Black-Box Concurrent Zero-Knowledge Requires (almost) Logarithmically Many Rounds[*]

Ran Canetti[†] Joe Kilian[‡] Erez Petrank[§] Alon Rosen[¶]

July 3, 2002

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 achieves a substantial improvement over previous lower bounds, and is the first bound 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 languages in $\mathcal{NP}$ (which is established via black-box simulation).

1 Introduction

Zero-knowledge proof systems, introduced by Goldwasser, Micali and Rackoff [21] 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 [19], who showed that every NP-statement can be proved in zero-knowledge provided that one-way functions exist [23, 27]. 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 that execute only one instance of the protocol at a time. A more realistic setting, especially in the age of the Internet, is one that allows the *concurrent* execution of zero-knowledge protocols. In the concurrent setting (see Feige [14], and more extensive treatment by Dwork, Naor and Sahai [12]), many protocols (sessions) are executed at the same time, involving many verifiers which may be talking with the same (or many) provers simultaneously (the so-called parallel composition considered in [18, 15, 17, 6, 4] is merely 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

[*]An extended abstract has appeared in the thirty third annual ACM Symposium on The Theory of Computing [8].

[†]IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA. E-mail: canetti@watson.ibm.com.

[‡]Yianilos Labs. Yianilos Labs 707 State Rd., Rt. 206, Suite 212, Princeton, NJ 08540, USA. E-mail: joe@pnylab.com.

[§]Dept. of Computer Science, Technion - Israel Institute of Technology, Haifa 32000, Israel. E-mail: erez@cs.technion.ac.il.

[¶]Dept. of Computer Science and Applied Math., Weizmann Institute of Science, Rehovot 76100, Israel. E-mail: alon@wisdom.weizmann.ac.il. Part of this work was done while visiting the IBM T.J. Watson Research Center.

the protocol. Since it seems unrealistic (and certainly undesirable) for honest provers to coordinate their actions so that zero-knowledge is preserved, we must 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*). 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 all languages in $\mathcal{NP}$, or even nontrivial languages (outside of $\mathcal{BPP}$) seems to be a challenging task. Intuition on the difficulty of this problem is given in [12], where it was argued that for a specific 4-round zero-knowledge protocol and 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 [26] who showed, building on the techniques of Goldreich and Krawczyk [18], 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 simulator is a simulator that has only black-box access to the adversarial verifier. Essentially all previously known proofs of security of zero-knowledge protocols use black-box simulators. An exception is the protocol of [22], which uses a non-standard assumption of a "non black-box" nature.) This lower bound was later improved by Rosen to seven rounds [29].

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 [28] 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 [25], who show that the Richardson-Kilian 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., $g(n) = \log\log n$).

We note that previously there was a considerable gap between the known upper and lower bounds on the round-complexity of concurrent zero-knowledge (i.e., [25, 29]): the best known protocol has $\tilde{O}(\log^2 n)$ rounds whereas the lower bound necessitates 7 rounds (via black-box simulation).[1] In particular, the question consisting of whether constant-round concurrent zero-knowledge protocols exist has been open.

[1] $f(n) = \tilde{O}(h(n))$ if there exist constants $c_1, c_2 > 0$ so that for all sufficiently large n, $f(n) \leq c_1 \cdot (\log h(n))^{c_2} \cdot h(n))$.

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 languages outside $\mathcal{BPP}$ ([25]). Our main result is stated in the following theorem.

Theorem 1.1 *Let $r : N \to N$ 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 1.1 builds on the works of Goldreich and Krawczyk [18], Kilian, Petrank and Rackoff [26], and Rosen [29]. 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 [18, 26, 29], the actual schedule of sessions, and its analysis, are new. One main idea that, together with other ideas, enables the proof of the bound is to have the verifier *abort* sessions depending on the history of the interaction. A more detailed outline, presenting both the general structure and the new ideas in the proof, appears in Section 3.

Remark: The concurrent schedule in our proof is fixed and known to everybody. As a consequence, Theorem 1.1 is actually stronger than stated. It will hold even if the simulator knows the schedule in advance (in particular, it knows the number of concurrent sessions), and even if the schedule of the messages does not change dynamically (as a function of the history of the interaction).

1.4 Conclusions and Open Problems

1.4.1 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 [12, 13]. Damgård uses a common random string [11], and Canetti et.al. use a public registry file [7].

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* [14, 15], which is preserved under concurrent composition.

[2] $f(n) = \tilde{\Omega}(h(n))$ if there exist constants $c_1, c_2 > 0$ so that for all sufficiently large n, $f(n) \geq c_1 \cdot h(n)/(\log h(n))^{c_2}$.

1.4.2 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 show 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. It is thus natural to think that a simulator that deviates from this paradigm (i.e., is non black-box, in the sense that is does not have to rewind the adversary 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.

Hada and Tanaka [22] have considered some weaker variants of zero-knowledge, and exhibited a three-round protocol for $\mathcal{NP}$ (whereas only $\mathcal{BPP}$ has three-round block-box zero-knowledge [18]). Their protocol was an example for a zero-knowledge protocol not proven secure via black-box simulation. Alas, their analysis was based in an essential way on a strong and highly non-standard hardness assumption.

In a recent breakthrough result, Barak [2] constructs a constant-round protocol for all languages in $\mathcal{NP}$ whose zero-knowledge property is proved using a *non black-box* simulator. Such a method of simulation enables him to bypass our impossiblity result (as well as [18, 26, 29]), and to perform cryptographic tasks otherwise considered inachievable. In particular, for every (predetermined) polynomial $p(\cdot)$, there exists a version of Barak's protocol that preserves its zero-knowledge property even when it is executed $p(n)$ times concurrently (where n denotes the size of the common input). As we show in our work, this task is unachievable via black-box simulation (unless $\mathcal{NP} \subseteq \mathcal{BPP}$).

1.4.3 Open problems

At first glance, it seems that Barak's protocol completely resolves the question of whether constant-round concurrent zero-knowledge protocol exist. Taking a closer look, however, one notices that the (polynomial) number of concurrent sessions relative to which the protocol should be secure is determined *before* the protocol is specified. Moreover, it turns out that the messages in the protocol are required to be longer than the number of concurrent sessions. Thus, from both a theoretical and a practical point of view, Barak's protocol is still not satisfactory. What we would like to have is a *single* protocol that preserves its zero-knowledge property even when it is executed concurrently for *any* (not predetermined) polynomial number of times. Such a property is indeed satisfied by the protocols of [28, 25] (alas these protocols are not constant-round). This leaves open the question of whether constant-round concurrent zero-knowledge protocol indeed exist for all languages in $\mathcal{NP}$.

2 Preliminaries

2.1 Probabilistic Notation

Denote by $x \overset{\text{R}}{\leftarrow} X$ the process of uniformly choosing an element x in a set X. If $B(\cdot)$ is an event depending on the choice of $x \overset{\text{R}}{\leftarrow} X$, then $\Pr_{x \leftarrow X}[B(x)]$ (alternatively, $\Pr_x[B(x)]$) denotes the probability that $B(x)$ holds when x is chosen with probability $1/|X|$. Namely,

$$\Pr_{x \leftarrow X}[B(x)] = \sum_x \frac{1}{|X|} \cdot \chi(B(x))$$

where χ is an indicator function so that $\chi(B) = 1$ if event B holds, and equals zero otherwise. This notation extends in the natural way for events $B(\cdot, \ldots, \cdot)$ that depend on k variables $x_1, x_2, \ldots, x_k$ that are uniformly chosen in k (possibly different) sets $X_1, X_2, \ldots, X_k$. That is, we denote by $\Pr_{x_1,x_2,\ldots,x_k}[B(x_1, x_2, \ldots, x_k)]$ the probability that $B(x_1, x_2, \ldots, x_k)$ holds when $x_1, x_2, \ldots, x_k$ are chosen with probability $1/(|X_1| \cdot |X_2| \cdots |X_k|)$.

2.2 Interactive proofs

We use the standard definitions of interactive proofs (interactive Turing machines) [21, 16] and arguments (a.k.a *computationally-sound* proofs) [5]. Given a pair of interactive Turing machines, P and V, we denote by $\langle P, V\rangle(x)$ the random variable representing the (local) output of V when interacting with machine P on common input x, when the random input to each machine is uniformly and independently chosen. We consider interactive proof systems in which the soundness error is negligible. The term negligible is used for denoting functions that are (asymptotically) smaller than one over any polynomial. More precisely, a function $\nu(\cdot)$ from non-negative integers to reals is called negligible if for every constant $c > 0$ and all sufficiently large n, it holds that $\nu(n) < n^{-c}$.

Definition 2.1 (Interactive Proof System) *A pair of interactive machines $\langle P, V\rangle$ is called an* interactive proof system *for a language L if machine V is polynomial-time and the following two conditions hold with respect to some negligible function $\nu(\cdot)$:*

- Completeness: *For every $x \in L$,*

$$\Pr[\langle P, V\rangle(x) = 1] \geq 1 - \nu(|x|)$$

- Soundness: *For every $x \notin L$, and every interactive machine B,*

$$\Pr[\langle B, V\rangle(x) = 1] \leq \nu(|x|)$$

Definition 2.1 can be relaxed to require only soundness error that is bounded away from $1 - \nu(|x|)$. This is so, since the soundness error can always be made negligible by sufficiently many parallel repetitions of the protocol (as such may occur anyhow in the concurrent model). However, we do not know whether this condition can be relaxed in the case of computationally sound proofs (i.e., when the soundness condition is required to hold only for machines B that are implementable by poly-size circuits). In particular, in this case parallel repetitions do not necessarily reduce the soundness error (cf. [3]).

2.3 Concurrent zero-knowledge

Let $\langle P, V\rangle$ be an interactive proof for a language L, and consider a concurrent adversary (verifier) V^* that, given input $x \in L$, interacts with an unbounded number of independent copies of P (all on common input x). The concurrent adversary V^* is allowed to interact with the various copies of P concurrently, without any restrictions over the scheduling of the messages in the different interactions with P (in particular, V^* has control over the scheduling of the messages in these interactions). The transcript of a concurrent interaction consists of the common input x, followed by the sequence of prover and verifier messages exchanged during the interaction. We denote by $\text{view}_{V^*}^{P}(x)$ a random variable describing the content of the random tape of V^* and the transcript of the concurrent interaction between P and V^* (that is, all messages that V^* sends and receives during the concurrent interactions with P, on common input x).

Remark: The actual definition of concurrent zero-knowledge requires that the concurrent adversary V^* explicitly specifies to which session the next scheduled message belongs. However, in the proof of Theorem 1.1 we consider a "weaker" concurrent adversary V^*, that is only running a fixed scheduling of sessions (and so does not determine the schedule dynamically). In particular, there will be no need to use a formalism for specifying to which session the next scheduled message belongs.

Definition 2.2 (Concurrent Zero-Knowledge) *Let $\langle P, V\rangle$ be an interactive proof system for a language L. We say that $\langle P, V\rangle$ is* concurrent zero-knowledge*, if for every polynomial-time concurrent adversary V^* there exists a probabilistic polynomial-time algorithm S_{V^*} such that the ensembles $\{\mathrm{view}_{V^*}^{P}(x)\}_{x\in L}$ and $\{S_{V^*}(x)\}_{x\in L}$ are computationally indistinguishable.*

2.4 Black-box concurrent zero-knowledge

Loosely speaking, the definition of black-box zero-knowledge requires that there exists a "universal" simulator, S, so that for every $x \in L$ and every probabilistic polynomial-time adversary V^*, the simulator S produces a distribution that is indistinguishable from $\mathrm{view}_{V^*}^{P}(x)$ while using V^* as an oracle (i.e., in a "black-box" manner). We assume concurrent adversaries V^* are modeled by poly-sized circuits (capturing non-uniform, deterministic verifiers viewed as an oracle, cf. [18, 16, 26]).

Before we proceed with the formal definition, we will have to overcome a technical difficulty arising from an inherent difference between the concurrent setting and "stand-alone" setting. In "stand-alone" zero-knowledge the length of the output of the simulator depends only on the protocol and the size of the common input x. It is thus reasonable to require that the simulator runs in time that depends only on the size of x, regardless of the running time of its black-box. However, in black-box concurrent zero-knowledge the output of the simulator is an entire schedule, and its length depends on the running time of the concurrent adversary. Therefore, if we naively require that the running time of the simulator is a fixed polynomial in the size of x, then we end up with an unsatisfiable definition. (As for any simulator S there is an adversary V^* that generates a transcript that is longer than the running time of S.)

One way to solve the above problem is to have for *each* fixed polynomial $q(\cdot)$, a simulator S_q that "only" simulates all $q(\cdot)$-sized circuits V^*. Clearly, the running time of the simulator now depends on the running time of V^* (which is an upper bound on the size of the schedule), and the above problem does not occur anymore. Another (more restrictive) way to overcome the above problem would be to consider a simulator S_q that "only" simulates all adversaries V^* which run at most $q(|x|)$ sessions during their execution (we stress that $q(\cdot)$ is chosen *after* the protocol is determined). Such simulators should run in worst-case time that is a fixed polynomial in $q(|x|)$ and in the size of the common input x. (Note that by letting S_q "know" $q(\cdot)$ in advance we actually *strengthen* the lower bound.) In the sequel we choose to adopt the latter formalization. We stress that both formalizations are general enough to include all *known* black-box zero-knowledge proofs.

Definition 2.3 (Black-Box Concurrent Zero-Knowledge) *Let $\langle P, V\rangle$ be an interactive proof system for a language L. We say that $\langle P, V\rangle$ is* black-box concurrent zero-knowledge*, if for every polynomial $q(\cdot)$, there exists a probabilistic polynomial-time*[3] *algorithm S_q, so that for every concurrent adversary circuit V^* that runs at most $q(|x|)$ concurrent sessions, $S_q(x)$ runs in time polynomial in $q(|x|)$ and $|x|$, and satisfies that the ensembles $\{\mathrm{view}_{V^*}^{P}(x)\}_{x\in L}$ and $\{S_q(x)\}_{x\in L}$ are computationally indistinguishable.*

[3]See below for a discussion on expected vs. strict probabilistic polynomial-time.

2.5 Additional conventions

Deviation gap and expected polynomial-time simulators: 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 $\text{view}_{V^*}^{P}(x)$ or to the 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 $\text{view}_{V^*}^{P}(x)$. The deviation gap of S is the deviation gap of the best polynomial time distinguisher D. In our definitions of concurrent zero-knowledge (Definitions 2.2 and 2.3) the deviation gap of the simulator is required to be negligible in $|x|$.

For our lower bound, we allow simulators that run in strict (worst case) polynomial time, and have deviation gap at most 1/4. As for expected polynomial time simulators, one can use a standard argument to show that any simulator running in expected polynomial time, and having deviation gap at most 1/8 can be transformed into a simulator that runs in strict (worst case) polynomial time, and has deviation gap at most 1/4. In particular, our lower bound (on simulators that run in strict polynomial time, and have deviation gap at most 1/4) extends to a lower bound on simulators running in expected polynomial time (and have deviation gap as large as 1/8).

Query 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] Clearly, any protocol in which $2k$ messages are exchanged can be modified to fit this form (by adding at most two messages).

We impose the following technical restrictions on the simulator (but claim that each of these restrictions can be easily satisfied): As in (cf. [18]), 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). We assume that the simulator never repeats the same query twice. In addition, 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 of Theorem 1.1. The actual proof will be given in Sections 4 and 5. To facilitate reading, we partition the outline into two parts: The first part reviews the general framework. (This part mainly follows previous works, namely [17, 26, 29].) The second part concentrates on the actual schedule and the specifics of our lower bound argument.

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

[5] 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 anyway).

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". To decide on an input x, run S with a cheating verifier V_h that was chosen at random from the constructed family, and 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 in this case S succeeds only with negligible probability. See details below.

3.1.1 Session-prefixes and useful session-prefixes

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 that ends at the point where some new session starts (including the first verifier message in that session). (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 a is called useful in a run of S if:

1. It was accepted (i.e., V_h sent an `ACCEPT` message for session-prefix a).
2. V_h has sent exactly $k+1$ messages for session-prefix a.

Loosely speaking, Condition 2 implies that S did not rewind the relevant session-prefix, where rewind 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. By rewinding session-prefix a, the simulator is able to obtain more than $k+1$ verifier messages for session-prefix a. This is contrast to an actual execution of the protocol $\langle P, V \rangle$ in which V sends exactly $k+1$ messages.

3.1.2 The construction of the cheating prover

Using the above terms, we sketch the construction of the cheating prover P^*. It first randomly chooses a function $h \overset{\mathrm{R}}{\leftarrow} H$ and an index (of a session-prefix) i. It then emulates 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 as the messages that P^* sends to the actual verifier it interacts with; similarly, it uses the messages received from the actual verifier V instead of V_h's messages in the i^{th} session-prefix. The strategy of the cheating prover is depicted in Figure 1 below.

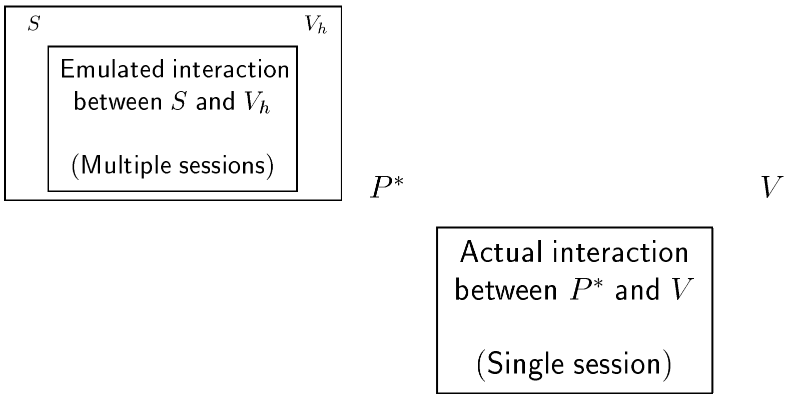

S
V_h
Emulated interaction between S and V_h
(Multiple sessions)
P^*
V
Actual interaction between P^* and V
(Single session)

Figure 1: Describes the strategy of the cheating prover P^*. The box on the left hand side represents the (multiple session) emulation of the interaction between S and V_h (executed "internally" by P^*). The box on the right hand side represents the actual execution of a single session between P^* and V.

