Bounded-Concurrent Secure Two-Party Computation Without Setup Assumptions

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

ABSTRACT

In this paper we study the feasibility of obtaining protocols for general two-party computation that remain secure under concurrent composition. (A general protocol can be used for obtaining secure computation of any functionality.) We consider a scenario where *no trusted setup* is assumed (and so, for example, there is no common reference string available to the parties); we call this the "plain model". We present both negative and positive results for this model. Specifically, we show that a general two-party protocol that remains secure for m concurrent executions and can be proven via *black-box* simulation, must have more than m rounds of communication. An important corollary of this result is that *there do not exist* protocols for black-box secure general two-party computation for the case of unbounded concurrency (where any polynomial number of concurrent executions may be run). On the positive side, we show that under general cryptographic assumptions, there exist secure protocols for general two-party computation in the model of bounded concurrent composition (in this model the number of concurrent executions is fixed and the protocol design may depend on this number). Our protocol has $O(m)$ rounds of communication, where m is the bound on the number of concurrent executions, and uses both black-box and non black-box techniques. We note that this protocol constitutes the first feasibility result for general two-party computation without setup assumptions *for any model of concurrency.*

Categories and Subject Descriptors

F.1.2 [**Theory of Computation**]: Interactive and reactive computation

General Terms

Theory

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 a malicious adversary who controls one of the parties and can arbitrarily deviate from the protocol instructions (i.e., in this work we consider 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 [26, 15]. However, these feasibility results relate only to the stand-alone setting, where a single pair of parties run a single execution.

Security under concurrent composition. 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, an important research goal is to re-establish the feasibility results of the stand-alone setting for the setting of concurrent composition.

We stress that the need for security under concurrent composition is not merely an issue of efficiency. Specifically, one cannot claim that in order to ensure security, all executions should just be carried out sequentially (recall that security under sequential composition is guaranteed by the stand-alone definitions [4]). This is because the honest parties must actively coordinate their executions in order to ensure sequentiality. However, the honest parties may not even know of each other's existence. Therefore, they cannot coordinate their executions, and a dishonest adversary that interacts with a number of parties can schedule the executions concurrently. The protocol being used must therefore be secure under concurrent composition.

The study of security under concurrent composition was initiated in the context of concurrent zero-knowledge [11]. In this scenario, a prover runs many copies of a protocol with many verifiers, and zero-knowledge must be preserved. The feasibility of concurrent zero-knowledge in the *plain model* (where no trusted setup or preprocessing phase is assumed) was first demonstrated by [23]. In general, the issue of concurrent zero-knowledge has received much attention, resulting in a rather exact understanding of the round-complexity of black-box concurrent zero-knowledge [20, 25, 7, 19, 22]. Another specific problem that has been studied in the con-

STOC'03, June 9–11, 2003, San Diego, California, USA.

text of concurrent composition is that of oblivious transfer [13].

In this paper, we consider the question of concurrent composition for *general* secure two-party computation. (A general protocol is one that can be used for obtaining secure computation of any efficient functionality.) The problem of concurrent composition can be defined in many different ways. One possibility is the model used in the framework of universal composability [5]. This framework considers a very general setting where many sets of (possibly different) parties run many protocols, and secure protocols may run concurrently with arbitrary other protocols. The problem of universally composable two-party computation has been studied with the following results. It is possible to securely compute *any* two-party functionality in a universally composable way, in the *common reference string model*[1] [9]. In contrast, in the plain model, the possibility of obtaining universally composable two-party computation has been ruled out for a very large class of functionalities [6, 5, 8]. However, these impossibility results relate specifically to the definition of universal composability and not necessarily to the composition operation guaranteed to hold under this definition. Thus it may be possible to obtain concurrent secure computation for other definitions and/or other models of concurrency. One specific model of interest is the model considered for concurrent zero-knowledge, where two parties run the same protocol many times concurrently. We remark that apart from the work on universal composability, there has been no work on the feasibility of concurrent secure two-party computation in the plain model (for any model of concurrency).

