th} execution is set to be the verification-key vk received by M. The forger M then perfectly emulates an ideal-model execution for $\mathcal{S}$. This involves emulating $\mathcal{A}$ and the trusted party. Note that the trusted party can be emulated with no problem in all executions except for Π_i (because M knows all the inputs, including the signing keys). In contrast, in order to emulate the trusted party for the i^{th} execution, M needs to be able to sign with the key that is associated with vk. M does this by querying its signing oracle. Thus, the emulation is perfect.

Now, assume that with non-negligible probability, there exist two queries $q, q' \in Q_i$ with different messages $\Pi_1(i-1)_1$ and $\Pi_1(i-1)'_1$, respectively. Since $\mathcal{A}$ does not abort, it must have received valid signatures for both $F_k(\Pi_1(i-1)_1)$ and $F_k(\Pi_1(i-1)'_1)$ under the verification key $vk_i = vk$. (Recall that in this case, $F_k(\Pi_1(i-1)_1) \neq F_k(\Pi_1(i-1)'_1)$.) Now, when $\mathcal{A}$ halts and its view includes the prefix q, its output includes a signature on $F_k(\Pi_1(i-1)_1)$ (because $\mathcal{A}$ outputs the output it received in all executions before it possibly aborts). Likewise, when $\mathcal{A}$ halts and its view includes the prefix q', its output includes a signature on $F_k(\Pi_1(i-1)'_1)$. We conclude that by continuing the emulation with both these prefixes, M can obtain $(F_k(\Pi_1(i-1)_1), \sigma)$ and $(F_k(\Pi_1(i-1)'_1), \sigma')$ that constitute two valid message/signature pairs for the key vk. However, during the ideal-model execution, $\mathcal{S}$ can only query the trusted party once for the i^{th} execution. Therefore, M queried its signing oracle with at most one of these messages. The other pair thus constitutes a valid forgery and M succeeds with non-negligible probability, in contradiction to the security of the signature scheme. ■

Next, we claim that $\mathcal{S}$ must obtain the input z that is used by $\mathcal{A}$ in the execution of Π_1. That is,

Claim 6.2 *At some point in the ideal execution, except with negligible probability, $\mathcal{S}$ must send the trusted party the pair $(*, z)$ where z is $\mathcal{A}$'s auxiliary input and $*$ denotes any string.*

Proof: In a real execution of $\Pi_1, \ldots, \Pi_m$, the output of $\mathcal{A}$ from Π_1 is a valid signature σ_1 on z; that is, σ_1 is such that $\mathsf{Verify}_{vk_1}(z, \sigma_1) = 1$. This is because $\mathcal{A}$ plays the honest party's strategy in this execution with input (vk_1, z), and thus the execution of Π_1 is exactly like an execution between two honest parties. (The fact that the output is obtained in such an execution follows from the fact that the secure protocol is non-trivial.) Therefore, it follows that in an ideal execution with $\mathcal{S}$, except with negligible probability, the output of $\mathcal{A}$ must also be a valid signature σ_1 on z. This holds because, otherwise, a distinguisher D who receives all the parties' inputs and outputs, including $\mathcal{A}$'s auxiliary input z, could easily distinguish the real and ideal executions.

Now, in an ideal execution, $\mathcal{S}$ is given no information about sk_1 and z. Thus, if $\mathcal{S}$ queries the trusted party with $(*, z)$, it will obtain the necessary signature σ_1 on z. Otherwise, it must forge a signature on z, so that verification with the key vk_1 succeeds. Formally, if $\mathcal{S}$ can obtain a correct σ_1 without querying the trusted party with $(*, z)$, then we can construct a forging algorithm M who breaks the one-time signature scheme. The actual reduction is essentially the same as in the proof of Claim 6.1 and the details are therefore omitted. ■

