Lower Bounds for Concurrent Self Composition

Yehuda Lindell

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

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 that there exist many functionalities that *cannot* be securely computed in this setting. 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. Our results hold for the plain model (where no trusted setup phase is assumed), and for the case that the parties may choose their inputs adaptively, based on previously obtained outputs. While proving our impossibility result, 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.

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 [21, 11].

Security under concurrent composition. The feasibility results of [21, 11] 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 being 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., [9, 20]).
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 [4].

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. We note that this additional adversarial capability has been shown to yield practical attacks against real protocols [13].

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 is that of bounded concurrency. In this case, a fixed bound on the number of concurrent executions is given, and the protocol need only remain secure when the number of concurrent execution does not exceed this bound. (When the bound is m, we call this m-bounded concurrency.) Note that the protocol may depend on this bound.

Feasibility of security under composition. The notion of concurrent general composition was first studied by [19] who considered the case that a secure protocol is executed *once* concurrently with another arbitrary protocol. (A definition and composition theorem were presented in [19], but no general feasibility results were demonstrated.) 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 [4]. 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, obtaining protocols that are secure by this definition is of great interest. 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 [4]. Furthermore, even when there is no honest majority, it is possible to securely compute any functionality in the *common reference string* (CRS) model [8]. (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 [5, 4, 7]. Recently, it was shown in [16] that these impossibility results extend to *any* security definition that guarantees security under concurrent general composition (including the definition of [19]).

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 [14, 18]. Unfortunately, however, these protocols are highly inefficient: The protocol of [14] has many rounds of communication and both the protocols of [14] and [18] have high bandwidth. (That is, in order to obtain security for m executions, the protocol of [14] has more than m rounds and communication complexity of at least mn^2. In contrast, the protocol of [18] has only a constant number of rounds, but still suffers from communication complexity of at least mn^2.) In addition to the above positive results, it has also been shown that there exist functionalities so that any protocol that securely computes one of them under m-bounded concurrent self composition, and is proven secure using *black-box simulation,* must have more than m rounds of communication [14]. 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 *efficient* protocols that remain secure under unbounded, or even m-bounded, concurrent self composition? (By efficient, we mean that at least, there should be no dependence on the bound m.)

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. We also prove a communication complexity lower bound for protocols that are secure under m-bounded concurrent self composition. 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.

Theorem 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, 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.

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. Essentially, any non-constant functionality that depends on both party's inputs, and where both parties receive output, has this property; see Section 2.3. We note that in a model where the parties can play different roles in the computation (e.g., if zero-knowledge is being computed, then in some executions a party plays the prover and in others it plays the verifier), then *any* functionality with the property that one party's output depends on the other party's input actually enables bit transmission. In Section 3, we prove the following theorem:

Theorem 2 (equivalence of self and general composition – informal): *Let f be a two-party functionality that enables bit transmission. Then, f can be securely computed under unbounded concurrent self composition if and only if it can be securely computed under concurrent general composition.*

The above equivalence holds for any functionality that enables bit transmission. In the full version of this paper, we show that an analogue of Theorem 2 does *not* hold for functionalities that do not enable bit transmission. In the full version, we also show that in the above-mentioned model where the parties can play different roles in the computation, then concurrent self composition is equivalent to concurrent general composition, for *all* functionalities.

Returning back to the proof of Theorem 1, impossibility is derived by combining the equivalence between concurrent self and general composition as stated in Theorem 2 with the impossibility results for concurrent general composition that were demonstrated in [16]. 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, we prove the following theorem that rules out the possibility of obtaining “efficient” protocols for m-bounded concurrency:

Theorem 3 (communication complexity lower bound – informal): *There exists a large class of two-party functionalities so 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 3 is essentially proven by directly combining the proof of Theorem 2 with proofs of impossibility from [16] and [7]; see Section 5.

Remarks. We stress that the above results are unconditional. That is, impossibility holds without any complexity assumptions. Furthermore, we assume nothing about the simulation, and in particular do not assume that it is “black-box”. We also note that although Theorems 1 and 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.

It is important to note 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 our impossibility results. 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).