Thus, one could loosely sum up our current understanding of concurrent composition as follows. In the common reference string model, the feasibility of universally composable two-party computation has been demonstrated. However, in the plain model, almost all known results relate specifically to concurrent zero-knowledge. In particular, *nothing* is known about the feasibility of obtaining concurrent general secure two-party computation in the plain model for definitions other than those of universal composability. Thus, a fundamental question that remains unanswered is the following:

> *Can general secure two-party computation that composes concurrently be obtained in the plain model?*

This question is of both theoretical interest and practical importance. (We note that in many settings, it is not clear how to reasonably obtain the trusted setup of a common reference string.) The aim of this work is take a first step in providing an answer to this question.

The model of concurrency. The notion of concurrent composition relates to a large number of possible scenarios. These scenarios range from two parties running the same protocol many times concurrently, to many pairs of different parties running many arbitrary protocols concurrently. In this work we consider the first case of two parties running the same protocol many times concurrently. We note that this is actually equivalent to the case where many different pairs of parties run a protocol concurrently, with the following adversarial limitation. Each party is designated to be either a first party P_1 or a second party P_2 (this defines the role that it plays in the protocol execution). The adversary is then only allowed to corrupt a subset of the parties playing P_1 or a subset of the parties playing P_2; but cannot simultaneously corrupt parties playing P_1 and parties playing P_2. Such a scenario can model some real-world scenarios, like a server concurrently interacting with many clients (and where we assume that there is no simultaneous corruption of a server and a client). From here on, when we talk of "concurrent composition", we mean this model. We remark that this is the exact model considered in the entire body of work on concurrent zero-knowledge. Although this is a rather limited model, we believe that it serves as a good first step to understanding concurrent composition of secure protocols. Furthermore, we stress that even in this limited model, nothing is currently known when no trusted setup is assumed.

An important parameter which must be considered when defining the model is the "amount of concurrency" that is allowed. In most of the literature on concurrent zero-knowledge, security must hold for *any* polynomial number of executions. We call this model unbounded concurrency. In contrast, [1] considered a model where the number of concurrent executions is a fixed polynomial in the security parameter, and the protocol design can depend on this number. This model is called bounded concurrency, and when m is the maximum number of concurrent executions, we talk about m-bounded concurrent composition ($m = m(n)$ is a fixed polynomial in the security parameter). Our protocol for general secure two-party computation is cast in the setting of bounded concurrent composition.

Our Results

As we have described, in this work we consider the case of a single pair of parties running the same protocol, in the setting of m-bounded concurrent composition. The definition of security that we use is a natural extension of the ideal-model based definitions of [16, 21, 3, 4] for the stand-alone setting. We present both negative and positive results relating to the feasibility of (bounded) concurrent secure two-party computation in the plain model, under this definition. We begin by stating our negative result, which is actually a black-box lower bound:

Negative results (black-box lower bound): We show that secure two-party computation with black-box simulation[2] is "hard" to achieve, even in the model of bounded concurrency:

THEOREM 1.1. (lower bound): *There exists an efficient functionality f such that every protocol Π that securely computes f under m-bounded concurrent composition, and is proven using* black-box simulation*, must have more than m rounds of communication.*

We remark that the specific functionality referred to in the theorem is a natural one. Specifically, we show our lower bound for blind signatures [10], and 1-out-of-2 oblivious

[1] In this model, all parties are given access to a string that is ideally chosen from some distribution.

[2] In the setting of secure computation, security is proven by demonstrating that for every real-model adversary $\mathcal{A}$ there exists an ideal-model adversary/simulator $\mathcal{S}$ such that the output distribution generated by $\mathcal{S}$ in an ideal execution is indistinguishable from that generated by $\mathcal{A}$ in a real execution of the protocol; see Definition 1. We say that an ideal-model simulator $\mathcal{S}$ is *black-box* if it is given only oracle access to the real-model adversary $\mathcal{A}$.

transfer [12]. This result seemingly contradicts the fact that protocols for concurrent oblivious transfer have been demonstrated [13]. However, in the model of [13], all the inputs in all the executions are 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.

The proof of Theorem 1.1 works by showing that when concurrent 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, a scheduling of messages can be shown that prevents the simulator from any rewinding in one of the executions. However, informally speaking, a black-box simulator must rewind in order to successfully simulate.