Essentially, Claim 6.1 tells us that for every i, there is only a single $\Pi_1(i-1)_1$ message sent by $\mathcal{S}$ to $\mathcal{A}$ for which $\mathcal{A}$ does not abort. Thus, any "rewinding" of $\mathcal{A}$ is of no help (i.e., in any rewinding, $\mathcal{S}$ will either have to replay the same messages as before, or $\mathcal{A}$ will just abort). Furthermore, Claim 6.2 tells us that $\mathcal{S}$ succeeds in "extracting" the input z used by $\mathcal{A}$ in the execution of Π_1. These claims are central to our construction of an adversary $\mathcal{A}'$ who corrupts P_1 and uses $\mathcal{S}$ to "attack" P_2. (Notice that we are switching the identity of the corrupted party here.) The adversary $\mathcal{A}'$ will internally invoke $\mathcal{S}$ and forward all messages from Π_1 between $\mathcal{S}$ and the honest P_2 (instead of between $\mathcal{S}$ and $\mathcal{A}$). Now, since $\mathcal{S}$ has only black-box access to $\mathcal{A}$ and cannot rewind $\mathcal{A}$, its interaction with $\mathcal{A}$ is the same as its interaction with P_2. Recall further that $\mathcal{A}$ actually plays the role of the honest P_2 in Π_1, and so it makes no difference to $\mathcal{S}$ whether it is running an ideal execution with $\mathcal{A}$ or a real execution with P_2. This implies that if $\mathcal{S}$ can extract $\mathcal{A}$'s input in the ideal model execution, then $\mathcal{A}'$ (internally running $\mathcal{S}$) can extract P_2's input in a real execution. However, this will result in a contradiction because P_2's input should not be revealed in a secure protocol execution (by the definition of the functionality f).

The adversary $\mathcal{A}'$: Let $\mathcal{A}'$ be an adversary for a single execution of Π who corrupts P_1 and interacts with P_2. In this execution, P_1's input is a signing key sk and P_2's input is the associated verification key vk and a uniformly chosen $\alpha \in_R \{0,1\}^n$. $\mathcal{A}'$ *internally* incorporates $\mathcal{S}$ and emulates the concurrent executions of $\Pi_1, \ldots, \Pi_m$ while *externally* interacting in a single execution of Π with P_2.[19]

Notice that $\mathcal{S}$ interacts with a trusted third party and with $\mathcal{A}$; therefore, $\mathcal{A}'$'s emulation involves emulating both of these entities. $\mathcal{S}$ also expects to receive the input $(vk_1, \alpha_1, \ldots, vk_{m(n)}, \alpha_{m(n)})$ that $\mathcal{A}$ receives, and so $\mathcal{A}'$ must provide this in the emulation as well. (Note that $\mathcal{S}$ does *not* receive $\mathcal{A}$'s auxiliary input z.) Below, we will refer to P_2 as the *external* P_2 in order to differentiate between real interaction with P_2 and internal emulation.

The executions $\Pi_2, \ldots, \Pi_m$ are all internally emulated by $\mathcal{A}'$. That is, $\mathcal{A}'$ selects all the inputs (for both parties) and emulates both $\mathcal{A}$ and the trusted party for $\mathcal{S}$. This emulation can be carried out perfectly because $\mathcal{A}'$ knows all inputs, including the signing keys. However, the emulation of Π_1 is carried out differently. That is, $\mathcal{A}'$ sets $\mathcal{S}$'s input in Π_1 to be (vk, α') where vk is the verification key associated with sk and $\alpha' \in_R \{0,1\}^n$. (Recall that sk is $\mathcal{A}'$'s input in the single execution with the external P_2 and that, by assumption, sk explicitly contains a description of vk. Notice also, that α' has no correlation with the external P_2's input α.) Having

