Black-Box Concurrent Zero-Knowledge Requires $\tilde{\Omega}(\log n)$ Rounds

(Extended Abstract)

Ran Canetti* Joe Kilian† Erez Petrank‡ Alon Rosen$^{\mathcal{S}}$

ABSTRACT

We show that any concurrent zero-knowledge protocol for a non-trivial language (i.e., for a language outside $\mathcal{BPP}$), whose security is proven via black-box simulation, must use at least $\tilde{\Omega}(\log n)$ rounds of interaction. This result substantially improves over previous lower bounds, and is the first bound to rule out the possibility of constant-round black-box concurrent zero-knowledge. Furthermore, the bound is polynomially related to the number of rounds in the best known concurrent zero-knowledge protocol for languages in $\mathcal{NP}$.

1. INTRODUCTION

Zero-knowledge proof systems, introduced by Goldwasser, Micali and Rackoff [17] are efficient interactive proofs that have the remarkable property of yielding nothing beyond the validity of the assertion being proved. The generality of zero-knowledge proofs has been demonstrated by Goldreich, Micali and Wigderson [15], who showed that every NP-statement can be proved in zero-knowledge provided that one-way functions exist [19, 22]. Since then, zero-knowledge proofs have turned out to be an extremely useful tool in the design of various cryptographic protocols.

The original setting in which zero-knowledge proofs were investigated consisted of a single prover and verifier which execute only one instance of the protocol at a time. A more realistic setting, especially in the time of the Internet, is one which allows the concurrent execution of zero-knowledge protocols. In the concurrent setting (see Feige [10], and more substantial treatment by Dwork, Naor and Sahai [8]), many protocols (sessions) are executed at the same time, involving many verifiers which may be talking with the same (or many) provers simultaneously. (The parallel composition operation considered in [14, 11, 13, 4, 2] is a special case.) This setting presents the new risk of a coordinated attack in which an adversary controls many verifiers, interleaving the executions of the protocols and choosing verifiers' messages based on other partial executions of the protocol. Since it seems unrealistic (and certainly undesirable) for honest provers to coordinate their actions so that zero-knowledge is preserved, we assume that in each prover-verifier pair the prover acts independently.

Loosely speaking, a zero-knowledge proof is said to be *concurrent zero-knowledge* if it remains zero-knowledge even when executed in the concurrent setting. Recall that in order to demonstrate that a certain protocol is zero-knowledge it is required to demonstrate that the view of every probabilistic polynomial-time adversary interacting with the prover can be simulated by a probabilistic polynomial-time machine (a.k.a. the *simulator*), based only on the fact that the proved theorem is correct. In the concurrent setting, the verifiers' view may include multiple sessions running at the same time. Furthermore, the verifiers may have control over the scheduling of the messages in these sessions (i.e., the order in which the interleaved execution of these sessions should be conducted). As a consequence, the simulator's task in the concurrent setting becomes considerably more complicated. In particular, standard techniques, based on "rewinding the adversary," run into trouble.

1.1 Previous Work

Constructing a "round-efficient" concurrent zero-knowledge protocol for languages outside $\mathcal{BPP}$ seems to be a challenging task. Intuition on the difficulty of this problem is given in [8], where it was argued that for a specific recursive scheduling of n sessions, the straightforward adaptation of the simulator to the concurrent setting requires time exponential in n. The first lower bound demonstrating the difficulty of concurrent zero-knowledge was given by Kilian, Petrank and Rackoff [21] who showed, building on the techniques of [14], that for every language outside $\mathcal{BPP}$ there is no 4-round protocol whose concurrent execution is simulatable in polynomial-time by a *black-box simulator*. (A black-box simulation is a simulator that has only black-box access to the adversarial verifier. Essentially all known proofs of security of zero-knowledge protocols use black-box simulators. An exception is the protocol of [18].) This lower bound was later improved by Rosen to seven rounds [24].

*IBM T.J. Watson Research Center. P.O. Box 704, Yorktown Heights, NY 10598, USA. canetti@watson.ibm.com.

†Yianilos Labs. Yianilos Labs 707 State Rd., Rt. 206, Suite 212, Princeton, NJ 08540, USA. joe@pnylab.com.

‡Dept. of Computer Science, Technion - Israel Institute of Technology, Haifa 32000, Israel. erez@cs.technion.ac.il.

$^{\mathcal{S}}$Dept. of Computer Science, Weizmann Institute of Science, Rehovot 76100, Israel. alon@wisdom.weizmann.ac.il. Part of this work was done while the author was visiting the IBM T.J. Watson Research Center.

STOC'01, July 6-8, 2001, Hersonissos, Crete, Greece.

Indeed, even ignoring issues of round efficiency, it was not clear whether there exists a concurrent zero-knowledge protocol for nontrivial languages, without modifying the underlying model. Richardson and Kilian [23] exhibited a family of concurrent zero-knowledge protocols (parameterized by the number of rounds) for all languages in $\mathcal{NP}$. Their original analysis showed how to simulate in polynomial-time $n^{O(1)}$ concurrent sessions only when the number of rounds in the protocol is at least n^{ϵ} (for some arbitrary $\epsilon > 0$). This result has recently been substantially improved by Kilian and Petrank [20], who show that the [23] protocol remains concurrent zero-knowledge even if it has $O(g(n) \cdot \log^2 n)$ rounds, where $g(\cdot)$ is any non-constant function (e.g., $g(n) = \log\log n$). Note that there was previously a considerable gap between the known upper and lower bounds on the round-complexity of concurrent zero-knowledge (i.e., [20, 24]): the best known protocol has $\tilde{O}(\log^2 n)$ rounds whereas the lower bound necessitates 7.[1]

1.2 Our Result

We substantially narrow the above gap by presenting a lower bound on the number of rounds required by concurrent zero-knowledge. We show that in the context of black-box concurrent zero-knowledge, $\tilde{\Omega}(\log n)$ rounds of interaction are essential for non-trivial proof systems.[2] This bound is the first to rule out the possibility of constant-round concurrent zero-knowledge, when proven via black-box simulation. Furthermore, the bound is polynomially related to the number of rounds in the best known concurrent zero-knowledge protocol for all languages in $\mathcal{NP}$ ([20]). Our main result is stated in the following theorem.

THEOREM 1. *Let $r : N \rightarrow N$ be a function so that $r(n) = o(\frac{\log n}{\log\log n})$. Suppose that $\langle P, V \rangle$ is an $r(\cdot)$-round proof system for a language L (i.e., on input x, the number of messages exchanged is at most $r(|x|)$), and that concurrent executions of P can be simulated in polynomial-time using black-box simulation. Then $L \in \mathcal{BPP}$. The theorem holds even if the proof system is only computationally-sound (with negligible soundness error) and the simulation is only computationally-indistinguishable (from the actual executions).*

1.3 Techniques

The proof of Theorem 1 builds on the works of Goldreich and Krawczyk [14], Kilian, Petrank and Rackoff [21], and Rosen [24]. On a very high level, the proof proceeds by constructing a specific concurrent schedule of sessions, and demonstrating that a black-box simulator cannot successfully generate a simulated accepting transcript for this schedule unless it "rewinds" the verifier *many times.* The work spent on these rewindings will be super-polynomial unless the number of rounds used by the protocol obeys the bound, or $L \in \mathcal{BPP}$. While the general outline of the proof remains roughly the same as in [14, 21, 24], the actual schedule of sessions, and its analysis, are new. One main idea that, together with other ideas, enables the bound is to have the verifier *abort* sessions depending on the history of the interaction.

[1] $f(n) = \tilde{O}(h(n))$ if there exists a constant $c > 0$ so that for all sufficiently large n, $f(n) \leq (\log h(n))^c \cdot h(n)$.

[2] $f(n) = \tilde{\Omega}(h(n))$ if there exists a constant $c > 0$ so that for all sufficiently large n, $f(n) \geq h(n)/(\log h(n))^c$.

