The Round-Complexity of Black-Box Concurrent Zero-Knowledge

Thesis for the Ph.D. Degree

by

Alon Rosen

Under the Supervision of
Professors Oded Goldreich and Moni Naor
Department of Computer Science and Applied Mathematics
The Weizmann Institute of Science

Submitted to the Feinberg Graduate School of
the Weizmann Institute of Science
Rehovot 76100, Israel

June 26, 2003

Abstract

Zero-knowledge proof systems are interactive protocols that enable one party, called the prover, to convince another party, called the verifier, in the truth of a statement without revealing anything beyond the validity of the assertion being proved. Besides being fascinating on their own right, zero-knowledge proofs serve as an extremely useful tool for the realization of many cryptographic tasks.

The original setting in which zero-knowledge proofs were investigated consisted of a single prover and verifier executing only one instance of the protocol at a time. A more realistic setting, especially in the era of the Internet, is one that allows the *concurrent* execution of zero-knowledge protocols.

The most common technique for proving the zero-knowledge property of a protocol is called *black-box simulation*. As it turns out, the usage of black-box simulation in the concurrent setting introduces many technical difficulties. The only known way to enable black-box simulation in the concurrent setting is to significantly increase the protocol's *round-complexity* (i.e., the number of messages exchanged in the protocol).

It has already been shown that for every language outside $\mathcal{BPP}$ there is no 4-round protocol whose concurrent zero-knowledge ($c\mathcal{ZK}$) property is proved via black-box simulation. In contrast, the most efficient $c\mathcal{ZK}$ protocol that we know of uses $\tilde{O}(\log^2 n)$ rounds, where n is a "security" parameter that is polynomially related to the number of concurrent executions. In this thesis we close the gap between these upper and lower bounds. Our main results are:

- Any $c\mathcal{ZK}$ proof system for a language outside $\mathcal{BPP}$, whose $c\mathcal{ZK}$ property is proved using black-box simulation, requires $\Omega(\log n/\log\log n)$ rounds of interaction.

- Assuming that perfectly hiding commitments exist, every language in $\mathcal{NP}$ has a $c\mathcal{ZK}$ proof system with $O(\alpha(n)\cdot\log n)$ rounds of interaction, where $\alpha(n)$ is any super-constant function. Moreover, the $c\mathcal{ZK}$ property of this proof system is proved using black-box simulation.

The above two results complement each other and yield an (almost) full characterization of the round-complexity of black-box $c\mathcal{ZK}$ protocols.

Acknowledgements

I would like to express my deepest gratitude to my thesis advisors Oded Goldreich and Moni Naor. Oded and Moni are very special individuals, and each one of them has affected my scientific development in his own distinctive way. It has been a privilege to study under their guidance.

Oded has agreed to take me under his supervision when Moni has left for a two year sabbatical. Soon thereafter, he suggested me a project to work on. Amazingly enough, Oded's initial suggestion turned out to yield all the results in this thesis. Oded has invested an unparalleled amount of time and effort to supply me with invaluable advice about technical issues, as well as on the way my work should be presented. There is no doubt that Oded has had a significant impact both on my scientific taste and on my approach to research. For that and for his devotion I thank him very much.

I am deeply indebted to Moni for treating me as a peer from the first moment. The credit he has given me has greatly contributed to my self-confidence as a researcher. Moni has always been available to discuss scientifical issues and has continuously provided me extremely interesting ideas for research. Moni's attitude to research, as well as his approach to people, make him an ideal advisor and a person that is fun to work with. I consider myself lucky for having spent so much time in the presence of someone as resourceful as Moni. I know that I have benefitted from it a lot.

I am most grateful to Shafi Goldwasser. It would be hard to underestimate the contribution of Shafi's advice and encouragement to my development as a researcher. Most people do not have the privilege of having two advisors. With Shafi around, I often felt as if I had three.

Of all the people I have interacted with during my studies, I am especially thankful to Ran Canetti, Omer Reingold and Ronen Shaltiel. Ran and Omer have been continuously guiding my steps from the earliest stages of my studies. Ronen has done so from a later stage. Thanks to Ran for being such a great collaborator, to Omer for his kindness and support from day one, and to Ronen for persistently sticking to the (hopeless) task of making me learn from his mistakes.

I would like to thank the faculty members at the Weizmann Insititute for making it, together with the students, such a great place to study in. Thanks to Itai Benjamini, Uri Feige, Ran Raz and Adi Shamir for their enlightening courses. Special thanks to Uri, Itai and Ran for conveying me their perspective about life as a researcher through many interesting conversations. Many thanks are due to my fellow students at the Weizmann Institute for having shared their knowledge and ideas with me. Special thanks to my closest collaborators: Danny Harnik, Yehuda Lindell, Rafael Pass and Boaz Barak. I enjoyed very much working with you and have learnt a great deal from it. I hope we will have the opportunity to work together in the future. I also thank Adi Akavia, Tzvika Hartman, Robi Krauthgamer, Michael Langberg, Kobbi Nissim, Eran Ofek, Benny Pinkas, Yoav Rodeh and Udi Wieder for many fruitful discussions.

I would like to thank my co-authors to the results that make up this thesis. Chapter 3 was done jointly with Ran Canetti, Joe Kilian and Erez Petrank [9, 10]. Chapter 4 is joint with Manoj

Prabhakaran and Amit Sahai [41]. I would especially like to mention Joe for his generosity and for contributing so many key ideas to the field of concurrent zero-knowledge.

I am indebted to the people at the IBM T.J. Watson research center for an enjoyable summer. Thanks to Marc Fischlin, Rosario Gennaro, Shai Halevi, Nick Howgrave-Graham, Eyal Kushilevitz and Tal Rabin. Special thanks to Tal for being so great and for her patience during a long period of uncertainty. I would like to thank Cynthia Dwork for her warm hospitality during a visit at MICROSOFT. Thanks also to Dalit, Yael, Efrat and Avner for the entertaining dinners during my two-week visit at IBM Almaden and Stanford.

Most importantly, I would like to thank the members of my family for their love and encouragement troughout the years. My parents Ora and Kalman, my brothers Erez and Oren, my wife Vered and my sons Yoav and Itamar. I wish to express the deepest love to Vered, Yoav and Itamar. Being in their presence is a wonderful experience and I consider it the greatest privilege of them all. Finally, thanks to Rivka and Yossi for their much appreciated help with raising the kids.

Contents

Chapter 1

Introduction

The past two and a half decades have witnessed an unprecedented progress in the field of Cryptography. During these years, many cryptographic tasks have been put under rigorous treatment and numerous constructions realizing these tasks have been proposed. By now, the scope of cryptographic constructions ranges from simple schemes that realize "atomic" tasks such as authentication, identification, encryption and digital signatures, to fairly complex protocols that realize "high-level" tasks such as general secure two-party computation (the latter being so general that it captures almost any conceivable cryptographic task in which two mutually distrustful parties interact).

The original setting in which cryptographic protocols were investigated consisted of a single execution of the protocol at a time (this is the so called *stand-alone* setting). A more realistic setting, especially in the era of the Internet, is one that allows the *concurrent* execution of protocols. In the concurrent setting many protocols are executed at the same time, involving multiple parties that may be talking with the same (or many) other parties simultaneously. The concurrent setting presents the new risk of a coordinated attack in which an adversary controls many parties, interleaving the executions of the protocols while trying to extract knowledge based on the existence of multiple concurrent executions. It would be most desirable to have cryptographic protocols retain their security properties even when executed concurrently. This would enable the realization of cryptographic tasks in a way that preserves security in a setting that is closer to the "real world".

Unfortunately, security of a specific protocol in the stand-alone setting does not necessarily imply its security in the (more demanding) concurrent setting. It is thus of great relevance to examine whether the original feasibility results for cryptographic protocols still hold when many copies of the protocol are executed concurrently.

1.1 Zero-Knowledge Proof Systems

In the course of developing tools for the design of complex cryptographic tasks, many innovative notions have emerged. One of the most basic (and important) examples for such notions is the one of *Zero-Knowledge Interactive Proofs*. Interactive proofs, introduced by Goldwasser, Micali and Rackoff [28], are efficient protocols that enable one party, known as the *prover*, to convince another party, known as the *verifier*, of the validity of an assertion. In the process of proving the assertion, the prover and the verifier exchange messages for a predetermined number of rounds. Throughout the interaction, both prover and verifier may employ proabilistic strategies and toss coins in order to determine their next message. At the end of the process, the verifier decides whether to accept or reject the proof based on his view of the interaction (as well as on his coin-tosses).

The basic requirement is that whenever the assertion is true, the prover always convinces the verifier (this is called the *completeness* condition), whereas if the assertion is false, then no matter what the prover does, the verifier is convinced with very small probability, where the probability is taken over the verifier's coin-tosses (this is called the *soundness* condition).

An interactive proof is said to be *zero-knowledge* ($\mathcal{ZK}$) if it yields nothing beyond the validity of the assertion being proved. This is formalized by requiring 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*). The idea behind this definition is that whatever an adversary verifier might have learned from interacting with the prover, he could have actually learned by himself (by running the simulator).

The concept of zero-knowledge was originally introduced by Goldwasser, Micali and Rackoff [28]. The generality of $\mathcal{ZK}$ has been demonstrated by Goldreich, Micali and Wigderson [25], who showed that every language in $\mathcal{NP}$ can be proved in $\mathcal{ZK}$, provided that one-way functions exist (cf. Naor [37], Håstad et al. [32]). Since then, $\mathcal{ZK}$ proof systems have turned out to be an extremely useful tool in the realization of increasingly many cryptographic tasks.

1.1.1 Applications of $\mathcal{ZK}$ in Cryptography

We illustrate the power of $\mathcal{ZK}$ proofs by giving two examples of their usefulness in Cryptography. The first example consists of the application of $\mathcal{ZK}$ proofs to the construction of *identification schemes* (due to Feige, Fiat and Shamir [18]). The second example demostrates the applicability of $\mathcal{ZK}$ to the task of cryptographic protocol design (due to Goldreich, Micali and Wigderson [25, 26]).

Identification schemes: One of the earliest examples for the applicability of $\mathcal{ZK}$ is the construction of identification schemes [18]. Such schemes are useful in a scenario where one party, Alice, wishes to repeatedly identify herself to another party, Bob. The presumption is that Alice and Bob are acquainted with one another, and that Bob possesses some (authenticated) public information that is associated with a secret value known only to Alice. In order to identify herself, Alice invokes an *identification protocol* and uses her secret information to convince Bob that she is indeed Alice. The requirement is that Alice will always be able to convince Bob, and that nobody else can fool Bob into believing that she/he is Alice (even after polynomially many invocations of the protocol).

A naïve solution to this problem would be to let Alice choose a secret password s, and to publish the value of $f(s)$, where f is some one-way function (i.e., a function that is "easy" to evaluate on all inputs but "hard" to invert on most inputs). When Alice wishes to identify herself to Bob, she simply sends her password s to Bob who computes $y = f(s)$ and compares it to the value that Alice has published. The security of the resulting scheme is supposedly guaranteed by the one-wayness of f (since it is hard to find the value of s given the value of $f(s)$). Unfortunately, this suggestion suffers from a severe drawback: Any adversary who impersonates Bob can obtain the value of s and later use it in order to identify himself as Alice.

The solution is to use $\mathcal{ZK}$ proofs (or actually a variant of $\mathcal{ZK}$ proofs, called *$\mathcal{ZK}$ proofs of knowledge* [18, 4]). Rather than sending over the value of s, Alice will now prove in $\mathcal{ZK}$ that she knows the preimage of $f(s)$. (Here one uses the fact that, once f is specified, proving the knowledge of a preimage under f is an $\mathcal{NP}$ assertion.) The $\mathcal{ZK}$ property then guarantees that, while Bob has been convinced of the validity of the assertion, no impersonator can obtain the value of s and misuse it at a later stage.

It should be noted that $\mathcal{ZK}$ seems a much stronger property than what is required here. In particular, besides being unable to impersonate Alice, an adversary interacting with Alice will not be able to do *anything* that he could have not been able to do prior to the interaction.

Enforcing honest behaviour on protocol participants: Perhaps the most important example for the power of $\mathcal{ZK}$ is the role they play in enforcing "honest" behaviour on parties that participate in a given protocol [25, 26]. Specifically, $\mathcal{ZK}$ proofs enable participants in a protocol to prove that their actions are indeed consistent with the protocol's prescribed instructions without taking any risk of compromising the security of their own secret information (e.g., cryptographic keys). This approach is made possible through the usage of $\mathcal{ZK}$ proofs for $\mathcal{NP}$ and proceeds according to the following two-step paradigm:

Semi-honest protocol design: Design a protocol Π_1 that is secure assuming that the adversaries follow the protocol's prescribed instructions (a.k.a. "semi-honest" adversaries).

Compilation to malicious model: "Compile" Π_1 into a protocol Π_2 that withstands *any* malicious behaviour by making the parties prove in $\mathcal{ZK}$ that they faithfully follow the instructions of Π_1. Here one relies on the fact that, once Π_1 is specified, consistency of the party's actions with the protocol's instructions is an $\mathcal{NP}$-statement, and so by [25] can be proved in $\mathcal{ZK}$.

Needless to say that this approach greatly facilitates the task of protocol design, since one only has to consider adversaries that follow the protocol's instructions. In fact, in many cases it is not clear how one could have approached the above task in a different way, since one would have to take into account any attack that an adversary might come up with.

1.1.2 Concurrent Composition of $\mathcal{ZK}$

The wide applicability of $\mathcal{ZK}$ proofs makes them a very useful "test case" for examining the behavior of cryptographic protocols in the concurrent setting. On the one hand, many of the difficulties that arise in the concurrent setting already appear in the (relatively basic) case of $\mathcal{ZK}$. On the other hand, positive solutions for the case of $\mathcal{ZK}$ may translate to positive solutions for much more complex cryptographic tasks (that use $\mathcal{ZK}$ protocols as a subroutine).