Theorem 1.1 implies two important corollaries. First, it demonstrates a black-box lower bound for *general* two-party computation (i.e., protocols that can be used for obtaining secure computation of any efficient functionality). Second, it shows that black-box secure protocols for the unbounded concurrent model *do not exist*. This is because the number of rounds of a protocol must be bounded by a fixed polynomial. However, Theorem 1.1 states that when the number of concurrent executions is greater than this number of rounds (as can happen in the model of unbounded concurrency), security cannot hold. In summary, we have the following corollary:

COROLLARY 1.2. *There do not exist secure protocols for general two-party computation in the unbounded concurrent model, that can be proven using black-box simulation.*

This corollary stands in stark contrast to the setting of concurrent zero-knowledge, where black-box simulation does suffice for obtaining unbounded concurrent composition [23]. Thus, obtaining concurrent secure computation is strictly harder than obtaining concurrent zero-knowledge. This result also highlights a limitation on concurrent zero-knowledge. Specifically, the model for concurrent zero-knowledge considers the case that the zero-knowledge protocol is run concurrently with itself only. However, arguably the most important application of zero-knowledge is its use as a building block in larger secure protocols. Thus, one would hope that concurrent zero-knowledge could be "plugged-in" to a larger protocol, in order to obtain concurrent security for the larger protocol. Our lower bound proves that this cannot be done in a generic way.[3] Despite this, our upper bound *does* use concurrent zero-knowledge protocols and techniques (in a non-generic way) in order to achieve bounded concurrent secure two-party computation; see Section 4.

We note that Theorem 1.1 holds even if at any given time, at most *two* executions are running. (Loosely speaking, in such a case the m-bounded concurrency means that m different protocol executions may overlap.) This shows that the lower bound does not stem from deep protocol nesting as in the case in concurrent zero-knowledge. Indeed, a nesting depth of at most two is needed here.

[3]To explain this more rigorously, we note that black-box secure protocols that use ideal calls to zero-knowledge and compose concurrently exist [9]. Thus, one would hope that a concurrent zero-knowledge protocol could be used to replace such an ideal call and the resulting protocol would remain secure. However, when the zero-knowledge protocol is black-box zero-knowledge, such a strategy would result in a black-box secure protocol that composes concurrently, in contradiction to Corollary 1.2.

Positive results (feasibility): Our positive result states that any efficient two-party functionality can be securely computed under bounded concurrent secure computation:

THEOREM 1.3. (upper bound): *Assume the existence of trapdoor permutations and collision resistant hash functions. Then, for any two party functionality f and for any m, there exists a protocol Π that securely computes f under m-bounded concurrent composition.*

We remark that Theorem 1.3 is the *first* feasibility result for general two-party computation in the plain model, *for any model of concurrency*. The protocol that we construct has $\widetilde{O}(m)$ rounds and uses both black-box and non black-box techniques (thus, our upper bound does not "match" the lower bound).

We conclude with a remark about *efficiency*. Our protocol requires $\widetilde{O}(m)$ rounds to obtain security for m concurrent executions. It is therefore far from being "reasonably efficient". Nevertheless, the focus of this paper is *not* efficiency. Rather, we believe that establishing the feasibility of obtaining secure computation in a reasonable model of concurrency is of great importance, irrespective of efficiency.

Organization. Due to lack of space in this abstract, we have chosen to present only a brief overview of the upper bound; see Section 4. A complete treatment appears in the full version of this paper.

2. DEFINITIONS

In this section we present the definition for m-bounded concurrent secure two-party computation. The basic description and definition of secure computation follows [16, 21, 3, 4]. We denote computational indistinguishability by $\stackrel{c}{\equiv}$, and the security parameter (and, for simplicity, the lengths of the parties' inputs) by n.

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 $((x, w), x) \mapsto (\lambda, R(x, w))$. In the context of concurrent composition, each party actually receives a vector of inputs of polynomial length, and the aim of the parties is to jointly compute $f(x_i, y_i)$ for every i. The fact that m-bounded concurrency is considered relates to the allowed scheduling of messages by the adversary in the protocol executions; see the description of the real model below.

We note that our results here also apply to *reactive functionalities* where inputs and outputs are supplied over a number of stages. Such a functionality can be modeled by a probabilistic polynomial-time interactive machine who receives inputs and supplies outputs. This machine can keep state and thus the inputs from previous computations can influence the outputs of later ones.