1.4 Organization

A detailed outline of the proof, presenting both the general structure and the new ideas, appears in Section 3. The proof of Theorem 1 appears in Section 4. Section 4.1 describes the strategy of the adversarial verifier, including the adversarial scheduling of messages. Section 4.2 describes the decision procedure for L given a black-box simulator for the proof-system $\langle P, V \rangle$. Section 5 reviews some methods to circumvent the lower bound. Due to lack of space, the analysis of the decision procedure is omitted from this extended abstract (see [6] for a full version of this paper).

2. PRELIMINARIES

On the concurrent ZK model: We use the standard definitions of interactive proofs [17, 12] and arguments (a.k.a computationally-sound proofs) [3]. We consider interactive proof systems in which the soundness error is negligible.[3] Let $\langle P, V \rangle$ be an interactive proof (resp. argument) for a language L, and consider a (non-uniform, deterministic, cf. [14, 12, 21]) adversary verifier V^* that, given input $x \in L$, interacts with an unbounded number of independent copies of P (all on input x). V^* is allowed to interact with the various copies of P concurrently, without any restrictions (in particular, V^* has control over the scheduling of the messages in the different interactions with P). A proof system $\langle P, V \rangle$ is said to be concurrent zero-knowledge (ZK) (cf. [8, 23, 21]), if for every such V^* there exists a probabilistic polynomial-time simulator S that generates a view that is indistinguishable from V^*'s view in the above interaction. If there exists a single simulator S that can generate the view of *any* adversary V^*, given only oracle access to V^*, then we say that $\langle P, V \rangle$ is black-box concurrent ZK.

On the concurrent ZK simulator: We consider black-box simulators S that receive in advance a parameter K denoting the number of sessions that the adversary will use in its execution. Such simulators should run in worst-case time that is a fixed polynomial in K and in the security parameter. (Usually the simulator does not receive K and is required to work for all adversaries. By letting S have K as input we *strengthen* the lower bound.) The deviation gap of a simulator S for a proof-system $\langle P, V \rangle$ is defined, somewhat informally, as follows. Consider a distinguisher D that is required to decide whether its input consists of a transcript of a real interaction of $\langle P, V^* \rangle$ for some cheating verifier V^*, or to a transcript that was produced by S. The deviation gap of D is the difference between the probability that D outputs 1 given an output of S, and the probability that D outputs 1 given a transcript of a real interaction of $\langle P, V^* \rangle$. The deviation gap of S is the deviation gap of the best polynomial time distinguisher D. We consider simulators that run in strict (worst case) polynomial time, and have deviation gap at most 1/4. Using a standard argument, a lower bound on such simulators extends to a lower bound on simulators running in expected polynomial time.

[3] In the case of unconditional soundness, this condition can be relaxed to require only soundness error that is bounded away from 1. This is so, since the soundness error can always be made negligible by sufficiently many parallel repetitions of the protocol. However, we do not know whether this condition can be relaxed in the case of computationally sound proofs. In particular, in this case parallel repetitions do not necessarily reduce the soundness error (cf. [1]).

Additional conventions: By k-round protocols we mean protocols in which $2k+2$ messages are exchanged subject to the following conventions. The first message is a fixed initiation message by the verifier, denoted $\mathtt{v}_1$, which is answered by the prover's first message denoted $\mathtt{p}_1$. The following verifier and prover messages are denoted $\mathtt{v}_2, \mathtt{p}_2, \ldots, \mathtt{v}_{k+1}, \mathtt{p}_{k+1}$, where $\mathtt{v}_{k+1}$ is an ACCEPT/REJECT message indicating whether the verifier has accepted its input, and the last message (i.e., $\mathtt{p}_{k+1}$) is a fixed acknowledgment message sent by the prover.[4] We impose the following technical restrictions on the simulator (but claim that each of these restrictions can be easily satisfied): First we assume that the simulator never repeats the same query twice. (We refer to the messages sent by the simulator to the adversary as queries and to the adversary's messages as answers.). As in (cf. [14]), the queries of the simulator are prefixes of possible execution transcripts (in the concurrent setting).[5] Such a prefix is a sequence of alternating prover and verifier messages (which may belong to different sessions as determined by the fixed schedule) that ends with a prover message. The answer to the queries made by the simulator consists of a single verifier message (which belongs to the next scheduled session). Secondly, we assume that before making a query $\overline{q} = (b_1, a_1, \ldots, b_t, a_t)$, where the a's are prover messages, the simulator has made queries to all relevant prefixes (i.e., $(b_1, a_1, \ldots, b_i, a_i)$, for every $i < t$), and has obtained the b_i's as answers. Finally, we assume that before producing output $(b_1, a_1, \ldots, b_T, a_T)$, the simulator makes the query $(b_1, a_1, \ldots, b_T, a_T)$.

3. PROOF OUTLINE

This section contains an outline of the proof. To facilitate reading, we partition the outline into two parts: The first part reviews the general framework. (This part mainly follows previous works, namely [13, 21, 24].) The second part concentrates on the actual schedule and the particularities of our lower bound.

3.1 The high-level framework

Consider a k-round Concurrent Zero Knowledge proof system $\langle P, V\rangle$ for language L, and let S be a black-box simulator for $\langle P, V\rangle$. We use S to construct a $\mathcal{BPP}$ decision procedure for L. For this purpose, we construct a family $\{V_h\}$ of "cheating verifiers". (Each V_h will run multiple interleaved interactions with the prover P.) To decide on an input x, run S on an interaction with a cheating verifier V_h that was chosen at random from the constructed family; decide that $x \in L$ iff S outputs an accepting transcript of V_h.

The general structure of the family $\{V_h\}$ is roughly as follows. A member V_h in the family is identified via a hash function h taken from a hash-function family H having "much randomness" (or high independence). Specifically, the independence of H will be larger than the running time of S. This guarantees that, for our purposes, a function drawn randomly from H behaves like a random function. We define some fixed concurrent schedule of a number of sessions between V_h and the prover. In each session, V_h runs the code of the honest verifier V on input x and random input $h(a)$, where a is the current history of the (*multi-session*) interaction at the point where the session starts. V_h accepts if all the copies of V accept.

The proof of validity of the decision procedure is structured as follows. Say that S *succeeds* if it outputs an accepting transcript of V_h. It is first claimed that if $x \in L$ then a valid simulator S must succeed with high probability. Roughly speaking, this is so because each session behaves like the original proof system $\langle P, V\rangle$, and $\langle P, V\rangle$ accepts x with high probability. Demonstrating that the simulator almost never succeeds when $x \notin L$ is much more involved. Given S we construct a "cheating prover" P^* that makes the honest verifier V accept x with probability that is polynomially related to the success probability of S. The soundness of $\langle P, V\rangle$ now implies that S succeeds only with negligible probability.

In order to complete the high-level description of the proof, we must first define the following notions, that play a central role in the analysis. Consider the conversation between V_h and a prover. A session-prefix a is a prefix of this conversation this conversation that ends at the point where some new session starts. (Recall that V's random input for that new session is set to $h(a)$.) Next, consider the conversation between S and V_h in some run of S. (Such a conversation may contain many interleaved and incomplete conversations of V_h with a prover.) Roughly speaking, a message sent by S to the simulated V_h is said to have session prefix a if it relates to the session where the verifier randomness is $h(a)$. A session-prefix is called useful in a run of S if it was accepted (i.e., V_h sent an accepting message for that session-prefix), and if V_h has sent exactly k messages for session-prefix a, where k is the number of rounds in the protocol (i.e., S did not "fork" that session-prefix, where forking session-prefix a is an informal term meaning that S rewinds V_h to a point where V_h provides a second continuation for session-prefix a).