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 [10, 9], where a prover runs many copies of a protocol with many verifiers. Thus, these works consider the question of security under *self* composition. This problem has received much attention; see [20, 6, 1] for just a few examples. Other specific problems have also been considered, but are not directly related to this paper.

2 Definitions

In this section, we present definitions for security under concurrent self composition and concurrent general composition, and we define the notion of functions that enable bit transmission. We denote the equivalence of distributions by $\equiv$, computational indistinguishability by $\overset{c}{\equiv}$, and the security parameter by n. The adversary always runs in time that is polynomial in n.

2.1 Concurrent Self Composition of Two-Party Protocols

We begin by presenting the definition for security under concurrent self composition. The basic description and definition of secure computation follows [12, 2, 17, 3]. Due to lack of space in this abstract, we present a slightly abridged definition and refer to the full version of this paper and [14] for full definitions. (Note that our definition here actually differs from [14] in that here the honest parties may adaptively choose their input to a session as a function of previously obtained outputs.)

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). We refer to such a process as a functionality and denote it $f : \{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^* \times \{0,1\}^*$, 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 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 time that is polynomial in the security parameter. 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 trivially 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 returns to each party its respective output. Unlike in the case of stand-alone computation, 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. In an ideal execution, the parties P_1 and P_2 interact with a trusted third party, sending it inputs and receiving back outputs. Party P_1 and P_2's inputs are determined by polynomial-size input-deciding circuit families $X = \{X_n\}_{n\in\mathsf{N}}$ and $Y = \{Y_n\}_{n\in\mathsf{N}}$, respectively. The circuits X_n and Y_n are polynomial in n and output exactly n bits. These circuits determine the length-n input values to be used, based on the current session number and previous outputs. Note that the number of previous outputs ranges from zero (for the case that no previous outputs have yet been obtained) to some fixed polynomial in n (that depends on the number of session initiated by the adversary).[1] Now, the ideal execution proceeds as follows. Whenever the adversary wishes to initiate a new session, it sends a start-session message to the trusted party. The trusted party then sends $(\mathsf{start\text{-}session}, i)$ to the honest party, where i is the index of the session (i.e., this is the i^{th} session to be started). Upon receiving $(\mathsf{start\text{-}session}, i)$ from the trusted party, the honest party applies its input-deciding circuit to (i) and its previous outputs, and obtains a new input v_i for this session. The honest party then sends (i, v_i) to the trusted party.

Whenever it wishes, the adversary can then send a message (i, w_i) to the trusted party, for any $w_i \in \{0,1\}^n$ of its choice. Upon sending this pair, it receives back its output from the trusted party, computed upon inputs (v_i, w_i). Following this, but again whenever it wishes, the adversary can instruct the trusted party to send the honest party its i^{th} output; the adversary does this by sending a $(\mathsf{send\text{-}output}, i)$ message to the trusted party. Finally, at the conclusion of the execution, the honest party outputs the vector of outputs that it received from the trusted party, and the adversary may output an arbitrary (probabilistic polynomial-time computable) function of its auxiliary input z, the corrupted party's input-deciding circuit and the outputs obtained from the trusted party.

Let $f : \{0,1\}^* \times \{0,1\}^* \mapsto \{0,1\}^* \times \{0,1\}^*$ be a functionality, and let $\mathcal{S}$ be a non-uniform probabilistic polynomial-time machine (representing the ideal-model adversary). Then, the ideal execution of f (on input-deciding circuits (X_n, Y_n) and auxiliary input z to $\mathcal{S}$), denoted $\text{IDEAL}_{f,\mathcal{S}}(X_n, Y_n, z)$, 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 does not differ for the case that unbounded concurrency or m-bounded concurrency is considered. This is because this bound is relevant only to the scheduling allowed to the adversary in the real model; see below.)

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). Let f be as above and let ρ be a polynomial-time two-party protocol for computing f. (We say that a protocol is polynomial-time if the running-time of the honest parties in a *single execution* is bound by a fixed polynomial.) In addition, let $\mathcal{A}$ be a non-uniform probabilistic polynomial-time machine that controls either P_1 or P_2. Then, the real concurrent execution of ρ (with input-deciding circuits (X_n, Y_n) and auxiliary input z to $\mathcal{A}$), denoted $\text{REAL}_{\rho,\mathcal{A}}(X_n, Y_n, z)$, 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 the i^{th} session is initiated by the adversary by sending a start-session message to the honest party.