Concurrent composition of $\mathcal{ZK}$ protocols was first considered by Feige [17]. A more extensive treatment was given by Dwork, Naor and Sahai [15], who also argued that the task of proving the $\mathcal{ZK}$ property of a protocol in the concurrent setting might encounter technical difficulties if approached in the straightforwad manner. Since then, concurrent composition of $\mathcal{ZK}$ protocols has received a considerable amount of attention (cf. [17, 15, 16, 36, 8, 42, 35]).

The scenario that is typically considered in the context of $\mathcal{ZK}$ involves a single (or many) honest provers that are running many concurrent executions of the same $\mathcal{ZK}$ protocol. The honest prover is trying to protect itself from a malicious adversary that controls a subset (or all) the verifiers it is interacting with. Since it seems unrealistic (and certainly undesirable) for honest provers to coordinate their actions so that security is preserved, one must assume that in each instance of the protocol the honest prover acts independently.

A $\mathcal{ZK}$ protocol is said to be *concurrent zero-knowledge* ($c\mathcal{ZK}$) if it remains zero-knowledge under concurrent composition. Recall that in order to demonstrate the $\mathcal{ZK}$ property of a protocol it is required to demonstrate that the view of every probabilistic polynomial-time adversary interacting with the prover can be simulated in probabilistic polynomial-time. 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 becomes considerably more complicated.

1.1.3 The round-complexity of black-box $c\mathcal{ZK}$

The most common technique for proving the $\mathcal{ZK}$ property of a protocol is called *black-box simulation.* (A black-box simulator is a simulator that has only black-box access to the adversary verifier.) A protocol whose $\mathcal{ZK}$ property is proved using black-box simulation is called *black-box $\mathcal{ZK}$.*

While black-box simulation has proved to be a very useful tool for proving security in the stand-alone setting, it does not seem to be suitable for use in the concurrent setting. The only known way to enable black-box simulation in the concurrent setting is to significantly increase the number of messages exchanged in the protocol [42] (a.k.a. the protocol's *round-complexity*). In particular, whereas the number of messages exchanged in the original (stand-alone) $\mathcal{ZK}$ protocols was a constant, the number of messages in the new $c\mathcal{ZK}$ protocols is required to grow with the number of concurrent executions.

In the context of $c\mathcal{ZK}$, the round-complexity of a protocol is measured as a function of some predetemined "security" parameter $n \in N$. The requirement is that the protocol will remain secure as long as the number of concurrent executions is bounded by some polynomial in n (we stress that the protocol is constructed *before* the polynomial bound is determined). We regard a protocol as having "high" round complexity if the number of rounds in this protocol depends on the value of n. This should be contrasted to constant-round protocols in which the number of messages is *not* required to increase as n grows.

Since the number of messages exchanged is perhaps the most important efficiency consideration for interactive protocols, it is natural to ask whether "high" round-complexity is inherent to the notion of $c\mathcal{ZK}$. The main objective of this thesis is to investigate this question in the context of black-box simulation. Arguably, this is not only an issue of theoretical interest, but rather has also significant practical consequences. Having "low" round-complexity is a fundamental property of any interactive protocol (especially if this protocol is a key ingredient for many other cryptographic protocols). Our main conclusions are:

- Any $c\mathcal{ZK}$ proof system for a language outside $\mathcal{BPP}$, whose $c\mathcal{ZK}$ property is proved using black-box simulation, requires $\Omega(\log n/\log\log n)$ rounds of interaction.

- Assuming that perfectly hiding commitments exist, every language in $\mathcal{NP}$ has a $c\mathcal{ZK}$ proof system with $O(\alpha(n)\cdot\log n)$ rounds of interaction, where $\alpha(n)$ is any super-constant function. Moreover, the $c\mathcal{ZK}$ property of this proof system is proved using black-box simulation.

The above two results complement each other and yield an (almost) full characterization of the round-complexity of black-box $c\mathcal{ZK}$ protocols.

It should be noted that subsequently to our work, new *non black-box* simulation techniques were devised [2]. These new techniques carry a great promise and give hope that a constant-round $c\mathcal{ZK}$ protocol can indeed be constructed. However, the results achieved so far in the context of concurrent composition are only partial (see Section 1.2.4 and Chapter 7 for more details).

1.2 Why Black-Box Simulation is Problematic

We now turn to describe the main technical difficulties that are encountered when trying to establish the $c\mathcal{ZK}$ property of a protocol via black-box simulation. We start by giving a high-level overview of a "typical" construction of a $\mathcal{ZK}$ protocol for $\mathcal{NP}$. Before we proceed, we (informally) define the notion of *commitment schemes* [37], which are a central tool in the construction.

Commitment schemes: Commitment schemes are the "digital" analog of sealed envelopes. They are used to enable a party, known as the *sender*, to commit itself to a value while keeping it secret from the *receiver* (this property is called *hiding*). Furthermore, the commitment is *binding* in the sense that in a later stage, when the commitment is opened, it is guaranteed that the "opening" can yield only a single value determined in the committing phase. For our purposes, it will be convenient to think of commitments as a non-interactive process in which the sender sends a single message to the receiver (somewhat analogously to an encryption scheme). The sender can then open the commitment by sending an additional message that reveals the value committed to.

1.2.1 The stand-alone case

On a high-level, the typical $\mathcal{ZK}$ protocol for $\mathcal{NP}$ is constructed by combining many atomic $\mathcal{ZK}$ protocols that proceed as follows.[1] Given a specific $\mathcal{NP}$ assertion, A, and a "proof" for the validity of this assertion (typically, an $\mathcal{NP}$-witness w for the validity of A), the prover uses his coin-tosses to generate two (related) messages M_0 and M_1 that depend on A and w. Letting P denote the prover and V denote the verifier, the protocol proceeds as follows:

$P \rightarrow V$: Commit to M_0 and M_1.

$V \rightarrow P$: Send a random $\sigma \in \{0,1\}$.

$P \rightarrow V$: Reveal M_σ.

The verifier accepts if and only if the revealed message M_σ is "valid" (i.e., if it passes a prespecfied validity inspection that is related to the commonly known assertion A). To insure that the resulting protocol is indeed an interactive proof, it is required that M_0, M_1 satisfy the following properties:

- If A is true, it is possible to make sure that both M_0 and M_1 are "valid".
- If A is false, then no matter what P does, either M_0 or M_1 is "invalid".

Thus if A is true then V always accepts, whereas if A is false then V accepts with probability at most $1/2$. (Here we rely on the fact that the commitment is binding and so P cannot reveal a value that is different than what he has committed to.) To insure that the protocol is also $\mathcal{ZK}$, the following property is required:

- Given the value of σ, it is always possible make sure that M_σ is "valid".

Indeed, the soundness property of the protocol heavily relies on the fact that P does not know the value of σ before the protocol starts (and so V can always "surprise" P by choosing σ at random). Otherwise, P (knowing σ in advance) would have always been able to make V accept in the protocol (regardless of whether A is true or not).

However, knowing σ in advance is the key for proving the $\mathcal{ZK}$ property of the protocol. Consider an adversary verifier V^* that is trying to extract knowledge from the interaction (by possibly deviating from the honest verifier strategy). All that has to be done in order to simulate the view of V^* is to let the simulator "guess" the value of σ in advance and generate M_0, M_1 so that M_σ is valid. The simulator can then "feed" V^* with a commitment to M_0, M_1 and obtain the value of some σ' that depends on this commitment. If indeed $\sigma' = \sigma$ then the simulator has suceeded

[1] As a side remark, we note that the type of protocols considered here resemble Blum's protocol for Hamiltonicity [6] (see Construction 4.3.1 in Page 68), and not the protocol by Goldreich, Micali and Wigderson for Graph 3-coloring [25].

in his task and will output a "valid" transcript in which V^* accepts. The hiding property of the commitment guarantees us that, no matter what is the strategy applied by V^*, the probability that $\sigma' = \sigma$ is $1/2$. In particular, after two attempts the simulator is expected to succeed in its task. Notice that the resulting simulator is "black-box" in the sense that the only way in which V^*'s strategy is used is through the examination of its input/output behaviour.

Reducing the error via parallel repetition: To make the above protocol useful, however, one must make sure that whenever A is false, V accepts only with small probability (rather than $1/2$). To achieve this, the atomic protocol described above is repeated many (say, k) times independently. V accepts if and only if it has accepted in all k repetitions. The probability of having V accept a false statement is now reduced to $1/2^k$ (by the independence of the repetitions).

The straighforward way to conduct the repetitions would be to perform the atomic protocols *sequentially* (i.e., one protocol after the other, see Figure 1.1.a). This approach suffers from the drawback that the resulting protocol has a fairly high round-complexity. To overcome this problem, the repetitions may be conducted in *parallel* (i.e., the j^{th} message of the atomic protocol is sent together in all the k repetitions, see Figure 1.1.b).

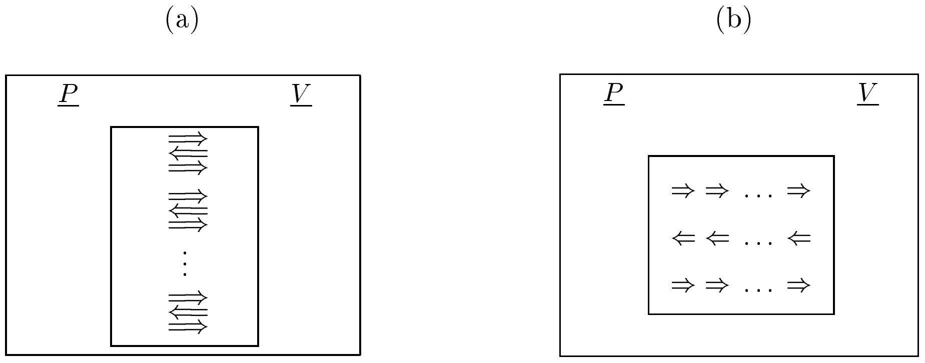


Figure 1.1: Sequential and parallel repetition.

Unfortunately, repeating the protocol many times in parallel brings up the following difficulty. Whereas in the case of a single execution, the probability that the $\mathcal{ZK}$ simulator "guesses" the value of σ correctly is at least $1/2$, the probability that he does so *simultaneously* for all k repetitions is $1/2^k$. For large k, this probability will be very small and might cause the simulator to run for too long. Thus, it is not clear that the $\mathcal{ZK}$ property of the protocol is preserved.

The solution to this problem is to let the verifier commit to all his "challenges" in advance. Specifically, consider the following protocol [23]:

$V \rightarrow P$ (`v1`): Commit to random $\sigma_1, \ldots, \sigma_k \in \{0,1\}$.

$P \rightarrow V$ (`p1`): Commit to $(M_0^1, M_1^1), (M_0^2, M_1^2), \ldots, (M_0^k, M_1^k)$.

$V \rightarrow P$ (`v2`): Reveal $\sigma_1, \ldots, \sigma_k$.

$P \rightarrow V$ (`p2`): Reveal $M_{\sigma_1}^1, M_{\sigma_2}^2, \ldots, M_{\sigma_k}^k$

The verifier accepts if and only if for all j, the message M_σ^j is "valid". By the hiding property of the commitment used in (`v1`), we are guaranteed that when sending (`p1`), the prover P has "no idea" about the values of $\sigma_1, \ldots, \sigma_k$, and so the soundness of the original protocol is preserved.

To see that the resulting protocol is $\mathcal{ZK}$, consider the following simulation technique. Start by obtaining the (v1) message from the verifier V^*. Then, playing the role of the prover, generate a sequence of k pairs $\{M_0^j, M_1^j\}_{j=1}^k$ each containing "garbage" (i.e., not ncessarily "valid"). Feed V^* with the commitments to these pairs and obtain the values of $\sigma_1, \ldots, \sigma_k$. Once these values are obtained, "rewind" the interaction to Step (p1) and recompute the values of $\{M_0^j, M_1^j\}_{j=1}^k$ so that for all j, the value of $M_{\sigma_j}^j$ is "valid". Since we have not rewound past (v1) (and thus did not modify its value), and since the commitment used in (v1) is binding, we are guaranteed that when reaching (v2) for the second time, the revealed values of $\sigma_1, \ldots, \sigma_k$ are identical to the ones revealed in the first time (v2) was reached (here we also use the fact that the commitment used in (p1) is hiding and so V^* cannot distinguish a commitment to "garbage" from a commitment to "valid" $M_{\sigma_j}^j$'s). Using the values of the $M_{\sigma_j}^j$'s, the simulator can thus output a "valid" transcript in which V^* accepts, as required.

But what if V^* refuses to reveal some (or all) of the committed values in Step (v2)? (Recall that V^* may behave in any adversarial manner.) In such a case, the simulator does not obtain all of the values of $\sigma_1, \ldots, \sigma_k$ and will supposedly fail in its task. Luckily, if V^* deviates from his prescribed strategy and does not reveal some σ_j value in (v2), then the prover in the protocol is not obligated to continue in the interaction (in particular, it aborts all k repetitions altogether). Using this fact, it is then possible to show (with some compications) that the simulator can eventually succeed in obtaining *all* of the values $\sigma_1, \ldots, \sigma_k$ and thus complete its task (cf. [23]).

1.2.2 Composition versus Repetition

Notice that the above analysis heavily relies on the fact that P is conducting a *single* execution of a given protocol (in which the behaviour of V^* in all repetitions is "linked" together). A more realistic scenario involves a single (or many) honest provers that are running *many* executions (sessions) of the same $\mathcal{ZK}$ protocol, but are not allowed to "link" between the various executions. This is called protocol *composition*. As in the case of protocol repetition, in the case of protocol composition the honest prover is trying to protect itself from a malicious adversary V^* that controls a subset (or all) the verifiers it is interacting with. However, unlike the case of protocol repetition, the honest prover is not allowed to coordinate its actions between different executions. As a consequence, a verifier in one execution of the protocol is not held accountable of the "misbehaviour" of a verifier in another execution.