[19]Notice that if the definition allows $\mathcal{S}$ to run in expected polynomial-time, then $\mathcal{A}'$ cannot internally run $\mathcal{S}$ (because it is limited to strict polynomial-time). Nevertheless, this can be overcome as follows. Let $q(n)$ be the polynomial that bounds the expected running time of $\mathcal{S}$. Then, if in the emulation, $\mathcal{S}$ exceeds $2q(n)$ steps, $\mathcal{A}'$ truncates the execution and outputs time-out. This ensures that $\mathcal{A}'$ runs in strict polynomial-time. Furthermore, by Markov's inequality, $\mathcal{A}'$ outputs time-out with probability at most 1/2. Thus, $\mathcal{A}'$'s success probability in its attack will be at most 1/2 of what it would be if it could run $\mathcal{S}$ without truncation. This suffices for our lower bound.

set the input, we now describe how $\mathcal{A}'$ emulates $\mathcal{S}$'s interaction with $\mathcal{A}$ in Π_1. This emulation works by having the external P_2 play the role of $\mathcal{A}$. That is, let q be an oracle query from $\mathcal{S}$ to $\mathcal{A}$, such that $\mathcal{A}$'s response to q is the i^{th} message of execution Π_1. Then, if $\mathcal{A}$ would abort upon receiving q or any prefix of q, adversary $\mathcal{A}'$ emulates $\mathcal{A}$ aborting. (Note that $\mathcal{A}'$ can know if $\mathcal{A}$ would abort because this depends only on $\mathcal{A}$'s output from executions $\Pi_2, \ldots, \Pi_m$, which are all internally emulated by $\mathcal{A}'$.) Otherwise, q is such that $\mathcal{A}$ does not abort, but rather replies with the i^{th} message of Π_1 (i.e., $\Pi_1(i)_2$). Now, if the external P_2 has already sent the i^{th} message of Π, then $\mathcal{A}'$ replays this same message to $\mathcal{S}$ as $\mathcal{A}$'s reply. Otherwise, $\mathcal{A}'$ extracts the $\Pi_1(i-1)_1$ message from q and sends it to the external P_2, who replies with $\Pi_1(i)_2$. $\mathcal{A}'$ records this message and passes it to $\mathcal{S}$ as $\mathcal{A}$'s reply from the query q. This describes how the role of $\mathcal{A}$ in Π_1 is emulated. (The above description implicitly assumes that, without loss of generality, $\mathcal{S}$ queries $\mathcal{A}$ with all prefixes of q before querying q.) In order to complete the emulation, $\mathcal{A}'$ also needs to emulate the trusted party in Π_1. However, this is straightforward because $\mathcal{A}'$ knows the signing key sk (it is its input), and can therefore provide the output as generated by the trusted party. At the conclusion, $\mathcal{A}'$ outputs its view, including all the messages sent by $\mathcal{S}$ to the trusted third party. This completes the description of $\mathcal{A}'$.

Deriving a contradiction. We first claim that $\mathcal{S}$'s view in the emulation by $\mathcal{A}'$ is statistically close to its view in an ideal execution with oracle-access to $\mathcal{A}$. The emulation of $\mathcal{A}$ and the trusted third party in executions $\Pi_2, \ldots, \Pi_m$ is clearly identical. We now demonstrate statistical closeness for execution Π_1. First, note that the input distribution for Π_1 is the same. That is, $\mathcal{S}$ is given a pair (vk, α') for P_2's input, and the external P_2 really uses the pair (vk, α). (Recall that α and α' are independent uniformly distributed n-bit strings.) This is exactly the same as in an ideal execution where $\mathcal{S}$'s input in Π_1 equals (vk, α); yet $\mathcal{A}$ replaces α with z and really uses the pair (vk, z) for input. Second, we claim that the distribution over the messages sent by P_2 is statistically close to the distribution over the non-abort messages answered by $\mathcal{A}$. This holds due to the fact that both $\mathcal{A}$ and P_2 follow the instructions of Π, and from the following observation. Denote by Q the set of all oracle queries q which result in $\mathcal{A}'$ sending a message to the external P_2. Then, except with negligible probability, the same $\Pi_1(i)_1$ messages appear in every $q \in Q$, for every i. This follows directly from Claim 6.1; i.e., otherwise, with non-negligible probability, there exists a j for which Q_j contains two different $\Pi_1(j-1)_1$ messages. Thus, $\mathcal{A}$'s non-abort replies are consistent with an honest party who receives a single series of messages $\Pi_1(1)_1, \ldots, \Pi_1(m)_1$. This also implies that the messages replayed from P_2 to $\mathcal{S}$ are the same as would be sent by $\mathcal{A}$ to $\mathcal{S}$, except with negligible probability. Third, $\mathcal{A}'$ sends $\perp$ to $\mathcal{S}$ in the emulation if and only if $\mathcal{A}$ would send $\perp$ to $\mathcal{S}$ in an ideal execution. Finally, note that since $\mathcal{A}'$ has the signing key sk, it plays the trusted party's role perfectly in the emulation. Combining the above, we have that $\mathcal{S}$'s view in the emulation by $\mathcal{A}'$ is statistically close to its view in an ideal execution. That is, all abort and non-abort oracle replies received by $\mathcal{S}$ in the emulation by $\mathcal{A}'$ are statistically close to what $\mathcal{S}$ would see in an ideal execution with $\mathcal{A}$. Furthermore, the output from the emulated trusted party that $\mathcal{S}$ receives from $\mathcal{A}'$ is identically distributed to the output that $\mathcal{S}$ would receive from the real trusted party in an ideal execution. In conclusion, all messages seen by $\mathcal{S}$ in the two scenarios are statistically close.