Adversarial behavior. In this work we consider a malicious, static adversary. That is, the 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. The focus of this work is not on fairness. We therefore present a definition where 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 secure by definition. 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 m 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 execution in the ideal model. Let $p(n)$ be a polynomial in the security parameter. Then, an ideal execution with an adversary who controls either P_1 or P_2 proceeds as follows:

Inputs: The honest party and adversary each obtain a vector of $p(n)$ inputs of length n; denote this vector by $\overline{w}$ (i.e., $\overline{w} = \overline{x}$ or $\overline{w} = \overline{y}$).

Honest party sends inputs to trusted party: The honest party sends its entire input vector $\overline{w}$ to the trusted party.

Adversary interacts with trusted party: For every $i = 1, \ldots, p(n)$, the adversary can send (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 based on w'_i and the input sent by the honest party. (That is, if P_1 is corrupted, then the adversary receives $f_1(w'_i, y_i)$ and if P_2 is corrupted then it receives $f_2(x_i, w'_i)$.) The adversary can send the (i, w'_i) pairs in any order it wishes and can also send them *adaptively* (i.e., choosing inputs based on previous outputs). The only limitation is that for any i, *at most one pair* indexed by i can be sent to the trusted party.

Trusted party answers honest party: Having received all of its own outputs, the adversary specifies which outputs the honest party receives. That is, the adversary sends the trusted party a set $I \subseteq \{1, \ldots, p(n)\}$. Then, the trusted party supplies the honest party with a vector $\overline{v}$ of length $p(n)$ such that for every $i \notin I$, $v_i = \perp$ and for every $i \in I$, v_i is the party's output from the i^{th} execution. (That is, if P_1 is honest, then for every $i \in I$, $v_i = f_1(x_i, w'_i)$ and if P_2 is honest, then $v_i = f_2(w'_i, y_i)$.)

Outputs: The honest party always outputs the vector $\overline{v}$ that it obtained from the trusted party. The adversary may output an arbitrary (probabilistic polynomial-time computable) function of its initial input and the messages 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 expected polynomial-time machine (representing the adversary). Then, the ideal execution of f (on input vectors $(\overline{x}, \overline{y})$ of length $p(n)$ and auxiliary input z to $\mathcal{S}$), denoted $\text{IDEAL}_{f,\mathcal{S}}(\overline{x}, \overline{y}, z)$, 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 does not include any reference to the bound m on the 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 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. (In particular, the adversary can schedule the executions sequentially, thereby learning previous outputs before defining the next input.)

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 $p(n)$ and $m = m(n)$ be polynomials, let f be as above and let Π be a two-party protocol for computing f. Furthermore, let $\mathcal{A}$ be a non-uniform probabilistic polynomial-time machine that controls either P_1 or P_2. Then, the real m-bounded concurrent execution of Π (on input vectors $(\overline{x}, \overline{y})$ of length $p(n)$ and auxiliary input z to $\mathcal{A}$), denoted $\text{REAL}^m_{\Pi,\mathcal{A}}(\overline{x}, \overline{y}, z)$, is defined as the output pair of the honest party and $\mathcal{A}$, resulting from $p(n)$ executions of the protocol interaction, where the honest party always inputs its i^{th} input into the i^{th} execution.

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, x) to the honest party. The honest party then adds x to the view of its i^{th} execution of Π and replies according to the instructions of Π and this view.[4] The adversary continues by sending another message (j, y), and so on. When unbounded concurrency is considered, *any* scheduling of the messages by the adversary is allowed. In contrast, in the setting of m-bounded concurrency, 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, and they possibly all overlap.

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, the definition asserts that a secure two-party protocol (in the real model) emulates the ideal model (in which a trusted party

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

exists). This is formulated by saying that for every real-model adversary there exists an ideal model adversary that can simulate an execution of the secure real-model protocol.

Definition 1. (security in the malicious model): *Let $m = m(n)$ be a polynomial and let f and Π be as above. Protocol Π is said to* securely compute f under m-bounded concurrent composition *if for every real-model non-uniform probabilistic polynomial-time adversary $\mathcal{A}$ controlling party P_i for $i \in \{1, 2\}$, there exists an ideal-model non-uniform probabilistic expected polynomial-time adversary $\mathcal{S}$ controlling P_i, such that for every polynomial $p(n)$,*

$$\left\{\text{IDEAL}_{f,\mathcal{S}}(\overline{x}, \overline{y}, z)\right\} \stackrel{c}{\equiv} \left\{\text{REAL}^m_{\Pi,\mathcal{A}}(\overline{x}, \overline{y}, z)\right\}$$

where the ensembles are indexed by $n \in \mathsf{N}$, $\overline{x}, \overline{y} \in (\{0,1\}^n)^{p(n)}$ and $z \in \{0,1\}^*$.