Returning to the proof, we sketch the construction of P^*. It first randomly chooses a function $h \overset{r}{\leftarrow} H$ and an index i. It then simulates an interaction between S and V_h, with the exception that P^* uses the messages sent by S that have the i^{th} session-prefix in that interaction as the messages that P^* sends to the actual verifier it interacts with; similarly, it uses the messages received from the actual verifier instead of V_h's messages in the i^{th} session-prefix in the simulated interaction. It can be seen that whenever the session-prefix chosen by P^* is useful, then $\langle P^*, V\rangle(x)$ accepts. Since the number of session-prefixes in an execution of S is bounded by a polynomial, it follows that if the conversation between S and V_h contains a useful session-prefix with non-negligible probability, then $\langle P^*, V\rangle(x)$ accepts with non-negligible probability. It is left to demonstrate that if S succeeds with non-negligible probability then the conversation between S and V_h contains a useful session-prefix with non-negligible probability. The above reasoning reduces the proof of the theorem to coming up with a construction of $\{V_h\}$, including the schedule of sessions, and demonstrating that $\{V_h\}$ satisfies the following two properties: (1) In an interaction between the honest prover P and V_h, the latter accepts with high probability. (2) If there exists a simulator S that succeeds with non-negligible probability then with non-negligible probability the conversation between S and V_h contains a useful session-prefix.

---

[4] The $\mathtt{p}_{k+1}$ message is an artificial message included in order to "streamline" the description of the adversarial schedule (see Section *4.1.1*).

[5] The transcript of an interaction consists of the common input x, followed by the sequence of prover and verifier messages exchanged during the interaction. For sake of simplicity, we choose to omit the input x from the transcript's representation (as it is implicit in the description of the verifier).

Remark: An alternative approach to proving Theorem 1 proceeds as follows. Given a concurrent ZK proof system $\langle P, V\rangle$ for L and a simulator S for $\langle P, V\rangle$, construct a prover P^* as described above. Now, on input x, run $\langle P^*, V\langle$ and accept x iff V accepts. Analyzing this decision procedure for L, it follows from the soundness of $\langle P, V\rangle$ that if $x \notin L$ then we will accept x only with non-negligible probability. The thrust of the proof is to show that if $x \in L$ then V accepts with non-negligible probability. It can be seen that this approach results in essentially the same analysis as the present one.

3.2 The schedule and additional ideas

We have seen that, using the above framework, the crux of the lower bound is to come up with a schedule that allows demonstrating properties (1) and (2). We describe our schedule. Our starting point is the schedule used in [21] to demonstrate the impossibility of black-box concurrent zero-knowledge in 2 rounds (namely, 4 messages). In that schedule there are n levels (n is polynomially related to the security parameter), where each level consists of a single session. The $(\ell+1)^{\text{th}}$-level session starts and ends in between the two rounds of the $(\ell)^{\text{th}}$-level session. Furthermore, V_h halts as soon as some session ends in V rejecting the input. That schedule suffices for demonstrating that any forking of an ℓ-level session past the "midpoint" (i.e., the point between the two rounds of that session) will cost the simulator work that is exponential in ℓ. More specifically, let $W(\ell)$ denote the work incurred by the simulator for a schedule of l levels; Then, roughly speaking, it is demonstrated that if the simulator violates condition (2) then $W(\ell) \geq 2 \cdot W(\ell-1)$ holds for all ℓ, with the implication that $W(n) \geq 2^{n-1}$.

A first attempt to generalize this schedule to the case of k rounds may proceed as follows. Start with a single session at level 1. Then, continue recursively where between any two consecutive rounds in a session at level ℓ start a new session at level $\ell+1$. The schedule ends when all n sessions were used. However, this schedule does not guarantee property (2): It can only be shown that a simulator that violates property (2) will satisfy $W(\ell) = \text{poly}(k) \cdot W(\ell-1)$. Furthermore, since the number of sessions in each level is k times the number of sessions in the previous level, there are only $O(\log_k n)$ levels. Thus the bound only requires that the work done by the simulator is $k^{O(\log_k n)} = n^{O(1)}$; this does not contradict the requirement that the simulator is poly-time. Indeed, this particular schedule can be efficiently simulated.

One method to circumvent this difficulty was used in [24]. However, that method extends the lower bound only up to 3 rounds (more precisely, 7 messages). Here we use a different method. We first add another, binary hash function, g, to the specification of V_h. This hash function is taken from a family G with sufficient independence, so that it looks like a random binary function. Now, before generating the next message in some session, $V_{g,h}$ first applies g to some predetermined part of the conversation so far. If g returns 0 then $V_{g,h}$ aborts the session by sending an "abort" message. If g returns 1 then V_h is run as usual. In addition, we replace each session in the above schedule (for k rounds) with a "block" of, say, n sessions. We now have n^2 sessions in a schedule. (This choice of parameters is made for convenience of presentation.) $V_{g,h}$ accepts a block of n sessions if at least 1/2 of the non-aborted sessions in this block were accepted (and not too many of the sessions in this block were aborted). Once a block is rejected, $V_{g,h}$ halts. At the end of the execution, $V_{g,h}$ accepts if all blocks were accepted.

The rationale behind the use of aborts can be explained as follows. Recall that a session-prefix a stops being useful only when $V_{g,h}$ sends more than k messages whose session-prefix is a. This means that a stops being useful only if S forks a *and in addition* g returns 1 in at least two of the continuations of a. This means that S is expected to fork session-prefix a several times before it stops being useful. Since each forking of a involves extra work of S on higher-level sessions, this may force S to invest considerably more work before a session stops being useful.

A bit more specifically, let p denote the probability, taken over the choice of g, that g returns 1 on a given input. In each attempt the session is not aborted with probability p. Thus S is expected to fork a session prefix $1/p$ times before it becomes non-useful. This gives hope that, in order to make sure that no session-prefix is useful, S must do work that satisfies a condition of the sort $W(\ell) \geq \Omega(1/p) \cdot W(\ell-1)$. This would mean that the work to simulate n sessions is at least $\Omega(p^{-\log_k n})$. Consequently, when the expression $p^{-\log_k n}$ is super-polynomial there is hope that condition (2) above is satisfied.

Clearly, the smaller p is chosen to be, the larger $p^{-\log_k n}$ is. However, p cannot be too small, or else the probability of a session to be ever completed will be too small, and condition (1) above will not be satisfied. Specifically, a k-round protocol is completed with probability p^k. We thus have to make sure that p^k is not negligible.

In the proof we set $p = n^{-1/2k}$. This guarantees that a session is completed with probability $p^k = n^{-1/2}$ (thus property (1) has hope to be satisfied). Furthermore, since $p^{-\log_k n}$ is super-polynomial whenever $k = o(\log n/\log\log n)$, there is hope that property (2) will be satisfied for $k = o(\log n/\log\log n)$.

3.3 The actual analysis

Demonstrating property (1) is straightforward. Demonstrating property (2) requires arguing on the dependency between the expected work done by the simulator and its success probability. This is a tricky business, since the choices made by the simulator (and in particular the amount of effort spent on making each session non-useful) may depend on past events. We go about this task by pinpointing a special property that holds for *any* successful run of the simulator, unless the simulator runs in super-polynomial time. Essentially, this property states that there exists a block of sessions such that none of the session-prefixes in this block were forked too many times. Using this property, we show that the probability (over the choices of $V_{g,h}$ and the simulator) that a run of the simulator contains no useful session-prefix is negligible.

4. PROOF OF THEOREM 1

Assuming towards the contradiction that a black-box simulator, denoted S, contradicting Theorem 1 exists, we will describe a probabilistic polynomial-time decision procedure for L, based on S. The first step towards describing the decision procedure for L involves the construction of an adversary verifier in the concurrent model.

4.1 The concurrent adversarial verifier

The description of the adversarial strategy proceeds in several steps. We start by describing the underlying (fixed) schedule of messages. Once the schedule is presented, we describe the adversary's strategy regarding the contents of the verifier messages.