[1] By convention, if the number of previously obtained outputs is greater than the maximum input length to the circuit, then we define the next input to be $\perp$.

The honest party then applies its input-deciding circuit on (i) and its previous outputs in order to obtain the input for this new session. (As in the ideal model, if the length of all previous outputs is greater than the maximum input length to the input-deciding circuit, then the next input is taken as $\perp$.) 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. 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 $(\mathsf{index}, \mathsf{message})$ 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 $\text{REAL}^m_{\rho,\mathcal{A}}(X_n, Y_n, z)$.

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 $\mathcal{A}$ there exists an ideal model adversary $\mathcal{S}$ such that for all polynomial-size input-deciding circuits, the outcome of an ideal execution with $\mathcal{S}$ is computationally indistinguishable from the outcome of a real protocol execution with $\mathcal{A}$. One important technical issue which arises here is due to the fact that the same $\mathcal{S}$ must work for *all* polynomial-size input-deciding circuits. In particular, this means that the honest parties (who compute their inputs in every execution from these circuits) may run longer than $\mathcal{S}$ can run (specifically, the size of the input-deciding circuits may be greater than $\mathcal{S}$'s running time).[2] This is an "unfair" requirement on $\mathcal{S}$ and we therefore allow a different ideal-model adversary $\mathcal{S}$ for every "size" circuit. That is, we require that for every real adversary $\mathcal{A}$ and polynomial $q(\cdot)$ there exists an ideal adversary $\mathcal{S}$ that works for all input-deciding circuit families $X = \{X_n\}$ and $Y = \{Y_n\}$ of size $O(q(n))$. We stress that any protocol that is secure when $\mathcal{S}$ must work for all polynomial-size input-deciding circuits is also

[2] We note that the *number* of executions is not a problem because this is determined by $\mathcal{A}$, and $\mathcal{S}$ comes after $\mathcal{A}$ in the order of quantifiers.

secure under this relaxation. This modification therefore only strengthens our impossibility results.[3] 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 for $i \in \{1,2\}$ and every polynomial $q(\cdot)$, there exists an ideal-model non-uniform probabilistic polynomial-time adversary $\mathcal{S}$ controlling P_i, such that for all families of input-deciding circuits $X = \{X_n\}_{n\in\mathsf{N}}$ and $Y = \{Y_n\}_{n\in\mathsf{N}}$ of size at most $O(q(n))$, and every auxiliary input $z \in \{0,1\}^*$,*

$$\left\{\text{IDEAL}_{f,\mathcal{S}}(X_n, Y_n, z)\right\}_{n\in\mathsf{N}} \stackrel{\text{c}}{\equiv} \left\{\text{REAL}_{\rho,\mathcal{A}}(X_n, Y_n, z)\right\}_{n\in\mathsf{N}}$$

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

$$\left\{\text{IDEAL}_{f,\mathcal{S}}(X_n, Y_n, z)\right\}_{n\in\mathsf{N}} \stackrel{\text{c}}{\equiv} \left\{\text{REAL}^{m}_{\rho,\mathcal{A}}(X_n, Y_n, z)\right\}_{n\in\mathsf{N}}$$

Non-trivial protocols. Notice that by the definition of security 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 introduce the notion of non-trivial protocols. 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.

2.2 Concurrent General Composition of Two-Party Protocols

Informally speaking, concurrent general composition considers the case that a secure protocol ρ runs concurrently with an arbitrary other protocol π. Furthermore, the inputs to ρ can be influenced (or actually determined) by protocol π. In the formalization of this setting, π is a "controlling protocol" that among other things, contains ideal calls to a trusted party that computes a functionality f. When these calls are replaced by executions of ρ, we denote the composed protocol by π^ρ. We stress that, in addition to representing a "controlling protocol", π can also represent arbitrary protocols that are running concurrently with ρ in the network. Therefore, by requiring that ρ remains secure for every calling protocol π, we derive that ρ remains secure when executed in any network with any set of protocols running. See [16] for more discussion.