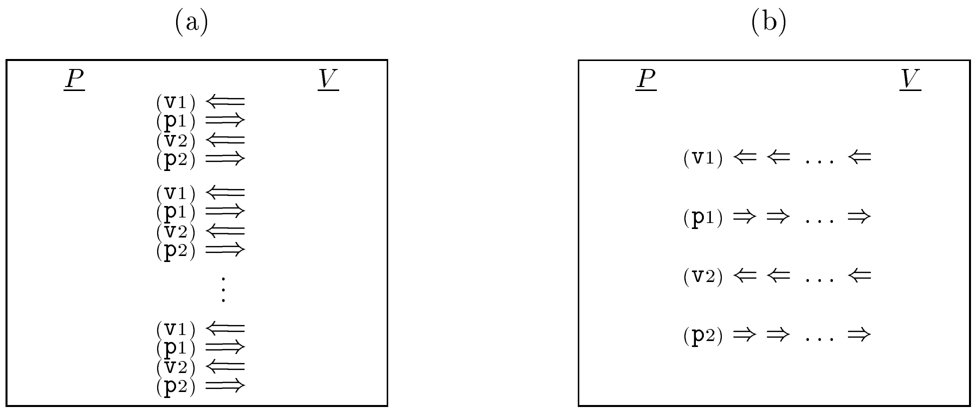


Figure 1.2: Sequential and parallel composition of a 4-round protocol.

Sequential and parallel composition: The most "basic" case of protocol composition is the one of sequential composition (Figure 1.2.a). This case has been treated in its most generality by Goldreich and Oren [27], who showed that *any* protocol that is (auxiliary input) $\mathcal{ZK}$ in a single execution will remain $\mathcal{ZK}$ under sequential composition. A more complicated case is the one of parallel composition (Figure 1.2.b). Here, a composition theorem is not known (and in fact does hold in general [24, 17]). Still, as recently shown by Goldreich [21], there exists a *specific* $\mathcal{ZK}$ protocol for $\mathcal{NP}$ (specifically, the protocol of [23]) that remains $\mathcal{ZK}$ under parallel composition.

Concurrent composition: A more general notion of protocol composition is the one of *concurrent* composition. Unlike the case of sequential and parallel composition (in which the scheduling of messages is defined in advance), the scheduling of messages in the case of concurrent composition is controlled by the adversary verifier, who determines the order in which the interleaved execution of the various sessions should be conducted. As observed by Dwork, Naor and Sahai [15], letting V^* control the scheduling and coordinate between sessions introduces technical difficulties that black-box simulation does not seem to handle very well. This is best seen by considering the following scheduling of messages for a 4-round protocol (suggested in [15]).

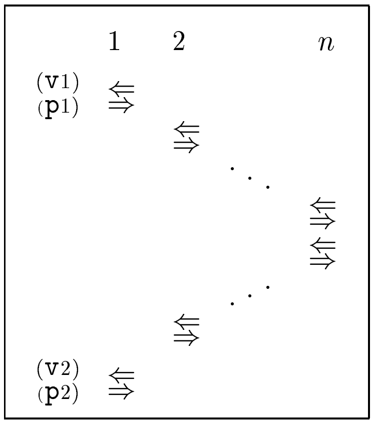


Figure 1.3: A concurrent schedule for n sessions of a 4-round protocol.

In this scheduling, the prover starts by sending the first two messages (i.e., (p1), (v1)) in all n sessions, only then proceeding to send the last two messages (i.e., (p2), (v2)) in the reverse order of sessions (i.e., starting at the n^{th} session and ending at the first). Suppose now that an adversary verifier V^* is sending messages according to the above schedule while applying the following "coordinated" strategy for all n sessions:

- V^* produces the various verifier messages according to the honest verifier strategy.
- The verifier coin-tosses used in a specific session depend on previous messages in the schedule.[2]
- Whenever V^* is not convinced in one session, he aborts the whole interaction altogether.[3]

[2]For example, V^* could obtain random coins by applying a $poly(n)$-wise independent hash function (or even a pseudorandom function) to the previous messages in the schedule. This would imply that the modification of even one of the previous messages, yields "fresh" randomness for the current session.

[3]Notice that this behaviour significantly deviates from the honest verifier strategy in which the decision of whether to reject or not is taken for each session independently of other sessions.

Since the view of V^* consists of the concurrent interaction in all n sessions in the schedule and since in each session V^* sends messages according to the honest verifier strategy, the simulator's task is to produce a transcript that contains n sessions in which V^* accepts (notice that the honest prover P would never cause V^* to reject, and so the simulator must do so as well).

The straightforward approach for simulation would be to use the 4-round protocol described above and let the simulator "rewind" the interaction with V^* in each session (just as it has done in the "stand-alone" case). However, by doing so the following problem is encountered. In order to suceed in the rewinding of the i^{th} session, the simulator must obtain the ($\mathtt{v2}$) message in this session. Since by the above scheduling, this message occurs after the end of session i' for all $i' > i$, the simulator will have to make V^* accept (and thus rewind) in all sessions $i' > i$ (otherwise, V^* would have aborted the interaction at the moment session i' ends, and the simulator would never obtain ($\mathtt{v2}$) in session i). Moreover, whenever the simulator rewinds session i, it modifies the value of ($\mathtt{p1}$) in this session. This will cause the randomness of all subsequent sessions (and so the verifier's "challenges" in sessions $i' > i$) to be modified. In particular, the simulation work done for all sessions $i' > i$ will be lost. To conclude:

- The simulator must rewind all n sessions.
- To rewind session i, the simulator must rewind session i' for all $i' > i$.
- By rewinding session i, the simulation work invested in sessions $i' > i$ is lost.

Denoting by $W(m)$ the amount of work that the simulator invests in m sessions, we obtain the recursion $W(m) \geq 2 \cdot W(m-1)$, which solves to $W(n) \geq 2^n$ (because $W(1) = 2$). This is clearly a too high running time for the simulator to afford.

The above example gives intuition to the difficulties that a "rewinding" simulator will encounter in the concurrent setting. At first glance it may seem that this still leaves open the possibility that an alternative black-box simulation technique might be found. Unfortunately, the technique of rewinding the interaction with V^* turns our to be inherent to black-box simulation. (Intuitively, this follows from the fact that rewinding is the only advantage that a black-box simulator might have over the honest prover.) Using this fact (and building on the work of Goldreich and Krawczyk [24]), Kilian, Petrank and Rackoff [36] have been able to trasform the above intuitive argment into an impossiblity result, and to prove that for every language outside $\mathcal{BPP}$ there is no 4-round protocol whose concurrent execution is simulatable in polynomial-time using black-box simulation. We note that our $\Omega(\log n/\log\log n)$ lower-bound on the round-complexity of black-box concurrent $\mathcal{ZK}$ (presented in Chpater 3) is obtained by employing a new, more sophisticated, scheduling of messages and by have the adversary verifier V^* occasionally *abort* sessions (i.e., refuse to decommit) depending on the history of the interaction.

1.2.3 The Richardson-Kilian protocol

For a while, it was not even clear whether there exists $c\mathcal{ZK}$ protocols for languages outside of $\mathcal{BPP}$. Several works have (successfully) attempted to overcome the above difficulties by augmenting the communication model with the so-called *timing assumption* [15, 16] or, alternatively, by using various set-up assumptions (such as the *public-key model* [8, 13]). The feasibility of $c\mathcal{ZK}$ in the plain model (i.e., without resorting to any set-up assumptions) has been established by Richardson and Kilian (RK for short) [42], who were the first to exhibit a family of $c\mathcal{ZK}$ protocols (parameterized by the number of rounds) for all languages in $\mathcal{NP}$.

The RK protocol: The idea underlying the RK protocol is to transform a given constant-round $\mathcal{ZK}$ protocol into $c\mathcal{ZK}$ by adding a k-round "preamble" to it (see Figure 1.4). This preamble (i.e., messages $(\mathrm{V}0), (\mathrm{P}1), (\mathrm{V}1), \ldots, (\mathrm{P}j), (\mathrm{V}j)$) is completely independent of the common input and its sole purpose is to enable a successful simulation in the concurrent setting. Every round in the preamble (i.e., every $(\mathrm{P}j), (\mathrm{V}j)$ pair) is viewed as a "rewinding opportunity". Having "successfully rewound" even one of the rounds in the preamble is sufficient in order to cheat arbitrarily in the actual proof (i.e., messages $(\mathtt{p1}), (\mathtt{v1}), (\mathtt{v2})$) and thus complete the simulation (cf. [20]).

The RK transformation reduces the problem of proving that the resulting protocol is $c\mathcal{ZK}$ to coming up with a simulator that, with overwhelming probability, manages to successfully rewind every session in the concurrent schedule (no matter what is the strategy applied by the verifier). Clearly, the larger is the number of rounds in the preamble, the easier the simulation task is. However, the main goal is to minimize the number of rounds in the protocol.

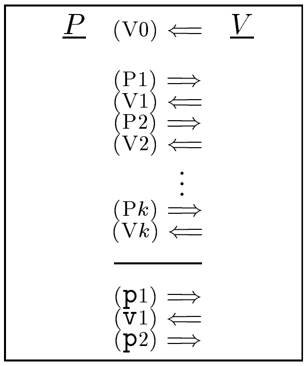


Figure 1.4: The Richardson-Kilian k-round protocol.

The RK simulator: Recall that rewinding a specific session in the concurrent setting may result in loss of work done for other sessions, and cause the simulator to do the same amount of work again. (Since all simulation work done for sessions starting after the point to which we rewind may be lost.) Considering a specific session of the RK protocol (out of $m = \mathrm{poly}(n)$ concurrent sessions), there must be an iteration (i.e., a $j \in \{1, \ldots, k\}$) so that at most $(m-1)/k$ sessions of the schedule start in the interval corresponding to the j^{th} iteration (of this specific session). So if we try to rewind on the correct j, we will invest (and so waste) only work proportional to $(m-1)/k$ sessions. The idea is to abort the rewinding attempt on the j^{th} iteration if more than $(m-1)/k$ sessions are initiated in the corresponding interval (as this will rule out the incorrect j's). The same reasoning applies *recursively* (i.e., to the rewinding in these $(m-1)/k$ sessions). Denoting by $W(m)$ the amount of work invested in m sessions, we obtain the recursion $W(m) = \mathrm{poly}(m) \cdot W(\frac{m-1}{k})$, which solves to $W(m) = m^{\Theta(\log_k m)}$. Thus, whenever $k = n$, we get $W(m) = m^{O(1)}$. This implies that for any $\epsilon > 0$, every language in $\mathcal{NP}$ has an n^{ϵ}-round $c\mathcal{ZK}$ proof system.

The RK analysis has been subsequently improved by Kilian and Petrank [35], who have employed a more sophisticated simulation technique (see Sections 4.1.2 and 4.4.1) to show that the RK protocol remains concurrent zero-knowledge even if it has $O(\alpha(n) \cdot \log^2 n)$ rounds, where $\alpha(\cdot)$ is any non-constant function (e.g., $\alpha(n) = \log\log n$). On a high level, the key idea underlying the Kilian-Petrank simulation strategy is that the order and timing of the simulator's rewindings are determined *obliviously* of the concurrent scheduling (which is determined "on the fly" by the adversary verifier). This is in contrast to the RK simulation strategy which heavily depends on the schedule as it is being revealed. Jumping ahead, we mention that our $O(\alpha(n) \cdot \log n)$ upper bound on the round-complexity of $c\mathcal{ZK}$ is obtained by conducting a fairly sophisticated analysis of the Kilian Petrank simulation technique (see Chapter 4) .

1.2.4 What About Non Black-Box Simulation?

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 prove results known impossible for black-box simulation. Specifically, for every (predetermined) polynomial $p(\cdot)$, there exists a constant-round protocol that preserves its zero-knowledge property even when it is executed $p(n)$ times concurrently (where n denotes the "security" parameter). As we show in Chapter 3, even this weaker notion is impossible to achieve when using black-box simulation, unless $\mathcal{NP} \subseteq \mathcal{BPP}$.

A major drawback of Barak's protocol is that the (polynomial) number of concurrent sessions relative to which the protocol should be secure must be fixed *before* the protocol is specified. Moreover, the length of the messages in the protocol grows linearly with 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 [42, 35] (as well as by the protocol presented in Chapter 4 of this thesis).

1.3 Organization

The main results in this thesis are presented in Chapters 3 and 4. The results presented in Chapters 5 and 6 have emerged as a result of the research efforts invested in the main results, but are interesting in their own right. It should be noted that, chronologically speaking, the result presented in Chapter 5 has been obtained prior to the results in presented in Chapters 3 and 4.

Chapter 2 - Preliminaries: We give formal definitions of interactive proofs and zero-knowledge. We then turn to define concurrent zero-knowledge, as well as black-box concurrent zero-knowledge. We also specify some conventions that are used in the proofs of the lower bound and the upper bound. Finally, we define the notion of bit-commitment, which will be used in the construction of our $c\mathcal{ZK}$ protocol.

Chapter 3 - Lower bound: We show that in the context of $c\mathcal{ZK}$, $\Omega(\log n / \log n \log n)$ rounds of interaction are essential for black-box simulation of proof systems for languages outside of $\mathcal{BPP}$ (Theorem 3.1). In addition, we note that the lower bound holds also for the case of $c\mathcal{ZK}$ arguments. In fact, it will hold even if the simulator knows the schedule in advance (in particular, it knows the number of concurrent sessions, which may just equal the security parameter), and even if the scheduling of the messages is fixed.

Chapter 4 - Upper bound: We show that, assuming the existence of perfectly-hiding commitment schemes, every language in $\mathcal{NP}$ can be proved in $c\mathcal{ZK}$ using only $O(\alpha(n) \cdot \log n)$ rounds of interaction, where $\alpha(n)$ is any super-constant function (Theorem 4.1). Our protocol retains its zero-knowledge property no matter how many times it is executed concurrently (as long as the number of concurrent sessions is polynomial in the size of the input). By considering so-called zero-knowledge arguments, we are also able to achieve a similar result assuming only the existence of one-way functions (Theorem 4.3). We also argue that our result in fact yields a generic transformation that takes any "standard" $\mathcal{ZK}$ protocol and transforms it into $c\mathcal{ZK}$ while paying only a "logarithmic" penatly in the round-complexity (Theorem 4.2). Additional results include the construction of a *resettable* $\mathcal{ZK}$ protocol with "logarithmic" number of rounds (Theorem 4.4), and the construction of $c\mathcal{ZK}$ arguments with polylogarithmic efficiency (Theorem 4.5).