4.1.1 The schedule

For each $x \in \{0,1\}^n$, we consider the following concurrent scheduling of n^2 sessions, all run on common input x.[6] The scheduling is defined recursively, where the scheduling of $m \leq n^2$ sessions (denoted $\mathcal{R}_m$) proceeds as follows:[7]

1. If $m \leq n$, sessions $1, \ldots, m$ are executed sequentially until they are all completed;

2. Otherwise, for $j = 1, \ldots, k+1$:

   Message exchange: Each of the first n sessions exchanges two messages (i.e., $\mathtt{v}_j, \mathtt{p}_j$);

   (These first n sessions out of $1, \ldots, m$ will be referred to as the main sessions of $\mathcal{R}_m$.)

   Recursive call: If $j < k+1$, the scheduling is applied recursively to $\lfloor \frac{m-n}{k} \rfloor$ new sessions;

   (This is done using the next $\lfloor \frac{m-n}{k} \rfloor$ remaining sessions out of $1, \ldots, m$.)

The schedule is depicted in Figure 1. Notice that the verifier typically postpones its answer (i.e., $\mathtt{v}_j$) to the last prover's message (i.e., $\mathtt{p}_{j-1}$) till after a recursive sub-schedule is executed, and that in the j^{th} iteration, $\lfloor \frac{m-n}{k} \rfloor$ new sessions are initiated (with the exception of the first iteration, in which the first n (main) sessions are initiated as well). The order in which the messages of various sessions are exchanged is fixed but immaterial. Say that we let the first session proceed, then the second and so on. That is, we have the order $\mathtt{v}_j^{(1)}, \mathtt{p}_j^{(1)}, \ldots, \mathtt{v}_j^{(n)}, \mathtt{p}_j^{(n)}$, where $\mathtt{v}_j^{(i)}$ (resp. $\mathtt{p}_j^{(i)}$) denotes the verifier's (resp. prover's) j^{th} message in the i^{th} session.

The set of n sessions that are explicitly executed during the message exchange phase of the recursive invocation (i.e., the main sessions) is called a recursive block. (Notice that each recursive block corresponds to exactly one recursive invocation of the schedule.) Taking a closer look at the schedule we observe that every session in the schedule is explicitly executed in exactly one recursive invocation (that is, belongs to exactly one recursive block). Since the total number of sessions in the schedule is n^2, and since the message exchange phase in each recursive invocation involves the explicit execution of n sessions (in other words, the size of each recursive block is n), we have that the total number of recursive blocks in the schedule is equal to n. Since each recursive invocation of the schedule involves the invocation of k additional sub-schedules, the recursion actually corresponds to a k-ary tree with n nodes. The depth of the recursion is thus $\lfloor \log_k((k-1)n+1) \rfloor$, and the number of "leaves" in the recursion (i.e., sub-schedules of size smaller that n) is at least $\lfloor \frac{(k-1)n+1}{k} \rfloor$.

[6] Recall that each session consists of $2k+2$ messages, and that $k = k(n) = o(\log n / \log\log n)$.

[7] In general, we may want to define a recursive scheduling for sessions $i_1, \ldots, i_s$ and denote it by $\mathcal{R}_{i_1,\ldots,i_s}$. We choose to simplify the exposition by renaming these sessions as $1, \ldots, m$ and denote the scheduling by $\mathcal{R}_m$.

Identifying sessions according to their block: To simplify the exposition of the proof, it will be convenient to associate every session appearing in the schedule with a pair of indices $(\ell, i) \in \{1, \ldots, n\} \times \{1, \ldots, n\}$ (rather than with a single index $s \in \{1, \ldots, n^2\}$). The value of $\ell = \ell(s) \in \{1, \ldots, n\}$ will represent the index of the recursive block to which session s belongs (according to some canonical enumeration of the n invocations in the recursive schedule, say according to the order in which they are invoked), whereas the value of $i = i(s) \in \{1, \ldots, n\}$ will represent the index of session s within the n sessions that belong to the ℓ^{th} recursive block. In other words, session (ℓ, i) is the i^{th} main session of the ℓ^{th} recursive invocation in the schedule. Typically, when we explicitly refer to messages of session (ℓ, i), the index of the corresponding recursive block (i.e., ℓ) is easily deducible from the context. In such cases, we will sometimes omit the index ℓ from the "natural" notation $\mathtt{v}_j^{(\ell,i)}$ (resp. $\mathtt{p}_j^{(\ell,i)}$), and stick to the notation $\mathtt{v}_j^{(i)}$ (resp. $\mathtt{p}_j^{(i)}$). Note that once the schedule is fixed (which is clearly the case in our proof), the values of (ℓ, i) and the session index s are completely interchangeable (in particular, $\ell = s \text{ div } n$ and $i = s \bmod n$).

Definition 1. (Identifiers of next message:) *The schedule defines a mapping from partial execution transcripts ending with a prover message to the* identifiers of the next verifier message*; that is, the session and round number to which the next verifier message belongs.* (Recall that such partial execution transcripts correspond to queries of a black-box simulator and so the mapping defines the identifier of the answer:) *For such a query $\overline{q} = (b_1, a_1, \ldots, b_t, a_t)$, we denote by $\pi_{\text{sn}}(\overline{q}) = (\ell, i) \in \{1, \ldots, n\} \times \{1, \ldots, n\}$ the session to which the next verifier message belongs, and by $\pi_{\text{msg}}(\overline{q}) = j \in \{1, \ldots, k+1\}$ its index within the verifier's messages in this session.*

Note that the identifiers of the next message are uniquely determined by the number of messages appearing in the query (and are not affected by the contents of these messages).

Once the identifiers of the next verifier message are deduced from the query's length, one has to specify a strategy according to which the contents of the next verifier message will be determined. Loosely speaking, our adversary verifier has two options: It will either send the answer that would have been sent by an honest verifier (given the messages in the query that are relevant to the current session), or it will choose to deviate from the honest verifier strategy and abort the interaction in the current session (this will be done by answering with a special `ABORT` message). Since in a non-trivial zero-knowledge proof system the honest verifier is always probabilistic, and since the "abort behaviour" of the adversary verifier should be "unpredictable" for the simulator, we have that both options require a source of randomness (either for computing the contents of the honest verifier answer or for deciding whether to abort the conversation). As is already customary in works of this sort [14, 21, 24], we let the source of randomness be a hash function with sufficiently high independence (which is "hard-wired" into the verifier's description).

Determining the randomness for a session: Focusing (first) on the randomness required to compute the honest verifier's answers, we ask what should the input of the above

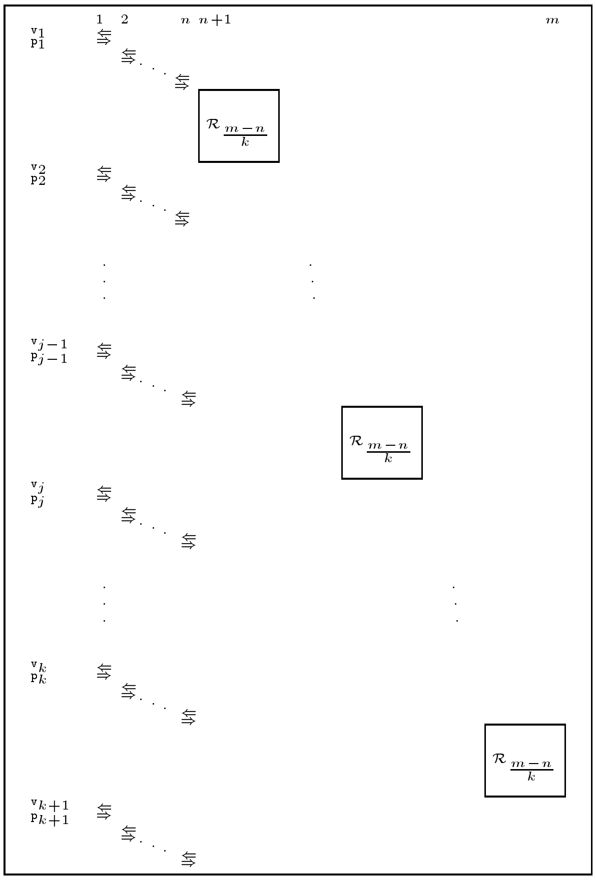

Figure 1: The recursive schedule $\mathcal{R}_m$ for m sessions. Columns correspond to m individual sessions and rows correspond to the time progression.

hash function be. A naive solution would be to let the randomness for a session depend on the session's index. That is, to obtain randomness for session $(\ell, i) = \pi_{\rm sn}(\overline{q})$ apply the hash function to the value (ℓ, i). This solution will indeed imply that every two sessions have independent randomness (as the hash function will have different inputs). However, the solution seems to fail to capture the difficulty arising in the simulation of multiple concurrent sessions. What we would like to have is a situation in which whenever the simulator rewinds a session (that is, feeds the adversary verifier with a different query of the same length), it causes the randomness of some other session (say, one level down in the recursive schedule) to be completely modified. To achieve this, we must cause the randomness of a session to depend also on the history of the entire interaction. Changing even a single message in this history would immediately result in an unrelated instance of the session, and would thus force the simulator to redo the simulation work on this session all over again.