Black-box security. We say that a protocol Π for securely computing f is proven using black-box simulation if $\mathcal{S}$ is given only oracle access to $\mathcal{A}$. That is, there exists an ideal-model adversary $\mathcal{S}$ such that for every real-model adversary $\mathcal{A}$,

$$\left\{\text{IDEAL}_{f,\mathcal{S}^{\mathcal{A}(z)}}(\overline{x}, \overline{y}, z)\right\} \stackrel{c}{\equiv} \left\{\text{REAL}^m_{\Pi,\mathcal{A}}(\overline{x}, \overline{y}, z)\right\}$$

We note that in this case, $\mathcal{S}$ is not given the auxiliary input z, but rather has oracle access to $\mathcal{A}(z)$.[5]

Remark. Definition 1 is rather simplistic in that it assumes that all inputs are fixed before any execution begins. We remark that our upper bound also holds for a more general definition where inputs for later executions can depend on outputs of earlier executions.

3. LOWER BOUND

In this section we prove our black-box lower bound for m-bounded concurrent secure two-party computation.

Motivating discussion. As described above, 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.

Next, 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. We now proceed to the formal proof.

Theorem 2. *There exists a probabilistic polynomial-time functionality f such that every protocol that securely computes f under m-bounded concurrent composition and is proven using* black-box simulation*, must have more than m rounds of communication.*

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 [18]. 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 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

[5]The proof of our lower bound (Theorem 1.1) relies on the fact that the black-box simulator $\mathcal{S}$ is not given the adversary's auxiliary input. Withholding the auxiliary input from $\mathcal{S}$ is standard practice, and is used in an essential way in all black-box lower bounds for zero-knowledge. Indeed, most known (zero-knowledge) black-box lower bounds are known to *not* hold for uniform verifiers [1].

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 [17] which can be constructed from any one-way function [24]. 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]: Party P_1's input is a signing-key sk for a secure one-time signature scheme, and P_2's input is a verification-key vk and a string $\alpha \in \{0,1\}^n$. The output of the functionality is such that P_2 obtains a signature $\sigma = \mathsf{Sign}_{sk}(\alpha)$, and P_1 obtains nothing. 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 the output of P_2, 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. We also assume that P_2's input 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 $p(n) = m(n)$ (see Definition 1) and thus there are exactly m executions overall; denote these m executions of Π by $\Pi_1, \ldots, \Pi_m$ and denote the i^{th} 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 i^{th} 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). We now define an adversary $\mathcal{A}$ for Π.

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.

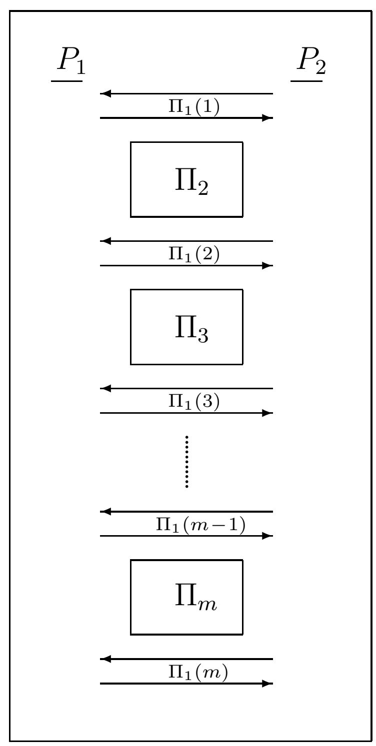

Figure 1: The schedule of the adversary. The arrows refer to messages from execution Π_1.

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 receiving 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 with a random key k [14], and replaces α_i with $F_k(\Pi_1(i-1)_1)$. (Recall that pseudorandom functions can be constructed from any one-way function.) Then, $\mathcal{A}$ follows the protocol instructions of Π_i exactly. 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 definition 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_1 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, $\mathcal{S}$ 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 3.1. *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.*[6] *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 resistance 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. The reduction is straightforward and is omitted due to lack of space.