Now, by Claim 6.2, except with negligible probability, $\mathcal{S}$ must send $(*, z)$ to the trusted third party at some stage of the ideal execution, where z is the auxiliary input of $\mathcal{A}$ that it uses as input in Π_1. Therefore, with probability at most negligibly less than 1, $\mathcal{S}$ must send $(*, \alpha)$ in the above emulation by $\mathcal{A}'$, where α is the input used by P_2 in Π. In such a case, α is included in $\mathcal{A}'$'s output. In summary, in a real execution of $\mathcal{A}'$ with P_2, adversary $\mathcal{A}'$ obtains P_2's input with probability almost 1. Intuitively, this yields a contradiction because P_2's input is not revealed by the functionality definition. Formally, consider the ideal-model execution for this scenario; let $\mathcal{S}'$ be the ideal-model adversary that is guaranteed to exist for $\mathcal{A}'$ in Π. By the security of the protocol, it must be that $\mathcal{S}'$ outputs P_2's input α with probability at most negligibly less than 1. (Otherwise, the real and ideal executions can be easily distinguished.) However, in such an ideal execution, $\mathcal{S}'$ is given no information on P_2's input α. Since α is uniformly distributed in $\{0,1\}^n$ it follows that $\mathcal{S}'$ can output this value with at most negligible probability. We therefore conclude that such a protocol Π does not exist. ■

Notice that Theorem 13 holds if the simulator for the case that P_2 is corrupted is black-box, irrespective of whether or not the simulator for a corrupt P_1 is also black-box (of course, the roles of P_1 and P_2 can be reversed). We remark that in the protocol of [25], one corruption case is proven using black-box simulation and the other is proven using non black-box simulation. Therefore, Theorem 13 states that any protocol using such a strategy must have round complexity greater than m.

Impossibility of black-box unbounded concurrent self composition. The following corollary is immediately derived from Theorem 13.

Corollary 14 *The blind signature functionality f defined above cannot be securely computed under* unbounded *concurrent self composition while using* black-box simulation.

Notice that Corollary 14 is significantly weaker than the impossibility results provided by Corollary 8 in Section 4. This is both due to the fact that Corollary 14 refers to only a single functionality, and in addition, only rules out protocols that are proven secure using black-box simulation. Nevertheless, it is important to note that the blind-signature functionality does not enable bit transmission. The functionality is therefore not ruled out by Corollary 8.