Chapter 5 - $c\mathcal{ZK}$ without aborts: We consider a "relaxation" of $c\mathcal{ZK}$ and only require that the $\mathcal{ZK}$ property is mantained if the verifier never "aborts" any specific execution of the protocol during the concurrent interaction. We show that even in this case, black-box simulation faces difficulties that are not encountered in the stand-alone setting. Specifically, we show that even if the verifier never "aborts", 7 rounds of interaction are essential for black-box simulation of proof systems for languages outside of $\mathcal{BPP}$ (Theorem 5.1). As a corollary, we obtain that the RK protocol with $k = 2$ is not black-box $c\mathcal{ZK}$, even in this restricted sense.

Chapter 6 - Constant-round $\mathcal{ZK}$ proofs for $\mathcal{NP}$ with a simpler proof of security: We consider the task of constructing a constant-round $\mathcal{ZK}$ proof system for all languages in $\mathcal{NP}$. This problem has been previously addressed by Goldreich and Kahan [23], who constructed such proof systems assuming the existence of a collection of claw-free functions. We show how to use a variant of the $c\mathcal{ZK}$ protocol presented in Chapter 4 in order to construct an alternative constant-round $\mathcal{ZK}$ proof system for $\mathcal{NP}$. The advantage of the new proof system over the one of [23] is that it admits a considerably simpler proof of security.

Chapter 7 - Conclusions and open problems: We discuss the issues arising from our results, as well as some open problems arising from Barak's non black-box simulation techniques [2]. We also suggest to investigate the round-complexity of $c\mathcal{ZK}$ without aborts as an interesting open-problem.

Declaration: The author, Alon Rosen, declares that this thesis summarizes his work under the supervision of Professors Oded Goldreich and Moni Naor. The results in Chapter 3 were obtained jointly with Ran Canetti, Joe Kilian and Erez Petrank [9, 10]. The results in Chapter 4 were obtained by myself and independently by Manoj Prabhakaran and Amit Sahai [41]. The results in Chapters 5 [43] and 6 were obtained by myself.

Chapter 2

Preliminaries

2.1 General

2.1.1 Basic notation

We let N denote the set of all integers. For any integer $k \in N$, denote by $[k]$ the set $\{1, 2, \ldots, k\}$. For any $x \in \{0,1\}^*$, we let $|x|$ denote the size of x (i.e., the number of bits used in order to write it). For two machines M, A, we let $M^A(x)$ denote the output of machine M on input x and given oracle access to A. 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 $\nu(n) < n^{-c}$.

2.1.2 Probabilistic notation

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

$$\mathrm{Pr}_{x \leftarrow X}\left[B(x)\right] = \sum_x \frac{1}{|X|} \cdot \chi(B(x))$$

where χ is an indicator function so that $\chi(B) = 1$ if event B holds, and equals zero otherwise. We denote by U_n the uniform distribution over the set $\{0,1\}^n$.

2.1.3 Computational indistinguishability

Let $S \subseteq \{0,1\}^*$ be a set of strings. A probability ensemble indexed by S is a sequence of random variables indexed by S. Namely, any $X = \{X_w\}_{w \in S}$ is a random variable indexed by S.

Definition 2.1.1 (Computational indistinguishability) *Two ensembles $X = \{X_w\}_{w \in S}$ and $Y = \{Y_w\}_{w \in S}$ are said to be* computationally indistinguishable *if for every probabilistic polynomial-time algorithm D, there exists a negligible function $\nu(\cdot)$ so that for every $w \in S$:*

$$\left|\mathrm{Pr}\left[D(X_w, w) = 1\right] - \mathrm{Pr}\left[D(Y_w, w) = 1\right]\right| < \nu(|w|)$$

The algorithm D is often referred to as the distinguisher. For more details on computational indistiguishability see Section 3.2 of [22].

2.2 Interactive Proofs

We use the standard definitions of interactive proofs (and interactive Turing machines) [28, 22] and arguments (a.k.a computationally-sound proofs) [7]. 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.

Definition 2.2.1 (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 $x \in L$,*

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

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

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

In case that the soundness condition is required to hold only with respect to a computationally bounded prover, the pair $\langle P, V\rangle$ is called an interactive argument *system.*

Definition 2.2.1 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, in the context of our lower bound, we do not know whether this condition can be relaxed when dealing with 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. [5]).

Definition 2.2.2 (Round-Complexity) *Let $\langle P, V\rangle$ be an interactive proof system for a language L and let $r : N \to N$ be an integer function. We say that $\langle P, V\rangle$ has* round-complexity *$r(\cdot)$ if for every input x the number of messages exchanged is at most $r(|x|)$. In such a case, we sometimes refer to $\langle P, V\rangle$ as an $r(\cdot)$-round interactive proof system.*

2.3 Zero-Knowledge

Loosely speaking, an interactive proof is said to be *zero-knowledge* ($\mathcal{ZK}$) if it yields nothing beyond the validity of the assertion being proved. This is formalized by requiring that the view of every probabilistic polynomial-time adversary V^* interacting with the honest prover P can be simulated by a probabilistic polynomial-time machine S_{V^*} (a.k.a. the *simulator*). The idea behind this definition is that whatever V^* might have learned from interacting with P, he could have actually learned by himself (by running the simulator S). The transcript of an 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 interaction between P and V^* (that is, all messages that V^* sends and receives during the interaction with P, on common input x).

Definition 2.3.1 (Zero-Knowledge) *Let $\langle P, V\rangle$ be an interactive proof system for a language L. We say that $\langle P, V\rangle$ is* zero-knowledge*, if for every probabilistic polynomial-time interactive machine V^* there exists a probabilistic polynomial-time algorithm S_{V^*} such that the ensembles $\{\text{view}_{V^*}^{P}(x)\}_{x\in L}$ and $\{S_{V^*}(x)\}_{x\in L}$ are computationally indistinguishable.*

To make Definition 2.3.1 useful in the context of protocol composition, Goldreich and Oren [27] suggested to augment the definition so that the corresponding conditions hold also with respect to all $z \in \{0,1\}^*$, where both V^* and S_{V^*} are allowed to obtain z as auxiliary input. Jumping ahead, we comment that in the context of black-box simulation,, the original definition implies the augmented one (i.e., any black-box $\mathcal{ZK}$ protocol is also $\mathcal{ZK}$ w.r.t. auxuliary inputs). Since in this work we only consider the notion of black-box $\mathcal{ZK}$, we may ignore the issue of auxiliary inputs while being guaranteed that all results hold with repsect to the augmented definition as well.

2.4 Concurrent Zero-Knowledge

Let $\langle P, V\rangle$ be an interactive proof (resp. argument) 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). In order to control the scheduling, the concurrent adversary V^* concatenates every message that it sends with the session and round number to which the next scheduled message belongs. The convention is that the reply sent by the prover should have session and message indices as specified in the preceding verifier message (in case it does not, the verifier V^* is allowed to reject the corresponding session). As before, 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).

Definition 2.4.1 (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 probabilistic polynomial-time concurrent adversary V^* there exists a probabilistic polynomial-time algorithm S_{V^*} such that the ensembles $\{\text{view}_{V^*}^{P}(x)\}_{x\in L}$ and $\{S_{V^*}(x)\}_{x\in L}$ are computationally indistinguishable.*

In the context of concurrent $\mathcal{ZK}$, the round-complexity of a protocol is measured as a function of some predetemined "security" parameter $n \in N$. The requirement is that the protocol will remain secure as long as the number of concurrent executions is bounded by some polynomial in n (we stress that the protocol is constructed *before* the polynomial bound is determined). In this work, we use the convention that the "security" paramter n is equal (or polynomially related) to $|x|$.

2.5 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 $\text{view}_{V^*}^{P}(x)$ while using V^* as an oracle (i.e., in a "black-box" manner). Essentially, the definition of black-box simulation says that the black-box simulator mimics the interaction of the prover P with any polynomial-time verifier V^* relative to any random input r it might choose. The simulator does so merely by using oracle calls to $V^*(x;r)$ (which specifies the next message that V^* sends on input x and random input r). The simulation is indistinguishable from the true interaction even if the distinguisher (i.e., D) is given access to the oracle $V^*(x;r)$. For more details see Section 4.5.4.2 of [22].

Before we proceed with the formal definition for the case of $c\mathcal{ZK}$, 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 (less 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. In the sequel we choose to adopt the latter formalization.

Definition 2.5.1 (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 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 $\{S_q^{V^*}(x)\}_{x\in L}$ are computationally indistinguishable.*

2.6 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 as follows. Consider a distinguisher D that is required to decide whether its input consists of $\mathrm{view}_{V^*}^{P}(x)$ 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 $\mathrm{view}_{V^*}^{P}(x)$. The deviation gap of S is the deviation gap of the best polynomial time distinguisher D. In our definitions of concurrent zero-knowledge (Definitions 2.4.1 and 2.5.1) 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: In the lower bound, k-round protocols consist of 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.[1] Clearly, any protocol in which $2k$ messages are exchanged can be modified to fit this form (by adding at most two messages).

Both in the lower bound and the upper bound, we impose the following technical restrictions on the simulator (we claim that each of these restrictions can be satisfied by any simulator): As in (cf. [24]), the queries of the simulator are prefixes of possible execution transcripts (in the concurrent setting).[2] 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), and is determined by the output of the machine V^* when applied to the corresponding query (that is, the answer to query $\overline{q}$ is the message $V^*(\overline{q})$). In the case of the upper bound, we assume that the verifier's answers are always sent along with the identifiers of the next scheduled message (as determined by V^*). That is, every verifier message is concatenated with the session and round number to which the next scheduled message belongs. In the case of the lower bound, this is not necessary since we are considering a fixed scheduling that is determined in advance and known to everybody. 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 $(b_1, a_1, \ldots, b_T, a_T)$.

On the simulator's "behaviour": Similarly to all known black-box simulators, the simulator presented in Chapter 4 will go about the simulation task by means of "rewinding" the adversary V^* to past points in the interaction. That is, the simulator will explore many possible concurrent interactions with V^* by feeding it with different queries of the same length (while examining V^*'s output on these queries).[3] As will turn out from our proof, before making a query $\overline{q} = (\mathtt{p}_1, \mathtt{v}_1, \ldots, \mathtt{v}_{t-1}, \mathtt{p}_t)$, where the $\mathtt{p}$'s are prover messages, the simulator will always make queries to all relevant prefixes (i.e., $(\mathtt{p}_1, \mathtt{v}_1, \ldots, \mathtt{v}_{i-1}, \mathtt{p}_i)$, for every $i < t$), and will obtain the $\mathtt{v}_i$'s as answers. In addition, the simulator will never make an illegal query (except with negligible probability). That is, the simulator will always feed the verifier with messages in the prescribed format, and will make sure that the session and message numbers of any prover message in the query are indeed consistent with the identifiers appearing in the preceding verifier message. Actually, in order to succed, the simulator *does* deviate from the prescribed prover strategy (and indeed sends messages that would have not been sent by an honest prover). However, it will do so in a way that cannot be noticed by any probabilistic polynomial-time procedure (unless perfectly-binding commitments do not exist). What we actually mean by saying that illegal queries are never made is that the simulator will never send an ill-formed message (i.e., one that would cause an honest verifier V to immediately reject the protocol).

[1] 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 3.2.1).

[2] 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] Recall that every query made by the simulator corresponds to a specific execution transcript, and that the query's length corresponds to the number of messages exchanged so far.

Dealing with ABORT messages: Since the adversary verifier V^* may arbitrarily deviate from the prescribed strategy, it may be the case that throughout its interaction with the prover (simulator), V^* occasionally sends ill-formed messages (in other words, V^* may potentially refuse to decommit to a previous commitment). Clearly, such an action on behalf of the verifier is considered illegal, and the interaction in the relevant session stops (i.e., there is no need to continue exchanging messages in this session). Without loss of generality, such ill-formed messages are always interpreted as some predetermined ABORT message. For the sake of concreteness, we assume that whenever an ABORT message is sent by the verifier, the prover and verifier keep exchanging ABORT messages until the relevant session is completed. We stress that, as far as the prover (simulator) is concerned, illegal actions on behalf of the verifier in one session do not have any effect on the interaction in other sessions (since in the concurrent setting each prover/verifier pair is assumed to act independently).

2.7 Commitment Schemes

Commitment schemes are used to enable a party, known as the *sender*, to commit itself to a value while keeping it secret from the *receiver* (this property is called hiding). Furthermore, the commitment is binding, and thus in a later stage when the commitment is opened, it is guaranteed that the "opening" can yield only a single value determined in the committing phase.

Perfectly-binding commitment schemes: In a perfectly binding commitment scheme, the binding property holds even for an all-powerful sender, while the hiding property is only guaranteed with respect to a polynomial-time bounded receiver. For simplicity, we present the definition for a non-interactive, commitment scheme for a single bit. String commitment can be obtained by separately committing to each bit in the string.

We denote by $C_r(b)$ the output of the commitment scheme C upon input $b \in \{0,1\}$ and using the random string $r \in_R \{0,1\}^n$ (for simplicity, we assume that C uses n random bits where $n \in N$ is the security parameter).

Definition 2.7.1 (Perfectly-binding commitment) *A* perfectly-binding bit commitment scheme *is a probabilistic algorithm C satisfying the following two conditions:*

- Perfect Binding: $C_r(0) \neq C_s(1)$ *for every* $r, s \in \{0,1\}^n$ *and* $n \in N$.
- Computational Hiding: *The probability ensembles* $\{C_{U_n}(0)\}_{n\in N}$ *and* $\{C_{U_n}(1)\}_{n\in N}$ *are computationally indistinguishable.*

Non-interactive perfectly-binding commitment schemes can be constructed using any 1–1 one-way function (see Section 4.4.1 of [22]). Allowing some minimal interaction (in which the receiver first sends a single message), (almost) perfectly-binding commitment schemes can be obtained from any one-way function [37, 32].