3.1.3 The success probability of the cheating prover

We next claim that if the session-prefix chosen by P^* is useful, then $\langle P^*, V\rangle(x)$ accepts. The key point is that whenever P^* chooses an useful session-prefix, the following two conditions (corresponding to the two conditions in the definition of a useful session-prefix) are satisfied:

1. The session corresponding to the i^{th} session-prefix is accepted by V_h (and so by V).
2. P^* manages to reach the end of the $\langle P^*, V\rangle$ interaction without "getting into trouble".[6]

Loosely speaking Item (1) is implied by Condition (1) in the definition of a useful session-prefix. As for Item (2), this just follows from the fact that S does not rewind the i^{th} session-prefix (as implied by Condition (2) in the definition of a useful session-prefix). In particular, P^* (playing the role of V_h) will not have to send the j^{th} verifier message with the i^{th} session-prefix more than once to S (since the number of messages sent by V_h for that session-prefix is precisely $k+1$).

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.

3.2 The schedule and additional ideas

Using the above framework, the crux of the lower bound is to come up with a schedule and V_h's that allow demonstrating that whenever S succeeds, the conversation between S and V_h contains a useful session-prefix (as we have argued above, it is in fact sufficient that the conversation between S and V_h contains a useful session-prefix with non-negligible probability). This is done next.

[6]The problem is that P^* does not know V's random coins, and so it cannot compute the verifier's answers by himself. Thus, whenever P^* is required to send the j^{th} verifier message in the protocol more than once to S it might get into trouble (since it gets the j^{th} verifier message only once from V).

3.2.1 The 2-round case

Our starting point is the schedule used in [26] to demonstrate the impossibility of black-box concurrent zero-knowledge with protocols in which 4 messages are exchanged (i.e., $\mathtt{v}_1, \mathtt{p}_1, \mathtt{v}_2, \mathtt{p}_2$). The schedule is recursive and consists of n concurrent sessions (n is polynomially related to the security parameter). Given parameter $m \leq n$, the scheduling on m sessions (denoted $\mathcal{R}_m$) proceeds as follows (see Figure 2 for a graphical description):

1. If $m = 1$, the relevant session exchanges all of its messages (i.e., $\mathtt{v}_1, \mathtt{p}_1, \mathtt{v}_2, \mathtt{p}_2$).
2. Otherwise (i.e., if $m > 1$):

   Message exchange: The first session (out of m sessions) exchanges 2 messages (i.e., $\mathtt{v}_1, \mathtt{p}_1$);

   Recursive call: The schedule is applied recursively on the remaining $m - 1$ sessions;

   Message exchange: The first session (out of m sessions) exchanges 2 messages (i.e., $\mathtt{v}_2, \mathtt{p}_2$).

At the end of each session V_h continues in the interaction if and only if the transcript of the session that has just terminated would have been accepted by the prescribed verifier V. This means that in order to proceed beyond the ending point of the ℓ^{th} session, the simulator must make the honest verifier accept the s^{th} session for all $s > \ell$.

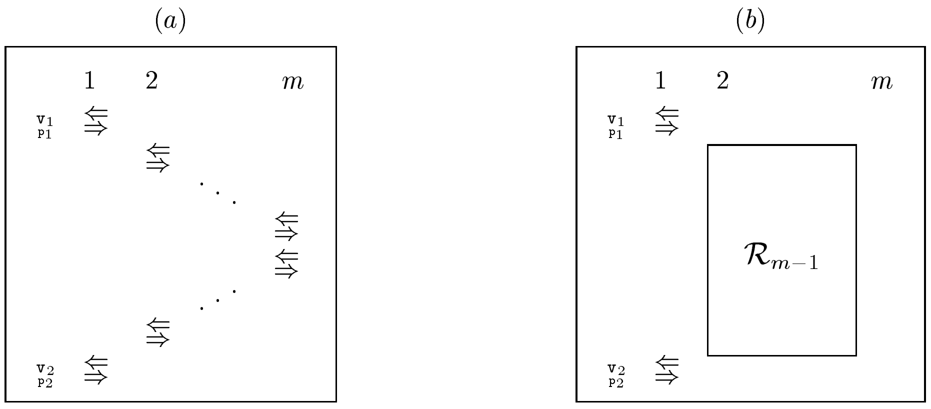


Figure 2: The "telescopic" schedule used by [26] to demonstrate impossibility of black-box concurrent zero-knowledge in 2 rounds. Columns correspond to n individual sessions and rows correspond to the time progression. (a) Depicts the schedule explicitly. (b) Depicts the schedule in a recursive manner ($\mathcal{R}_m$ denotes the recursive schedule for m sessions).

Suppose now that S suceeds in simulating the above V_h but the conversation between S and V_h does not contain a useful session-prefix. Since V_h proceeds beyond the ending point of a session only if this session is accepted, then the only reason for which the corresponding session-prefix can be non-useful is because S has rewound that session-prefix. Put in other words, a session-prefix becomes non-useful if and only if S resends the first prover message in the protocol (i.e., $\mathtt{p}_1$).[7] This shuld cause V_h to resend the second verifier message (i.e., $\mathtt{v}_2$), thus violating Condition (2) in the definition of a useful session-prefix (see Section 3.1.1).

[7]Notice that the first prover message in the protocol (i.e., $\mathtt{p}_1$) is the only place in which rewinding the interaction may cause a session-prefix to be non-useful. The reason for this is that the first verifier message in the protocol (i.e., $\mathtt{v}_1$) is part of the session-prefix. Rewinding past this message (i.e., $\mathtt{v}_1$) would modify the session-prefix itself.

The key observation is that whenever the first prover message in the ℓ^{th} session is modified, then so is the session-prefix of the s^{th} session for *all* $s > \ell$. Thus, whenever S resends the first prover message in the ℓ^{th} session, it must do so also in the s^{th} session for all $s > \ell$ (since otherwise the "fresh" session-prefix of the s^{th} session, that is induced by resending the above message, will be useful). But this means that the work $W(m)$, invested in the simulation of a schedule with m levels, must satisfy $W(m) \geq 2 \cdot W(m-1)$ for all m. Thus, either the conversation between V_h and S contains a useful session-prefix (in which case we are done), or the simulation requires exponential-time (since $W(m) \geq 2 \cdot W(m-1)$ solves to $W(n) \geq 2^{n-1}$).

3.2.2 The k-round case – first attempt

A first attempt to generalize this schedule to the case of k rounds may proceed as follows. Given parameter $m \leq n$ (denoting the number of sessions in $\mathcal{R}_m$) do:

1. If $m = 1$, the relevant session exchanges all of its messages (i.e., $\mathtt{v}_1, \mathtt{p}_1, \ldots, \mathtt{v}_{k+1}, \mathtt{p}_{k+1}$).

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

   Message exchange: The first session (out of m) exchanges two messages (i.e., $\mathtt{v}_j, \mathtt{p}_j$);

   Recursive call: If $j < k+1$, the scheduling is applied recursively on $\lfloor \frac{m-1}{k} \rfloor$ new sessions;
   (This is done using the next $\lfloor \frac{m-n}{k} \rfloor$ remaining sessions out of $1, \ldots, m$.)

As before, at the end of each session V_h continues in the interaction if and only if the transcript of the session that has just terminated would have been accepted by the prescribed verifier V. The schedule is depicted in Figure 3.

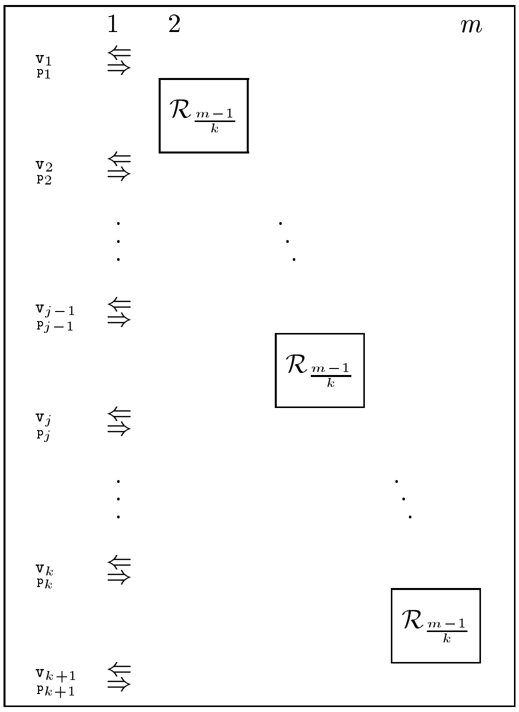


Figure 3: First attempt to generalize the recursive schedule ($\mathcal{R}_m$ with m sessions) for k-round protocols. Columns correspond to m individual sessions and rows correspond to the time progression.

The crucial problem of the above schedule is that one can come up with a k-round protocol and a corresponding simulator that manages to succesfully simulate V_h *and* cause all session-prefixes in its conversation with V_h to be non-useful. Specifically, there exist protocols (cf. [28]) in which the simulator is required to successfully rewind an honestly behaving verifier exactly once for every session. Whereas in the case of 2-rounds this could have had devastating consequences (since, in the case of the previous schedule, it would have implied $W(m) \geq (k+1) \cdot W(m-1) = 2 \cdot W(m-1)$, which solves to $W(n) \geq 2^{n-1}$), in the general case (i.e., when $k+1 > 2$) any rewinding of the schedule that we have suggested would have forced the simulator to re-invest simulation "work" only for $\frac{m-1}{k}$ sessions. Note that such a simulator satisfies $W(m) = (k+1) \cdot W(\frac{m-1}{k})$, which solves to $k^{O(\log_k n)} = n^{O(1)}$. In particular, by investing polynomial amount of work the simulator is able to make all session-prefixes not useful while succesfully simulating all sessions.

3.2.3 The k-round case – second attempt

One method to circumvent this difficulty was used in [29]. However, that method extends the lower bound only up to 3 rounds (more precisely, 7 messages). Here we use a different method. What we do is let the cheating verifier abort (i.e., refuse to answer) every message in the schedule with some predetermined probability (independently of other messages). To do this, 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 $V_{g,h}$ is run as usual.

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 rewinds the session-prefix a *and in addition* g returns 1 in at least two of the continuations of a. This means that S is expected to rewind session-prefix a several times before it stops being useful. Since each rewinding 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 rewind 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:

$$W(m) \geq \Omega(1/p) \cdot W\left(\tfrac{m-1}{k}\right) \qquad (1)$$

This would mean that the work required to successfully simulate n sessions *and* make all session-prefixes non-useful is at least $\Omega(p^{-\log_k n})$. Consequently, when the expression $p^{-\log_k n}$ is super-polynomial there is hope that the conversation between S and V_h contains a useful session-prefix with non-negligible probability.

3.2.4 The k-round case – final version

However, demonstrating Eq. (1) brings up the following difficulty. Once the verifier starts aborting sessions, the probability that a session is ever completed may become too small. As a consequence, it is not clear anymore that the simulator must invest simulation "work" for all sessions in the schedule. It may very well be the case that the simulator will go about the simulation task while "avoiding" part of the simulation "work" in some recursive invocations (as some of these invocations may be aborted anyway during the simulation). In other words, there is no guarantee that the recursive "work" invested by the simulator behaves like Eq. (1).

To overcome this problem, we replace each session in the above schedule (for k rounds) with a "block" of, say, n sessions (see Figure 4 in Page 15). We now have n^2 sessions in a schedule. (This choice of parameters is arbitrary, and 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 above modification guarantees us that, by a careful setting of the parameters, the simulator's recursive "work" must satisfy Eq. (1), at least with overwhelming probability.

3.2.5 Setting the value of p

Once Eq. (1) is established, it remains to set the value of p. 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) in the definition of a useful session-prefix (Section 3.1.1) 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 (and furthermore that $p^k \cdot n \gg 1$).

In the proof we set $p = n^{-1/2k}$. This will guarantee that a session is completed with probability $p^k = n^{-1/2}$ (thus Condition (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 Condition (2) in the definition of a useful session-prefix (Section 3.1.1) will be satisfied for $k = o(\log n / \log\log n)$.

3.3 The actual analysis

Demonstrating that there exist many accepted session-prefixes is straightforward. Demonstrating that one of these session-prefixes is useful 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 (combinatorial) property that holds for *any* successful run of the simulator, unless the simulator runs in super-polynomial time (Lemma 5.9). Essentially, this property states that there exists a block of sessions such that none of the session-prefixes in this block were rewound too many times. Using this property, we show (in Lemma 5.7) 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 The Actual Proof (of Theorem 1.1)

Assuming towards the contradiction that a black-box simulator, denoted S, contradicting Theorem 1.1 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. This is done next.

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.[8] The scheduling is defined recursively, where the scheduling of $m \leq n^2$ sessions (denoted $\mathcal{R}_m$) proceeds as follows:[9]

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 on $\lfloor \frac{m-n}{k} \rfloor$ 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 4. We stress 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 of Step 2, $\lfloor \frac{m-n}{k} \rfloor$ 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 (in the first part of Step 2) 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 equals 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 at most n) is at least $\lfloor \frac{(k-1)n+1}{k} \rfloor$.

Identifying sessions according to their recursive 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

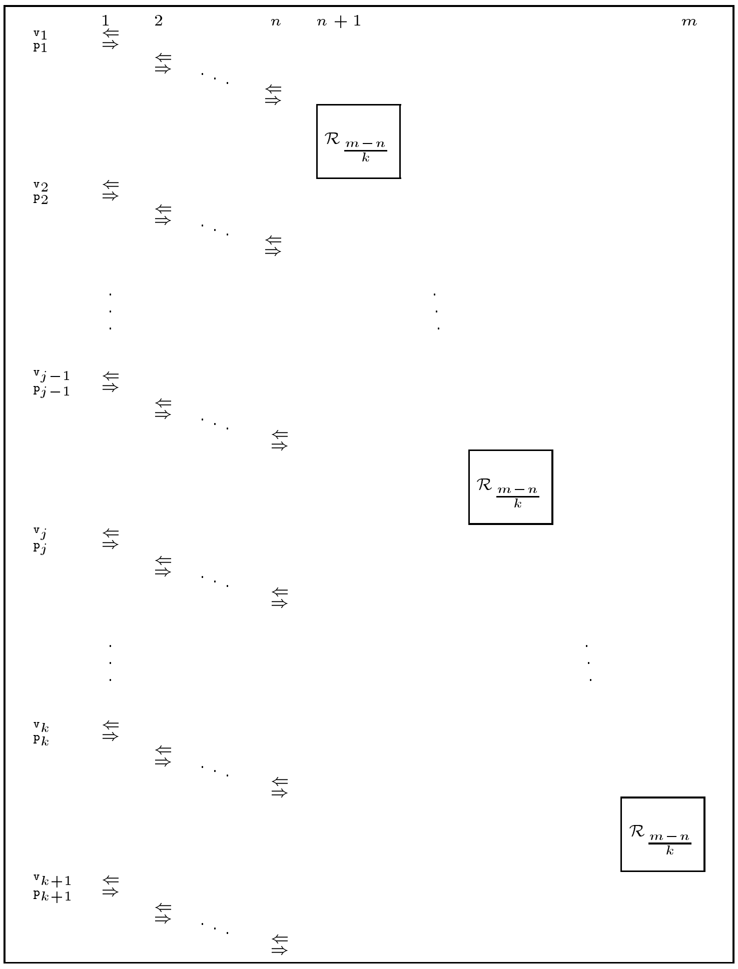
[8] Recall that each session consists of $2k+2$ messages, where $k \stackrel{\text{def}}{=} k(n) = o(\log n / \log\log n)$.

[9] In general, we may want to define a recursive scheduling for sessions $i_1, \ldots, i_m$ and denote it by $\mathcal{R}_{i_1, \ldots, i_m}$ (see Section A in the Appendix for a more formal description of the schedule). We choose to simplify the exposition by renaming these sessions as $1, \ldots, m$ and denote the scheduling by $\mathcal{R}_m$.

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 $\mathsf{v}_j^{(\ell,i)}$ (resp. $\mathsf{p}_j^{(\ell,i)}$), and stick to the notation $\mathsf{v}_j^{(i)}$ (resp. $\mathsf{p}_j^{(i)}$). Note that 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 4.1 (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* $\pi_{\mathrm{sn}}(\overline{q}) = (\ell, i) \in \{1, \ldots, n\} \times \{1, \ldots, n\}$ *the session to which the next verifier message belongs, and by* $\pi_{\mathrm{msg}}(\overline{q}) = j \in \{1, \ldots, k+1\}$ *its index within the verifier's messages in this session.*

We stress 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).



1
2
n
n + 1
m
v1
p1
v2
p2
vj−1
pj−1
vj
pj
vk
pk
vk+1
pk+1
$\mathcal{R}_{\frac{m-n}{k}}$
$\mathcal{R}_{\frac{m-n}{k}}$
$\mathcal{R}_{\frac{m-n}{k}}$

Figure 4: The recursive schedule $\mathcal{R}_m$ for m sessions. Columns correspond to m individual sessions and rows correspond to the time progression.

4.1.2 Towards constructing an adversarial verifier

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 (cf. [20]), 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 [18, 26, 29], we let the source of randomness be a hash function with sufficiently high independence (which is "hard-wired" into the verifier's description), and consider the execution of a black-box simulator that is given access to such a random verifier. (Recall that the simulator's queries correspond to partial execution transcripts and thus contain the whole history of the interaction so far.)

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 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 $(\ell, i) = \pi_{\text{sn}}(\overline{q})$ apply the hash function on 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 current 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 (since the protocol's specification may require that the verifier's randomness is completely determined before the first verifier message is sent). For technical reasons, 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 $(\ell, i) = \pi_{\text{sn}}(\overline{q})$ we feed the hash function with the prefix of query $\overline{q}$ that ends just before the first message in block number ℓ (this prefix is called the block-prefix of query $\overline{q}$ and is defined below). In order to achieve independence with 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 us 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 (that is, once the first verifier message is sent). (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.

Definition 4.2 (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.*[10] (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 on 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 occurred. 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. (As before, the hash function is also fed with the value of i (in order to achieve independence from other sessions in the block).) This prefix is called the iteration-prefix of query $\overline{q}$ and is defined next (see Figure 5 for a graphical description of the block-prefix and iteration-prefix of a query).

Definition 4.3 (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 iteration-prefix of $\overline{q} = (b_1, a_1, \ldots, b_t, a_t)$ if a_δ is of the form $\mathsf{p}_{j-1}^{(n)}$ (where $\mathsf{p}_{j-1}^{(n)}$ denotes the $(j-1)^{\rm st}$ prover message in the $n^{\rm th}$ 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 the iteration-prefix is defined only for $\pi_{\rm msg}(\overline{q}) > 1$.)

We stress that two queries $\overline{q}_1, \overline{q}_2$ may have the same iteration-prefix even if they do not correspond to the same session. This could happen whenever $bp(\overline{q}_1) = bp(\overline{q}_2)$ and $\pi_{\rm msg}(\overline{q}_1) = \pi_{\rm msg}(\overline{q}_2)$ (which is possible even if $\pi_{\rm sn}(\overline{q}_1) \neq \pi_{\rm sn}(\overline{q}_2)$).