[3] The reason that we insist on allowing a different $\mathcal{S}$ for every $q(\cdot)$ is due to the fact that, otherwise, it would turn out that concurrent general composition *does not imply* concurrent self composition. This would be absurd. We stress that our proof that concurrent self composition implies concurrent general composition holds in any case.

Let ρ be as above and assume that it computes a functionality f. Then, the security of ρ when composed with π in the real model is formalized by comparing the π^ρ composition to a hybrid execution where π uses ideal calls to a trusted party computing the functionality f. If the results of the hybrid and real executions are indistinguishable, then this means that a real execution of ρ behaves like an ideal call to f, even when run concurrently with π.

The hybrid model. Let π be an arbitrary polynomial-time protocol 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 and associates all ideal messages with f. Notice that the computation of π is a "hybrid" between the ideal model (where a trusted party carries out the entire computation) and the real model (where the parties interact with each other only). Specifically, the messages of π are sent directly between the parties, and the trusted party is only used in the ideal calls to 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 and ideal messages that it wishes in the name of this party (where the format of the ideal messages is as defined in the ideal execution in Section 2.1). 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 $x, y \in \{0,1\}^n$ be the parties' respective inputs to π. Then, the hybrid execution of π with functionality f, denoted $\text{HYBRID}^f_{\pi,\mathcal{S}}(x, y, z)$, is defined as the output of the adversary $\mathcal{S}$ and of the honest party from the above hybrid execution.

The real model – general composition. Let ρ be a 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. 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 this execution of ρ, it forwards to this ITM, and 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/symbol for π.) Finally, when an execution of ρ concludes and a value β is written on the output tape of an 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 $x, y \in \{0,1\}^n$ be the parties' respective inputs to π. Then, the real execution of π with ρ, denoted $\text{REAL}_{\pi^\rho,\mathcal{A}}(x, y, z)$, is defined as the output of the adversary $\mathcal{A}$ and of 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. 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 polynomial-time two-party protocol and f a two-party functionality. Then, ρ* securely realizes f under concurrent general composition *if for every 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 for all inputs $x, y \in \{0,1\}^n$ and all auxiliary inputs $z \in \{0,1\}^*$,*

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

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.

2.3 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 "equality functionality", where both parties receive the output, enables bit transmission as follows. The party who wishes to receive a bit inputs a predetermined value, say 1. Then, if the sending party wishes to send a bit 0, it inputs 0 (in this case, the inputs are not equal and so the output of the computation is 0). On the other hand, if the sending party wishes to send the bit 1, then it inputs 1 (thus, the inputs are equal and the output is 1). Notice that a functionality enables bit transmission only if both parties are able to send bits to each other. Therefore, functionalities like oblivious transfer and zero-knowledge do not enable bit transmission, because only one party receives output. Nevertheless, by considering a more general setting where both parties can play both roles in the functionality (e.g., both parties can prove statements in zero-knowledge and both parties can play the sender in the oblivious transfer), we obtain that *any* functionality with the property that one party's output depends on the other party's input actually enables bit transmission. This generalization is dealt with in the full version of this paper. 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.

3 Self Composition Versus General Composition

In this section 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 [14, 18], but cannot be achieved for general composition [16]. (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 calling protocol π.)

Theorem 4 *Let f be a 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.*

Intuitively, security under general composition implies security under self composition because in both cases, many copies of the secure protocol are run; the only difference is that in the setting of general composition, other protocols may *also* be run. The other, more interesting direction, is proven as follows. Loosely speaking, the parties use the "bit transmission property" of f in order to emulate an execution of π^ρ, while only running copies of ρ (recall that π^ρ denotes the concurrent general composition of a secure protocol ρ with an arbitrary other protocol π). This can be carried out by sending the messages of π one bit at a time, via executions of the protocol ρ that computes f. Thus, it is possible to emulate the setting of concurrent general composition, within the context of concurrent self composition. The proof of Theorem 4 appears in the full version of this paper. As we have mentioned, we also show that in a model where the parties can play different roles in the computation, *full equivalence* holds between concurrent self composition and concurrent general composition.