Non-uniform versus uniform adversaries. Our proof of the lower bound uses the non-uniformity of the adversary $\mathcal{A}$ in an essential way. That is, we require that $\mathcal{A}$ use its auxiliary input z in the execution of Π_1. This is important because, on the one hand, the simulator $\mathcal{S}$ must not have access to the input used by $\mathcal{A}$. On the other hand, the distinguisher must know which input is used by $\mathcal{A}$ in order to check that the signature generated by $\mathcal{S}$ is on the "correct" input. We solve this by using $\mathcal{A}$'s auxiliary input which is not given to $\mathcal{S}$ (but *is* given to the distinguisher). We remark that we do not know whether or not Theorem 13 holds also for uniform adversaries.

6.2 Impossibility For Concurrent Oblivious Transfer

As we have mentioned, Theorem 13 and Corollary 14 refer to the specific blind signature functionality. In this section, we extend the black-box impossibility result to the oblivious transfer functionality; a highly important building block in many secure protocols. Recall that 1-out-of-2 oblivious transfer is defined by: $((x_0, x_1), \sigma) \mapsto (\lambda, x_\sigma)$, where $x_0, x_1 \in \{0,1\}^n$ and $\sigma \in \{0,1\}$ [12].

Corollary 15 *There does not exist a non-trivial protocol that securely computes the* 1-out-of-2 oblivious transfer *functionality under unbounded concurrent self composition and can be proven using black-box simulation.*

Proof Sketch: We show that a protocol for 1-out-of-2 oblivious transfer that remains secure under unbounded concurrent self composition can be used to obtain a protocol for the blind signature functionality that remains secure under unbounded concurrent self composition. Since we have already proven an impossibility result for the blind signature functionality, this implies an impossibility result for the oblivious transfer functionality as well.

We now describe the protocol for blind signatures that uses the 1-out-of-2 oblivious transfer functionality. We remark that since the concurrent setting that we consider is where only a single protocol is run many times concurrently, our protocol for blind signatures must use invocations of the oblivious transfer functionality and *nothing else.* Our protocol uses the specific construction of [24] for a one-time signature scheme. The signature scheme of [24] is defined as follows. Let f be a one-way function. Then, the signing-key equals $2n$ random strings $x_1^0, x_1^1 \ldots, x_n^0, x_n^1$ each of length n, and the verification key equals $y_1^0, y_1^1 \ldots, y_n^0, y_n^1$, where for every i, $y_i^0 = f(x_i^0)$ and $y_i^1 = f(x_i^1)$. Now, a signature on the message $w = w_1 \cdots w_n \in \{0,1\}^n$ equals $x_1^{w_1}, \ldots, x_n^{w_n}$. The verification of a signature is carried by simply checking that for every i, $f(x_i^{w_i}) = y_i^{w_i}$. We are now ready to present the protocol:

Protocol 1 (blind signatures from oblivious transfer):

- Inputs: *Party P_1 (the signer) has $(x_1^0, x_1^1 \ldots, x_n^0, x_n^1)$, and P_2 has $(y_1^0, y_1^1 \ldots, y_n^0, y_n^1)$ and $w \in \{0,1\}^n$.*
- The Protocol:
  1. *P_1 and P_2 run n copies of a protocol for 1-out-of-2 oblivious transfer that remains secure under concurrent self composition.*
  2. *In the i^{th} execution, P_1 inputs (x_i^0, x_i^1) and P_2 inputs w_i.*
  3. *P_2 checks that for every i, $f(x_i^{w_i}) = y_i^{w_i}$ and if yes, outputs $x_1^{w_1}, \ldots, x_n^{w_n}$.*

The proof of security of the protocol is straightforward. That is, in the case that P_1 is corrupted, the ideal-model simulator/adversary obtains all the x_i^b's from $\mathcal{A}$. This is because $\mathcal{A}$ must send each x_i^b to the oblivious transfer functionality. These strings constitute the entire signing key and thus the simulator sends them to the trusted party. Likewise, in the case that P_2 is corrupted, the w_i's sent to each copy of the oblivious transfer protocol constitute the entire input w. Therefore, the ideal-model simulator can send the input to the trusted party and obtain the required signature. The key point is that since the oblivious transfer functionality remains secure under unbounded concurrent self composition, the same is true of the blind signature functionality. However, this contradicts Corollary 14.