Motivating Definitions 4.2 and 4.3: The choices made in Definitions 4.2 and 4.3 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 so 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]

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

[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.

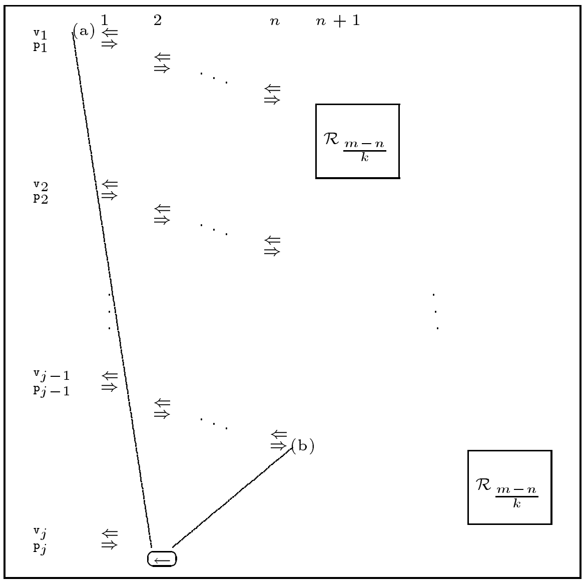


Figure 5: Determining the prefixes of query $\overline{q}$ (in this example, query $\overline{q}$ ends with a $\mathsf{p}_j^{(1)}$ message and is to be answered by $\mathsf{v}_j^{(2)}$, represented by the marked arrow): (a) indicates the block-prefix of $\overline{q}$ (i.e., messages up to this point are used by $V_{g,h}$ to determine the randomness to be used for computing message $\mathsf{v}_j^{(2)}$). (b) indicates the iteration-prefix of $\overline{q}$ (i.e., messages up to this point are used by $V_{g,h}$ to determine whether or not message $\mathsf{v}_j^{(2)}$ will be set to ABORT).

To see that Definitions 4.2 and 4.3 indeed lead to the fulfillment of the above requirements, we consider the following 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 $(\ell, i) = \pi_{\text{sn}}(\overline{q})$ (or at most, to messages that belong to other sessions in block number ℓ). However, 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 on 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 himself doing the simulation work for these lower-level 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 $(\ell, i) = \pi_{\text{sn}}(\overline{q})$ itself. One possible consequence of changes in iteration-prefix $ip(\overline{q})$ is that they may also effect 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 completely modified, 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 $\pi_{\text{sn}}(\overline{q}) = (\ell, i)$, and $\pi_{\text{msg}}(\overline{q}) = j$). 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.

4.1.3 The actual 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 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 $\Pr_{g \leftarrow G}[g(\alpha) = 1] = n^{-1/2k}$. 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, so that for every α we have $\Pr_{h \leftarrow H}[h(\alpha) = 1] = 2^{-\rho_V(n)}$ (where $\rho_V(n)$ is the number of random bits used by an honest verifier V on an input $x \in \{0,1\}^n$).[12] 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 (i.e., corresponds to a possible execution prefix), and halts with a special ERROR message if the query is not legal.[13]

2. Next, $V_{g,h}$ determines the block-prefix, $bp(\overline{q}) = (b_1, a_1, \ldots, b_\gamma, a_\gamma)$, 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 next message is the j^{th} verifier message in the i^{th} session of block ℓ. 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., $\mathtt{v}_1^{(i)}$).

[12]We stress that functions in such families can be described by strings of polynomial length in a way that enables polynomial time evaluation (cf. [24, 9, 10, 1]).

[13]In particular, $V_{g,h}$ checks whether the query is of the prescribed format (as described in Section 2.5, 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 $V_{g,h}(\overline{q}')$.)

4. If $j > 1$, then $V_{g,h}$ determines $b_{i,j} = g(i, ip(\overline{q}))$ (i.e., 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. Specifically, if the j^{th} verifier message in session i is `ABORT` then all subsequent prover and verifier messages in that session will also equal `ABORT`.

On the arguments to g and h: The hash function h, which determines the random input for V in a session, is applied both on i (the identifier of the relevant session within the current block) and on 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 on i and on 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 i and the block-prefix are fixed (and is uneffected by mesages that belong to that session), the input of g *varies* depending on previous messages sent in that session. 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 an *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[14] 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 from feeding

[14]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.

$V_{g,h}$ 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 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 $\frac{n^{-1/2}}{2}$. 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 $n \cdot \frac{n^{-1/2}}{2} = \frac{n^{1/2}}{2}$, and with overwhelming high probability at least $\frac{n^{1/2}}{4}$ sessions are accepted by $V_{g,h}$.

A slight modification of the verifier strategy: To facilitate the analysis, we slightly modify the verifier strategy $V_{g,h}$ 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 $\frac{n^{1/2}}{4}$ 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 $\overline{q} = (b_1, a_1, \ldots, b_t, a_t)$ ends with the last prover message of the nth 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 $\frac{n^{1/2}}{4}$ 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 $\frac{n^{1/2}}{2}$, the probability that the block contains less than $\frac{n^{1/2}}{4}$ 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 $\frac{n^{1/2}}{4}$ 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 at least $\frac{n^{1/2}}{4}$ 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 the corresponding condition can be easily checked by the simulator (which can easily produce this special message by itself),[15] 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.

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 $g \overset{\mathrm{R}}{\leftarrow} G$ and $h \overset{\mathrm{R}}{\leftarrow} H$.
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 $V_{g,h}$, where g and h are as determined in Step 1.
3. Accept if and only if S outputs a legal transcript (as determined by Steps 1 and 1' of $V_{g,h}$).[16]

By our hypothesis, the above procedure runs in probabilistic polynomial-time. We next analyze its performance.

Lemma 4.4 (performance on YES-instances): *For all but finitely many $x \in L$, the above procedure accepts x with probability at least* $2/3$.

Proof Sketch: Let $x \in L$, $g \overset{\mathrm{R}}{\leftarrow} G$, $h \overset{\mathrm{R}}{\leftarrow} H$, and consider the honest prover P. We show below that, except for negligible probability (where the probability is taken over the random choices of g, h, and P's coin tosses), when $V_{g,h}$ interacts with P, all recursive blocks in the resulting transcript contain the accepting conversations of at least $\frac{n^{1/2}}{4}$ main sessions. Since for *every* g and h the simulator $S^{V_{g,h}}(x)$ must generate a transcript whose deviation gap from $\langle P, V_{g,h}\rangle(x)$ is at most $1/4$, it follows that $S^{V_{g,h}}(x)$ has deviation gap at most $1/4$ from $\langle P, V_{g,h}\rangle(x)$ also when $g \overset{\mathrm{R}}{\leftarrow} G$ and $h \overset{\mathrm{R}}{\leftarrow} H$. Consequently, when S is run by the decision procedure for L, the transcript $S^{V_{g,h}}(x)$ will not be legal with probability at most $1/3$. Details follow.

Let τ denote the random variable describing the transcript of the interaction between the honest prover P and $V_{g,h}$, where the probability is taken over the choices of g, h, and P. Let $s \in \{1,\ldots,n^2\}$. We first calculate the probability that the s^{th} session in τ is completed and accepted (i.e., $V_{g,h}$ sends the message $\mathtt{v}_{k+1}^{(s)} = \mathtt{ACCEPT}$), conditioned on the event that $V_{g,h}$ did not abandon the interaction beforehand (i.e., $V_{g,h}$ did not send the DEVIATION message before).[17] For uniformly selected $g \overset{\mathrm{R}}{\leftarrow} G$, the probability that $V_{g,h}$ does not abort the session in each of the k rounds, given that it has not

[15]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.

[16]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 $(b_1, a_1, \ldots, b_T, a_T)$, cheking the legality of the transcript in 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 either produces execution transcripts in which every recursive block contains at least $n^{1/2}/4$ sessions that were accepted by $V_{g,h}$, or it does not produce any output at all.

[17]Note that, since we are dealing with the honest prover P, there is no need to consider the ERROR message at all (since in an interaction with the honest prover P, the adversary verifier $V_{g,h}$ will never output ERROR anyway).

already aborted, is $n^{-1/2k}$. Thus, conditioned on the event that $V_{g,h}$ did not output DEVIATION beforehand, the session is completed (without being aborted) with probability $(n^{-1/2k})^k = n^{-1/2}$.

The key observation is that if h is uniformly chosen from H then, conditioned on the event that $V_{g,h}$ did not output DEVIATION beforehand and the current session is not aborted, the conversation between $V_{g,h}$ and P is distributed identically to the conversation between the honest verifier V and P on input x. By the completeness requirement for zero-knowledge protocols, we have that V accepts in such an interaction with probability at least $1/2$ (this probability is actually higher, but $1/2$ is more than enough for our purposes). Consequently, for uniformly selected g and h, conditioned on the event that $V_{g,h}$ did not output DEVIATION beforehand, the probability that a session is accepted by $V_{g,h}$ is at least $\frac{n^{-1/2}}{2}$.

We calculate the probability that τ contains a block such that less than $\frac{n^{1/2}}{4}$ of its sessions are accepted. Say that a block B in a transcript has been completed if all the messages of sessions in B have been sent during the interaction. Say that B is admissible if the number of accepted sessions that belong to block B in the transcript is at least $\frac{n^{1/2}}{4}$. Enumerating blocks in the order in which they are completed (that is, when we refer to the ℓ^{th} block in τ, we mean the ℓ^{th} block that is completed in τ), we denote by γ_ℓ the event that all the blocks up to and including the ℓ^{th} block are admissible in τ.

For $i \in \{1, \ldots, n\}$ define a boolean indicator α_i^ℓ to be 1 if and only if the i^{th} session in the ℓ^{th} block is accepted by $V_{g,h}$. We have seen that, conditioned on the event $\gamma_{\ell-1}$, each α_i^ℓ is 1 w.p. at least $\frac{n^{-1/2}}{2}$. As a consequence, for every ℓ, the expectation of $\sum_{i=1}^n \alpha_i^\ell$ (i.e., the number of accepted main sessions in block number ℓ) is at least $\frac{n^{1/2}}{2}$. Since, conditioned on $\gamma_{\ell-1}$, the α_i^ℓ's are independent of each other, we can apply the Chernoff bound, and infer that $\Pr[\gamma_\ell | \gamma_{\ell-1}] > 1 - e^{-\Omega(n^{1/2})}$. Furthermore, since no session belongs to more than one block, we have: $\Pr[\gamma_\ell] \geq \Pr[\gamma_l | \gamma_{\ell-1}] \cdot \Pr[\gamma_{l-1}]$. It follows (by induction on the number of completed blocks in a transcript), that all blocks in τ are admissible with probability at least $(1 - e^{-\Omega(n^{1/2})})^n > 1 - n \cdot e^{-\Omega(n^{1/2})}$. The lemma follows. ■

Lemma 4.5 (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 positive 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 soundness (and even computational-soundness) of the proof system.

5 Proof of Lemma 4.5 (performance on NO-instances)

Let us fix an $x \in \{0,1\}^n \setminus L$ as above.[18] Denote by $\texttt{AC} = \texttt{AC}_x$ the set of triplets (σ, g, h) so that on input x, internal coins σ and oracle access to $V_{g,h}$, the simulator outputs a legal transcript (which we denote by $S_\sigma^{V_{g,h}}(x)$). Recall that our contradiction assumption is that $\Pr_{\sigma,g,h}[(\sigma, g, h) \in \texttt{AC}] > 1/p(n)$, for some fixed positive polynomial $p(\cdot)$. Before proceeding with the proof of Lemma 4.5, we formalize what we mean by referring to the "execution of the simulator".

[18] Actually, we need to consider infinitely many such x's.

Definition 5.1 (Execution of simulator) *Let $x, \sigma \in \{0,1\}^*$, $g \in G$ and $h \in H$. The* execution *of simulator S, denoted $\text{EXEC}_x(\sigma, g, h)$, is the sequence of queries made by S, given input x, random coins σ, and oracle access to $V_{g,h}(x)$.*

Since the simulator has the ability to "rewind" the verifier $V_{g,h}$ and explore $V_{g,h}$'s output on various execution prefixes (i.e., queries) of the same length, the number of distinct block-prefixes that appear in $\text{EXEC}_x(\sigma, g, h)$ may be strictly larger than n (recall that the schedule consists of n invocations to recursive blocks, and that in an interaction between the honest prover P and $V_{g,h}$ there is a one-to-one correspondence between recursive blocks and block-prefixes). As a consequence, the ℓ^{th} distinct block-prefix appearing in $\text{EXEC}_x(\sigma, g, h)$ does not necessarily correspond to the ℓ^{th} recursive block in the schedule. Nevertheless, given $\text{EXEC}_x(\sigma, g, h)$ and ℓ, one can easily determine for the ℓ^{th} distinct block-prefix in the execution of the simulator the index of its corresponding block in the schedule (say, by extracting the ℓ^{th} distinct block-prefix in $\text{EXEC}_x(\sigma, g, h)$, and then analyzing its length).

In the sequel, given a specific block-prefix $\overline{bp}$, we let $\ell^{(\overline{bp})} \in \{1, \ldots, n\}$ denote the index of its corresponding block in the schedule (as determined by $\overline{bp}$'s length). Note that two different block-prefixes $\overline{bp}_1$ and $\overline{bp}_2$ in $\text{EXEC}_x(\sigma, g, h)$ may satisfy $\ell^{(\overline{bp}_1)} = \ell^{(\overline{bp}_2)}$ (as they may correspond to two different instances of the same recursive block). In particular, session $(\ell^{(\overline{bp}_1)}, i)$ may have more than a single occurrence during the execution of the simulator (whereas in an interaction of the honest prover P with $V_{g,h}$ each session index will occur exactly once). This means that whenever we refer to an instance of session (ℓ, i) in the simulation, we will also have to explicitly specify to which block-prefix this instance corresponds.

In order to avoid cumbersome statements, we will abuse the notation $\ell^{(\overline{bp})}$ and also use it in order to specify to which instance the recursive block $\ell^{(\overline{bp})}$ corresponds. That is, whenever we refer to recursive block number $\ell^{(\overline{bp})}$ we will actually mean: "the specific instance of recursive block number ℓ $(= \ell^{(\overline{bp})})$ that corresponds to block-prefix $\overline{bp}$ in $\text{EXEC}_x(\sigma, g, h)$". Viewed this way, for $\ell^{(\overline{bp}_1)} = \ell^{(\overline{bp}_2)}$, sessions $(\ell^{(\overline{bp}_1)}, i)$ and $(\ell^{(\overline{bp}_2)}, i)$ actually correspond to two different instances of the same session in the schedule.

5.1 The cheating prover

The cheating prover (denoted P^*) starts by uniformly selecting a triplet (σ, g, h) while hoping that $(\sigma, g, h) \in \texttt{AC}$. It next selects uniformly a pair $(\xi, \eta) \in \{1, \ldots, t_S(n)\} \times \{1, \ldots, n\}$, where the simulator's running time, $t_S(n)$, acts as a bound on the number of (different block-prefixes induced by the) queries made by S on input $x \in \{0,1\}^n$. The prover next emulates an execution of $S_\sigma^{V_{g,h^{(r)}}}(x)$ (where $h^{(r)}$, which is essentially equivalent to h, will be defined below), while interacting with $V(x, r)$ (that is, the honest verifier, running on input x and using coins r). The prover handles the simulator's queries as well as the communication with the verifier as follows: Suppose that the simulator makes query $\overline{q} = (b_1, a_1, \ldots, b_t, a_t)$, where the a's are prover messages.

1. Operating as $V_{g,h}$, the cheating prover determines the block-prefix $bp(\overline{q}) = (b_1, a_1, \ldots, b_\gamma, a_\gamma)$. It also determines $(\ell, i) = \pi_{\text{sn}}(\overline{q})$, $j = \pi_{\text{msg}}(\overline{q})$, the iteration-prefix $ip(\overline{q}) = (b_1, a_1, \ldots, b_\delta, \mathtt{p}_{j-1}^{(n)})$, and the j–1 prover messages $\mathtt{p}_1^{(i)}, \ldots, \mathtt{p}_{j-1}^{(i)}$ appearing in the query $\overline{q}$ (as done by $V_{g,h}$ in Step 2). (Note that by the modification of S there is no need to perform Steps 1 and 1' of $V_{g,h}$.)

2. If $j = 1$, the cheating prover answers the simulator with the verifier's fixed initiation message for session i (as done by $V_{g,h}$ in Step 3).

3. If $j > 1$, the cheating prover determines $b_{i,j} = g(i, ip(\overline{q}))$ (as done by $V_{g,h}$ in Step 4).
4. If $bp(\overline{q})$ is the ξ^{th} distinct block-prefix resulting from the simulator's queries so far and if, in addition, i equals η, then the cheating prover operates as follows:
   (a) If $b_{i,j} = 0$, then the cheating prover answers the simulator with ABORT.

   Motivating discussion for substeps b and c: The cheating prover has now reached a point in the schedule in which it is supposed to feed the simulator with $\mathtt{v}_j^{(i)}$. To do so, it first forwards $\mathtt{p}_{j-1}^{(i)}$ to the honest verifier $V(x,r)$, and only then feeds the simulator with the verifier's answer $\mathtt{v}_j^{(i)}$ (as if it were the answer given by $V_{g,h^{(r)}}$). We stress the following two points: (1) The cheating prover cannot forward more than one $\mathtt{p}_{j-1}^{(i)}$ message to V (since P^* and V engage in an actual execution of the protocol $\langle P, V\rangle$). (2) The cheating prover will wait and forward $\mathtt{p}_{j-1}^{(i)}$ to the verifier only when $\mathtt{v}_j^{(i)}$ is the next scheduled message.

   (b) If $b_{i,j} = 1$, and the cheating prover has only sent $j-2$ messages to the actual verifier, the cheating-prover forwards $\mathtt{p}_{j-1}^{(i)}$ to the verifier, and feeds the simulator with the verifier's response (i.e., which is of the form $\mathtt{v}_j^{(i)}$).[19]

   (We comment that by our conventions regarding the simulator, it cannot be the case that the cheating prover has sent less than $j-2$ prover messages to the actual verifier. The prefixes of the current query dictate $j-2$ sequences of prover messages with distinct lengths, so that none of these sequences was answered with ABORT. In particular, the last message of each one of these sequences was already forwarded to the verifier.)

   (c) If $b_{i,j} = 1$, and the cheating prover has already sent $j-1$ messages (or more) to the actual verifier then it retrieves the $(j-1)^{\text{st}}$ answer it has received and feeds it to the simulator.