So where in the schedule should the randomness of session (ℓ, i) be determined? On the one hand, we would like to determine the randomness of a session as late as possible (in order to maximize the effect of changes in the history of the interaction on the randomness of the session). On the other hand, we cannot afford to determine the randomness after the session's initiating message is scheduled (as the protocol's specification may require that the verifier's randomness is completely determined before the first message in the protocol is exchanged). The point in which we choose to determine the randomness of session (ℓ, i) is the point in which recursive block number ℓ is invoked. That is, to obtain the randomness of session $(\ell, i) = \pi_{\rm sn}(\overline{q})$ we feed the hash function with the prefix of query $\overline{q}$ that ends just before the first message in block number ℓ (remember that queries correspond to partial execution transcripts and thus contain the whole history of the interaction so far).[8] This prefix is called the block-prefix of query $\overline{q}$ and is defined next.

Definition 2. (Block-prefix:) *The* block-prefix *of a query $\overline{q}$ satisfying $\pi_{\rm sn}(\overline{q}) = (\ell, i)$, is the prefix of $\overline{q}$ that is answered with the first verifier message of session $(\ell, 1)$ (that is, the first main session in block number ℓ). More formally, $bp(\overline{q}) = (b_1, a_1, \ldots, b_\gamma, a_\gamma)$, is the block-prefix of $\overline{q} = (b_1, a_1, \ldots, b_t, a_t)$ if $\pi_{\rm sn}(bp(\overline{q})) = (\ell, 1)$ and $\pi_{\rm msg}(bp(\overline{q})) = 1$. The block-prefix will be said to correspond to recursive block number ℓ.*[9] (Note that i may be any index in $\{1, \ldots, n\}$, and that a_t need not belong to session (ℓ, i).)

Determining whether and when to abort sessions: Whereas the randomness that is used to compute the honest verifier's answers in each session is determined before a session begins, the randomness that is used in order to decide whether to abort a session is chosen independently every time the execution of the schedule reaches the next verifier message in this session. As before, the required randomness is obtained by applying a hash function to the suitable prefix of the execution transcript. This time, however, the length of the prefix increases each time the execution of the session reaches the next verifier message (rather than being fixed for the whole execution of the session). This way, the decision of whether to abort a session also depends on the contents of messages that were exchanged after the initiation of the session has occured. Specifically, in order to decide whether to abort session $(\ell, i) = \pi_{\rm sn}(\overline{q})$ at the $j^{\rm th}$ message (where $j = \pi_{\rm msg}(\overline{q})$), we feed the hash function with the prefix of query $\overline{q}$ that ends with the $(j-1)^{\rm st}$ prover message in the $n^{\rm th}$ main session of block number ℓ. This prefix is called the iteration-prefix of query $\overline{q}$ and is defined next (see Figure 2 for a graphical description of the block-prefix and iteration-prefix of a query).[10]

Definition 3. (Iteration-prefix:) *The* iteration-prefix *of a query $\overline{q}$ satisfying $\pi_{\rm sn}(\overline{q}) = (\ell, i)$ and $\pi_{\rm msg}(\overline{q}) = j > 1$, is the prefix of $\overline{q}$ that ends with the $(j-1)^{\rm st}$ prover message in session (ℓ, n) (that is, the $n^{\rm th}$ main session in block number ℓ). More formally, $ip(\overline{q}) = (b_1, a_1, \ldots, b_\delta, a_\delta)$, is the*

[8] In order to achieve independence from other sessions in block number ℓ, we will also feed the hash function with the value of i. This (together with the above choice) guarantees the following properties: (1) The input to the hash function (and thus the randomness for session (ℓ, i)) does not change once the interaction in the session begins. (2) For every pair of different sessions, the input to the hash function is different (and thus the randomness for each session is independent). (3) Even a single modification in the prefix of the interaction up to the first message in block number ℓ, induces fresh randomness for all sessions in block number ℓ.

[9] In the special case that $\ell = 1$ (that is, we are in the first block of the schedule), we define $bp(\overline{q}) = \perp$.

[10] As before, the hash function is also fed with the value of i (in order to achieve independence from other sessions).

iteration-prefix of $\overline{q} = (b_1, a_1, \ldots, b_t, a_t)$ *if* a_δ *is of the form* $\mathrm{p}_{j-1}^{(n)}$ *(where* $\mathrm{p}_{j-1}^{(n)}$ *denotes the* $(j-1)^{\mathrm{st}}$ *prover message in the* n^{th} *main session of block number* ℓ*). This iteration-prefix is said to correspond to the block-prefix of* $\overline{q}$*.* (Again, note that i may be any index in $\{1, \ldots, n\}$, and that a_t need not belong to session (ℓ, i). Also note that two queries $\overline{q}_1, \overline{q}_2$ that satisfy $bp(\overline{q}_1) = bp(\overline{q}_2)$, and $\pi_{\mathrm{msg}}(\overline{q}_1) = \pi_{\mathrm{msg}}(\overline{q}_2)$ may have the same iteration-prefix (even if $\pi_{\mathrm{sn}}(\overline{q}_1) \neq \pi_{\mathrm{sn}}(\overline{q}_2)$). Finally, note that the iteration-prefix is defined only for $\pi_{\mathrm{msg}}(\overline{q}) > 1$.)

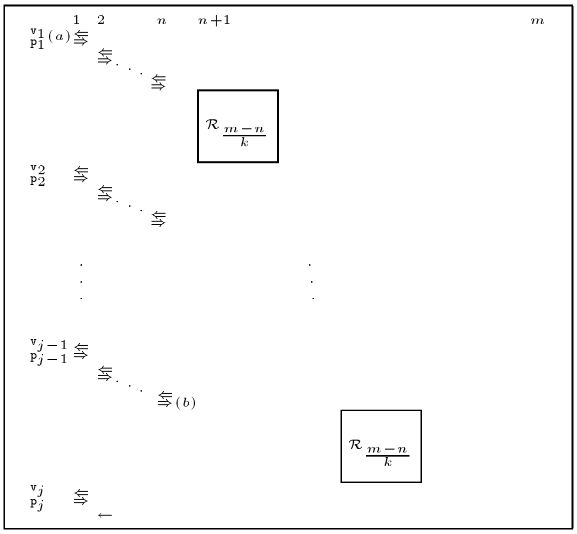

Figure 2: Determining the prefixes of query $\overline{q}$ (in this example, query $\overline{q}$ ends with a $\mathrm{p}_j^{(1)}$ message): (a) The block-prefix of $\overline{q}$ - messages up to this point are used by $V_{g,h}$ to determine the randomness to be used for computing message $\mathrm{v}_j^{(2)}$. (b) The iteration-prefix of $\overline{q}$ - messages up to this point are used by $V_{g,h}$ to determine whether message $\mathrm{v}_j^{(2)}$ will be set to ABORT.