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 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

[6]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., $\Pi_1(i)_2$).

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 3.2. *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 equals σ_1 where $\mathsf{Verify}_{vk_1}(z, \sigma_1) = 1$. 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. (It is easy to show that in an execution of a secure protocol with two honest parties, except with negligible probability, the output of the parties must be according to the functionality definition.) 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 for 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, the only input given to $\mathcal{S}$ is vk_1 and z (ignoring the other vk_i's which are irrelevant). Thus, if $\mathcal{S}$ queries the trusted party with $(*, z)$, it will obtain the necessary signature σ_1. Otherwise, it must have forged a signature on z with key vk_1. 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 3.1 and the details are therefore omitted. ■

Essentially, Claim 3.1 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 3.2 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. A technicality that arises with such a strategy is that $\mathcal{A}'$ is a strict polynomial-time machine, whereas $\mathcal{S}$ is allowed to run in expected polynomial-time. Therefore, 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 $\mathsf{time\text{-}out}$. This ensures that $\mathcal{A}'$ runs in strict polynomial-time. Furthermore, by Markov's inequality, $\mathcal{A}'$ outputs $\mathsf{time\text{-}out}$ with probability at most $1/2$.

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 vector $(vk_1, \alpha_1, \ldots, vk_{p(n)}, \alpha_{p(n)})$ 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 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 emulates 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. 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. (Recall that α and z are also independent and uniformly distributed n-bit strings.) 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 follows from 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 3.1; 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 3.2, except with negligible probability, $\mathcal{S}$ must obtain $(*, z)$ in an 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 $1/2$, $\mathcal{S}$ must obtain $(*, \alpha)$ in the above emulation by $\mathcal{A}'$, where α is the input used by P_2 in Π. (Recall that $\mathcal{A}'$ truncates $\mathcal{S}$'s execution after $2q(n)$ steps. Therefore, with probability $1/2$, $\mathcal{S}$ may not terminate and so may not output $(*, z)$.) In summary, in a real execution of $\mathcal{A}'$ with P_2, adversary $\mathcal{A}'$ obtains P_2's input with probability almost $1/2$. 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 $1/2$. (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 2 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 for our upper bound (Theorem 1.3), one corruption case is proven using black-box simulation and the other is proven using non black-box simulation. Therefore, Theorem 2 states that any protocol using such a strategy must have round complexity greater than m.

Impossibility of (black-box) unbounded concurrent composition. The following important corollary demonstrates a severe limitation of black-box simulation in the setting of concurrent secure two-party computation:

COROLLARY 3. *There exists a probabilistic polynomial-time functionality f for which there does not exist a protocol that securely computes f under* unbounded *concurrent composition and is proven using* black-box simulation.

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 2 holds also for uniform adversaries.

Impossibility of concurrent oblivious transfer. Although Theorem 2 suffices for ruling out general protocols, we cannot infer a lower bound from it to any specific functionality other than that of blind signatures. Nevertheless, we can show that, as a corollary to the above proof, concurrent secure computation of the 1-out-of-2 oblivious transfer functionality [12] *can* be ruled out. This is demonstrated by constructing a protocol for blind signatures that uses calls to oblivious transfer only. See the full version for details.

We remark that our impossibility result seems to contradict the concurrent oblivious transfer upper bound of [13]. However, the model of concurrency of [13] is fundamentally different to ours. Specifically, they assume that the inputs of the parties are chosen independently of each other and independently of the inputs in other executions. In contrast, our model follows the more standard definition where quantification is over all inputs (and, in particular, over correlated inputs). Furthermore, we use the fact that P_1 and P_2 have correlated inputs in an essential way in the proof of Theorem 2.

4. OVERVIEW OF THE UPPER BOUND

In this section, we provide a very brief and informal outline of the proof of Theorem 1.3. We begin by observing that secure two-party computation that composes concurrently can be obtained in a hybrid model where the parties have (concurrent) access to an ideal zero-knowledge functionality.