   (We comment that this makes sense provided that the simulator never makes two queries with the same block-prefix and the same number of prover messages, but with a different sequence of such messages. However, for $j \geq 2$ it may be the case that a previous query regarding the same block-prefix had a different $\mathtt{p}_{j-1}^{(i)}$ message. This is the case in which the cheating prover may fail to conduct Step 4c (see further discussion below).)
5. If either $bp(\overline{q})$ is NOT the ξ^{th} distinct block-prefix resulting from the queries so far, or if i is NOT equal to η, the prover emulates $V_{g,h}$ in the obvious manner (i.e., as in Step 4 of $V_{g,h}$):
   (a) If $b_{i,j} = 0$, then the cheating prover answers the simulator with ABORT.

   (b) If $b_{i,j} = 1$, then the cheating prover determines $r_i = h(i, bp(\overline{q}))$, and then answers the simulator with $V(x, r_i; \mathtt{p}_1^{(i)}, \ldots, \mathtt{p}_{j-1}^{(i)})$, where all notations are as above.

On the efficiency of the cheating prover: Notice that the strategy of the cheating prover can be implemented in polynomial-time (that is, given that the simulator's running time, $t_S(\cdot)$, is polynomial as well). Thus, Lemma 4.5 (and so Theorem 1.1) will also hold if $\langle P, V\rangle$ is an *argument* system (since, in the case of argument systems, the existence of an *efficient* P^* leads to contradiction of the computational soundness of $\langle P, V\rangle$).

[19]Note that in the special case that $j = 1$ (i.e., when the verifier's response is the fixed initiation message $\mathtt{v}_1^{(i)}$), the cheating prover cannot really forward $\mathtt{p}_{j-1}^{(i)}$ to the honest verifier (since no such message exists). Still, since $\mathtt{v}_1^{(i)}$ is a fixed initiation message, the cheating prover can produce $\mathtt{v}_1^{(i)}$ without actually having to interact with the honest verifier (as it indeed does in Step 2 of the cheating prover strategy).

The cheating prover may "do nonsense" in Step 4c: The cheating prover is hoping to convince an honest verifier by focusing on the η^{th} session in recursive block number $\ell^{(\overline{bp}_\xi)}$, where $\overline{bp}_\xi$ denotes the ξ^{th} distinct block-prefix in the simulator's execution. Prover messages in session $(\ell^{(\overline{bp}_\xi)}, \eta)$ are received from the (multi-session) simulator and are forwarded to the (single-session) verifier. The honest verifier's answers are then fed back to the simulator as if they were answers given by $V_{g,h^{(r)}}$ (defined below). For the cheating prover to succeed in convincing the honest verifier the following two conditions must be satisfied: (1) Session $(\ell^{(\overline{bp}_\xi)}, \eta)$ is eventually accepted by $V_{g,h^{(r)}}$. (2) The cheating prover never "does nonsense" in Step 4c during its execution. Let us clarify the meaning of this "nonsense".

One main problem that the cheating prover is facing while conducting Step 4c emerges from the following fact: Whereas the black-box simulator is allowed to "rewind" $V_{g,h^{(r)}}$ (impersonated by the cheating prover) and attempt different execution prefixes before proceeding with the interaction of a session, the prover cannot do so while interacting with the actual verifier. In particular, the cheating prover may reach Step 4c with a $\mathtt{p}_{j-1}^{(\eta)}$ message that is different from the $\mathtt{p}_{j-1}^{(\eta)}$ message that was previously forwarded to the honest verifier (in Step 4b). Given that the verifier's answer to the current $\mathtt{p}_{j-1}^{(\eta)}$ message is most likely to be different than the answer which was given to the "previous" $\mathtt{p}_{j-1}^{(\eta)}$ message, by answering (in Step 4c) in the same way as before, the prover action "makes no sense".[20]

We stress that, at this point in its execution, the cheating prover might as well have stopped with some predetermined "failure" message (rather than "doing nonsense"). However, for simplicity of presentation, it is more convenient for us to let the cheating prover "do nonsense".

The punchline of the analysis is that with noticeable probability (over choices of (σ, g, h)), there exists a choice of (ξ, η) so that the above "bad" event will not occur for session $(\ell^{(\overline{bp}_\xi)}, \eta)$. That is, using the fact that the success of a "rewinding" also depends on the output of g (which determines whether and when sessions are aborted) we show that, with non-negligible probability, Step 4c is never reached with two different $\mathtt{p}_{j-1}^{(\eta)}$ messages. Specifically, for every $j \in \{2, \ldots, k+1\}$, once a $\mathtt{p}_{j-1}^{(\eta)}$ message is forwarded to the verifier (in Step 4b), all subsequent $\mathtt{p}_{j-1}^{(\eta)}$ messages are either equal to the forwarded message or are answered with ABORT (here we assume that session $(\ell^{(\overline{bp}_\xi)}, \eta)$ is eventually accepted by $V_{g,h^{(r)}}$, and every $\mathtt{p}_{j-1}^{(\eta)}$ message is forwarded to the verifier at least once).

Defining $h^{(r)}$ (mentioned above): Let (σ, g, h) and (ξ, η) be the initial choices made by the cheating prover, let $\overline{bp}_\xi$ be the ξ^{th} block-prefix appearing in $\textsc{exec}_x(\sigma, g, h)$, and suppose that the honest verifier uses coins r. Then, the function $h^{(r)} = h^{(r,\sigma,g,h,\xi,\eta)}$ is defined to be uniformly distributed among the functions h' which satisfy the following conditions: The value of h' when applied on $(\eta, \overline{bp}_\xi)$ equals r, whereas for $(\eta', \xi') \neq (\eta, \xi)$ the value of h' when applied on $(\eta', \overline{bp}_{\xi'})$ equals the value of h on this prefix. (The set of such functions h' is not empty due to the hypothesis that the functions are selected in a family of $t_S(n)$-wise independent hash functions.) We note that replacing h by $h^{(r)}$ does not effect Step 5 of the cheating prover, and that the cheating prover does not know $h^{(r)}$. In particular, whenever the honest verifier V uses coins r, one may think of the

[20]We stress that the cheating prover does not know the random coins of the honest verifier, and so it cannot compute the verifier's answers by himself. In addition, since P^* and V are engaging in an actual execution of the specified protocol $\langle P, V \rangle$ (in which every message is sent exactly once), the cheating prover cannot forward the "recent" $\mathtt{p}_{j-1}^{(\eta)}$ message to the honest verifier in order to obtain the corresponding answer (because it has already forwarded the previous $\mathtt{p}_{j-1}^{(\eta)}$ message to the honest verifier).

cheating prover as if it is answering the simulator's queries with the answers that would have been given by $V_{g,h^{(r)}}$.

Claim 5.2 *For every value of σ, g, ξ and η, if h and r are uniformly distributed then so is $h^{(r)}$.*

Proof Sketch: Fix some simulator coins $\sigma \in \{0,1\}^*$, $g \in G$, block-prefix index $\xi \in \{1, \ldots, t_S(n)\}$, and session index $\eta \in \{1, \ldots, n\}$. The key for proving Claim 5.2 is to view the process of picking a function $h \in H$ as consisting of two stages. The first stage is an iterative process in which up to $t_S(n)$ different arguments are adversarially chosen, and for each such argument the value of h on this argument is uniformly selected in its range. In the second stage, a function h is chosen uniformly from all h's in H under the constraints that are introduced in the first stage. The iterative process in which the arguments are chosen (that is, the first stage above) corresponds the simulator's choice of the various block-prefixes $\overline{bp}$ (along with the indices i), on which h is applied.

At first glance, it seems obvious that the function $h^{(r)}$, which is uniformly distributed amongst all functions that are defined to be equal to h on all inputs (except for the input $(\eta, \overline{bp}_\xi)$ on which it equals r) is uniformly distributed in H. Taking a closer look, however, one realizes that a rigorous proof for the above claim is more complex than one may initially think, since it is not even clear that an h that is defined by the above process actually belongs to the family H.

The main difficulty in proving the above lies in the fact that the simulator's queries may "adaptively" depend on previous answers it has received (which, in turn, may depend on previous outcomes of h). The key obervation used in order to overcome this difficulty is that for *every* family of $t_S(n)$-wise independent functions and for *every* sequence of at most $t_S(n)$ arguments (and in particular, for an adaptively chosen sequence), the values of a uniformly chosen function when applied to the arguments in the sequence are uniformly and independently distributed. Thus, as long as the values assigned to the function in the first stage of the above process are uniformly and independently distributed (which is indeed the case, even if we constraint one output to be equal to r), the process will yield a uniformly distributed function from H. ■

5.2 The success probability of the cheating prover

We start by introducing two important notions that will play a central role in the analysis of the success probability of the cheating prover.

5.2.1 Grouping queries according to their iteration-prefixes

In the sequel, it will be convenient to group the queries of the simulator into different classes based on different iteration-prefixes. (Recall that the iteration-prefix of a query $\overline{q}$ (satisfying $\pi_{\text{sn}}(\overline{q}) = (\ell, i)$ and $\pi_{\text{msg}}(\overline{q}) = j > 1$) is the prefix of $\overline{q}$ that ends with the $(j-1)^{\text{st}}$ prover message in session (ℓ, n).). Grouping by iteration-prefixes particularly makes sense in the case that two queries are of the same length (see discussion below). Nevertheless, by Definition 4.3, two queries may have the same iteration-prefix even if they are of *different* lengths (see below).

Definition 5.3 (ip-different queries) *Two queries, $\overline{q}_1$ and $\overline{q}_2$ (of possibly different lengths), are said to be* ip-different*, if and only if they have different iteration-prefixes (that is, $ip(\overline{q}_1) \neq ip(\overline{q}_2)$).*

By Definition 4.3, if two queries, $\overline{q}_1$ and $\overline{q}_2$, satisfy $ip(\overline{q}_1) = ip(\overline{q}_2)$, then the following two conditions must hold: (1) $\pi_{\text{sn}}(\overline{q}_1) = (\ell, i_1)$, $\pi_{\text{sn}}(\overline{q}_2) = (\ell, i_2)$ and; (2) $\pi_{\text{msg}}(\overline{q}_1) = \pi_{\text{msg}}(\overline{q}_2)$. However, it is not necessarily true that $i_1 = i_2$. In particular, it may very well be the case that q_1, q_2 have different lengths (i.e., $i_1 \neq i_2$) but are *not* ip-different (note that if $i_1 = i_2$ then q_1 and q_2 are of equal length). Still, even if two queries are of the same length and have the same iteration-prefix, it is not necessarily true that they are equal, as they may be different at some message which occurs after their iteration-prefixes.

Motivating Definition 5.3: Recall that a necessary condition for the success of the cheating prover is that for every j, once a $\mathsf{p}_{j-1}^{(\eta)}$ message has been forwarded to the verifier (in Step 4b), all subsequent $\mathsf{p}_{j-1}^{(\eta)}$ messages (that are not answered with ABORT) are equal to the forwarded message. In order to satisfy the above condition it is sufficient to require that the cheating prover never reaches Steps 4b and 4c with two ip-different queries of equal length. The reason for this is that if two queries of the same length have the same iteration-prefix, then they contain the *same* sequence of prover messages for the corresponding session (since all such messages are contained in the iteration-prefix), and so they agree on their $\mathsf{p}_{j-1}^{(\eta)}$ message. In particular, once a $\mathsf{p}_{j-1}^{(\eta)}$ message has been forwarded to the verifier (in Step 4b), all subsequent queries that reach Step 4c and are of the same lenght will have the same $\mathsf{p}_{j-1}^{(\eta)}$ messages as the first such query (since they have the same iteration-prefix).

In light of the above discussion, it is only natural to require that the number of ip-different queries that reach Step 4c of the cheating prover is exactly one (as, in such a case, the above necessary condition is indeed satified).[21] Jumping ahead, we comment that the smaller is the number of ip-different queries that correspond to block-prefix $\overline{bp}_\xi$, the smaller is the probability that more than one ip-different query reaches Step 4c. The reason for this lies in the fact that the number of ip-different queries that correspond to block-prefix $\overline{bp}_\xi$ is equal to the number of different iteration-prefixes that correspond to $\overline{bp}_\xi$. In particular, the smaller is the number of such iteration-prefixes, the smaller is the probability that g will evaluate to 1 on more than a single iteration-prefix (thus reaching Step 4c with more than one ip-different query).

5.2.2 Useful block-prefixes

The probability that the cheating prover makes the honest verifier accept will be lower bounded by the probability that the ξ^{th} distinct block-prefix in $\textsc{exec}_x(\sigma, g, h)$ is η-useful (in the sense hinted above and defined next):

Definition 5.4 (Useful block-prefix) *A specific block-prefix $\overline{bp} = (b_1, a_1, \ldots, b_\gamma, a_\gamma)$, appearing in $\textsc{exec}_x(\sigma, g, h)$, is called i-*useful *if it satisfies the following two conditions:*

1. *For every $j \in \{2, .., k+1\}$, the number of* ip*-different queries $\overline{q}$ in $\textsc{exec}_x(\sigma, g, h)$ that correspond to block-prefix $\overline{bp}$ and satisfy $\pi_{\text{sn}}(\overline{q}) = (\ell^{(\overline{bp})}, i)$, $\pi_{\text{msg}}(\overline{q}) = j$, and $g(i, ip(\overline{q})) = 1$, is exactly one.*
2. *The (only) query $\overline{q}$ in $\textsc{exec}_x(\sigma, g, h)$ that corresponds to block-prefix $\overline{bp}$ and that satisfies $\pi_{\text{sn}}(\overline{q}) = (\ell^{(\overline{bp})}, i)$, $\pi_{\text{msg}}(\overline{q}) = k+1$, and $g(i, ip(\overline{q})) = 1$, is answered with* ACCEPT *by $V_{g,h}$.*

If there exists an $i \in \{1, \ldots, n\}$, so that a block-prefix is i-useful, then this block-prefix is called useful.

Condition 1 in Definition 5.4 states that for every fixed value of j there exists exactly one iteration-prefix, $\overline{ip}$, that corresponds to queries of the block-prefix $\overline{bp}$ and the the j^{th} message so that $g(i, \overline{ip})$ evaluates to 1. Condition 2 asserts that the last verifier message in the i^{th} main session of recursive block number $\ell = \ell^{(\overline{bp})}$ is equal to ACCEPT. It follows that if the cheating prover happens to select

[21] In order to ensure the cheating prover's success, the above requirement should be augmented by the condition that session $(\ell^{(\overline{bp}_\xi)}, \eta)$ is accepted by $V_{g,h^{(r)}}$.

$(\sigma, g, h, \xi, \eta)$ so that block-prefix $\overline{bp}_\xi$ (i.e., the $\xi^{\rm th}$ distinct block-prefix in $\textsc{exec}_x(\sigma, g, h^{(r)})$) is η-useful, then it convinces $V(x,r)$; the reason being that (by Condition 2) the last message in session $(\ell^{(\overline{bp}_\xi)}, \eta)$ is answered with ACCEPT,[22] and that (by Condition 1) the emulation does not get into trouble in Step 4c of the cheating prover (to see this, notice that each prover message in session $(\ell^{(\overline{bp}_\xi)}, \eta)$ will end up reaching Step 4c only once).

Let $\langle P^*, V\rangle(x) = \langle P^*(\sigma, g, h, \xi, \eta), V(r)\rangle(x)$ denote the random variable representing the (local) output of the honest verifier V when interacting with the cheating prover P^* on common input x, where σ, g, h, ξ, η are the initial random choices made by the cheating prover P^*, and r is the randomness used by the honest verifier V. Adopting this notation, we will say that the cheating prover $P^* = P^*(x, \sigma, g, h, \xi, \eta)$ has convinced the honest verifier $V = V(x,r)$ if $\langle P^*, V\rangle(x) = \texttt{ACCEPT}$. With these notations, we are ready to formalize the above discussion.

Claim 5.5 *If the cheating prover happens to select $(\sigma, g, h, \xi, \eta)$ so that the $\xi^{\rm th}$ distinct block-prefix in $\textsc{exec}_x(\sigma, g, h^{(r)})$ is η-useful, then the cheating prover convinces $V(x,r)$ (i.e., $\langle P^*, V\rangle(x) = \texttt{ACCEPT}$).*

Proof: Let us fix $x \in \{0,1\}^n$, $\sigma \in \{0,1\}^*$, $g \in G$, $h \in H$, $r \in \{1, \ldots, \rho_V(n)\}$, $\eta \in \{1, \ldots, n\}$, and $\xi \in \{1, \ldots, t_S(n)\}$. We show that if the $\xi^{\rm th}$ distinct block-prefix in $\textsc{exec}_x(\sigma, g, h^{(r)})$ is η-useful, then the cheating prover $P^*(x, \sigma, g, h, \xi, \eta)$ convinces the honest verifier $V(x,r)$.

By definition of the cheating-prover, the prover messages that are actually forwarded to the honest verifier (in Step 4b) correspond to session $(\ell^{(\overline{bp}_\xi)}, \eta)$. Specifically, messages that are forwarded by the cheating prover are of the form $\mathsf{p}_{j-1}^{(\eta)}$, and correspond to queries $\overline{q}$, that satisfy $\pi_{\rm sn}(\overline{q}) = (\ell^{(\overline{bp}_\xi)}, \eta)$, $\pi_{\rm msg}(\overline{q}) = j$ and $g(\eta, ip(\overline{q})) = 1$. Since the $\xi^{\rm th}$ distinct block-prefix in $\textsc{exec}_x(\sigma, g, h^{(r)})$ is η-useful, we have that for every $j \in \{2, \ldots, k{+}1\}$, there is exactly one query $\overline{q}$ that satisfies the above conditions. Thus, for every $j \in \{2, \ldots, k{+}1\}$, the cheating prover never reaches Step 4c with two different $\mathsf{p}_{j-1}^{(\eta)}$ messages. Here we use the fact that if two queries of the same length are not ip-different (i.e., have the same iteration-prefix) then the answers given by $V_{g,h^{(r)}}$ to these queries are identical (see discussion above). This in particular means that P^* is answering the simulator's queries with the answers that would have been given by $V^{g,h^{(r)}}$ itself. (Put in other words, whenever the $\xi^{\rm th}$ distinct block-prefix in $\textsc{exec}_x(\sigma, g, h^{(r)})$ is η-useful, the emulation does not "get into trouble" in Step 4c of the cheating prover.)