Motivating discussion: The choices made in Definitions 2 and 3 are designed to capture the difficulties encountered whenever many sessions are to be simulated concurrently. As was previously mentioned, we would like to create a situation in which every attempt of the simulator to rewind a specific session will result in loss of work done for other sessions, and will cause the simulator to do the same amount of work all over again. In order to force the simulator to repeat each such rewinding attempt many times, we make each rewinding attempt fail with some predetermined probability (by letting the verifier send an ABORT message instead of a legal answer).[11]

To see that Definitions 2 and 3 indeed lead to the fulfillment of the above requirements, we consider the following (informal) example. Suppose that at some point during the simulation, the adversary verifier aborts session (ℓ, i) at the j^{th} message (while answering query $\overline{q}$). Further suppose that (for some unspecified reason) the simulator wants to to get a "second chance" in receiving a legal answer to the j^{th} message in session (ℓ, i) (hoping that it will not receive the ABORT message again). Recall that the decision of whether to abort a session depends on the outcome of a hash function when applied to the iteration-prefix $ip(\overline{q})$, of query $\overline{q}$. In particular, to obtain a "second chance", the black-box simulator has no choice but to change at least one prover message in the above iteration-prefix (in other words, the simulator must rewind the interaction to some message occurring in iteration-prefix $ip(\overline{q})$). At first glance it may seem that the effect of changes in the iteration-prefix of query $\overline{q}$ is confined to the messages that belong to session $(\ell, i) = \pi_{\mathrm{sn}}(\overline{q})$ (or at most, to messages that belong to other sessions in block number ℓ). Taking a closer look at the schedule, we observe that every iteration-prefix (and in particular $ip(\overline{q})$) can also be viewed as the block-prefix of a recursive block one level down in the recursive construction. Viewed this way, it is clear that the effect of changes in $ip(\overline{q})$ is not confined only to messages that correspond to recursive block number ℓ, but rather extends also to sessions at lower levels in the recursive schedule. By changing even a single message in iteration-prefix $ip(\overline{q})$, the simulator is actually modifying the block-prefix of all recursive blocks in a sub-schedule one level down in the recursive construction. This means that the randomness for all sessions in these blocks is completely modified (recall that the randomness of a session is determined by applying a hash function to the corresponding block-prefix), and that all the simulation work done for these sessions is lost. In particular, by changing even a single message in iteration-prefix $ip(\overline{q})$, the simulator will find itself doing the simulation work for these sessions all over again.

Having established the effect of changes in iteration-prefix $ip(\overline{q})$ on sessions at lower levels in the recursive schedule, we now turn to examine the actual effect on session $(\ell, i) = \pi_{\mathrm{sn}}(\overline{q})$ itself. One possible consequence of changes in iteration-prefix $ip(\overline{q})$ is that they may also affect the contents of the block-prefix $bp(\overline{q})$ of query $\overline{q}$ (notice that, by definition, the block-prefix $bp(\overline{q})$ of query $\overline{q}$ is contained in the iteration-prefix $ip(\overline{q})$ of query $\overline{q}$). Whenever this happens, the randomness used for session (ℓ, i) is effectively "chosen anew" and all simulation work done for this session will be lost. A more interesting consequence of a change in the contents of iteration-prefix $ip(\overline{q})$, is that it will result in a completely independent decision of whether session (ℓ, i) is to be aborted at the j^{th} message (the decision of whether to abort is taken whenever the simulator makes a query $\overline{q}$ satisfying $\pi_{\mathrm{sn}}(\overline{q}) = (\ell, i)$, and $\pi_{\mathrm{msg}}(\overline{q}) = j$). In other words, each time the simulator attempts to get a "second chance" in receiving a legal answer to the j^{th} message in session (ℓ, i) (by rewinding the interaction to a message that belongs to iteration-prefix $ip(\overline{q})$), it faces the risk of being answered with an ABORT message independently of all previous rewinding attempts.

Indeed, as we demonstrate in the proof of Theorem 1, the choices made in our definitions lead to the fulfillment of the requirements that we would like to meet.

4.1.2 *The verifier strategy* $V_{g,h}$

We consider what happens when a simulator S (for the above schedule) is given oracle access to a verifier strategy $V_{g,h}$ defined as follows (depending on hash functions g, h and the input x). Recall that we may assume that S

[11] Recall that all of the above is required in order to make the simulator's work accumulate to too much, and eventually cause its running time to be super-polynomial.

runs in strict polynomial time: we denote such time bound by $t_S(\cdot)$. Let G denote a small family of $t_S(n)$-wise independent hash functions mapping poly(n)-bit long sequences into a single bit of output, so that for every α we have $\Pr_g[g(\alpha) = 1] = n^{-1/2k}$. Let H denote a small family of $t_S(n)$-wise independent hash functions mapping poly(n)-bit long sequences to $\rho_V(n)$-bit sequences, where $\rho_V(n)$ is the number of random bits used by an honest verifier V on an input $x \in \{0,1\}^n$. We describe a family $\{V_{g,h}\}_{g\in G, h\in H}$ of adversarial verifier strategies (where x is implicit in $V_{g,h}$). On query $\overline{q} = (b_1, a_1, \ldots, a_{t-1}, b_t, a_t)$, the verifier acts as follows:

1. First, $V_{g,h}$ checks if the execution transcript given by the query is legal, and halts with a special `ERROR` message if the query is not legal.[12]

2. Next, $V_{g,h}$ determines the block-prefix, $bp(\overline{q})$, of query $\overline{q}$. It also determines the identifiers of the next-message $(\ell, i) = \pi_{\text{sn}}(\overline{q})$ and $j = \pi_{\text{msg}}(\overline{q})$, the iteration-prefix $ip(\overline{q}) = (b_1, a_1, \ldots, b_\delta, \mathtt{p}_{j-1}^{(n)})$, and the $j-1$ prover messages of session i appearing in query $\overline{q}$ (which we denote by $\mathtt{p}_1^{(i)}, \ldots, \mathtt{p}_{j-1}^{(i)}$).

   (**Motivating discussion:** The value of the block-prefix, $bp(\overline{q})$, is used in order to determine the randomness of session (ℓ, i), whereas the value of the iteration-prefix, $ip(\overline{q})$, is used in order to determine whether session (ℓ, i) is about to be aborted at this point (i.e., j^{th} message) in the schedule (by answering with a special `ABORT` message).)

3. If $j = 1$, then $V_{g,h}$ answers with the verifier's fixed initiation message for session i (i.e., $\mathtt{v}_1^{(i)}$).

4. If $j > 1$, then $V_{g,h}$ determines $b_{i,j} = g(i, ip(\overline{q}))$ (as a bit deciding whether to abort session i):

   (a) If $b_{i,j} = 0$, then $V_{g,h}$ sets $\mathtt{v}_j^{(i)} = \mathtt{ABORT}$ (indicating that $V_{g,h}$ aborts session i).

   (b) If $b_{i,j} = 1$, then $V_{g,h}$ determines $r_i = h(i, bp(\overline{q}))$ (as coins to be used by V), and computes the message $\mathtt{v}_j^{(i)} = V(x, r_i; \mathtt{p}_1^{(i)}, \ldots, \mathtt{p}_{j-1}^{(i)})$ that would have been sent by the honest verifier on common input x, random-pad r_i, and prover's messages $\mathtt{p}_1^{(i)}, \ldots, \mathtt{p}_{j-1}^{(i)}$.

   (c) Finally, $V_{g,h}$ answers with $\mathtt{v}_j^{(i)}$.

Dealing with `ABORT` messages: Note that, once $V_{g,h}$ has aborted a session, the interaction in this session essentially stops, and there is no need to continue exchanging messages in this session. However, for simplicity of exposition we assume that the verifier and prover stick to the fixed schedule of Section *4.1.1* and exchange `ABORT` messages whenever an aborted session is scheduled.