Perfectly-hiding commitment schemes: In a perfectly hiding commitment scheme, the binding property is guaranteed to hold only with respect to a probabilistic polynomial-time sender. On the other hand, the hiding property is information-theoretic. That is, the distributions of commitments to 0 and commitments to 1 are identical (statistically-close), and thus even an all-powerful receiver cannot know the value committed to by the sender. We stress that the binding property guarantees that a cheating probabilistic polynomial-time sender can find only one decommitment, even though decommitments to both 0 and 1 exist (which in particular means that an all powerful sender can always decommit both to 0 and to 1). See [22] (Section 4.8.2) for a full definition.

Perfectly hiding commitment schemes can be constructed from any one-way permutation [38]. However, *constant-round* schemes are only known to exist under stronger assumptions; specifically, assuming the existence of collision-resistant hash functions [39, 14] or the existence of a collection of certified clawfree functions [23] (see also [22], Section 4.8.2.3).

Chapter 3

Black-Box $c\mathcal{ZK}$ Requires (almost) Logarithmically many Rounds

In this chapter we prove that in the context of black-box $c\mathcal{ZK}$, $\Omega(\log n/\log\log n)$ rounds of interaction are essential for non-trivial proof systems. This bound is the first to rule out the possibility of constant-round $c\mathcal{ZK}$, when proven via black-box simulation. Furthermore, the bound almost matches the number of rounds in the best known $c\mathcal{ZK}$ protocol for languages outside $\mathcal{BPP}$ [41] (see Chapter 4). The central result of this chapter is stated in the following theorem.

Theorem 3.1 (Lower Bound) *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, 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).*

The proof of Theorem 3.1 builds on the works of Goldreich and Krawczyk [24], Kilian, Petrank and Rackoff [36], and Rosen [43]. 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 [24, 36, 43], 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 the next section.

Remark: *The concurrent schedule in our proof is fixed and known to everybody. As a consequence, Theorem 3.1 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). Moreover, the actual scheduling and the number of sessions are known even before the protocol itself is determined.*

3.1 Proof Outline

This section contains an outline of the proof of Theorem 3.1. The actual proof will be given in Sections 3.2 and 3.3. To facilitate reading, we partition the outline into two parts: The first part reviews the general framework. (This part mainly follows previous works, namely [24, 36, 43].) The second part concentrates on the actual schedule and the specifics of our lower bound argument.

3.1.1 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.

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.

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 3.1.

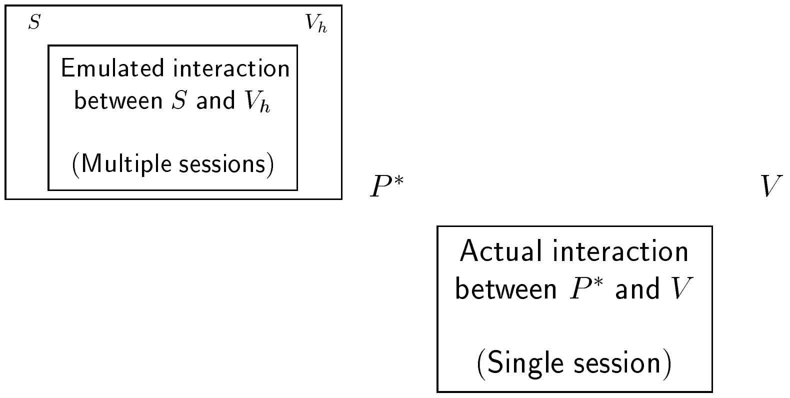

S
V_h
Emulated interaction between S and V_h
(Multiple sessions)
P^*
V
Actual interaction between P^* and V
(Single session)

Figure 3.1: 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. (Recall that P^* relays some of the actual interaction messages to its internal emulation.)

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".[1]

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 exactly $k+1$).

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.

3.1.2 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.

[1]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 in the emulation 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).

The 2-round case: Our starting point is the schedule used in [36] 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 3.2 for a graphical description):

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

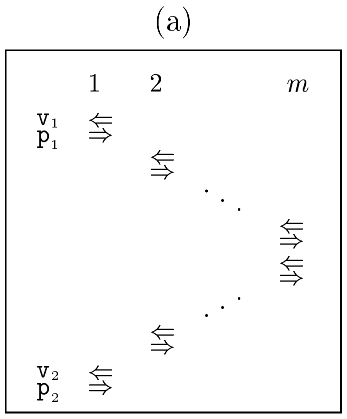
2. Otherwise (i.e., if $m > 1$):

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

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

   Final message exchange: The first session (out of m) 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 $s > \ell$.



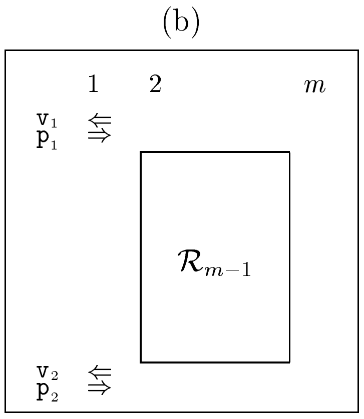


Figure 3.2: The "telescopic" schedule used to demonstrate impossibility of black-box concurrent zero-knowledge in 2 rounds. Columns correspond to n individual sessions and rows correspond to the time progression. (a) Depicts the schedule explicitly. (b) Depicts the schedule in a recursive manner ($\mathcal{R}_m$ denotes the recursive schedule for m sessions).

Suppose now that S succeeds 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., $\mathtt{p}_1$).[2] 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 22).

[2]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. As for $\mathtt{p}_2$, it is clear that rewinding this message would not cause any change in verifier messages that correspond to the relevant session-prefix (since, $\mathtt{v}_1$ and $\mathtt{v}_2$ occur after $\mathtt{p}_2$ anyway).

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 $W(n) \geq 2^{n-1}$).

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 $m \leq n$ 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 $\lfloor \frac{m-1}{k} \rfloor$ remaining sessions out of $2, \ldots, m$.)

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 3.3.

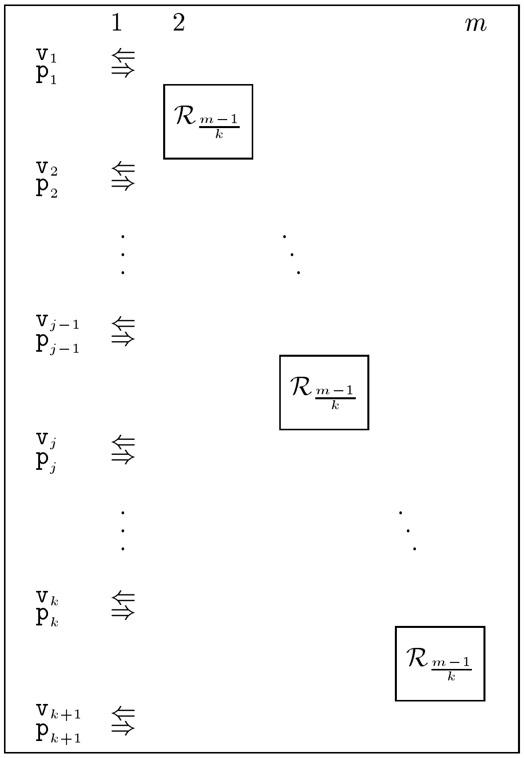


Figure 3.3: First attempt to generalize the recursive schedule for k-round protocols.

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. [42]) 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.

The k-round case – second attempt: One method to circumvent this difficulty was used in [43]. 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:

$$W(m) \geq \Omega(1/p) \cdot W\left(\frac{m-1}{k}\right) \tag{3.1}$$

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.

The k-round case – final version: However, demonstrating Eq. (3.1) 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. (3.1).

To overcome this problem, we replace each session in the above schedule (for k rounds) with a "block" of, say, n sessions (see Figure 3.4 in Page 29). 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 that, with a careful setting of the parameters, the simulator's recursive "work" must satisfy Eq. (3.1), at least with overwhelming probability.

Setting the value of p: It now remains to set the value of p so that Eq. (3.1) is established. 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 (See Page 22) 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 (See Page 22) will be satisfied for $k = o(\log n/\log\log n)$.

3.1.3 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 3.3.9). 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 3.3.7) 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.

3.2 The Actual Proof (of Theorem 3.1)

Assuming towards the contradiction that a black-box simulator, denoted S, contradicting Theorem 3.1 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.

3.2.1 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.

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.[3] The scheduling is defined recursively, where the scheduling of $m \leq n^2$ sessions (denoted $\mathcal{R}_m$) proceeds as follows:[4]

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 3.4. 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

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

[4]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 8.1 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, $\ell = s \textsf{ div } n$ and $i = s \textsf{ mod } n$).

Definition 3.2.1 (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 $\pi_{\rm sn}(\overline{q}) = (\ell, i) \in \{1, \ldots, n\} \times \{1, \ldots, n\}$ the session to which the next verifier message belongs, and by $\pi_{\rm msg}(\overline{q}) = j \in \{1, \ldots, k+1\}$ 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).

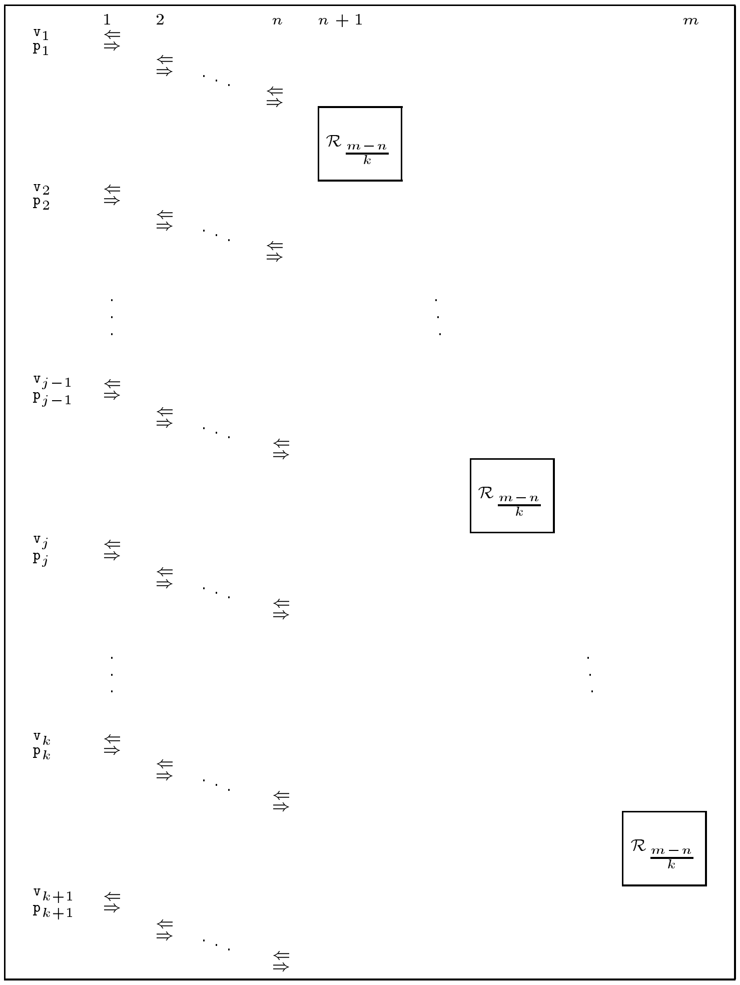


Figure 3.4: The recursive schedule $\mathcal{R}_m$ for m sessions. Columns correspond to m individual sessions and rows correspond to the time progression.

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. [27]), 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 [24, 36, 43], 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 $(\ell, i) = \pi_{\mathrm{sn}}(\overline{q})$ 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 $(\ell, i) = \pi_{\mathrm{sn}}(\overline{q})$ 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 $\textsf{block-prefix}$ 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 3.2.2 (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.*[5] (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 3.5 for a graphical description of the block-prefix and iteration-prefix of a query).

Definition 3.2.3 (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 $\mathtt{p}_{j-1}^{(n)}$ (where $\mathtt{p}_{j-1}^{(n)}$ 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 3.2.2 and 3.2.3: The choices made in Definitions 3.2.2 and 3.2.3 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).[6]

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

[6] 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.

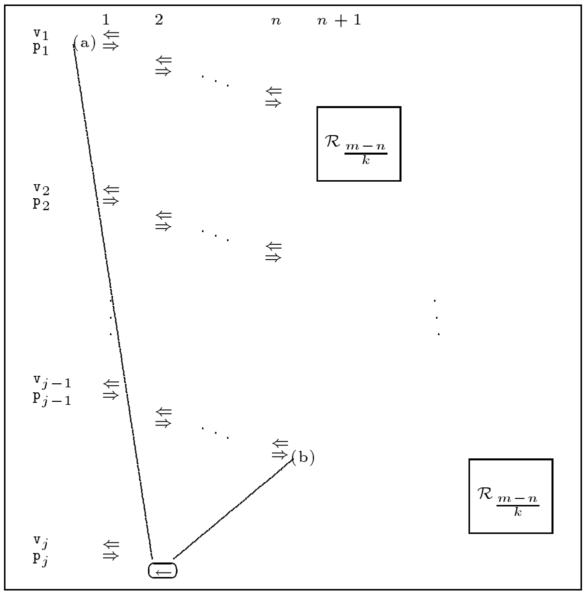


Figure 3.5: Determining the prefixes of query $\overline{q}$ (in this example, query $\overline{q}$ ends with a $\mathsf{p}_j^{(1)}$ message and is to be answered by $\mathsf{v}_j^{(2)}$, represented by the marked arrow): (a) indicates the block-prefix of $\overline{q}$ (i.e., messages up to this point 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 3.2.2 and 3.2.3 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.

3.2.2 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 $x \in \{0,1\}^n$).[7] 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.[8]

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., $\mathtt{v}_1^{(i)}$).