At this point, we have that the cheating prover never fails to perform Step 4c, and so the interaction that it is conducting with $V(x,r)$ reaches "safely" the $(k+1)^{\rm st}$ verifier message in the protocol. To complete the proof we have to show that at the end of the interaction with the cheating-prover, $V(x,r)$ outputs ACCEPT. This is true since, by Condition 2 of Definition 5.4, the query $\overline{q}$, that corresponds to block-prefix $\overline{bp}_\xi$, satisfies $\pi_{\rm sn}(\overline{q}) = (\ell^{(\overline{bp}_\xi)}, \eta)$, $\pi_{\rm msg}(\overline{q}) = j$ and $g(\eta, ip(\overline{q})) = 1$, is answered with ACCEPT. Here we use the fact that $V(x,r)$ behaves exactly as $V_{g,h^{(r)}}$ behaves on queries that correspond to the $\xi^{\rm th}$ distinct block-prefix in $\textsc{exec}_x(\sigma, g, h^{(r)})$. ■

5.2.3 Reduction to rareness of legal transcripts without useful block-prefixes

The following lemma (Lemma 5.6) establishes the connection between the success probability of the simulator and the success probability of the cheating-prover. Loosely speaking, the lemma asserts that if S outputs a legal transcript with non-negligible probability, then the cheating prover will

[22]Notice that $V(x,r)$ behaves exactly as $V_{g,h^{(r)}}$ behaves on queries that correspond to the $\xi^{\rm th}$ distinct iteration-prefix in $\textsc{exec}_x(\sigma, g, h^{(r)})$.

succeed in convincing the honest verifier with non-negligible probability. Since this is in contradiction to the computational soundness of the proof system, we have that Lemma 5.6 actually implies the correctness of Lemma 4.5 (recall that the contradiction hypothesis of Lemma 4.5 is that the probability that the simulator outputs a legal transcript is non-negligible).

Lemma 5.6 *Suppose that* $\Pr_{\sigma,g,h}[(\sigma,g,h) \in \mathtt{AC}] > 1/p(n)$ *for some fixed polynomial* $p(\cdot)$. *Then the probability (taken over* $\sigma, g, h, \xi, \eta, r$*), that* $\langle P^*, V\rangle(x) = \mathtt{ACCEPT}$ *is at least* $\frac{1}{2 \cdot p(n) \cdot t_S(n) \cdot n}$.

Proof: Define a Boolean indicator $\mathsf{useful}_{\xi,\eta}(\sigma,g,h)$ to be true if and only if the ξ^{th} distinct block-prefix in $\text{EXEC}_x(\sigma,g,h)$ is η-useful. Using Claim 5.5, we have:

$$\Pr_{\sigma,g,h,\xi,\eta,r}\left[\langle P^*, V\rangle(x) = \mathtt{ACCEPT}\right] \geq \Pr_{\sigma,g,h,\xi,\eta,r}\left[\mathsf{useful}_{\xi,\eta}(\sigma,g,h^{(r)})\right] \tag{2}$$

where the second probability refers to an interaction between S and $V_{g,h^{(r)}}$. Since for every value of σ, g, η and ξ, when h and r are uniformly selected the function $h^{(r)}$ is uniformly distributed (see Claim 5.2), we infer that:

$$\Pr_{\sigma,g,h,\xi,\eta,r}\left[\mathsf{useful}_{\xi,\eta}(\sigma,g,h^{(r)})\right] = \Pr_{\sigma,g,h',\xi,\eta}\left[\mathsf{useful}_{\xi,\eta}(\sigma,g,h')\right] \tag{3}$$

On the other hand, since ξ and η are distributed independently of (σ,g,h), we have:

$$\begin{aligned}
\Pr_{\sigma,g,h,\xi,\eta}\left[\mathsf{useful}_{\xi,\eta}(\sigma,g,h)\right] &= \sum_{d=1}^{t_S(n)} \sum_{i=1}^{n} \Pr_{\sigma,g,h,\xi,\eta}\left[\mathsf{useful}_{d,i}(\sigma,g,h) \;\&\; (\xi = d \;\&\; \eta = i)\right] \\
&= \sum_{d=1}^{t_S(n)} \sum_{i=1}^{n} \Pr_{\sigma,g,h}\left[\mathsf{useful}_{d,i}(\sigma,g,h)\right] \cdot \Pr_{\xi,\eta}\left[\xi = d \;\&\; \eta = i\right] \\
&= \sum_{d=1}^{t_S(n)} \sum_{i=1}^{n} \Pr_{\sigma,g,h}\left[\mathsf{useful}_{d,i}(\sigma,g,h)\right] \cdot \frac{1}{t_S(n) \cdot n} \\
&\geq \Pr_{\sigma,g,h}\left[\exists d, i \text{ s.t. } \mathsf{useful}_{d,i}(\sigma,g,h)\right] \cdot \frac{1}{t_S(n) \cdot n} \qquad (4)
\end{aligned}$$

where $t_S(n)$ is the bound used by the cheating prover (for the number of distinct block-prefixes in $\text{EXEC}_x(\sigma,g,h)$). Combining Eq. (2), (3), (4) we get:

$$\Pr_{\sigma,g,h,\xi,\eta,r}\left[\langle P^*, V\rangle(x) = \mathtt{ACCEPT}\right] \;\geq\; \Pr_{\sigma,g,h}\left[\exists d, i \text{ s.t. } \mathsf{useful}_{d,i}(\sigma,g,h)\right] \cdot \frac{1}{t_S(n) \cdot n} \tag{5}$$

Recall that by our hypothesis, $\Pr[(\sigma,g,h) \in \mathtt{AC}] > 1/p(n)$ for some fixed polynomial $p(\cdot)$. We can thus rewrite and lower bound the value of $\Pr_{\sigma,g,h}\left[\exists d, i \text{ s.t. } \mathsf{useful}_{d,i}(\sigma,g,h)\right]$ in the following way:

$$\begin{aligned}
&\Pr\left[\exists d, i \text{ s.t. } \mathsf{useful}_{d,i}(\sigma,g,h)\right] \\
= \;& 1 - \Pr\left[\forall d, i \; \neg\mathsf{useful}_{d,i}(\sigma,g,h)\right] \\
= \;& 1 - \Pr\left[(\forall d, i \; \neg\mathsf{useful}_{d,i}(\sigma,g,h)) \;\&\; ((\sigma,g,h) \notin \mathtt{AC})\right] - \Pr\left[(\forall d, i \; \neg\mathsf{useful}_{d,i}(\sigma,g,h)) \;\&\; ((\sigma,g,h) \in \mathtt{AC})\right] \\
\geq \;& 1 - \Pr\left[(\sigma,g,h) \notin \mathtt{AC}\right] - \Pr\left[(\forall d, i \; \neg\mathsf{useful}_{d,i}(\sigma,g,h)) \;\&\; (\sigma,g,h) \in \mathtt{AC}\right] \\
> \;& 1/p(n) - \Pr\left[(\forall d, i \; \neg\mathsf{useful}_{d,i}(\sigma,g,h)) \;\&\; (\sigma,g,h) \in \mathtt{AC}\right]
\end{aligned}$$

where all the above probabilities are taken over (σ, g, h). It follows that in order to show that $\Pr_{\sigma,g,h,\xi,\eta,r}[\langle P^*, V\rangle(x) = \texttt{ACCEPT}] > \frac{1}{2\cdot p(n)\cdot t_S(n)\cdot n}$, it will be sufficient to prove that for every fixed polynomial $p'(\cdot)$ it holds that:

$$\Pr_{\sigma,g,h}[(\forall d, i\ \neg\mathsf{useful}_{d,i}(\sigma, g, h))\ \&\ (\sigma, g, h) \in \texttt{AC}] \quad < \quad 1/p'(n)$$

Thus, Lemma 5.6 is satisfied provided that $\Pr_{\sigma,g,h}[\forall d, i\ \neg\mathsf{useful}_{d,i}(\sigma, g, h)\ \&\ (\sigma, g, h) \in \texttt{AC}]$ is negligible. Consequently, Lemma 5.6 will follow by establishing Lemma 5.7, stated next.

Lemma 5.7 *The probability (taken over σ, g, h), that for all pairs (d, i) $\mathsf{useful}_{d,i}(\sigma, g, h)$ does not hold* **and** *that $(\sigma, g, h) \in \texttt{AC}$, is negligible. That is, the probability that $\text{EXEC}_x(\sigma, g, h)$ does not contain a useful block-prefix* and *S outputs a legal transcript is negligible.*

This completes the proof of Lemma 5.6. The rest of this section is devoted to proving Lemma 5.7. ■

5.3 Proof of Lemma 5.7 (existence of useful block-prefixes in legal transcripts)

The proof of Lemma 5.7 will proceed as follows. We first define a special kind of block-prefixes, called potentially-useful block-prefixes. Loosely speaking, these are block-prefixes in which the simulator does not make too many "rewinding" attempts (each "rewinding" corresponds to a different iteration-prefix). Intuitively, the larger the number of "rewindings" is, the smaller is the probability that a specific block-prefix is useful. A block-prefix with a small number of "rewindings" is thus more likely to cause its block-prefix to be useful. Thus our basic approach will be to show that:

1. In *every* "successful" execution (i.e., producing a legal transcript), the simulator generates a potentially-useful block-prefix. This is proved by demonstrating, based on the structure of the schedule, that if no potentially-useful block-prefix exists, then the simulation must take super-polynomial time.

2. Any potentially-useful block-prefix is in fact useful with considerable probability. The argument that demonstrates this claim proceeds basically as follows. Consider a specific block-prefix $\overline{bp}$, let $\ell = \ell^{(\overline{bp})}$, and focus on a specific instance of session (ℓ, i) (that is, the specific instance of session (ℓ, i) that corresponds to block-prefix $\overline{bp}$). Suppose that block-prefix $\overline{bp}$ is potentially-useful and that the above instance of session (ℓ, i) happens to be accepted by $V_{g,h}$. This means that there exist k queries with block-prefix $\overline{bp}$ that consist of the "main thread" that leads to acceptance (i.e., all queries that were not answered with ABORT). Recall that the decision to abort a session (ℓ, i) is made by applying the function g to i and the iteration-prefix of the corresponding query. Thus, if there are only few different iteration-prefixes that correspond to block-prefix $\overline{bp}$ (which, as we said, is potentially-useful), then there is considerable probability that *all* the queries having block-prefix $\overline{bp}$, but which do not belong to that "main thread", will be answered with ABORT (that is, g will evaluate to 0 on the corresponding input). If this lucky event occurs, then block-prefix $\overline{bp}$ will indeed be useful (recall that for a block-prefix to be useful we require that there exists a corresponding session that is accepted by $V_{g,h}$ and satisfies that for every $j \in \{2, \ldots, k+1\}$ there is a single iteration-prefix that makes g evaluate to 1 at the j^{th} message of this session).

Returning to the actual proof, we start by introducing the necessary definition (of a potentially-useful block-prefix). Recall that, for any $g \in G$ and $h \in H$, the running time of the simulator S with oracle access to $V_{g,h}$ is bounded by $t_S(n)$. Let c be a constant such that $t_S(n) \leq n^c$ for all sufficiently large n.

Definition 5.8 (Potentially-useful block-prefix) *A specific block-prefix $\overline{bp} = (b_1, a_1, .., b_\gamma, a_\gamma)$, appearing in* $\text{EXEC}_x(\sigma, g, h)$, *is called* potentially-useful *if it satisfies the following two conditions:*

1. *The number of* ip-*different queries that correspond to block-prefix $\overline{bp}$ is at most k^{c+1}.*
2. *The execution of the simulator reaches the end of the block that corresponds to block-prefix $\overline{bp}$. That is,* $\text{EXEC}_x(\sigma, g, h)$ *contains a query $\overline{q}$, that ends with the $(k+1)^{\text{st}}$ prover message in the n^{th} main session of recursive block number $\ell^{(\overline{bp})}$ (i.e., some $\mathsf{p}_{k+1}^{(\ell^{(\overline{bp})},n)}$ message).*

We stress that the bound k^{c+1} in Condition 1 above refers to the same constant $c > 0$ that is used in the time bound $t_S(n) \leq n^c$. Using Definition 5.3 (of ip-different queries), we have that a bound of k^{c+1} on the number of ip-different queries that correspond to block-prefix $\overline{bp}$ induces an upper bound on the total number of iteration-prefixes that correspond to block-prefix $\overline{bp}$. Note that this is in contrast to the definition of a useful block-prefix (Definition 5.4), in which we only have a bound on the number of ip-different queries of a specific length (i.e., the number of ip-different queries that correspond to specific message in a specific session).

Turning to Condition 2 of Definition 5.8 we recall that the query $\overline{q}$ ends with a $\mathsf{p}_{k+1}^{(\ell^{(\overline{bp})},n)}$ message (i.e., the last prover message of recursive block number $\ell^{(\overline{bp})}$). Technically speaking, this means that $\overline{q}$ does not actually correspond to block-prefix $\overline{bp}$ (since, by definition of the recursive schedule, the answer to query $\overline{q}$ is a message that does not belong to recursive block number $\ell^{(\overline{bp})}$). Nevertheless, since before making query $\overline{q}$, the simulator has made queries to all prefixes of $\overline{q}$, we are guaranteed that for every $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, k+1\}$, the simulator has made a query $\overline{q}_{i,j}$ that is a prefix of $\overline{q}$, corresponds to block-prefix $\overline{bp}$, and satisfies $\pi_{\text{sn}}(\overline{q}) = (\ell^{(\overline{bp})}, i)$ and $\pi_{\text{msg}}(\overline{q}) = j$. (In other words, all messages of all sessions in recursive block number $\ell^{(\overline{bp})}$ have occurred during the execution of the simulator.) Furthermore, since the (modified) simulator does not make a query that is answered with a `DEVIATION` message (in Step 1' of $V_{g,h}$) and it does make the query $\overline{q}$, we are guaranteed that the partial execution transcript induced by the query $\overline{q}$ contains the accepting conversations of at least $\frac{n^{1/2}}{4}$ sessions in recursive block number $\ell^{(\overline{bp})}$. (The latter observation will be used only at a later stage (while proving Lemma 5.7).)

It is worth noting that whereas the definition of a useful block-prefix refers to the contents of iteration-prefixes (induced by the queries) that are sent by the simulator, the definition of a potentially-useful block-prefix refers only to their quantity (neither to their contents nor to the effect of the application of g on them).[23] It is thus natural that statements referring to potentially-useful block-prefixes tend to have a combinatorial flavor. The following lemma is no exception. It asserts that *every* "successful" execution of the simulator must contain a potentially-useful block-prefix (or, otherwise, the simulator will run in super-polynomial time).

Lemma 5.9 *For any $(\sigma, g, h) \in \mathtt{AC}_x$,* $\text{EXEC}_x(\sigma, g, h)$ *contains a potentially-useful block-prefix.*

5.3.1 Proof of Lemma 5.9 (existence of potentially-useful block-prefixes)

The proof of Lemma 5.9 is by contradiction. We assume the existence of a triplet $(\sigma, g, h) \in \mathtt{AC}$ so that every block-prefix in $\text{EXEC}_x(\sigma, g, h)$ is not potentially-useful, and show that this implies that $S_\sigma^{V_h}(x)$ made strictly more than n^c queries (which contradicts the explicit hypothesis that the running time of S is bounded by n^c).

[23]In particular, whereas the definition of a useful block-prefix refers to the outcome of g on iteration-prefixes that correspond to the relevant block-prefix, the definition of a potentially-useful block-prefix refers only to the number of ip-different queries that correspond to the block-prefix (ignoring the outcomes of g on the relevant iteration-prefixes).

The query–and–answer tree: Throughout the proof of Lemma 5.9, we will fix an arbitrary $(\sigma, g, h) \in \mathtt{AC}$ as above, and study the corresponding $\textsc{exec}_x(\sigma, g, h)$. A key vehicle in this study is the notion of a query–and–answer tree introduced in [26] (and also used in [29]).[24] This is a rooted tree (corresponding to $\textsc{exec}_x(\sigma, g, h)$) in which vertices are labeled with verifier messages and edges are labeled with prover's messages. The root is labeled with the fixed verifier message initializing the first session, and has outgoing edges corresponding to the prover's messages initializing this session. In general, paths down the tree (i.e., from the root to some vertices) correspond to queries. The query associated with such a path is obtained by concatenating the labeling of the vertices and edges along the path in the order traversed. We stress that each vertex in the query–and–answer tree corresponds to a query actually made by the simulator.

The index of the verifier (resp., prover) message labeling a specific vertex (resp., edge) in the tree is completely determined by the level in which the vertex (resp., edge) lies. That is, all vertices (resp., edges) in the ω^{th} level of the tree are labeled with the ω^{th} verifier (resp., prover) message in the schedule (out of a total of $n^2 \cdot (k+1)$ scheduled messages). For example, if $\omega = n^2 \cdot (k+1)$ all vertices (resp., edges) at the ω^{th} level (which is the lowest possible level in the tree) are labeled with $\mathsf{v}_{k+1}^{(n,n)}$ (resp., $\mathsf{p}_{k+1}^{(n,n)}$). The difference between "sibling" vertices in the same level of the tree lies in the difference in the labels of their incoming edges (as induced by the simulator's "rewindings"). Specifically, whenever the simulator "rewinds" the interaction to the ω^{th} verifier message in the schedule (i.e., makes a new query that is answered with the ω^{th} verifier message), the corresponding vertex in the tree (which lies at the ω^{th} level) will have multiple descendants one level down in the tree (i.e., at the $(\omega+1)^{\text{st}}$ level). The edges to each one of these descendants will be labeled with a different prover message.[25] We stress that the difference between these prover messages lies in the contents of the corresponding message (and not in its index).

By the above discussion, the outdegree of every vertex in the query–and–answer tree corresponds to the number of "rewindings" that the simulator has made to the relevant point in the schedule (the order in which the outgoing edges appear in the tree does not necessarily correspond to the order in which the "rewindings" were actually performed by the simulator). Vertices in which the simulator does not perform a "rewinding" will thus have a single outgoing edge. In particular, in case that the simulator follows the prescribed prover strategy P (sending each scheduled message exactly once), all vertices in the tree will have outdegree one, and the tree will actually consist of a single path of total length $n^2 \cdot (k+1)$ (ending with an edge that is labeled with a $\mathsf{p}_{k+1}^{(n,n)}$ message).

Recall that, by our conventions regarding the simulator, before making a query $\overline{q}$ the simulator has made queries to all prefixes of $\overline{q}$. Since every query corresponds to a path down the tree, we have that every particular path down the query–and–answer tree is developed from the root downwards (that is, within a specific path, a level $\omega < \omega'$ vertex is always visited before a level ω' vertex). However, we cannot say anything about the order in which *different* paths in the tree are developed (for example, we cannot assume that the simulator has made all queries that end at a level ω vertex before making any other query that ends at a level $\omega' > \omega$ vertex, or that it has visited all vertices of level ω in some specific order). To summarize, the only guarantee that we have about the order in which the query–and–answer tree is developed is implied by the convention that before making a specific query, the simulator has made queries to all relevant prefixes.