We note that the exact lower bound on the number of rounds of oblivious transfer derived above is as follows: any protocol that securely computes the oblivious transfer functionality under m-bounded concurrent self composition must have more than m/n rounds. This loss of a factor of n is due to the fact that n copies of the oblivious transfer functionality are needed for every blind signature execution. ■

Remark. Notice that Corollary 15 seems to contradict [14], where a protocol for concurrent oblivious transfer was demonstrated. However, the model of concurrency of [14] is different to ours in an essential way. Specifically, they assume that the inputs of the parties are chosen *independently in every execution.* For simplicity, one can think of the inputs in each execution as being chosen uniformly from a fixed and given domain. This implies that the parties' inputs are not correlated, and furthermore, that each party's inputs in different executions are also not correlated.

In contrast, our model follows the more standard definition where quantification is over all inputs (and, in particular, over correlated inputs). Specifically, our definition implies security even in the case that both parties' fixed-input vectors are generated from a *single* distribution $\overline{XY}$ (with a single random tape). Furthermore, we use this correlation in an essential way in our lower bound.

6.3 Extensions of the Black-Box Lower Bounds

We now show extensions of our lower bounds to more relaxed definitions of security.

Restricting the input correlations. As described above, the current formulation of the lower bound considers a case where all inputs can be correlated. However, the bound can also be shown to hold when some independence of inputs exists. We consider a number of possible models:

1. First consider a model where the inputs of P_1 and P_2 may be correlated in any given execution, but the inputs between different executions are independent. Formally, in this model the fixed-input vectors are chosen according to a series of fixed and known distributions $\overline{XY} = XY_1, \ldots, XY_m$, where for each i, $XY_i(r_i)$ outputs a pair of inputs (x_i, y_i) for the parties, and for every $i \neq j$, the random coins used by XY_i and XY_j are independent. The fact that the distributions are "known" is formalized by saying that for every adversary $\mathcal{A}$ and series of distributions $\overline{XY}$, there exists a simulator $\mathcal{S}$ and so on. Thus, a different ideal-model simulator is allowed for every series of distributions.

   We claim that the lower bound holds also for this model. In fact, the proof of Theorem 13 holds without any modification. This follows from the fact that the input distribution considered in the proof is such that the inputs in different executions are chosen independently of each other. Specifically, in a given execution, the inputs are a pair of signature keys $(vk, sk) \leftarrow \mathsf{Gen}(1^n)$ and a random string $\alpha \in_R \{0,1\}^n$. However, in different executions, different (and independently chosen) signing keys are used. Therefore, the random coins used in generating the inputs are independent in every execution.

2. In the above model, the parties have correlated inputs, but the auxiliary input of the adversary is independent. However, one can also consider a model where the parties' inputs are independent of each other, but the adversary's auxiliary input may be correlated. That is, define the (known) input distributions as $\overline{XYZ} = X_1, Y_1, Z_1, \ldots, X_m, Y_m, Z_m$, where the random coins used in each X_i and Y_j are independent (including when $i = j$). However, Z_i may be correlated to X_i and Y_i (formally, give Z_i the coins r_i^x and r_i^y used by X_i and Y_i, respectively). As above, the proof of Theorem 13 also works here. The only difference is that $\mathcal{A}$'s auxiliary input is defined to be $(z, vk_1, \ldots, vk_m)$, where $z \in_R \{0,1\}^n$ and vk_i is the verification key associated with sk_i that P_1 receives for input in the i^{th} execution. In such a case, the input of P_2 does *not* include vk_i (since this would constitute correlated input); rather, P_2 receives a random α_i only. The rest of the proof remains the same.

In contrast to the above models where Theorem 13 holds, consider a further relaxation where the inputs of the parties are chosen independently of each other and independently of the inputs in other executions. This is the model considered by [14]. Since they demonstrated the feasibility of oblivious transfer in the model, we know that Theorem 13 does *not* hold here. We remark, however, that this model is rather restricted as it does not allow any correlation of inputs between the parties, or any correlated auxiliary knowledge given to the adversary.