[12]In particular, $V_{g,h}$ checks whether the query is of the prescribed format (as described in Section 2, and as determined by the schedule), and that the contents of its messages is consistent with $V_{g,h}$'s prior answers. (That is, for every proper prefix $\overline{q}' = (b_1, a_1, \ldots, b_u, a_u)$ of query $\overline{q} = (b_1, a_1, \ldots, b_t, a_t)$, the verifier checks whether the value of b_{u+1} (as it appears in $\overline{q}$) is indeed equal to the value of $V_{g,h}(\overline{q}')$.)

On the arguments to g and h: The hash function h, which determines the random input for V in a session, is applied both to i (the identifier of the relevant session in the current block) and to the entire block-prefix of the query $\overline{q}$. This means that even though all sessions in a specific block have the same block-prefix, for every pair of two different sessions, the corresponding random inputs of V will be independent of each other (as long as the number of applications of h does not exceed $t_S(n)$, which is indeed the case in our application). The hash function g, which determines whether and when the verifier aborts sessions, is applied both to i and to the entire iteration-prefix of the query $\overline{q}$. As in the case of h, the decision whether to abort a session is independent from the same decision for other sessions (again, as long as g is not applied more than $t_S(n)$ times). However, there is a significant difference between the inputs of h and g: Whereas the input of h is *fixed* once (ℓ, i) is fixed (for every possible value of the message number index j), the input of g *varies* depending on the value of j. In particular, whereas the randomness of a session is completely determined once the session begins, the decision of whether to abort a session is taken independently each time that the schedule reaches the next verifier message of this session.

On the number of different prefixes that occur in interactions with $V_{g,h}$: Since the number of recursive blocks in the schedule is equal to n, and since there is a one-to-one correspondence between recursive blocks and block-prefixes, we have that the number of different block-prefixes that occur during an interaction between and honest prover P and the verifier $V_{g,h}$ is always equal to n. Since the number of iterations in the message exchange phase of a recursive invocation of the schedule equals $k+1$, and since there is a one-to-one correspondence between such iterations and iteration-prefixes[13] we have that the number of different iteration-prefixes that occur during an interaction between and honest prover P and the verifier $V_{g,h}$, is always equal to $k \cdot n$ (that is, k different iteration-prefixes for each one of the n recursive invocations of the schedule). In contrast, the number of different block-prefixes (resp. iteration-prefixes), that occur during an execution of a black-box simulator S that is given oracle access to $V_{g,h}$, may be considerably larger than n (resp. $k \cdot n$). The reason for this is that there is nothing that prevents the simulator to feed $V_{g,h}$ with different queries of the same length (this corresponds to the so called rewinding of an interaction). Still, the number of different prefixes in an execution of S is always upper bounded by the running time of S; that is, $t_S(n)$.

On the probability that a session is never aborted: A typical interaction between an honest prover P and the verifier $V_{g,h}$ will contain sessions whose execution has been aborted prior to completion. Recall that at each point in the schedule, the decision of whether or not to abort the next scheduled session depends on the outcome of g. Since the function g returns 1 with probability $n^{-1/2k}$, a specific session is never aborted with probability $(n^{-1/2k})^k = n^{-1/2}$. Using the fact that whenever a session is not aborted, $V_{g,h}$ operates as the honest verifier, we infer that the probability

[13]The only exception is the first iteration in the message exchange phase. Since only queries $\overline{q}$ that satisfy $\pi_{\text{msg}}(\overline{q}) > 1$ have an iteration-prefix, the first iteration will never have a corresponding iteration-prefix.

that a specific session is eventually accepted by $V_{g,h}$ is at least $1/2$ times the probability that the very same session is never aborted (where $1/2$ is an arbitrary lower bound on the completeness probability of the protocol). In other words, the probability that a session is accepted by $V_{g,h}$ is at least $n^{-1/2}/2$. In particular, for every set of n sessions, the expected number of sessions that are eventually accepted by $V_{g,h}$ (when interacting with the honest prover P) is at least $n \cdot n^{-1/2}/2 = n^{1/2}/2$, and with overwhelming probability at least $n^{1/2}/4$ sessions are accepted by $V_{g,h}$.

A slight modification of the verifier strategy: To facilitate the analysis, we slightly modify the verifier strategy so that it does not allow the number of accepted sessions in the history of the interaction to deviate much from its "expected behavior". Loosely speaking, given a prefix of the execution transcript (ending with a prover message), the verifier will check whether the recursive block that has just been completed contains at least $n^{1/2}/4$ accepted sessions. (To this end, it will be sufficient to inspect the history of the interaction only when the execution of the schedule reaches the end of a recursive block. That is, whenever the schedule reaches the last prover message in the last session of a recursive block (i.e., some $\mathtt{p}_{k+1}^{(n)}$ message).) The modified verifier strategy (which we continue to denote by $V_{g,h}$), is obtained by adding to the original strategy an additional Step 1' (to be executed after Step 1 of $V_{g,h}$):

1'. If a_t is of the form $\mathtt{p}_{k+1}^{(n)}$ (i.e., in case query $\overline{q}$ ends with the last prover message of the n^{th} main session of a recursive block), $V_{g,h}$ checks whether the transcript $\overline{q} = (b_1, a_1, \ldots, b_t, \mathtt{p}_{k+1}^{(n)})$ contains the accepting conversations of at least $n^{1/2}/4$ main sessions in the block that has just been completed. In case it does not, $V_{g,h}$ halts with a special `DEVIATION` message (indicating that the number of accepted sessions in the block that has just been completed deviates from its expected value).

Motivating discussion: Since the expected number of accepted sessions in a specific block is at least $n^{1/2}/2$, the probability that the block contains less than $n^{1/2}/4$ accepted sessions is negligible. Still, the above modification is not superfluous (even though it refers to events that occur only with negligible probability): It allows us to assume that every recursive block that is completed *during the simulation* (including those that do not appear in the simulator's output) contains at least $n^{1/2}/4$ accepted sessions. In particular, whenever the simulator feeds $V_{g,h}$ with a partial execution transcript (i.e., a query), we are guaranteed that for every completed block in this transcript, the simulator has indeed "invested work" to simulate the $n^{1/2}/4$ accepted sessions in the block.

A slight modification of the simulator: Before presenting the decision procedure, we slightly modify the simulator so that it never makes a query that is answered with either the `ERROR` or `DEVIATION` messages by the verifier $V_{g,h}$. Note that this condition can be easily checked by the simulator itself,[14] and that the modification does not effect the simulator's output. From this point on, when we talk of the simulator (which we continue to denote by S) we mean the modified one.

[14]We stress that, as opposed to the `ERROR` and `DEVIATION` messages, the simulator cannot predict whether its query is about to be answered with the `ABORT` message.

4.2 The decision procedure for L

We are now ready to describe a probabilistic polynomial-time decision procedure for L, based on the black-box simulator S and the verifier strategies $V_{g,h}$. On input $x \in \{0,1\}^n$, the procedure operates as follows:

1. Uniformly select hash functions $g \overset{r}{\leftarrow} G$ and $h \overset{r}{\leftarrow} H$.
2. Invoke S on input x providing it black-box access to $V_{g,h}$ (as defined above). That is, the procedure emulates the execution of the oracle machine S on input x along with emulating the answers of $V_{g,h}$.
3. Accept if and only if S outputs an accepting transcript (as determined by Steps 1 and 4b' of $V_{g,h}$).[15]

By our hypothesis, the above procedure runs in probabilistic polynomial-time. The performance of the procedure is established by the following lemmata. Due to lack of space the proofs of both lemmas are omitted from this extended abstract. They appear in [6].