Satisfied path: A path from one node in the tree to some of its descendants is said to satisfy session i if the path contains edges (resp., vertices) for each of the messages sent by the prover

[24]The query–and–answer tree should not be confused with the tree that is induced by the recursive schedule.

[25]In particular, the shape of the query–and–answer tree is completely determined by the contents of prover messages in $\textsc{exec}_x(\sigma, g, h)$ (whereas the contents of verifier answers given by $V_{g,h}$ have no effect on the shape of the tree).

(resp., verifier) in session i. A path is called satisfied if it satisfies all sessions for which the verifier's first message appears along the path. One important example for a satisfied path is the path that starts at the root of the query–and–answer tree and ends with an edge that is labeled with a $\mathsf{p}_{k+1}^{(n,n)}$ message. This path contains all $n^2 \cdot (k+1)$ messages in the schedule (and so satisfies all n^2 sessions in the schedule). We stress that the contents of messages (occurring as labels) along a path are completely irrelevant to the question of whether the path is satisfied or not. In particular, a path may be satisfied even if some (or even all) of the vertices along it are labeled with ABORT.

Recall that, by our conventions, the simulator never makes a query that is answered with the DEVIATION message. We are thus guaranteed that, for every completed block along a path in the tree, at least $\frac{n^{1/2}}{4}$ sessions are accepted by $V_{g,h}$. In particular, the vertices corresponding to messages of these accepted sessions cannot be labeled with ABORT.

Good sub-tree: Consider an arbitrary sub-tree (of the query–and–answer tree) that satisfies the following two conditions:

1. The sub-tree is rooted at a vertex corresponding to the first message of some session so that this session is the first main session of some recursive invocation of the schedule.

2. Each path in the sub-tree is truncated at the last message of the relevant recursive invocation.

The full tree (i.e., the tree rooted at the vertex labeled with the first message in the schedule) is indeed such a tree, but we will need to consider sub-trees which correspond to m sessions in the recursive schedule construction (i.e., correspond to $\mathcal{R}_m$). We call such a sub-tree m-good if it contains a satisfied path starting at the root of the sub-tree. Since $(\sigma, g, h) \in \mathtt{AC}$, we have that the simulator has indeed produced a "legal" transcript as output. It follows that the full tree contains a path from the root to a leaf that contains vertices (resp., edges) for each of the messages sent by the verifier (resp., prover) in all n^2 sessions of the schedule (as otherwise the transcript $S_\sigma^{V_{g,h}}(x)$ would have not been legal). In other words, the full tree contains a satisfied path and is thus n^2-good.

Note that, by the definition of the recursive schedule, two m-good sub-trees are always disjoint. On the other hand, if $m' < m$, it may be the case that an m'-good sub-tree is contained in another m-good sub-tree. As a matter of fact, since an m-good sub-tree contains all messages of all sessions in a recursive block corresponding to $\mathcal{R}_m$, then it must contain at least k disjoint $\frac{m-n}{k}$-good sub-trees (i.e., that correspond to k the recursive invocations of $\mathcal{R}_{\frac{m-n}{k}}$ made by $\mathcal{R}_m$).

The next lemma (which can be viewed as the crux of the proof) states that, if the contradiction hypothesis of Lemma 5.9 is satisfied, then the number of disjoint $\frac{m-n}{k}$-good sub-trees that are contained in an m-good sub-tree is actually considerably larger than k.

Lemma 5.10 *Suppose that every block-prefix that appears in* $\text{EXEC}_x(\sigma, g, h)$ *is not potentially-useful. Then for every* $m \geq n$*, every* m*-good sub-tree contains at least* k^{c+1} *disjoint* $\frac{m-n}{k}$*-good sub-trees.*

Denote by $W(m)$ the size of an m-good sub-tree. (That is, $W(m)$ actually represents the work performed by the simulator on m concurrent sessions in our fixed scheduling.) It follows (from Lemma 5.10) that any m-good sub-tree must satisfy:

$$W(m) \geq \begin{cases} 1 & \text{if } m \leq n \\ k^{c+1} \cdot W\left(\frac{m-n}{k}\right) & \text{if } m > n \end{cases} \tag{6}$$

Since for all but finitely many n, Eq. (6) solves to $W(n^2) > n^c$ (see Section B in the Appendix), and since every vertex in the query–and–answer tree corresponds to a query actually made by the simulator, it follows that the hypothesis that the simulator runs in time that is bounded by n^c (and hence the full n^2-good tree must have been of size at most n^c) is contradicted. Thus, Lemma 5.9 will actually follow from Lemma 5.10.

Proof (of Lemma 5.10): Let T be an arbitrary m-good sub-tree of the query–and–answer tree. Considering the m sessions corresponding to an m-good sub-tree, we focus on the n main sessions of this level of the recursive construction. Let B_T denote the recursive block to which the indices of these n sessions belong. A T-query is a query $\overline{q}$ whose corresponding path down the query–and–answer tree ends with a node that belongs to T (recall that every query $\overline{q}$ appearing in $\text{EXEC}_x(\sigma, g, h)$ corresponds to a path down the full tree), and that satisfies $\pi_{\text{sn}}(\overline{q}) \in B_T$.[26] We first claim that all T-queries $\overline{q}$ in $\text{EXEC}_x(\sigma, g, h)$ have the same block-prefix. This block-prefix corresponds to the path from the root of the full tree to the root of T, and is denoted by $\overline{bp_T}$.

Fact 5.11 *All T-queries in $\text{EXEC}_x(\sigma, g, h)$ have the same block-prefix (denoted $\overline{bp_T}$).*

Proof: Assume, towards contradiction, that there exist two different T-queries $\overline{q}_1, \overline{q}_2$ so that $bp(\overline{q}_1) \neq bp(\overline{q}_2)$. In particular, $bp(\overline{q}_1)$ and $bp(\overline{q}_2)$ must differ in a message that precedes the first message of the first main session in B_T. (Note that if two block-prefixes are equal in all messages preceding the first message of the first session of the relevant block then, by definition, they are equal.[27]) This means that the paths that correspond to $\overline{q}_1$ and $\overline{q}_2$ split from each other before they reach the root of T (remember that T is rooted at a node corresponding to the first main session of recursive block B_T). But this contradicts the fact that both paths that correspond to these queries end with a node in T, and the fact follows. $\Box$

Using the hypothesis that no block-prefix in $\text{EXEC}_x(\sigma, g, h)$ is potentially-useful, we prove:

Claim 5.12 *Let T be an m-good sub-tree. Then the number of* ip*-different queries that correspond to block-prefix $\overline{bp_T}$ is at least k^{c+1}.*

Proof: Since all block-prefixes that appear in $\text{EXEC}_x(\sigma, g, h)$ are not potentially-useful (by the hypothesis of Lemma 5.10), this holds as a special case for block-prefix $\overline{bp_T}$. Let $\ell = \ell^{(\overline{bp_T})}$ be the index of the recursive block that corresponds to block-prefix $\overline{bp_T}$ in $\text{EXEC}_x(\sigma, g, h)$. Since block-prefix $\overline{bp_T}$ is not potentially-useful, at least one of the two conditions of Definition 5.8 is violated. In other words, one of the following two conditions is satisfied:

1. The number of ip-different queries that correspond to block-prefix $\overline{bp_T}$ is at least k^{c+1}.
2. The execution of the simulator does not reach the end of the block that corresponds to block-prefix $\overline{bp_T}$ (i.e., there is no query in $\text{EXEC}_x(\sigma, g, h)$ that ends with a $\mathsf{p}_{k+1}^{(\ell,n)}$ message that corresponds to block-prefix $\overline{bp_T}$).

Now, since T is an m-good sub-tree, then it must contain a satisfied path. Such a path starts at the root of T and satisfies all sessions whose first verifier message appears along the path. The key observation is that every satisfied path that starts at the root of sub-tree T must satisfy all the

[26]Note that queries $\overline{q}$ that satisfy $\pi_{\text{sn}}(\overline{q}) \in B_T$ do not necessarily correspond to a path that ends with a node in T (as $\text{EXEC}_x(\sigma, g, h)$ may contain a different sub-tree T' that satisfies $B_T = B_{T'}$). Also note that there exist queries $\overline{q}$, whose corresponding path ends with a node that belongs to T, but satisfy $\pi_{\text{sn}}(\overline{q}) \notin B_T$. This is so, since T may also contain vertices that correspond to messages in sessions which are not main sessions of B_T (in particular, all sessions that belong to the lower level recursive blocks that are invoked by block B_T).

[27]Recall that the index of the relevant block is determined by the length of the corresponding block-prefix

main sessions in B_T (to see this, notice that the first message of all main sessions in B_T will always appear along such a path), and so it contains all messages of all main session in recursive block B_T. In particular, the sub-tree T contains a path that starts at the root of T and ends with an edge that is labeled with the last prover message in session number (ℓ, n) (i.e., a $\mathsf{p}_{k+1}^{(\ell,n)}$ message). In other words, the execution of the simulator *does* reach the end of the block that corresponds to block-prefix $\overline{bp_T}$ (since, for the above path to exist, the simulator must have made a query that ends with a $\mathsf{p}_{k+1}^{(\ell,n)}$ message that corresponds to block-prefix $\overline{bp_T}$), and so Condition 2 above does not apply. Thus, the only reason that may cause block-prefix $\overline{bp_T}$ not to be potentially-useful is Condition 1. We conclude that the number of ip-different queries that correspond to block-prefix $\overline{bp_T}$ is at least k^{c+1}, as required. □

The following claim establishes the connection between the number of ip-different queries that correspond to block-prefix $\overline{bp_T}$ and the number of $\frac{m-n}{k}$-good sub-trees contained in T. Loosely speaking, this is achieved based on the following three observations: (1) Two queries are said to be ip-different if and only if they have different iteration-prefixes. (2) Every iteration-prefix is a block-prefix of some sub-schedule one level down in the recursive construction (consisting of $\frac{m-n}{k}$ sessions). (3) Every such distinct block-prefix yields a distinct $\frac{m-n}{k}$-good sub-tree.

Claim 5.13 *Let T be an m-good sub-tree. Then for every pair of* ip*-different queries that correspond to block-prefix $\overline{bp_T}$, the sub-tree T contains two disjoint $\frac{m-n}{k}$-good sub-trees.*

Once Claim 5.13 is proved, we can use it in conjunction with Claim 5.12 to infer that T contains at least k^{c+1} disjoint $\frac{m-n}{k}$-good sub-trees.

Proof: Before we proceed with the proof of Claim 5.13, we introduce the notion of an iteration-suffix of a query $\overline{q}$. This is the suffix of $\overline{q}$ that starts at the ending point of the query's iteration-prefix. A key feature satisfied by an iteration-suffix of a query is that it contains all the messages of all sessions belonging to some invocation of the schedule one level down in the recursive construction (this follows directly from the structure of our fixed schedule).

Definition 5.14 (Iteration-suffix) *The* iteration-suffix *of a query $\overline{q}$ (satisfying $j = \pi_{\mathrm{msg}}(\overline{q}) > 1$), denoted $is(\overline{q})$, is the suffix of $\overline{q}$ that begins at the ending point of the iteration-prefix of query $\overline{q}$. That is, for $\overline{q} = (b_1, a_1, \ldots, a_t, b_t)$ if $ip(\overline{q}) = (b_1, a_1, \ldots, b_{\delta-1}, a_\delta)$ then $is(\overline{q}) = (a_\delta, b_{\delta+1}, \ldots, a_t, b_t)$.*[28]

Let $\overline{q}$ be a query, and let $(\ell, i) = \pi_{\mathrm{sn}}(\overline{q})$, $j = \pi_{\mathrm{msg}}(\overline{q})$. Let $\mathcal{P}(\overline{q})$ denote the path corresponding to query $\overline{q}$ in the query-and-answer tree. Let $\mathcal{P}_{ip}(\overline{q})$ denote the sub-path of $\mathcal{P}(\overline{q})$ that corresponds to the iteration-prefix $ip(\overline{q})$ of $\overline{q}$, and let $\mathcal{P}_{is}(\overline{q})$ denote the sub-path of $\mathcal{P}(\overline{q})$ that corresponds to the iteration-suffix $is(\overline{q})$ of $\overline{q}$. That is, the sub-path $\mathcal{P}_{ip}(\overline{q})$ starts at the root of the full tree, and ends at a $\mathsf{p}_{j-1}^{(\ell,n)}$ message, whereas the sub-path $\mathcal{P}_{is}(\overline{q})$ starts at a $\mathsf{p}_{j-1}^{(\ell,n)}$ message and ends at a $\mathsf{v}_j^{(\ell,i)}$ message (in particular, path $\mathcal{P}(\overline{q})$ can be obtained by concatenating $\mathcal{P}_{ip}(\overline{q})$ with $\mathcal{P}_{is}(\overline{q})$[29]).

Fact 5.15 *For every query $\overline{q} \in \mathrm{EXEC}_x(\sigma, g, h)$, the sub-path $\mathcal{P}_{is}(\overline{q})$ is satisfied. Moreover:*

1. *The sub-path $\mathcal{P}_{is}(\overline{q})$ satisfies all $\frac{m-n}{k}$ sessions of a recursive invocation one level down in the recursive construction (i.e., corresponding to $\mathcal{R}_{\frac{m-n}{k}}$).*

2. *If $\overline{q}$ corresponds to block-prefix $\overline{bp_T}$, then the sub-path $\mathcal{P}_{is}(\overline{q})$ is contained in T.*

[28]This means that a_δ is of the form $\mathsf{p}_{j-1}^{(\ell,n)}$, where $(\ell, i) = \pi_{\mathrm{sn}}(\overline{q})$ and $j = \pi_{\mathrm{msg}}(\overline{q})$.

[29]To be precise, one should delete from the resulting concatenation one of the two consecutive edges which are labeled with $a_\delta = \mathsf{p}_{j-1}^{(\ell,n)}$ (one edge is the last in $\mathcal{P}_{ip}(\overline{q})$ and the other edge is the first in $\mathcal{P}_{is}(\overline{q})$).

Proof: Let $(\ell,i)=\pi_{\rm sn}(\overline{q})$ and $j=\pi_{\rm msg}(\overline{q})$. By nature of our fixed scheduling, the vertex in which sub-path $\mathcal{P}_{is}(\overline{q})$ begins precedes the first message of all (nested) sessions in the $(j-1)^{\rm st}$ recursive invocation made by recursive block number ℓ (i.e., an instance of $\mathcal{R}_{\frac{m-n}{k}}$ which is invoked by $\mathcal{R}_m$). Since query $\overline{q}$ is answered with a $\mathsf{v}_j^{(\ell,i)}$ message, we have that the sub-path $\mathcal{P}_{is}(\overline{q})$ eventually reaches a vertex labeled with $\mathsf{v}_j^{(\ell,i)}$. In particular, the sub-path $\mathcal{P}_{is}(\overline{q})$ (starting at a $\mathsf{p}_{j-1}^{(\ell,n)}$ edge and ending at a $\mathsf{v}_j^{(\ell,i)}$ vertex) contains the first and last messages of each of the above (nested) sessions, and so contains edges (resp., vertices) for each prover (resp., verifier) message in these sessions. But this means (by definition) that all these (nested) sessions are satisfied by $\mathcal{P}_{is}(\overline{q})$. Since the above (nested) sessions are the only sessions whose first message appears along the sub-path $\mathcal{P}_{is}(\overline{q})$, we have that $\mathcal{P}_{is}(\overline{q})$ is satisfied. To see that whenever $\overline{q}$ corresponds to block-prefix $\overline{bp_T}$ the sub-path $\mathcal{P}_{is}(\overline{q})$ is contained in the sub-tree T, we observe that both its starting point (i.e., a $\mathsf{p}_{j-1}^{(\ell,n)}$ edge) and its ending point (i.e., a $\mathsf{v}_j^{(\ell,i)}$ vertex) are contained in T. □

Fact 5.16 *Let $\overline{q}_1,\overline{q}_2$ be two* ip*-different queries. Then $\mathcal{P}_{is}(\overline{q}_1)$ and $\mathcal{P}_{is}(\overline{q}_2)$ are disjoint.*

Proof: Let $\overline{q}_1$ and $\overline{q}_2$ be two ip-different queries, let $(\ell_1,i_1)=\pi_{\rm sn}(\overline{q}_1)$, $(\ell_2,i_2)=\pi_{\rm sn}(\overline{q}_2)$, and let $j_1=\pi_{\rm msg}(\overline{q}_1)$, $j_2=\pi_{\rm msg}(\overline{q}_2)$. Recall that queries $\overline{q}_1$ and $\overline{q}_2$ are said to be ip-different if and only if they have different iteration-prefixes. Since $\overline{q}_1$ and $\overline{q}_2$ are assumed to be ip-different, then so are iteration-prefixes $ip(\overline{q}_1)$ and $ip(\overline{q}_2)$. In particular, the paths $\mathcal{P}_{ip}(\overline{q}_1)$ and $\mathcal{P}_{ip}(\overline{q}_2)$ are different. We distinguish between the following two cases:

1. **Path $\mathcal{P}_{ip}(\overline{q}_1)$ splits from $\mathcal{P}_{ip}(\overline{q}_2)$:** In such a case, the ending points of paths $\mathcal{P}_{ip}(\overline{q}_1)$ and $\mathcal{P}_{ip}(\overline{q}_2)$ must belong to different sub-trees of the query–and–answer tree. Since the starting point of an iteration-suffix is the ending point of the corresponding iteration-prefix, we must have that paths $\mathcal{P}_{is}(\overline{q}_1)$ and $\mathcal{P}_{is}(\overline{q}_2)$ are disjoint.