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):

[7] We stress that functions in such families can be described by strings of polynomial length in a way that enables polynomial time evaluation (cf. [34, 11, 12, 1]).

[8] In particular, $V_{g,h}$ checks whether the query is of the prescribed format (as described in Section 2.6, 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 $V_{g,h}(\overline{q}')$.)

(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 3.2.1 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[9] 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 $V_{g,h}$ 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, $t_S(n)$.

[9]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.

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),[10] and that

[10]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.

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.

3.2.3 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 $V_{g,h}$).[11]

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

Lemma 3.2.4 (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).[12] 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 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

[11]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.

[12]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).

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 3.2.5 (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.

3.3 Proof of Lemma 3.2.5 (performance on NO-instances)

Let us fix an $x \in \{0,1\}^n \setminus L$ as above.[13] Denote by $\mathtt{AC} = \mathtt{AC}_x$ 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 $\Pr_{\sigma,g,h}[(\sigma, g, h) \in \mathtt{AC}] > 1/p(n)$, for some fixed positive polynomial $p(\cdot)$. Before proceeding with the proof of Lemma 3.2.5, we formalize what we mean by referring to the "execution of the simulator".

Definition 3.3.1 (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 $V_{g,h}(x)$.*

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

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 $\textsc{exec}_x(\sigma, g, h)$ 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 $\textsc{exec}_x(\sigma, g, h)$ does not necessarily correspond to the ℓ^{th} recursive block in the schedule. Nevertheless, given $\textsc{exec}_x(\sigma, g, h)$ 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 $\textsc{exec}_x(\sigma, g, h)$, 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 $\textsc{exec}_x(\sigma, g, h)$ 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 $\textsc{exec}_x(\sigma, g, h)$". 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.

3.3.1 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 $\mathtt{v}_j^{(i)}$).[14]

(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 3.2.5 (and so Theorem 3.1) 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 $\langle P, V\rangle$).

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

[14]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).

$(\ell^{(\overline{bp_\xi})}, \eta)$ 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".[15]

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

[15]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).

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 3.3.2 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. ■

3.3.2 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.

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 3.2.3, two queries may have the same iteration-prefix even if they are of *different* lengths (see below).

Definition 3.3.3 (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 3.2.3, 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 3.3.3: 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).[16] 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).

Useful block-prefixes

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

Definition 3.3.4 (Useful block-prefix) *A block-prefix* $\overline{bp} = (b_1, a_1, \ldots, b_\gamma, a_\gamma)$, *that appears in* $\text{EXEC}_x(\sigma, g, h)$, *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* $\text{EXEC}_x(\sigma, g, h)$ *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* $\text{EXEC}_x(\sigma, g, h)$ *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 3.3.4 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 jth message so that $g(i, \overline{ip})$ evaluates to 1. Condition 2 asserts that the last verifier message in the ith main session of recursive block number $\ell = \ell^{(\overline{bp})}$ is equal to ACCEPT. It follows that if the cheating prover happens to select $(\sigma, g, h, \xi, \eta)$ so that block-prefix $\overline{bp}_\xi$ (i.e., the ξth distinct block-prefix in $\text{EXEC}_x(\sigma, g, h^{(r)})$) is η-useful, then it convinces $V(x, r)$; the reason being that (by Condition 2) the last message in session

[16] 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 $V_{g,h^{(r)}}$.

$(\ell^{(\overline{bp}_\xi)}, \eta)$ is answered with ACCEPT,[17] 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 3.3.5 *If the cheating prover happens to select $(\sigma, g, h, \xi, \eta)$ so that the ξ^{th} distinct block-prefix in $\text{EXEC}_x(\sigma, g, h^{(r)})$ 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 ξ^{th} distinct block-prefix in $\text{EXEC}_x(\sigma, g, h^{(r)})$ 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 $\pi_{\text{sn}}(\overline{q}) = (\ell^{(\overline{bp}_\xi)}, \eta)$, $\pi_{\text{msg}}(\overline{q}) = j$ and $g(\eta, ip(\overline{q})) = 1$. Since the ξ^{th} distinct block-prefix in $\text{EXEC}_x(\sigma, g, h^{(r)})$ 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 ξ^{th} distinct block-prefix in $\text{EXEC}_x(\sigma, g, h^{(r)})$ 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 $(k+1)^{\text{st}}$ 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 3.3.4, the query $\overline{q}$, that corresponds to block-prefix $\overline{bp}_\xi$, satisfies $\pi_{\text{sn}}(\overline{q}) = (\ell^{(\overline{bp}_\xi)}, \eta)$, $\pi_{\text{msg}}(\overline{q}) = j$ 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 ξ^{th} distinct block-prefix in $\text{EXEC}_x(\sigma, g, h^{(r)})$. ■

Reduction to rareness of legal transcripts without useful block-prefixes

The following lemma (Lemma 3.3.6) 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 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 3.3.6 actually

[17]Notice that $V(x, r)$ behaves exactly as $V_{g,h^{(r)}}$ behaves on queries that correspond to the ξ^{th} distinct iteration-prefix in $\text{EXEC}_x(\sigma, g, h^{(r)})$.

implies the correctness of Lemma 3.2.5 (recall that the contradiction hypothesis of Lemma 3.2.5 is that the probability that the simulator outputs a legal transcript is non-negligible).

Lemma 3.3.6 *Suppose that* $\Pr_{\sigma,g,h}[(\sigma,g,h)\in \texttt{AC}] > 1/p(n)$ *for some fixed polynomial* $p(\cdot)$. *Then the probability (taken over* σ,g,h,ξ,η,r*), that* $\langle P^*,V\rangle(x) = \texttt{ACCEPT}$ *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 3.3.5, we have:

$$\Pr_{\sigma,g,h,\xi,\eta,r}\left[\langle P^*,V\rangle(x) = \texttt{ACCEPT}\right] \geq \Pr_{\sigma,g,h,\xi,\eta,r}\left[\mathsf{useful}_{\xi,\eta}(\sigma,g,h^{(r)})\right] \tag{3.2}$$

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 3.3.2), we infer that:

$$\Pr_{\sigma,g,h,\xi,\eta,r}\left[\mathsf{useful}_{\xi,\eta}(\sigma,g,h^{(r)})\right] = \Pr_{\sigma,g,h',\xi,\eta}\left[\mathsf{useful}_{\xi,\eta}(\sigma,g,h')\right] \tag{3.3}$$

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

$$\begin{aligned}
\Pr_{\sigma,g,h,\xi,\eta}\left[\mathsf{useful}_{\xi,\eta}(\sigma,g,h)\right] &= \sum_{\ell=1}^{t_S(n)}\sum_{i=1}^{n}\Pr_{\sigma,g,h,\xi,\eta}\left[\mathsf{useful}_{\ell,i}(\sigma,g,h)\ \&\ (\xi=\ell\ \&\ \eta=i)\right]\\
&= \sum_{\ell=1}^{t_S(n)}\sum_{i=1}^{n}\Pr_{\sigma,g,h}\left[\mathsf{useful}_{\ell,i}(\sigma,g,h)\right]\cdot\Pr_{\xi,\eta}\left[\xi=\ell\ \&\ \eta=i\right]\\
&= \sum_{\ell=1}^{t_S(n)}\sum_{i=1}^{n}\Pr_{\sigma,g,h}\left[\mathsf{useful}_{\ell,i}(\sigma,g,h)\right]\cdot\frac{1}{t_S(n)\cdot n}\\
&\geq \Pr_{\sigma,g,h}\left[\exists \ell,i \text{ s.t. } \mathsf{useful}_{\ell,i}(\sigma,g,h)\right]\cdot\frac{1}{t_S(n)\cdot n}
\end{aligned} \tag{3.4}$$

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. (3.2), (3.3), (3.4) we get:

$$\Pr_{\sigma,g,h,\xi,\eta,r}\left[\langle P^*,V\rangle(x) = \texttt{ACCEPT}\right] \;\geq\; \Pr_{\sigma,g,h}\left[\exists \ell,i \text{ s.t. } \mathsf{useful}_{\ell,i}(\sigma,g,h)\right]\cdot\frac{1}{t_S(n)\cdot n} \tag{3.5}$$

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

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

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:

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

Thus, Lemma 3.3.6 is satisfied provided that $\Pr_{\sigma,g,h}[\forall \ell, i \ \neg\mathsf{useful}_{\ell,i}(\sigma,g,h) \ \& \ (\sigma,g,h) \in \texttt{AC}]$ is negligible. Consequently, Lemma 3.3.6 will follow by establishing Lemma 3.3.7, stated next.

Lemma 3.3.7 *The probability (taken over σ, g, h), that for all pairs (ℓ, i) $\mathsf{useful}_{\ell,i}(\sigma,g,h)$ 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 3.3.6. In the rest of this section we prove Lemma 3.3.7. ■

3.3.3 Proof of Lemma 3.3.7 (legal transcripts yield useful block-prefixes)

The proof of Lemma 3.3.7 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 "rewinds" is, the smaller is the probability that a specific block-prefix is useful. A block-prefix with a small number of "rewinds" 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 3.3.8 (Potentially-useful block-prefix) *A block-prefix $\overline{bp} = (b_1, a_1, .., b_\gamma, a_\gamma)$, that appears 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 3.3.3 (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 3.3.4), 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 3.3.8 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 3.3.7).)

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).[18] 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 3.3.9 *For any $(\sigma, g, h) \in \mathtt{AC}_x$,* $\text{EXEC}_x(\sigma, g, h)$ *contains a potentially-useful block-prefix.*

Proof of Lemma 3.3.9 (existence of potentially-useful block-prefixes)

The proof of Lemma 3.3.9 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).

[18]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 3.3.9, 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 [36] (and also used in [43]).[19] 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 "rewinds"). 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.[20] 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 "rewinds" 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 "rewinds" 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

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

[20]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 $\mathtt{p}_{k+1}^{(n,n)}$ 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 3.3.9 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 3.3.10 *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 3.3.10) that any m-good sub-tree must satisfy:

$$W(m) \geq \begin{cases} 1 & \text{if } m \leq n \\ k^{c+1} \cdot W\left(\frac{m-n}{k}\right) & \text{if } m > n \end{cases} \tag{3.6}$$

Since for all but finitely many n, Eq. (3.6) solves to $W(n^2) > n^c$ (see Section 8.2 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 3.3.9 will actually follow from Lemma 3.3.10.

Proof (of Lemma 3.3.10): 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 $\pi_{\text{sn}}(\overline{q}) \in B_T$.[21] 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 3.3.11 *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.[22]) 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. ■

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

Claim 3.3.12 *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 3.3.10), 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 3.3.8 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

[21]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).

[22]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 3.3.13 *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 3.3.13 is proved, we can use it in conjunction with Claim 3.3.12 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 3.3.13, 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 3.3.14 (Iteration-suffix) *The* iteration-suffix *of a query $\overline{q}$ (satisfying $j = \pi_{\text{msg}}(\overline{q}) > 1$), 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 $is(\overline{q}) = (a_\delta, b_{\delta+1}, \ldots, a_t, b_t)$.*[23]

Let $\overline{q}$ be a query, and let $(\ell, i) = \pi_{\text{sn}}(\overline{q})$, $j = \pi_{\text{msg}}(\overline{q})$. 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 $\mathcal{P}_{is}(\overline{q})$[24]).

Fact 3.3.15 *For every query $\overline{q} \in \text{EXEC}_x(\sigma, g, h)$, 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.*

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

[24]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 3.3.16 *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 $ip(\overline{q}_2)$,[25] 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 3.3.13, 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 3.3.13), 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 3.3.15, T_1 and T_2 are indeed sub-trees

[25]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.

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 recursive construction (i.e., to an instance of $\mathcal{R}_{\frac{m-n}{k}}$). Using Fact 3.3.15 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 3.3.16) 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 3.3.10 (using Claims 3.3.12 and 3.3.13). By Claim 3.3.12, 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 3.3.13), 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 3.3.10 follows. ■

Back to the Proof of Lemma 3.3.7 (existence of useful block-prefixes)

Once the correctness of Lemma 3.3.9 is established, we may proceed with the proof of Lemma 3.3.7. 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 $\ell \in \{1, \ldots, t_S(n)\}$ and all $i \in \{1, \ldots, n\}$, the ℓ^{th} distinct block-prefix in $\text{EXEC}_x(\sigma, g, h)$ is not i-useful. Specifically, we would like to show that:

$$\Pr_{\sigma,g,h}\left[(\forall \ell, i \ \neg\mathsf{useful}_{\ell,i}(\sigma,g,h)) \ \& \ ((\sigma,g,h) \in \texttt{AC})\right] \tag{3.7}$$

is negligible. Define a Boolean indicator $\mathsf{pot\!-\!use}_\ell(\sigma, g, h)$ to be true if and only if the ℓ^{th} distinct block-prefix in $\text{EXEC}_x(\sigma, g, h)$ is potentially-useful. As proved in Lemma 3.3.9, for any $(\sigma, g, h) \in \texttt{AC}$ there exists an index $\ell \in \{1, \ldots, t_S(n)\}$, so that the ℓ^{th} block-prefix in $\text{EXEC}_x(\sigma, g, h)$ is potentially-useful. In other words, for every $(\sigma, g, h) \in \texttt{AC}$, $\mathsf{pot\!-\!use}_\ell(\sigma, g, h)$ holds for some value of ℓ. Thus, Eq. (3.7) is upper bounded by:

$$\Pr_{\sigma,g,h}\left[\bigvee_{\ell=1}^{t_S(n)} \mathsf{pot\!-\!use}_\ell(\sigma,g,h) \ \& \ (\forall i \in \{1,\ldots,n\} \ \neg\mathsf{useful}_{\ell,i}(\sigma,g,h))\right] \tag{3.8}$$

Consider a specific $\ell \in \{1, \ldots, t_S(n)\}$ so that $\mathsf{pot\!-\!use}_\ell(\sigma, g, h)$ is satisfied (i.e., the ℓ^{th} block prefix in $\text{EXEC}_x(\sigma, g, h)$ is potentially-useful). By Condition 2 in the definition of a potentially-useful block-prefix (Definition 3.3.8), the execution of the simulator reaches the end of the corresponding block in the schedule. In other words, there exists a query $\overline{q} \in \text{EXEC}_x(\sigma, g, h)$ that ends with the $(k+1)^{\text{st}}$ prover message in the n^{th} main session of recursive block number $\ell^{(\overline{bp}_\ell)}$, where $\overline{bp}_\ell$ denotes the ℓ^{th} distinct block-prefix in $\text{EXEC}_x(\sigma, g, h)$, and $\ell^{(\overline{bp}_\ell)}$ denotes the index of the recursive block that corresponds to block-prefix $\overline{bp}_\ell$ in $\text{EXEC}_x(\sigma, g, h)$. Since, by our convention and the modification of the simulator, S never generates a query that is answered with a `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 $\ell^{(\overline{bp}_\ell)}$ (as otherwise query $\overline{q}$ would have been answered with the `DEVIATION` message in Step 1' of $V_{g,h}$).