Acknowledgements

I would like to express deep thanks to Ran Canetti for his help in this work. Among other things, some key ideas in the proof of the black-box lower bound are due to Ran. In addition, the formulation of the definitions of security under composition was greatly influenced by Ran. Thanks also to Jonathan Katz for discussions regarding the feasibility of securely computing the zero-knowledge proof of knowledge functionality under concurrent self composition. I would also like to thank Boaz Barak and Tal Rabin for many helpful discussions. Finally, thanks to Phil MacKenzie for clarifications regarding [14].

References

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

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

[3] 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).

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

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

[6] R. Canetti and M. Fischlin. Universally Composable Commitments. In *CRYPTO'01*, Springer-Verlag (LNCS 2139), pages 19–40, 2001.

[7] R. Canetti, J. Kilian, E. Petrank, and A. Rosen. Black-Box Concurrent Zero-Knowledge Requires $\tilde{\Omega}(\log n)$ Rounds. In 33*rd STOC*, pages 570–579, 2001.

[8] 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.

[9] 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.

[10] D. Chaum. Blind Signatures for Untraceable Payments. In *CRYPTO'82*, pages 199–203, 1982.

[11] C. Dwork, M. Naor, and A. Sahai. Concurrent Zero-Knowledge. In 30*th STOC*, pages 409–418, 1998.

[12] S. Even, O. Goldreich and A. Lempel. A Randomized Protocol for Signing Contracts. In *Communications of the ACM,* 28(6):637–647, 1985.

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

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

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

[16] O. Goldreich. *Secure Multi-Party Computation.* Manuscript, version 1.4, 2002. Available from `http://www.wisdom.weizmann.ac.il/∼oded/pp.html`.

[17] O. Goldreich, S. Goldwasser and S. Micali. How to Construct Random Functions. *Journal of the ACM*, 33(4):792–807, 1986.

[18] O. Goldreich and A. Kahan. How To Construct Constant-Round Zero-Knowledge Proof Systems for NP. *Journal of Cryptology*, 9(3):167–190, 1996.

[19] 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. For details see [16].

[20] 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.

[21] S. Goldwasser, S. Micali, and R.L. Rivest. A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks. *SIAM Journal on Computing*, 17(2):281–308, 1988.

[22] R. Impagliazzo and M. Luby. One-way Functions are Essential for Complexity Based Cryptography. In 30*th FOCS*, pages 230–235, 1989.

[23] J. Kelsey, B. Schneier and D. Wagner. Protocol Interactions and the Chosen Protocol Attack. In 5*th International Workshop on Security Protocols*, Springer-Verlag (LNCS 1361), pages 91–104, 1997.

[24] L. Lamport. Constructing Digital Signatures from One-Way Functions. *SRI International,* CSL-98, 1979.

[25] Y. Lindell. Bounded-Concurrent Secure Two-Party Computation Without Setup Assumptions. In 35*th STOC*, pages 683–692, 2003. (See [26] for a full version of the upper bound from this paper.)

[26] Y. Lindell. Protocols for Bounded-Concurrent Secure Two-Party Computation Without Setup Assumptions. Available from the *Cryptology ePrint Archive,* Report 2003/100, 2003. `http://eprint.iacr.org/`.

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

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

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

[30] 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.

[31] 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.

[32] M. Prabhakaran, A. Rosen, and A. Sahai. Concurrent Zero Knowledge with Logarithmic Round-Complexity. In 33*rd FOCS*, pages 366–375, 2002.

[33] M. Rabin. How to Exchange Secrets by Oblivious Transfer. Tech. Memo TR-81, Aiken Computation Laboratory, Harvard U., 1981.

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

[35] J. Rompel. One-Way Functions are Necessary and Sufficient for Secure Signatures. In 22*nd STOC*, pages 387–394, 1990.

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