2. **Path $\mathcal{P}_{ip}(\overline{q}_1)$ is a prefix of path $\mathcal{P}_{ip}(\overline{q}_2)$:** That is, both $\mathcal{P}_{ip}(\overline{q}_1)$ and $\mathcal{P}_{ip}(\overline{q}_2)$ reach a $\mathsf{v}_{j_1-1}^{(\ell_1,n)}$ vertex, while path $\mathcal{P}_{ip}(\overline{q}_2)$ continues down the tree and reaches a $\mathsf{v}_{j_2-1}^{(\ell_2,n)}$ vertex. The key observation in this case is that either ℓ_1 is strictly smaller than ℓ_2, or j_1 is strictly smaller than j_2. The reason for this is that in case both $\ell_1=\ell_2$ and $j_1=j_2$ hold, iteration-prefix $ip(\overline{q}_1)$ must be equal to iteration-prefix $ip(\overline{q}_2)$,[30] in contradiction to our hypothesis. Since path $\mathcal{P}_{is}(\overline{q}_1)$ starts at a $\mathsf{p}_{j_1-1}^{(\ell_1,n)}$ vertex and ends with a $\mathsf{v}_{j_1}^{(\ell_1,i_1)}$ vertex, and since path $\mathcal{P}_{is}(\overline{q}_2)$ starts with a $\mathsf{p}_{j_2-1}^{(\ell_2,n)}$ vertex, we have that the ending point of path $\mathcal{P}_{is}(\overline{q}_1)$ precedes the starting point of path $\mathcal{P}_{is}(\overline{q}_2)$ (this is so since if $j_1<j_2$, the $\mathsf{p}_{j_1}^{(\ell_1,i_1)}$ message will always precede/equal the $\mathsf{p}_{j_2-1}^{(\ell_2,n)}$ message). In particular, paths $\mathcal{P}_{is}(\overline{q}_1)$ and $\mathcal{P}_{is}(\overline{q}_2)$ are disjoint.

It follows that for every two ip-different queries, $\overline{q}_1$ and $\overline{q}_2$, sub-paths $\mathcal{P}_{is}(\overline{q}_1)$ and $\mathcal{P}_{is}(\overline{q}_2)$ are disjoint, as required. □

Back to the proof of Claim 5.13, let $\overline{q}_1$ and $\overline{q}_2$ be two ip-different queries that correspond to block-prefix $\overline{bp_T}$ (as guaranteed by the hypothesis of Claim 5.13), and let $\mathcal{P}_{is}(\overline{q}_1)$ and $\mathcal{P}_{is}(\overline{q}_2)$ be as above. Consider the two sub-trees, T_1 and T_2, of T that are rooted at the starting point of sub-paths $\mathcal{P}_{is}(\overline{q}_1)$ and $\mathcal{P}_{is}(\overline{q}_2)$ respectively (note that by, Fact 5.15, T_1 and T_2 are indeed sub-trees of T). By definition of our recursive schedule, T_1 and T_2 correspond to $\frac{m-n}{k}$ sessions one level down in the

[30]That is, unless $bp(\overline{q}_1)\neq bp(\overline{q}_2)$. But in such a case, paths $\mathcal{P}_{ip}(\overline{q}_1)$ and $\mathcal{P}_{ip}(\overline{q}_2)$ must split from each other (since they differ in some message that belongs to their block-prefix), and we are back to Case 1.

recursive construction (i.e., to an instance of $\mathcal{R}_{\frac{m-n}{k}}$). Using Fact 5.15 we infer that sub-path $\mathcal{P}_{is}(\overline{q}_1)$ (resp., $\mathcal{P}_{is}(\overline{q}_2)$) contains all messages of all sessions in T_1 (resp., T_2), and so the sub-tree T_1 (resp., T_2), is $\frac{m-n}{k}$-good. In addition, since sub-paths $\mathcal{P}_{is}(\overline{q}_1)$ and $\mathcal{P}_{is}(\overline{q}_2)$ are disjoint (by Fact 5.16) and since, by definition of an $\frac{m-n}{k}$-good tree, two different $\frac{m-n}{k}$-good trees are always disjoint, then T_1 and T_2 (which, being rooted at different vertices, must be different) are also disjoint. It follows that for every pair of different queries that correspond to block-prefix $\overline{bp}_T$, the sub-tree T contains two disjoint $\frac{m-n}{k}$-good sub-trees. ■

We are finally ready to establish Lemma 5.10 (using Claims 5.12 and 5.13). By Claim 5.12, we have that the number of different queries that correspond to block-prefix $\overline{bp}_T$ is at least k^{c+1}. Since (by Claim 5.13), for every pair of different queries that correspond to block-prefix $\overline{bp}_T$ the sub-tree T contains two disjoint $\frac{m-n}{k}$-good sub-trees, we infer that T contains a total of at least k^{c+1} disjoint $\frac{m-n}{k}$-good sub-trees (corresponding to the (at least) k^{c+1} different queries mentioned above). Lemma 5.10 follows. ■

5.3.2 Back to the Proof of Lemma 5.7 (existence of useful block-prefixes)

Once the correctness of Lemma 5.9 is established, we may proceed with the proof of Lemma 5.7. Let $x \in \{0,1\}^n$. We bound from above the probability, taken over the choices of $\sigma \in \{0,1\}^*, g \overset{\text{R}}{\leftarrow} G$ and $h \overset{\text{R}}{\leftarrow} H$, that $(\sigma, g, h) \in \texttt{AC}$ and that for all $d \in \{1, \ldots, t_S(n)\}$ and all $i \in \{1, \ldots, n\}$, the d^{th} distinct block-prefix in $\textsc{exec}_x(\sigma, g, h)$ is not i-useful. Specifically, we would like to show that:

$$\Pr_{\sigma,g,h}\left[(\forall d, i \ \neg\mathsf{useful}_{d,i}(\sigma, g, h)) \ \& \ ((\sigma, g, h) \in \texttt{AC})\right] \tag{7}$$

is negligible. Define a Boolean indicator $\mathsf{pot{-}use}_d(\sigma, g, h)$ to be true if and only if the d^{th} distinct block-prefix in $\textsc{exec}_x(\sigma, g, h)$ is potentially-useful. As proved in Lemma 5.9, for any $(\sigma, g, h) \in \texttt{AC}$ there exists an index $d \in \{1, \ldots, t_S(n)\}$, so that the d^{th} block-prefix in $\textsc{exec}_x(\sigma, g, h)$ is potentially-useful. In other words, for every $(\sigma, g, h) \in \texttt{AC}$, $\mathsf{pot{-}use}_d(\sigma, g, h)$ holds for some value of d. Thus, Eq. (7) is upper bounded by:

$$\Pr_{\sigma,g,h}\left[\bigvee_{d=1}^{t_S(n)} \mathsf{pot{-}use}_d(\sigma, g, h) \ \& \ (\forall i \in \{1, \ldots, n\} \ \neg\mathsf{useful}_{d,i}(\sigma, g, h))\right] \tag{8}$$

Consider a specific $d \in \{1, \ldots, t_S(n)\}$ so that $\mathsf{pot{-}use}_d(\sigma, g, h)$ is satisfied (i.e., the d^{th} block prefix in $\textsc{exec}_x(\sigma, g, h)$ is potentially-useful). By Condition 2 in the definition of a potentially-useful block-prefix (Definition 5.8), the execution of the simulator reaches the end of the corresponding block in the schedule. In other words, there exists a query $\overline{q} \in \textsc{exec}_x(\sigma, g, h)$ that ends with the $(k+1)^{\text{st}}$ prover message in the n^{th} main session of recursive block number $\ell^{(\overline{bp}_d)}$, where $\overline{bp}_d$ denotes the d^{th} distinct block-prefix in $\textsc{exec}_x(\sigma, g, h)$, and $\ell^{(\overline{bp}_d)}$ denotes the index of the recursive block that corresponds to block-prefix $\overline{bp}_d$ in $\textsc{exec}_x(\sigma, g, h)$. Since, by our convention and the modification of the simulator, S never generates a query that is answered with a $\texttt{DEVIATION}$ message, we have that the partial execution transcript induced by query $\overline{q}$ must contain the accepting conversations of at least $\frac{n^{1/2}}{4}$ main sessions in block number $\ell^{(\overline{bp}_d)}$ (as otherwise query $\overline{q}$ would have been answered with the $\texttt{DEVIATION}$ message in Step 1' of $V_{g,h}$).

Let $\overline{q}^{(\overline{bp}_d)} = \overline{q}^{(\overline{bp}_d)}(\sigma, g, h)$ denote the first query in $\textsc{exec}_x(\sigma, g, h)$ that is as above (i.e., that ends with the $(k+1)^{\text{st}}$ prover message in the n^{th} main session of recursive block number $\ell^{(\overline{bp}_d)}$, where $\overline{bp}_d$ denotes the d^{th} block-prefix appearing in $\textsc{exec}_x(\sigma, g, h)$).[31] Define an additional Boolean indicator $\mathsf{accept}_{d,i}(\sigma, g, h)$ to be true if and only if query $\overline{q}^{(\overline{bp}_d)}$ contains an accepting conversation for session $(\ell^{(\overline{bp}_d)}, i)$ (that is, no prover message in session $(\ell^{(\overline{bp}_d)}, i)$ is answered with `ABORT`, and the last verifier message of this session equals `ACCEPT`).[32] It follows that for every $d \in \{1, \ldots, t_S(n)\}$ that satisfies $\mathsf{pot\text{--}use}_d(\sigma, g, h)$ (as above), there exists a set $\mathcal{S} \subset \{1, \ldots, n\}$ of size $\frac{n^{1/2}}{4}$ such that $\mathsf{accept}_{d,i}(\sigma, g, h)$ holds for every $i \in \mathcal{S}$. Thus, Eq. (8) is upper bounded by:

$$\Pr_{\sigma,g,h}\left[\bigvee_{d=1}^{t_S(n)} \bigvee_{\substack{\mathcal{S}\subset\{1,\ldots,n\}\\ |\mathcal{S}|=\frac{n^{1/2}}{4}}} \Big(\mathsf{pot\text{--}use}_d(\sigma, g, h) \;\&\; \Big(\forall i \in \mathcal{S},\ \neg\mathsf{useful}_{d,i}(\sigma, g, h) \;\&\; \mathsf{accept}_{d,i}(\sigma, g, h)\Big)\Big)\right] \qquad (9)$$

Using the union bound, we upper bound Eq. (9) by:

$$\sum_{d=1}^{t_S(n)} \sum_{\substack{\mathcal{S}\subset\{1,\ldots,n\}\\ |\mathcal{S}|=\frac{n^{1/2}}{4}}} \Pr_{\sigma,g,h}\Big[\mathsf{pot\text{--}use}_d(\sigma, g, h) \;\&\; \Big(\forall i \in \mathcal{S},\ \neg\mathsf{useful}_{d,i}(\sigma, g, h) \;\&\; \mathsf{accept}_{d,i}(\sigma, g, h)\Big)\Big] \qquad (10)$$

The last expression is upper bounded using the following lemma, that bounds the probability that a specific set of different sessions corresponding to the same (in index) potentially-useful block-prefix are accepted (at the first time that the recursive block to which they belong is completed), but still do not turn it into a useful block-prefix. In fact, we prove something stronger:

Lemma 5.17 *For every $\sigma \in \{0,1\}^*$, every $h \in H$, every $d \in \{1, \ldots, t_S(n)\}$, and every set of indices $S \subset \{1, \ldots, n\}$, so that $|S| > k$:*

$$\Pr_g\Big[\mathsf{pot\text{--}use}_d(\sigma, g, h) \;\&\; \Big(\forall i \in S,\ \neg\mathsf{useful}_{d,i}(\sigma, g, h) \;\&\; \mathsf{accept}_{d,i}(\sigma, g, h)\Big)\Big] < \Big(n^{-\left(\frac{1}{2}+\frac{1}{4k}\right)}\Big)^{|S|}$$

Proof: Let $x \in \{0,1\}^*$. Fix some $\sigma \in \{0,1\}^*$, $h \in H$, $d \in \{1, \ldots, t_S(n)\}$ and a set $S \subset \{1, \ldots, n\}$. Denote by $\overline{bp}_d = \overline{bp}_d(g)$ the d^{th} distinct block-prefix in $\textsc{exec}_x(\sigma, h, g)$, and by $\ell^{(\overline{bp}_d)}$ the index of its corresponding recursive block in the schedule. We bound the probability, taken over the choice of $g \overset{\text{R}}{\leftarrow} G$, that for all $i \in S$ block-prefix $\overline{bp}_d$ is not i-useful, even though it is potentially-useful and for all $i \in S$ the query $\overline{q}^{(\overline{bp}_d)}$ contains an accepting conversation for session $(\ell^{(\overline{bp}_d)}, i)$.

[31] Since the simulator is allowed to feed $V_{g,h}$ with different queries of the same length, we have that the execution of the simulator may reach the end of the corresponding block more than once (and thus, $\textsc{exec}_x(\sigma, g, h)$ may contain more than a single query that ends with the $(k+1)^{\text{st}}$ prover message in the n^{th} main session of block number $\ell^{(\overline{bp}_d)}$). Since each time that the simulator reaches the end of the corresponding block, the above set of accepted sessions may be different, we are not able to pinpoint a specific set of accepted sessions without explicitly specifying to which one of the above queries we are referring. We solve this problem by explicitly referring to the first query that satisfies the above conditions (note that, in our case, such a query is always guaranteed to exist).

[32] Note that the second condition implies the first one. Namely, if the last verifier message of session $(\ell^{(\overline{bp}_d)}, i)$ equals `ACCEPT`, then no prover message in this session could have been answered with `ABORT`.

A technical problem resolved: In order to prove Lemma 5.17 we need to focus on the d^{th} distinct block-prefix in $\text{EXEC}_x(\sigma, h, g)$ (denoted by $\overline{bp}_d$) and analyze the behaviour of a uniformly chosen g when applied to the various iteration-prefixes that correspond to $\overline{bp}_d$. However, trying to do so we encounter a technical problem. This problem is caused by the fact that the contents of block-prefix $\overline{bp}_d$ depends on g.[33] In particular, it does not make sense to analyze the behaviour of a uniformly chosen g on iteration-prefixes that correspond to an "undetermined" block-prefix (since it is not possible to determine the iteration-prefixes that correspond to $\overline{bp}_d$ when $\overline{bp}_d$ itself is not determined). To overcome the above problem, we rely on the following observations:

1. Whenever σ, h and d are fixed, the contents of block-prefix $\overline{bp}_d$ is completely determined by the output of g on inputs that have occurred *before* $\overline{bp}_d$ has been reached (i.e., has appeared as a block-prefix of some query) for the first time.

2. All iteration-prefixes that correspond to block-prefix $\overline{bp}_d$ occur *after* $\overline{bp}_d$ has been reached for the first time.

It is thus possible to carry out the analysis by considering the output of g only on inputs that have occurred *after* $\overline{bp}_d$ has been determined. That is, fixing σ, h and d we distinguish between: (a) the outputs of g that have occurred *before* the d^{th} distinct block-prefix in $\text{EXEC}_x(\sigma, g, h)$ (i.e., $\overline{bp}_d$) has been reached, and (b) the outputs of g that have occurred *after* $\overline{bp}_d$ has been reached. For every possible outcome of (a) we will analyze the (probabilistic) behaviour of g only over the outcomes of (b). (Recall that once (a)'s outcome has been determined, the identities (but not the contents) of all relevant prefixes are well defined.) Since for *every* possible outcome of (a) the analysis will hold, it will in particular hold over all choices of g.

More formally, consider the following (alternative) way of describing a uniformly chosen $g \in G$ (at least as far as $\text{EXEC}_x(\sigma, g, h)$ is concerned). Let g_1, g_2 be two $t_S(n)$-wise independent hash functions uniformly chosen from G and let σ, h, d be as above. We define $g^{(g_1,g_2)} = g^{(\sigma,h,d,g_1,g_2)}$ to be uniformly distributed among the functions g' that satisfy the following conditions: the value of g' when applied to an input α that has occurred *before* $\overline{bp}_d$ has been reached (in $\text{EXEC}_x(\sigma, g, h)$) is equal to $g_1(\alpha)$, whereas the value of g' when applied to an input α that has occurred *after* $\overline{bp}_d$ has been reached is equal to $g_2(\alpha)$.

Similarly to the proof of Claim 5.2 it can be shown that for every σ, h, d as above, if g_1 and g_2 are uniformly distributed then so is $g^{(g_1,g_2)}$. In particular:

$$\begin{aligned} & \Pr_g \Big[\mathsf{pot\text{-}use}_d(\sigma, g, h) \ \& \ \Big(\forall i \in S,\ \neg\mathsf{useful}_{d,i}(\sigma, g, h) \ \& \ \mathsf{accept}_{d,i}(\sigma, g, h)\Big)\Big] \\ = \ & \Pr_{g_1,g_2} \Big[\mathsf{pot\text{-}use}_d(\sigma, g^{(g_1,g_2)}, h) \ \& \ \Big(\forall i \in S,\ \neg\mathsf{useful}_{d,i}(\sigma, g^{(g_1,g_2)}, h) \ \& \ \mathsf{accept}_{d,i}(\sigma, g^{(g_1,g_2)}, h)\Big)\Big] \end{aligned}$$

By fixing g_1 and then analyzing the behaviour of a uniformly chosen g_2 on the relevant iteration-prefixes the above technical problem is resolved. This is due to the following two reasons: (1) For every choice of σ, h, d and for *every* fixed value of g_1, the block-prefix $\overline{bp}_d$ is completely determined (and the corresponding iteration-prefixes are well defined). (2) Once $\overline{bp}_d$ has been reached, the outcome of $g^{(g_1,g_2)}$ when applied to the *relevant* iteration-prefixes is completely determined by the

[33] Clearly, the contents of queries that appear in $\text{EXEC}_x(\sigma, g, h)$ may depend on the choice of the hash function g. (This is because the simulator may dynamically adapt its queries depending on the outcome of g on iteration-prefixes of past queries.) As a consequence, the contents of $\overline{bp}_d = \overline{bp}_d(g)$ may vary together with the choice of g.

choice of g_2. Thus, all we need to show in order to prove Lemma 5.17 is that for *every* choice of g_1, the value of:

$$\Pr_{g_2}\left[\mathsf{pot\text{-}use}_d(\sigma, g^{(g_1,g_2)}, h) \;\&\; \Big(\forall i \in S,\ \neg\mathsf{useful}_{d,i}(\sigma, g^{(g_1,g_2)}, h) \;\&\; \mathsf{accept}_{d,i}(\sigma, g^{(g_1,g_2)}, h)\Big)\right] \qquad (11)$$

is upper bounded by $(n^{-(1/2+1/4k)})^{|S|}$.

Back to the actual proof of Lemma 5.17: Consider the block-prefix $\overline{bp}_d$, as determined by the choices of σ, h, d and g_1, and focus on the iteration-prefixes that correspond to $\overline{bp}_d$ in $\text{EXEC}_x(\sigma, g, h)$. We next analyze the implications of $\overline{bp}_d$ being not i-useful, even though it is potentially-useful and for all $i \in S$ query $\overline{q}^{(\overline{bp}_d)}$ contains an accepting conversation for session $(\ell^{(\overline{bp}_d)}, i)$.