LEMMA 1. (performance on YES-instances): *For all but finitely many $x \in L$, the above procedure accepts x with probability at least $2/3$.*

LEMMA 2. (performance on NO-instances): *For all but finitely many $x \notin L$, the above procedure rejects x with probability at least $2/3$.*

We can actually prove that for every polynomial $p(\cdot)$ and for all but finitely many $x \notin L$, the above procedure accepts x with probability at most $1/p(|x|)$. Assuming towards contradiction that this is not the case, we will construct a (probabilistic polynomial-time) strategy for a cheating prover that fools the honest verifier V with success probability at least $1/\text{poly}(n)$ (in contradiction to the computational-soundness of the proof system). Loosely speaking, the argument capitalizes on the fact that rewinding of a session requires the simulator to work on a new simulation sub-problem (one level down in the recursive construction). New work is required since each different message for the rewinded session forms an unrelated instance of the simulation sub-problem (by virtue of definition of $V_{g,h}$). Furthermore, since each message in the protocol is aborted with probability $1 - n^{-1/2k}$, each rewinding attempt is bound to fail with the same probability. As a consequence, the simulator (which must succeed to rewind with reasonably high probability) is forced to repeat the rewinding attempts sufficiently many times. The schedule causes work involved in such multiple rewindings to accumulate too much, and so it must be the case that the simulator does not make too many attempts to rewind some (full instance of some) session. But in this case, the cheating prover may attempt to use such a session in order to fool the verifier.

[15]Recall that we are assuming that the simulator never makes a query that is ruled out by Steps 1 and 1' of $V_{g,h}$. Since before producing output $(b_1, a_1, \ldots, b_T, a_T)$ the simulator makes the query $(b_1, a_1, \ldots, b_T, a_T)$, Step 3 is not really necessary (as, in case that the modified simulator indeed reaches the output stage "safely", we are guaranteed that it will produce a legal output). In particular, we are always guaranteed that the simulator produces execution transcripts in which every recursive block contains at least $n^{1/2}/4$ sessions that were accepted by $V_{g,h}$.

5. CONCLUSIONS

5.1 Alternative models

The lower bound presented here draws severe limitations on the ability of black-box simulators to cope with the standard concurrent zero-knowledge setting, and provides motivation to consider relaxations of and augmentations to the standard model. Indeed, several works have managed to "bypass" the difficulty in constructing concurrent zero-knowledge protocols by modifying the standard model in a number of ways. Dwork, Naor and Sahai augment the communication model with assumptions on the maximum delay of messages and skews of local clocks of parties [8, 9]. Damgård uses a common random string [7], and Canetti et.al. use a public registry file [5].

A different approach would be to try and achieve security properties that are weaker than zero-knowledge but are still useful. For example, Feige and Shamir consider the notion of *witness indistinguishability* [10, 11], which is preserved under concurrent composition.

5.2 Alternative simulation techniques

Loosely speaking, the only advantage that a black-box simulator may have over the honest prover is the ability to rewind the interaction and explore different execution paths before proceeding with the simulation (as its access to the verifier's strategy is restricted to the examination of input/output behavior). As we have seen in our proof, such a mode of operation (i.e., the necessity to rewind every session) is a major contributor to the hardness of simulating many concurrent sessions. Coming up with a simulator that deviates from this paradigm (i.e., is non black-box, in the sense that it does not necessarily have to rewind the adversary verifier in order to obtain a faithful simulation of the conversation), would essentially bypass the main problem that arises while trying to simulate many concurrent sessions. We stress that our result does not rule out non black-box simulators and that, as far as we know, they could conceivably exist. Unfortunately, such simulators seem difficult to construct.

Hada and Tanaka [18] consider some weaker variants of zero-knowledge, and exhibit a three-round protocol for $\mathcal{NP}$ (whereas only languages in $\mathcal{BPP}$ have three-round block-box zero-knowledge protocols [14]). Their protocol is the only known example of a zero-knowledge protocol not proven secure via black-box simulation. We remark that their proof can be extended to show that their protocol remains zero-knowledge for any language in $\mathcal{NP}$ even in the concurrent setting. However, their analysis is based in an essential way on a strong and highly non-standard hardness assumption.

6. ACKNOWLEDGEMENTS

We are indebted to Oded Goldreich for his devoted help and technical contribution to this project.

7. REFERENCES

[1] M. Bellare, R. Impagliazzo and M. Naor. Does Parallel Repetition Lower the Error in Computationally Sound Protocols? In *38th FOCS*, pages 374–383, 1997.

[2] M. Bellare, S. Micali, and R. Ostrovsky. Perfect zero-knowledge in constant rounds. In *22nd STOC*, pages 482–493, 1990.

[3] G. Brassard, D. Chaum and C. Crépeau. Minimum Disclosure Proofs of Knowledge. *JCSS*, Vol. 37, No. 2, pages 156–189, 1988.

[4] G. Brassard, C. Crépeau and M. Yung. Constant-Round Perfect Zero-Knowledge Computationally Convincing Protocols. *Theoret. Comput. Sci.* , Vol. 84, pp. 23-52, 1991.

[5] R. Canetti, O. Goldreich, S. Goldwasser, and S. Micali. Resettable Zero-Knowledge. In *32nd STOC*, pages 235–244 ,2000.

[6] R. Canetti, J. Kilian, E. Petrank and Alon Rosen. Black-Box Concurrent Zero-Knowledge Requires $\tilde{\Omega}(\log n)$ rounds. In IACR eprint archive, 2001.

[7] I. Damgard. Eficient Concurrent Zero-Knowledge in the Auxiliary String Model. In *EuroCrypt2000*, LNCS 1807, pages 418–430, 2000.

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

[9] C. Dwork, and A. Sahai. Concurrent Zero-Knowledge: Reducing the Need for Timing Constraints. In *Crypto98*, Springer LNCS 1462 , pages 442–457, 1998.

[10] U. Feige. Ph.D. thesis, Weizmann Institute of Science, 1990.

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

[12] O. Goldreich. Foundations of Cryptography – Fragments of a Book. Available from `http://theory.lcs.mit.edu/~oded/frag.html`.

[13] O. Goldreich and A. Kahan. How to Construct Constant-Round Zero-Knowledge Proof Systems for NP. *Jour. of Cryptology*, Vol. 9, No. 2, pages 167–189, 1996.

[14] O. Goldreich and H. Krawczyk. On the Composition of Zero-Knowledge Proof Systems. *SIAM J. Computing*, Vol. 25, No. 1, pages 169–192, 1996.

[15] O. Goldreich, S. Micali and A. Wigderson. Proofs that Yield Nothing But Their Validity or All Languages in NP Have Zero-Knowledge Proof Systems. *JACM*, Vol. 38, No. 1, pp. 691–729, 1991.

[16] O. Goldreich and Y. Oren. Definitions and Properties of Zero-Knowledge Proof Systems. *Jour. of Cryptology*, Vol. 7, No. 1, pages 1–32, 1994.

[17] S. Goldwasser, S. Micali and C. Rackoff. The Knowledge Complexity of Interactive Proof Systems. *SIAM J. Comput.*, Vol. 18, No. 1, pp. 186–208, 1989.

[18] S. Hada and T. Tanaka. On the Existence of 3-Round Zero-Knowledge Protocols. In *Crypto98*, Springer LNCS 1462, pages 408–423, 1998.

[19] J. Hastad, R. Impagliazzo, L.A. Levin and M. Luby. Construction of Pseudorandom Generator from any One-Way Function. *SIAM Jour. on Computing*, Vol. 28 (4), pages 1364–1396, 1999.

[20] J. Kilian and E. Petrank. Concurrent and Resettable Zero-Knowledge in Poly-logarithmic Rounds. In *33rd STOC*, 2001.

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

[22] M. Naor. Bit Commitment using Pseudorandomness. *Jour. of Cryptology*, Vol. 4, pages 151–158, 1991.

[23] R. Richardson and J. Kilian. On the Concurrent Composition of Zero-Knowledge Proofs. In *EuroCrypt99*, Springer LNCS 1592, pages 415–431, 1999.

[24] A. Rosen. A note on the round-complexity of Concurrent Zero-Knowledge. In *Crypto2000*, Springer LNCS 1880, pages 451–468, 2000.