Let $\overline{q}^{(\overline{bp}_\ell)} = \overline{q}^{(\overline{bp}_\ell)}(\sigma, g, h)$ denote the first query in $\text{EXEC}_x(\sigma, g, h)$ 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 $\ell^{(\overline{bp}_\ell)}$, where $\overline{bp}_\ell$ denotes the d^{th} block-prefix appearing in $\text{EXEC}_x(\sigma, g, h)$).[26] Define an additional Boolean indicator $\mathsf{accept}_{\ell,i}(\sigma, g, h)$ to be true if and only if query $\overline{q}^{(\overline{bp}_\ell)}$ contains an accepting conversation for session $(\ell^{(\overline{bp}_\ell)}, i)$ (that is, no prover message in session $(\ell^{(\overline{bp}_\ell)}, i)$ is answered with ABORT, and the last verifier message of this session equals ACCEPT).[27] It follows that for every $\ell \in \{1, \ldots, t_S(n)\}$ that satisfies $\mathsf{pot\text{-}use}_\ell(\sigma, g, h)$ (as above), there exists a set $\mathcal{S} \subset \{1, \ldots, n\}$ of size $\frac{n^{1/2}}{4}$ such that $\mathsf{accept}_{\ell,i}(\sigma, g, h)$ holds for every $i \in \mathcal{S}$. Thus, Eq. (3.8) is upper bounded by:

$$\Pr_{\sigma,g,h}\left[\bigvee_{\ell=1}^{t_S(n)} \bigvee_{\substack{\mathcal{S}\subset\{1,\ldots,n\} \\ |\mathcal{S}|=\frac{n^{1/2}}{4}}} \Big(\mathsf{pot\text{-}use}_\ell(\sigma, g, h) \ \& \ \Big(\forall i \in \mathcal{S}, \ \neg\mathsf{useful}_{\ell,i}(\sigma, g, h) \ \& \ \mathsf{accept}_{\ell,i}(\sigma, g, h)\Big)\Big)\right] \tag{3.9}$$

Using the union bound, we upper bound Eq. (3.9) by:

$$\sum_{\ell=1}^{t_S(n)} \sum_{\substack{\mathcal{S}\subset\{1,\ldots,n\} \\ |\mathcal{S}|=\frac{n^{1/2}}{4}}} \Pr_{\sigma,g,h}\Big[\mathsf{pot\text{-}use}_\ell(\sigma, g, h) \ \& \ \Big(\forall i \in \mathcal{S}, \ \neg\mathsf{useful}_{\ell,i}(\sigma, g, h) \ \& \ \mathsf{accept}_{\ell,i}(\sigma, g, h)\Big)\Big] \tag{3.10}$$

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

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

Proof: Let $x \in \{0,1\}^*$. Fix some $\sigma \in \{0,1\}^*$, $h \in H$, $\ell \in \{1, \ldots, t_S(n)\}$ and a set $S \subset \{1, \ldots, n\}$. Denote by $\overline{bp}_\ell = \overline{bp}_\ell(g)$ the ℓ^{th} distinct block-prefix in $\text{EXEC}_x(\sigma, h, g)$, and by $\ell^{(\overline{bp}_\ell)}$ 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 $\overline{bp}_\ell$ is not i-useful, even though it is potentially-useful and for all $i \in S$ the query $\overline{q}^{(\overline{bp}_\ell)}$ contains an accepting conversation for session $(\ell^{(\overline{bp}_\ell)}, i)$.

[26] 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, $\text{EXEC}_x(\sigma, g, h)$ 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 $\ell^{(\overline{bp}_\ell)}$). 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).

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

A technical problem resolved: In order to prove Lemma 3.3.17 we need to focus on the ℓ^{th} distinct block-prefix in $\text{EXEC}_x(\sigma, h, g)$ (denoted by $\overline{bp_\ell}$) and analyze the behaviour of a uniformly chosen g when applied to the various iteration-prefixes that correspond to $\overline{bp_\ell}$. However, trying to do so we encounter a technical problem. This problem is caused by the fact that the contents of block-prefix $\overline{bp_\ell}$ depends on g.[28] 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 $\overline{bp_\ell}$ when $\overline{bp_\ell}$ itself is not determined). To overcome the above problem, we rely on the following observations:

1. Whenever σ, h and ℓ are fixed, the contents of block-prefix $\overline{bp_\ell}$ is completely determined by the output of g on inputs that have occurred *before* $\overline{bp_\ell}$ 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 $\overline{bp_\ell}$ occur *after* $\overline{bp_\ell}$ 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* $\overline{bp_\ell}$ has been determined. That is, fixing σ, h and ℓ we distinguish between: (a) the outputs of g that have occurred *before* the ℓ^{th} distinct block-prefix in $\text{EXEC}_x(\sigma, g, h)$ (i.e., $\overline{bp_\ell}$) has been reached, and (b) the outputs of g that have occurred *after* $\overline{bp_\ell}$ 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, ℓ be as above. We define $g^{(g_1,g_2)} = g^{(\sigma,h,\ell,g_1,g_2)}$ 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* $\overline{bp_\ell}$ 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* $\overline{bp_\ell}$ has been reached is equal to $g_2(\alpha)$.

Similarly to the proof of Claim 3.3.2 it can be shown that for every σ, h, ℓ as above, if g_1 and g_2 are uniformly distributed then so is $g^{(g_1,g_2)}$. In particular:

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

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, ℓ and for *every* fixed value of g_1, the block-prefix $\overline{bp_\ell}$ is completely determined (and the corresponding iteration-prefixes are well defined). (2) Once $\overline{bp_\ell}$ has been reached, the outcome of $g^{(g_1,g_2)}$ when applied to the *relevant* iteration-prefixes is completely determined by the choice of g_2. Thus, all we need to show to prove Lemma 3.3.17 is that for *every* choice of g_1

$$
\Pr_{g_2} \Big[\mathsf{pot\text{-}use}_\ell(\sigma, g^{(g_1,g_2)}, h) \ \& \ \Big(\forall i \in S,\ \neg\mathsf{useful}_{\ell,i}(\sigma, g^{(g_1,g_2)}, h) \ \& \ \mathsf{accept}_{\ell,i}(\sigma, g^{(g_1,g_2)}, h)\Big)\Big] \tag{3.11}
$$

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

[28] 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 $\overline{bp_\ell} = \overline{bp_\ell}(g)$ may vary together with the choice of g.

Back to the actual proof of Lemma 3.3.17: Consider the block-prefix $\overline{bp}_\ell$, as determined by the choices of σ, h, ℓ and g_1, and focus on the iteration-prefixes that correspond to $\overline{bp}_\ell$ in $\text{EXEC}_x(\sigma, g, h)$. We next analyze the implications of $\overline{bp}_\ell$ being not i-useful, even though it is potentially useful and for all $i \in S$ query $\overline{q}^{(\overline{bp}_\ell)}$ contains an accepting conversation for session $(\ell^{(\overline{bp}_\ell)}, i)$.

Claim 3.3.18 *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 $\left(\mathsf{pot{-}use}_\ell(\sigma, g, h) \ \& \ (\forall i \in S, \neg\mathsf{useful}_{\ell,i}(\sigma, g, h) \ \& \ \mathsf{accept}_{\ell,i}(\sigma, g, h))\right)$ is true. Then:*

1. *The number of different iteration-prefixes that correspond to block-prefix $\overline{bp}_\ell$ 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 $\overline{bp}_\ell$), 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 $\overline{bp}_\ell$), 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 3.3.18 will be invoked with $g = g^{(g_1, g_2)}$.

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

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

Proof of Item 2: Let $i \in S$ and recall that $\mathsf{accept}_{\ell,i}(\sigma, g, h)$ holds. In particular, we have that query $\overline{q}^{(\overline{bp}_\ell)}$ (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 $\ell^{(\overline{bp}_\ell)}$) contains an accepting conversation for session $(\ell^{(\overline{bp}_\ell)}, i)$. That is, no prover message in session $(\ell^{(\overline{bp}_\ell)}, i)$ is answered with ABORT, and the last verifier message of this session equals ACCEPT. Since by our conventions regarding the simulator, before making query $\overline{q}^{(\overline{bp}_\ell)}$ the simulator has made queries to all relevant prefixes, then it must be the case that all prefixes of query $\overline{q}^{(\overline{bp}_\ell)}$ 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 $\overline{q}^{(\overline{bp}_\ell)}$ and that satisfies $bp(\overline{q}_{i,j}) = \overline{bp}_\ell$, $\pi_{\text{sn}}(\overline{q}_{i,j}) = (\ell^{(\overline{bp}_\ell)}, i)$, $\pi_{\text{msg}}(\overline{q}_{i,j}) = j$, and $g(i, ip(\overline{q}_{i,j})) = 1$. (If $g(i, ip(\overline{q}_{i,j}))$ would have been equal to 0, query $\overline{q}^{(\overline{bp}_\ell)}$ would have contained a prover message in session $(\ell^{(\overline{bp}_\ell)}, i)$ that is answered with ABORT, in contradiction to the fact that $\mathsf{accept}_{\ell,i}(\sigma, g, h)$ 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 $\overline{bp}_\ell$ (as queries $\overline{q}_{i,j}$ all have block-prefix $\overline{bp}_\ell$), 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 $\mathsf{accept}_{\ell,i}(\sigma, g, h)$ holds, we have that $\mathsf{useful}_{\ell,i}(\sigma, g, h)$ does not hold. Notice that the only reason for which $\mathsf{useful}_{\ell,i}(\sigma, g, h)$

can be false (i.e., the ℓ^{th} block-prefix is not i-useful), is that Condition 1 in Definition 3.3.4 is violated by $\text{EXEC}_x(\sigma, g, h)$. (Recall that $\mathsf{accept}_{\ell,i}(\sigma, g, h)$ holds, and so Condition 2 in Definition 3.3.4 is indeed satisfied by query $\overline{q}_{i,k+1}$ (as defined above): This query corresponds to block-prefix $\overline{bp}_\ell$, satisfies $\pi_{\text{sn}}(\overline{q}_{i,k+1}) = (\ell^{(\overline{bp}_\ell)}, i)$, $\pi_{\text{msg}}(\overline{q}_{i,k+1}) = k+1$, $g(i, ip(\overline{q}_{i,k+1})) = 1$, and is answered with `ACCEPT`.)

For Condition 1 in Definition 3.3.4 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 $\overline{bp}_\ell$, satisfy $\pi_{\text{sn}}(\overline{q}_1) = \pi_{\text{sn}}(\overline{q}_2) = (\ell^{(\overline{bp}_\ell)}, i)$, $\pi_{\text{msg}}(\overline{q}_1) = \pi_{\text{msg}}(\overline{q}_2) = j$, 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 $\overline{bp}_\ell$ 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 $\mathsf{useful}_{\ell,i}(\sigma, g, h)$ & $\mathsf{accept}_{\ell,i}(\sigma, g, h)$), 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 $\overline{bp}_\ell$, 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 3.3.18. ■

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 $\overline{bp}_\ell$ 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 $\overline{bp}_\ell$ as an independently executed experiment that succeeds with probability $n^{-1/2k}$.[29]

Using Claim 3.3.18.1 (i.e., Item 1 of Claim 3.3.18), the applications of g_2 which are relevant to sessions $\{(\ell^{(\overline{bp}_\ell)}, i)\}_{i \in S}$ 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 3.3.18.2 now implies that at least k of the above experiments will fully succeed (that is, all of their sub-experiments will succeed), while Claim 3.3.18.3 implies that for every $i \in S$ 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. (3.11) is upper bounded by the probability that the following two events occur (these events correspond to Claims 3.3.18.2 and 3.3.18.3 respectively):

[29] 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 $\overline{bp}_\ell$ has been determined (as, in the above case, all inputs for g_2 are iteration-prefixes that correspond to block-prefix $\overline{bp}_\ell$, and such iteration-prefixes will occur only after $\overline{bp}_\ell$ has already been determined).

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 3.3.18.2).)

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 3.3.18.3).)