Claim 5.18 *Let $\sigma \in \{0,1\}^*$, $g \in G$, $h \in H$, $d \in \{1, \ldots, t_S(n)\}$ and $S \subset \{1, \ldots, n\}$. Suppose that the indicator $\Big(\mathsf{pot\text{-}use}_d(\sigma, g, h) \;\&\; (\forall i \in S, \neg\mathsf{useful}_{d,i}(\sigma, g, h) \;\&\; \mathsf{accept}_{d,i}(\sigma, g, h))\Big)$ is true. Then:*

1. *The number of different iteration-prefixes that correspond to block-prefix $\overline{bp}_d$ is at most k^{c+1}.*
2. *For every $j \in \{2, \ldots, k+1\}$, there exists an iteration-prefix $\overline{ip}_j$ (corresponding to block-prefix $\overline{bp}_d$), so that for every $i \in S$ we have $g(i, \overline{ip}_j) = 1$.*
3. *For every $i \in S$, there exist an (additional) iteration-prefix $\overline{ip}^{(i)}$ (corresponding to block-prefix $\overline{bp}_d$), so that for every $j \in \{2, \ldots, k+1\}$, we have $\overline{ip}^{(i)} \neq \overline{ip}_j$, and $g(i, \overline{ip}^{(i)}) = 1$.*

In accordance with the discussion above, Claim 5.18 will be invoked with $g = g^{(g_1,g_2)}$.

Proof: Loosely speaking, Item (1) follows directly from the hypothesis that block-prefix $\overline{bp}_d$ is potentially-useful. In order to prove Item (2) we also use the hypothesis that for all $i \in S$ query $\overline{q}^{(\overline{bp}_d)}$ contains an accepting conversation for session $(\ell^{(\overline{bp}_d)}, i)$, and in order to to prove Item (3) we additionally use the hypothesis that for all $i \in S$ block-prefix $\overline{bp}_d$ is not i-useful. Details follow.

Proof of Item 1: The hypothesis that block-prefix $\overline{bp}_d$ is potentially-useful (i.e., $\mathsf{pot\text{-}use}_d(\sigma, g, h)$ holds), implies that the number of iteration-prefixes that correspond to block-prefix $\overline{bp}_d$ is at most k^{c+1} (as otherwise, the number of ip-different queries that correspond to $\overline{bp}_d$ would have been greater than k^{c+1}).

Proof of Item 2: Let $i \in S$ and recall that $\mathsf{accept}_{d,i}(\sigma, g, h)$ holds. In particular, we have that query $\overline{q}^{(\overline{bp}_d)}$ (i.e., the first query in $\text{EXEC}_x(\sigma, g, h)$ that ends with the $(k+1)^{\text{st}}$ prover message in the n^{th} main session of recursive block number $\ell^{(\overline{bp}_d)}$) contains an accepting conversation for session $(\ell^{(\overline{bp}_d)}, i)$. That is, no prover message in session $(\ell^{(\overline{bp}_d)}, i)$ is answered with ABORT, and the last verifier message of this session equals ACCEPT. Since by our conventions regarding the simulator, before making query $\overline{q}^{(\overline{bp}_d)}$ the simulator has made queries to all relevant prefixes, then it must be the case that all prefixes of query $\overline{q}^{(\overline{bp}_d)}$ have previously occurred as queries in $\text{EXEC}_x(\sigma, g, h)$. In particular, for every $i \in S$ and for every $j \in \{2, \ldots, k+1\}$, the execution of the simulator must contain a query $\overline{q}_{i,j}$ that is a prefix of $\overline{q}^{(\overline{bp}_d)}$ and that satisfies $bp(\overline{q}_{i,j}) = \overline{bp}_d$, $\pi_{\text{sn}}(\overline{q}_{i,j}) = (\ell^{(\overline{bp}_d)}, i)$, $\pi_{\text{msg}}(\overline{q}_{i,j}) = j$, and $g(i, ip(\overline{q}_{i,j})) = 1$. (If $g(i, ip(\overline{q}_{i,j}))$ would have been equal to 0, query $\overline{q}^{(\overline{bp}_d)}$ would have contained a prover message in session $(\ell^{(\overline{bp}_d)}, i)$ that is answered with ABORT, in contradiction to the fact that $\mathsf{accept}_{d,i}(\sigma, g, h)$ holds.) Since for every $j \in \{2, \ldots, k+1\}$ and for every $i_1, i_2 \in S$ we have that $ip(\overline{q}_{i_1,j}) = ip(\overline{q}_{i_2,j})$ (as queries $\overline{q}_{i,j}$ are all prefixes of $\overline{q}_\ell$ and $|ip(\overline{q}_{i_1,j})| = |ip(\overline{q}_{i_2,j})|$), we can set $\overline{ip}_j = ip(\overline{q}_{i,j})$. It follows that for every $j \in \{2, \ldots, k+1\}$, iteration-prefix $\overline{ip}_j$ corresponds to block-prefix $\overline{bp}_d$ (as queries $\overline{q}_{i,j}$ all have block-prefix $\overline{bp}_d$), and for every $i \in S$ we have that $g(i, \overline{ip}_j) = 1$.

Proof of Item 3: Let $i \in S$ and recall that in addition to the fact that $\mathsf{accept}_{d,i}(\sigma, g, h)$ holds, we have that $\mathsf{useful}_{d,i}(\sigma, g, h)$ does not hold. Notice that the only reason for which $\mathsf{useful}_{d,i}(\sigma, g, h)$ can be false (i.e., the d^{th} block-prefix is not i-useful), is that Condition 1 in Definition 5.4 is violated by $\text{EXEC}_x(\sigma, g, h)$. (Recall that $\mathsf{accept}_{d,i}(\sigma, g, h)$ holds, and so Condition 2 in Definition 5.4 is indeed satisfied by query $\overline{q}_{i,k+1}$ (as defined above): This query corresponds to block-prefix $\overline{bp}_d$, satisfies $\pi_{\text{sn}}(\overline{q}_{i,k+1}) = (\ell^{(\overline{bp}_d)}, i)$, $\pi_{\text{msg}}(\overline{q}_{i,k+1}) = k+1$, $g(i, ip(\overline{q}_{i,k+1})) = 1$, and is answered with ACCEPT.)

For Condition 1 in Definition 5.4 to be violated, there must exists a $j \in \{2, \ldots, k{+}1\}$, with two ip-different queries, $\overline{q}_1$ and $\overline{q}_2$, that correspond to block-prefix $\overline{bp}_d$, satisfy $\pi_{\text{sn}}(\overline{q}_1) = \pi_{\text{sn}}(\overline{q}_2) = (\ell^{(\overline{bp}_d)}, i)$, $\pi_{\text{msg}}(\overline{q}_1) = \pi_{\text{msg}}(\overline{q}_2) = j$, and $g(i, ip(\overline{q}_1)) = g(i, ip(\overline{q}_2)) = 1$. Since, by definition, two queries are considered ip-different only if they differ in their iteration-prefixes, we have that there exist two different iteration-prefixes $\overline{ip}(\overline{q}_1)$ and $\overline{ip}(\overline{q}_2)$ (of the same length) that correspond to block-prefix $\overline{bp}_d$ and satisfy $g(i, \overline{ip}(\overline{q}_1)) = g(i, \overline{ip}(\overline{q}_2)) = 1$. Since iteration-prefixes $\overline{ip}_2, \ldots, \overline{ip}_{k+1}$ (from Item 2 above) are all of distinct length, and since the only iteration-prefix in $\overline{ip}_2, \ldots, \overline{ip}_{k+1}$ that can be equal to either $\overline{ip}(\overline{q}_1)$ or $\overline{ip}(\overline{q}_2)$ is $\overline{ip}_j$ (note that this is the only iteration-prefix having the same length as $\overline{ip}(\overline{q}_1)$ and $\overline{ip}(\overline{q}_2)$), then it must be the case that at least one of $\overline{ip}(\overline{q}_1), \overline{ip}(\overline{q}_2)$ is different from all of $\overline{ip}_2, \ldots, \overline{ip}_{k+1}$ (recall that $\overline{ip}(\overline{q}_1)$ and $\overline{ip}(\overline{q}_2)$ are different, which means that they cannot be both equal to $\overline{ip}_j$). In particular, for every $i \in S$ (that satisfies $\mathsf{useful}_{d,i}(\sigma, g, h)$ & $\mathsf{accept}_{d,i}(\sigma, g, h)$), there exists at least one (extra) iteration-prefix, $\overline{ip}^{(i)} \in \{\overline{ip}(\overline{q}_1), \overline{ip}(\overline{q}_2)\}$, that corresponds to block-prefix $\overline{bp}_d$, differs from $\overline{ip}_j$ for every $j \in \{2, \ldots, k+1\}$, and satisfies $g_2(i, \overline{ip}^{(i)}) = 1$.

This completes the proof of Claim 5.18. ■

Recall that the hash function g_2 is chosen at random from a $t_S(n)$-wise independent family. Since for every pair of different iteration-prefixes the function g_2 will have different inputs, then g_2 will have independent outputs when applied to different iteration-prefixes (since no more than $t_S(n)$ queries are made by the simulator). Similarly, for every pair of different $i, i' \in S$, g_2 will have different input, and thus independent output. Put in other words, all outcomes of g_2 that are relevant to block-prefix $\overline{bp}_d$ are independent of each other. Since a uniformly chosen g_2 will output 1 with probability $n^{-1/2k}$, we may view every application of g_2 on iteration-prefixes that correspond to $\overline{bp}_d$ as an independently executed experiment that succeeds with probability $n^{-1/2k}$.[34]

Using Claim 5.18.1 (i.e., Item 1 of Claim 5.18), the applications of g_2 which are relevant to sessions $\{(\ell^{(\overline{bp}_d)}, i)\}_{i \in S}$ can be viewed as a sequence of at most k^{c+1} experiments (corresponding to at most k^{c+1} different iteration-prefixes). Each of these experiments consists of $|S|$ independent sub-experiments (corresponding to the different $i \in S$), and each sub-experiment succeeds with probability $n^{-1/2k}$. Claim 5.18.2 now implies that at least k of the above experiments will fully succeed (that is, all of their sub-experiments will succeed), while Claim 5.18.3 implies that for every

[34]We may describe the process of picking $g_2 \overset{\text{R}}{\leftarrow} G$ as the process of independently letting the output of g_2 be equal to 1 with probability $n^{-1/2k}$ (each time a new input is introduced). Note that we will be doing so only for inputs that occur after block-prefix $\overline{bp}_d$ has been determined (as, in the above case, all inputs for g_2 are iteration-prefixes that correspond to block-prefix $\overline{bp}_d$, and such iteration-prefixes will occur only after $\overline{bp}_d$ has already been determined).

$i \in S$ there exists an additional successful sub-experiment (that is, a sub-experiment of one of the $k^{c+1}-k$ remaining experiments). Using the fact that the probability that a sub-experiment succeeds is $n^{-1/2k}$, we infer that the probability that an experiment fully succeeds is equal to $(n^{-1/2k})^{|S|}$. In particular, the probability in Eq. (11) is upper bounded by the probability that the following two events occur (these events correspond to Claims 5.18.2 and 5.18.3 respectively):

Event 1: *In a sequence of (at most k^{c+1}) experiments, each succeeding with probability $(n^{-1/2k})^{|S|}$, there exist k successful experiments.* (The success probability corresponds to the probability that for every $i \in S$, we have $g_2(i, \overline{ip}_j) = 1$ (see Claim 5.18.2).)

Event 2: *For every one out of $|S|$ sequences of the remaining (at most $k^{c+1}-k$) sub-experiments, each succeeding with probability $n^{-1/2k}$, there exists at least one successful experiment.* (In this case, the success probability corresponds to the probability that iteration-prefix $\overline{ip}^{(i)}$ satisfies $g_2(i, \overline{ip}^{(i)}) = 1$ (see Claim 5.18.3).)

For $i \in |S|$ and $j \in [k^{c+1}]$, let us denote the success of the i^{th} sub-experiment in the j^{th} experiment by $\chi_{i,j}$. By the above discussion for every i, j, the probability that $\chi_{i,j}$ holds is $n^{-1/2k}$ (independently of other $\chi_{i,j}$'s). We now have that, for Event 1 above to suceed, there must exists a set of k experiments, $K \subseteq [k^{c+1}]$, so that for all $(i,j) \in S \times K$, the event $\chi_{i,j}$ holds. For Event 2 to suceed, it must be the case that, for every $i \in S$, there exist one additional experiment (i.e., some $j \in [k^{c+1}] \setminus K$) so that $\chi_{i,j}$ holds. It follows that Eq. (11) is upper bounded by:

$$\begin{aligned}
&\sum_{\substack{K \subseteq [k^{c+1}] \\ |K| = k}} \Pr\Big[\forall j \in K,\ \forall i \in S \text{ s.t. } \chi_{i,j}\Big] \cdot \Pr\Big[\forall i \in S,\ \exists j \in [k^{c+1}] \setminus K \text{ s.t. } \chi_{i,j}\Big] \\
&= \binom{k^{c+1}}{k} \cdot \left(\left(n^{-\frac{1}{2k}}\right)^{|S|}\right)^k \cdot \left(1 - \left(1 - n^{-\frac{1}{2k}}\right)^{k^{c+1}-k}\right)^{|S|} \\
&< \left(k^{c+1}\right)^k \cdot \left(\left(n^{-\frac{1}{2k}}\right)^{|S|}\right)^k \cdot \left(k^{c+1} \cdot n^{-\frac{1}{2k}}\right)^{|S|} \qquad (12) \\
&= \left(k^{c+1}\right)^{k+|S|} \cdot \left(n^{-\frac{1}{2k}}\right)^{k\cdot|S|+|S|} \\
&= \left(k^{c+1}\right)^{k+|S|} \cdot \left(n^{-\frac{1}{4k}}\right)^{|S|} \left(n^{-(\frac{1}{2}+\frac{1}{4k})}\right)^{|S|} \\
&< \left(n^{-(\frac{1}{2}+\frac{1}{4k})}\right)^{|S|} \qquad (13)
\end{aligned}$$

where Eq. (12) holds whenever $k^{c+1}-k = o(n^{1/2k})$ (which is satisfied if $k = o(\frac{\log n}{\log\log n})$), and Eq. (13) holds whenever $(k^{c+1})^{k+|S|} \cdot (n^{-1/4k})^{|S|} < 1$ (which is satisfied if both $|S| > k$ and $k = o(\frac{\log n}{\log\log n})$). This means that Eq. (11) is upper bounded by $(n^{-(1/2+1/4k)})^{|S|}$, and the proof of Lemma 5.17 is complete. ■

Using Lemma 5.17, we upper bound Eq. (10) by

$$\begin{aligned}
t_S(n) \cdot \binom{n}{\frac{n^{1/2}}{4}} \cdot \left(n^{-(\frac{1}{2}+\frac{1}{4k})}\right)^{\frac{n^{1/2}}{4}} &< t_S(n) \cdot \left(\frac{4 \cdot e \cdot n}{n^{1/2}}\right)^{\frac{n^{1/2}}{4}} \cdot \left(n^{-(\frac{1}{2}+\frac{1}{4k})}\right)^{\frac{n^{1/2}}{4}} \\
&= t_S(n) \cdot \left(\frac{4 \cdot e}{n^{1/4k}}\right)^{\frac{n^{1/2}}{4}} \\
&< t_S(n) \cdot 2^{-\frac{n^{1/2}}{4}} \qquad (14)
\end{aligned}$$

where Inequality 14 holds whenever $8 \cdot e < n^{1/4k}$ (which holds for $k < \frac{\log n}{4\cdot(3+\log e)}$). This completes the proof of Lemma 5.7 (since $\mathrm{poly}(n) \cdot 2^{-\Omega(n^{1/2})}$ is negligible).

Acknowledgements

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

References

[1] N. Alon, L. Babai, and A. Itai A Fast and Simple Randomized Parallel Algorithm for the Maximal Independent Set Problem. *Journal of ALgorithms*, 7, pages 567–583, 1986.

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

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

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

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

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

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

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

[9] M.N. Wegman, and J.L. Carter. New Hash Functions and Their Use in Authentication and Set Equality. *JCSS 22*, 1981, pages 265–279.

[10] B. Chor, and O. Goldreich On the power of Two-Point Based Sampling. *Jour. of Complexity*, Vol. 5, 1989, pages 96-106.

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

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

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

[14] U. Feige. Alternative Models For Zero-Knowledge Interactive Proofs. Ph.D. thesis, Weizmann Institute of Science, 1990.

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

[16] O. Goldreich. Foundations of Cryptography - Basic Tools. *Cambridge University Press,* 2001.

[17] 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.

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

[19] 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.

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

[21] 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.

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

[23] 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.

[24] A. Joffe. On a set of Almost Deterministic k-Independent Random Variables. *The annals of Probability*, 1974, Vol. 2, No. 1, pages 161-162.

[25] J. Kilian and E. Petrank. Concurrent and Resettable Zero-Knowledge in Poly-logarithmic Rounds. In *33rd STOC*, pages 560–569, 2001.

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

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

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

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

Appendix

A Alternative Description of the Recursive Schedule

The schedule consists of n^2 sessions (each session consists of $k+1$ prover messages and $k+1$ verifier messages). It is defined recursively, where for each $m \leq n^2$, the schedule for sessions $i_1, \ldots, i_m$ (denoted $\mathcal{R}_{i_1,\ldots,i_m}$) proceeds as follows:

1. If $m \leq n$, execute sessions $i_1, \ldots, i_m$ sequentially until they are all completed;
2. Otherwise, For $j = 1, \ldots, k+1$:
   (a) For $\ell = 1, \ldots, n$:
      i. Send the j^{th} verifier message in session i_ℓ (i.e., $\mathtt{v}_j^{(i_\ell)}$);
      ii. Send the j^{th} prover message in session i_ℓ (i.e., $\mathtt{p}_j^{(i_\ell)}$);

   (b) If $j < k+1$, invoke a recursive copy of $\mathcal{R}_{i_{(n+(j-1)\cdot t+1)},\ldots,i_{(n+j\cdot t)}}$ (where $t \overset{\text{def}}{=} \lfloor \frac{m-n}{k} \rfloor$); (Sessions $i_{(n+(j-1)\cdot t+1)}, \ldots, i_{(n+j\cdot t)}$ are the next t remaining sessions out of $i_1, \ldots, i_m$.)

B Solving the Recursion

Claim B.1 *Suppose that Eq. (6) holds. Then for all sufficiently large n, $W(n^2) > n^c$.*

Proof: By applying Eq. (6) iteratively $\log_k(n-1)$ times, we get:

$$\begin{aligned} W(n^2) &\geq \left(k^{c+1}\right)^{\log_k(n-1)} \cdot W(n) \\ &\geq \left(k^{c+1}\right)^{\log_k(n-1)} \cdot 1 \\ &= (n-1)^{c+1} \\ &> n^c \qquad (15) \end{aligned}$$

where Eq. (15) holds for all sufficiently large n. □