In the full version of this paper, we also show a *separation* between concurrent self composition and concurrent general composition, for functions that do *not* enable bit transmission. Specifically, we show that the zero-knowledge proof of knowledge functionality (for an NP-complete language) can be securely computed under concurrent self composition. However, in [16], it has been shown that this cannot be achieved under concurrent general composition.

4 Impossibility for Concurrent Self Composition

An important ramification of Theorem 4 is that known impossibility results for concurrent *general* composition apply also to unbounded concurrent *self* composition, as long as the functionality in question enables bit transmission. 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 [16]:

1. Let $f:\{0,1\}^*\times\{0,1\}^*\rightarrow\{0,1\}^*$ be a deterministic functionality. If f depends on both parties' inputs,[4] then the functionality $(x,y) \rightarrow (f(x,y), f(x,y))$ 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 $f:\{0,1\}^*\times\{0,1\}^*\rightarrow\{0,1\}^*\times\{0,1\}^*$ be a deterministic functionality and denote $f = (f_1, f_2)$. If f is *not completely revealing*,[5] then the functionality $(x,y) \rightarrow (f_1(x,y), f_2(x,y))$ 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 4 to the results of [16], we obtain the following corollary:

[4] 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 $f(x, y_1) \neq f(x, y_2)$.

[5] The definition of completely revealing functionalities can be found in Section 5.

Corollary 5 *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.*

The set of functionalities $\Phi \cap \Psi$ contains all the functionalities ruled out in [16] 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 the output, is included in this set.

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 result stated in Corollary 5. The proof of our lower bound combines ideas from [7] and [16], together with the proof of Theorem 4.

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 (deterministic) functionality completely reveals party P_1's input, if party P_2 can choose an input that will enable it to *completely determine* P_1's input (no matter what P_1's input is). That is, a functionality f completely reveals P_1's input 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 functionality 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 functionality 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 [7] 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 $x, x' \in X$. We say that x and x' are equivalent with respect to f_2 if for every $y \in Y$ it holds that $f_2(x, y) = f_2(x', y)$. Notice that if x and x' are equivalent with respect to f_2, then x can always be used instead of x' (at least regarding P_2's output). We now define completely revealing functionalities:

Definition 6 (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 functionality f_2* completely reveals P_1's input *if there exists a single input $y \in Y$ for P_2, such* *that for every pair of values $x, x' \in X$ that are not equivalent with respect to f_2, it holds that $f_2(x, y) \neq f_2(x', y)$. Complete revealing for P_2's input is defined analogously. We say that a functionality is* completely revealing *if f_1 completely reveals P_2's input or f_2 completely reveals P_1's input.*

If a functionality completely reveals P_1's input, then party P_2 can set its own input to be y from the definition, and then P_2 will always obtain the exact input used by P_1, or one that is equivalent to it. Specifically, given y and $f_2(x, y)$, it can traverse over all X and find the unique x that must be P_1's input (or one equivalent to it). Thus we see that P_1's input is completely revealed by f_2. In contrast, if a functionality f_2 does *not* completely reveal P_1's input, 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 and x' such that $f_2(x, y) = f_2(x', y)$. Therefore, if P_1's input is x or x', 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 the lower bound. Note that we require that x and x' be non-equivalent because in such a case, x and x' are really the same input and so, essentially, both x and x' are P_1's input.

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 7 *Let $f = (f_1, f_2)$ be a deterministic two-party functionality over a finite domain that is not completely revealing and enables bit transmission. If a non-trivial 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 4 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 to equal the total number of bits sent by the parties to *each other*, plus a *single bit* for each call to f.[6] 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 4, 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:

[6] 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.