This was formally demonstrated in [9]. Given this observation, the question is whether or not one can transform protocols that use this ideal zero-knowledge functionality into protocols that run in the real model (without any trusted help). Of course, this transformation must preserve the concurrent composition, or bounded concurrent composition, of the protocol.

A naive implementation of the above idea is to replace the ideal zero-knowledge calls with any concurrent zero-knowledge protocol. However, two problems arise with this idea. First, as we have mentioned, concurrent zero-knowledge considers a scenario where the protocol is run concurrently with itself only. Thus, it is not clear that the zero-knowledge simulation can be accomplished if the protocol is run concurrently to other protocols as well. Second, a problem relating to the malleability of protocols arises. That is, during the concurrent executions, the adversary may simultaneously verify and prove zero-knowledge proofs. Thus, it can execute a man-in-the-middle attack on the zero-knowledge proofs, possibly enabling it to prove false statements. This is a problem because in order for the simulation of the secure protocol to work, we must be sure that the proofs provided by the adversary are sound. This second problem is solved by having the parties use fundamentally different zero-knowledge proofs. Specifically, one party proves using the black-box zero-knowledge protocol of [22, 23], while the other party proves using the non black-box zero-knowledge protocol of [1, 2]. It turns out that a careful choice of parameters for these protocols solves the problem of malleability. However, the first problem (of concurrency with other protocols) still remains. This is solved as follows. The protocol of [1] can easily be modified so that it remains zero-knowledge when run concurrently with other protocols (intuitively, this is the case because there is no rewinding). However, the protocol of [22] is more problematic. We therefore devise a *new* black-box simulation strategy for [22] in the setting of m-bounded concurrency, that enables it to be used directly as a sub-protocol of another protocol. Having solved these problems, we are able to replace the ideal zero-knowledge calls in any protocol with the specific m-bounded concurrent zero-knowledge protocols described above. Putting this transformation together with a protocol that is secure when given access to the ideal zero-knowledge functionality, we obtain a protocol that is secure under m-bounded concurrent composition in the real model.

Acknowledgements

We express our deep thanks to Ran Canetti for all his help in this work. Among other things, some key ideas in the proof of the lower bound are due to Ran. We would also like to thank Jonathan Katz, Tal Rabin, and Alon Rosen for helpful discussions. Finally, we thank Phil MacKenzie for clarifications regarding [13].

5. REFERENCES

[1] B. Barak. How to go beyond the black-box simulation barrier. In 42*nd FOCS*, pages 106–115, 2001.

[2] B. Barak and O. Goldreich. Universal Arguments and their Applications. 17*th IEEE Conference on Computational Complexity*, pages 194–203, 2002.

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

[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 Universally Composable Two-Party Computation Without Set-Up Assumptions. In *Eurocrypt*, 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] J. Garay and P. Mackenzie. Concurrent Oblivious Transfer. In 41*st FOCS*, pages 314–324, 2000.

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

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

[16] S. Goldwasser and L. Levin. Fair Computation of General Functions in Presence of Immoral Majority. In *CRYPTO'90,* pages 77–93, 1990.

[17] S. Goldwasser, S. Micali, and R.L. Rivest. A digital signature scheme secure against adaptive chosen- message attacks. *SICOMP*, 17(2):281–308, 1988.

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

[19] J. Kilian and E. Petrank. Concurrent and resettable zero-knowledge in poly-logartihmic rounds. In 33*rd STOC,* pages 560–569, 2001.

[20] J. Kilian, E. Petrank and C. Rackoff. Lower Bounds for Zero Knowledge on the Internet. In 39*th FOCS,* pages 484–492, 1998.

[21] S. Micali and P. Rogaway. Secure computation. Unpublished manuscript, 1992. Preliminary version in *CRYPTO'91*, pages 392–404, 1991.

[22] M. Prabhakaran, A. Rosen and A. Sahai. Concurrent Zero Knowledge With Logarithmic Round Complexity. In 43*rd FOCS,* pages 366–375, 2002.

[23] R. Richardson and J. Kilian. On the Concurrent Composition of Zero-Knowledge Proofs. In *Eurocrypt'99*, pages 415–413, 1999.

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

[25] A. Rosen. A Note on the Round-Complexity of Concurrent Zero-Knowledge. In *CRYPTO 2000,* Springer-Verlag (LNCS 1880), pages 451–468, 2000.

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