Claim 8 *Let f be a 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 for all $x, y \in \{0,1\}^n$ and all $z \in \{0,1\}^*$,*

$$\{\text{HYBRID}^f_{\pi,\mathcal{S}}(x,y,z)\}_{n\in\mathsf{N}} \overset{\text{c}}{\equiv} \{\text{REAL}_{\pi^\rho,\mathcal{A}}(x,y,z)\}_{n\in\mathsf{N}} \qquad (1)$$

We now proceed with the actual proof of Theorem 7. Let $f = (f_1, f_2)$ be a deterministic two-party functionality over a finite domain, such that f is not completely revealing and enables bit transmission. 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. (1) of Claim 8 to hold. This thereby contradicts Claim 8, 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 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, in order for π to make sense, 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 (if ρ has bandwidth $m-1$, then π will have bandwidth m by adding 1 for 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 [15, 18] that use asymmetrical instructions in an inherent way). Nevertheless, by Claim 8, 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,[7] adversary $\mathcal{A}$ forwards this same message back to P_2 as if it is P_1's first π-message to P_2. Then, when P_2 answers this 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 that this output must equal $f_2(x, y)$. (This follows from the guaranteed behaviour of such a 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 8, we have that there exists a probabilistic polynomial-time hybrid-model adversary $\mathcal{S}$ such that:

$$\{\text{HYBRID}^f_{\pi,\mathcal{S}}(\lambda, (x, y), \lambda)\} \overset{c}{\equiv} \{\text{REAL}_{\pi^\rho,\mathcal{A}}(\lambda, (x, y), \lambda)\} \tag{2}$$

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. (2). 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 a $\tilde{y}$ so that $f_2(\tilde{x}, \tilde{y}) \neq f_2(x, \tilde{y})$, then with probability $1/|Y|$ 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). 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

[7] We assume without loss of generality that the first message in ρ is sent by P_2.

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, 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 $y \in Y$, $f_2(\tilde{x}, y) = f_2(x, y)$, 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 P_1's input 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 $x \in_R X$ 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. ■

Acknowledgements

I would like to thank Boaz Barak and Jonathan Katz for helpful discussions.

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. *Journal of Cryptology*, 13(1):143–202, 2000.
4. R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In 42*nd FOCS*, pages 136–145, 2001.
5. R. Canetti and M. Fischlin. Universally Composable Commitments. In *CRYPTO'01*, Springer-Verlag (LNCS 2139), pages 19–40, 2001.
6. 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.
7. R. Canetti, E. Kushilevitz and Y. Lindell. On the Limitations of Universal Composition Without Set-Up Assumptions. In *EUROCRYPT'03,* Springer-Verlag (LNCS 2656), pages 68–86, 2003.
8. 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.
9. C. Dwork, M. Naor, and A. Sahai. Concurrent Zero-Knowledge. In 30*th STOC*, pages 409–418, 1998.
10. U. Feige and A. Shamir. Witness Indistinguishability and Witness Hiding Protocols. In *22nd STOC*, pages 416–426, 1990.
11. 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.
12. S. Goldwasser and L. Levin. Fair Computation of General Functions in Presence of Immoral Majority. In *CRYPTO'90,* Springer-Verlag (LNCS 537), 1990.
13. 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.
14. Y. Lindell. Bounded-Concurrent Secure Two-Party Computation Without Setup Assumptions. In 35*th STOC*, pages 683–692, 2003. (See [15] for a full version of the upper bound from this paper.)
15. Y. Lindell. Protocols for Bounded-Concurrent Secure Two-Party Computation Without Setup Assumptions. *Cryptology ePrint Archive,* Report #2003/100, `http://eprint.iacr.org/2003/100`.
16. Y. Lindell. General Composition and Universal Composability in Secure Multi-Party Computation. In 44*th FOCS*, pages 394–403, 2003.
17. S. Micali and P. Rogaway. Secure Computation. Unpublished manuscript, 1992. Preliminary version in *CRYPTO'91*, Springer-Verlag (LNCS 576), 1991.
18. R. Pass and A. Rosen Bounded-Concurrent Secure Two-Party Computation in a Constant Number of Rounds. In 44*th FOCS*, 2003.
19. B. Pfitzmann and M. Waidner. Composition and Integrity Preservation of Secure Reactive Systems. In *7th CCS*, pages 245–254, 2000.
20. R. Richardson and J. Kilian. On the Concurrent Composition of Zero-Knowledge Proofs. In *EUROCRYPT'99*, Springer-Verlag (LNCS 1592), pp. 415–431, 1999.
21. A. Yao. How to Generate and Exchange Secrets. 27*th FOCS*, pp. 162–167, 1986.