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. (3.11) is upper bounded by:

$$\begin{aligned}
&\sum_{\substack{K\subseteq[k^{c+1}]\\|K|=k}} \Pr\Big[\forall j \in K,\ \forall i \in S \text{ s.t. } \chi_{i,j}\Big] \cdot \Pr\Big[\forall i \in S,\ \exists j \in [k^{c+1}] \setminus K \text{ s.t. } \chi_{i,j}\Big] \\
&= \binom{k^{c+1}}{k} \cdot \left(\left(n^{-\frac{1}{2k}}\right)^{|S|}\right)^k \cdot \left(1 - \left(1 - n^{-\frac{1}{2k}}\right)^{k^{c+1}-k}\right)^{|S|} \\
&< \left(k^{c+1}\right)^k \cdot \left(\left(n^{-\frac{1}{2k}}\right)^{|S|}\right)^k \cdot \left(k^{c+1} \cdot n^{-\frac{1}{2k}}\right)^{|S|} \qquad (3.12) \\
&= \left(k^{c+1}\right)^{k+|S|} \cdot \left(n^{-\frac{1}{2k}}\right)^{k\cdot|S|+|S|} \\
&= \left(k^{c+1}\right)^{k+|S|} \cdot \left(n^{-\frac{1}{4k}}\right)^{|S|} \left(n^{-\left(\frac{1}{2}+\frac{1}{4k}\right)}\right)^{|S|} \\
&< \left(n^{-\left(\frac{1}{2}+\frac{1}{4k}\right)}\right)^{|S|} \qquad (3.13)
\end{aligned}$$

where Eq. (3.12) holds whenever $k^{c+1} - k = o(n^{1/2k})$ (which is satisfied if $k = o(\frac{\log n}{\log\log n})$), and Eq. (3.13) 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. (3.11) is upper bounded by $(n^{-(1/2+1/4k)})^{|S|}$, and the proof of Lemma 3.3.17 is complete. ■

Using Lemma 3.3.17, we upper bound Eq. (3.10) by

$$\begin{aligned}
t_S(n) \cdot \binom{n}{\frac{n^{1/2}}{4}} \cdot \left(n^{-\left(\frac{1}{2}+\frac{1}{4k}\right)}\right)^{\frac{n^{1/2}}{4}} &< t_S(n) \cdot \left(\frac{4 \cdot e \cdot n}{n^{1/2}}\right)^{\frac{n^{1/2}}{4}} \cdot \left(n^{-\left(\frac{1}{2}+\frac{1}{4k}\right)}\right)^{\frac{n^{1/2}}{4}} \\
&= t_S(n) \cdot \left(\frac{4 \cdot e}{n^{1/4k}}\right)^{\frac{n^{1/2}}{4}} \\
&< t_S(n) \cdot 2^{-\frac{n^{1/2}}{4}} \qquad (3.14)
\end{aligned}$$

where Inequality 3.14 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 3.3.7 (since $\text{poly}(n) \cdot 2^{-\Omega(n^{1/2})}$ is negligible).

Chapter 4

$c\mathcal{ZK}$ in Logarithmically many Rounds

In this chapter we present a black-box $c\mathcal{ZK}$ protocol whose number of rounds matches the lower bound established in Chapter 3. Specifically, assuming the existence of perfectly-hiding commitment schemes (which exist assuming the existence of a collection of claw-free functions [29]), we show that every language in $\mathcal{NP}$ can be proved in $c\mathcal{ZK}$ using logarithmically many rounds of interaction. This is formally stated in the following theorem.

Theorem 4.1 (Upper Bound) *Assume the existence of perfectly-hiding commitment schemes, and let $\alpha : N \rightarrow N$ be any super-constant function. Then, there exists an $O(\alpha(n) \cdot \log n)$-round black-box concurrent zero-knowledge proof system for every language $L \in \mathcal{NP}$.*

The proof of Theorem 4.1 builds on the protocol by Richardson and Kilian [42] and on the simulator by Kilian and Petrank [35]. However, our analysis of the simulator's execution is more sophisticated and thus yields a stronger result. We introduce a novel counting argument that involves a direct analysis of the underlying probability space. This is in contrast to previous results that required subtle manipulations of conditional probabilities. We also present a new variant of the RK protocol [42] which is both simpler and more amenable to analysis than the original version.

4.1 A $c\mathcal{ZK}$ proof system for $\mathcal{NP}$

We start by presenting a high-level description of our protocol, as well as a description of the black-box simulator that establishes its zero-knowledge property. Our protocol is inspired by the Richardson-Kilian (RK) protocol [42] and uses the well known 3-round protocol for Hamiltonicity by Blum [6] as a building block. The crucial property of Blum's protocol that we need in order to construct a concurrent zero-knowledge simulator is that the simulation task becomes trivial as soon as the verifier's message is known in advance. That is, if the prover knows the verifier's "challenge" prior to the beginning of the protocol then it can always make the verifier accept (regardless of whether the graph is Hamiltonian). This is done by adjusting the prover's messages according to the contents of the verifier's "challenge" (which, as we said, is known in advance). We stress that the choice of Blum's protocol as a building block is arbitrary (and is made just for clarity of presentation). In fact, the above property is satisfied by many other known protocols (in particular, any one of these protocols could have been used as a building block for our construction).

Much alike the RK protocol [42], our protocol is designed to overcome difficulties that are encountered whenever many sessions are to be black-box simulated concurrently. This is done by adding a "preamble" to the protocol, which is completely independent of the common input and whose sole purpose is to enable a successful simulation in the concurrent setting. Every round in the preamble is viewed as a "rewind opportunity". Having successfully rewound even one of the rounds in the preamble is sufficient in order to reveal the verifier's "challenge" in the Hamiltonicity proof system. As mentioned above, knowing the verifier's "challenge" in advance enables the simulator to cheat arbitrarily in the Hamiltonicity proof (regardless of whether the graph is Hamiltonian).

4.1.1 The protocol

We let k be any super-logarithmic function in n. Our protocol consists of two stages. In the first stage (or preamble), which is independent of the actual common input, the verifier commits to a random n-bit string σ, and to two sequences, $\{\sigma^0_{i,j}\}_{i,j=1}^k$, and $\{\sigma^1_{i,j}\}_{i,j=1}^k$, each consisting of k^2 random n-bit strings (this first message is called the initial commitment of the protocol). The sequences are chosen under the constraint that for every i,j the value of $\sigma^0_{i,j} \oplus \sigma^1_{i,j}$ equals σ. This is followed by k iterations so that in the j^{th} iteration the prover sends a random k-bit string, $r_j = r_{1,j}, \ldots, r_{k,j}$, and the verifier decommits to $\sigma^{r_{1,j}}_{1,j}, \ldots, \sigma^{r_{k,j}}_{k,j}$.

In the second stage, the prover and verifier engage in the 3-round protocol for Hamiltonicity, where the "challenge" sent by the verifier in the second round of the Hamiltonicity protocol equals σ (at this point the verifier also decommits to all the values $\sigma, \{\sigma^{1-r_{i,j}}_{i,j}\}_{i,j=1}^k$ that were not revealed in the first stage). The protocol is depicted in Figure 4.1.

A $c\mathcal{ZK}$ Proof System for $\mathcal{NP}$

Common Input: A directed graph $G = (V, E)$ with $n \stackrel{\text{def}}{=} |V|$.

Auxiliary Input to Prover: A directed Hamiltonian Cycle, $C \subset E$, in G.

Additional parameter: A super-logarithmic function $k(n)$.

Stage 1: Commitment to challenge $\sigma \in \{0,1\}^n$ (independent of common input):

$P \rightarrow V$: Send first message for perfectly hiding commitment scheme.

$V \rightarrow P$: Commit to random $\sigma, \{\sigma^0_{i,j}\}_{i,j=1}^k, \{\sigma^1_{i,j}\}_{i,j=1}^k$ s.t. $\sigma^0_{i,j} \oplus \sigma^1_{i,j} = \sigma$ for all i,j.

For $j = 1, \ldots, k$:

$P \rightarrow V$: Send a random k-bit string $r_j = r_{1,j}, \ldots, r_{k,j}$.

$V \rightarrow P$: Decommit to $\sigma^{r_{1,j}}_{1,j}, \ldots, \sigma^{r_{k,j}}_{k,j}$.

end (for);

Stage 2: Engage in Blum's 3-round Hamiltonicity protocol using σ as challenge:

$P \rightarrow V$: Use C to produce first prover message of Hamiltonicity protocol.

$V \rightarrow P$: Decommit to σ and to $\{\sigma^{1-r_{i,j}}_{i,j}\}_{i,j=1}^k$.

$P \rightarrow V$: Answer σ with second prover message of Hamiltonicity protocol.

Figure 4.1: Our concurrent zero-knowledge protocol. The first stage is independent of the common input. The second stage consists of a 3-round proof of Hamiltonicity, where the "challenge" sent by V is the n-bit string σ committed to in the first message of the first stage.

Intuitively, since in an actual execution of the protocol, the prover does not know the value of σ, the protocol constitutes a proof system for Hamiltonicity (with soundness error 2^{-n}). However, knowing the value of σ in advance allows the simulation of the protocol: Whenever the simulator may cause the verifier to reveal both $\sigma^0_{i,j}$ and $\sigma^1_{i,j}$ for some i,j (this is done by the means of *rewinding* the verifier after the values $\sigma^{r_{1,j}}_{1,j}, \ldots, \sigma^{r_{k,j}}_{k,j}$ have been revealed), it can simulate the rest of the protocol (and specifically Stage 2) by adjusting the first message of the Hamiltonicity protocol according to the value of $\sigma = \sigma^0_{i,j} \oplus \sigma^1_{i,j}$ (which, as we said, is obtained before entering the Stage 2).

4.1.2 The simulator

(V0), (P1), (V1), ..., (Pk), (Vk) denote the $2k+1$ first stage messages in our protocol and let (`p1`), (`v1`), (`p2`) denote the three (second stage) messages in the Hamiltonicity proof system. Loosely speaking, the simulator is said to rewind the the j^{th} round if after receiving a (Vj) message, it "goes back" to some point preceding the corresponding (Pj) message and "re-executes" the relevant part of the interaction until (Vj) is reached again.

The simulator is said to successfully rewind the j^{th} round, if it manages to receive (Vj) as answer to two *different* (Pj) messages. Note that, once this happens, the simulator has obtained both $\sigma^0_{i,j}$ and $\sigma^1_{i,j}$ for some $i \in \{1, \ldots, k\}$. Thus, if the simulator successfully rewinds even one of the rounds in the first stage then it reveals the verifier's "challenge" (which is equal to $\sigma^0_{i,j} \oplus \sigma^1_{i,j}$). Once the "challenge" is revealed, the simulator can cheat arbitrarily in the second stage of the protocol.

To simplify the analysis, we let the simulator always pick the (Pj)'s uniformly at random. Since the length of the (Pj) messages is super-logarithmic, the probability that *any* two (Pj) messages sent during the simulation are equal is negligible.

Motivating discussion: The binding property of the initial commitment guarantees us that, once $\sigma^0_{i,j}$ and $\sigma^1_{i,j}$ have been revealed, the verifier cannot "change his mind" and decommit to $\sigma \neq \sigma^0_{i,j} \oplus \sigma^1_{i,j}$ at a later stage. However, this remains true *only* if we have not rewound past the initial commitment. As observed by Dwork, Naor and Sahai [15], rewinding a specific session in the concurrent setting may result in rewinding past the initial commitment of other sessions. This means that the "work" done for these sessions may be lost (since once we rewind past the initial commitment of a session all $\sigma^{r_{i,j}}_{i,j}$ values that we have gathered in this session become irrelevant). Consequently, the simulator may find himself doing the same amount of "work" again.

The big question is how to design a simulation strategy that will manage to overcome the above difficulty. One possible approach would be to try and rewind every session at the location that will "minimize the damage". This is the approach taken by Richardson and Kilian [42]. Specifically, for every specific session (out of m concurrent sessions), there must be a $j \in \{1, \ldots, k\}$ so that at most $(m-1)/k$ other sessions start in the interval corresponding to the j^{th} iteration (of this specific session). So if we try to rewind on the correct j, we will invest (and so waste) only work proportional to $(m-1)/k$ sessions. The idea is to avoid the rewinding attempt on the j^{th} iteration if more than $(m-1)/k$ sessions are initiated in the corresponding interval (this will rule out the incorrect j's). The same reasoning applies *recursively* (i.e., to the rewinding in these $(m-1)/k$ sessions).

The drawback of this approach is that it works only when the number of iterations in the preamble is polynomially related to the number of concurrent sessions. Specifically, denoting by $W(m)$ the amount of work invested in m sessions, we obtain the recursion $W(m) = \text{poly}(m) \cdot W(\frac{m-1}{k})$, which solves to $W(m) = m^{\Theta(\log_k m)}$. Thus, whenever $k = n$, we get $W(m) = m^{O(1)}$, whereas taking k to be a constant (or even poly-logarithmic) will cause $W(m)$ to be quasi-polynomial.

A totally different approach is taken by Kilian and Petrank [35]. Rather than concentrating on each session separately and decide on the rewindings according to the schedule as it is being revealed, determine the rewindings *obliviously* of the concurrent scheduling (which is determined "on the fly" by the adversary verifier). Specifically, the order and timing of the simulator's rewindings are determined recursively and depend only on: (1) The length of the execution transcript determined so far. (2) The total number of concurrent sessions (which, by definition, is determined prior to the simulation process). This is essentially the approach taken by us.

```
Procedure SOLVE($\ell$, hist, $\mathcal{T}$):
  1. Bottom level ($\ell = 1$):
      • If session $s$ does not appear in hist, delete all session $s$ messages from $\mathcal{T}$.
      • Uniformly choose a first stage prover message p, and feed $V^*$ with (hist, p).
      • Store $V^*$'s answer v, in $\mathcal{T}$.
      • Output $\mathcal{T}$, (p, v).
  2. Recursive step ($\ell > 1$):
      • Set $\mathcal{T}_1$, $(\tilde{p}_1, \tilde{v}_1, \ldots, \tilde{p}_{\ell/2}, \tilde{v}_{\ell/2}) \leftarrow$ SOLVE($\ell/2$, hist, $\mathcal{T}$).
      • Set $\mathcal{T}_2$, $(p_1, v_1, \ldots, p_{\ell/2}, v_{\ell/2}) \leftarrow$ SOLVE($\ell/2$, hist, $\mathcal{T}_1$).
      • Set $\mathcal{T}_3$, $(\tilde{p}_{\ell/2+1}, \tilde{v}_{\ell/2+1}, \ldots, \tilde{p}_\ell, \tilde{v}_\ell) \leftarrow$ SOLVE($\ell/2$, (hist, $p_1, v_1, \ldots, p_{\ell/2}, v_{\ell/2}$), $\mathcal{T}_2$).
      • Set $\mathcal{T}_4$, $(p_{\ell/2+1}, v_{\ell/2+1}, \ldots, p_\ell, v_\ell) \leftarrow$ SOLVE($\ell/2$, (hist, $p_1, v_1, \ldots, p_{\ell/2}, v_{\ell/2}$), $\mathcal{T}_3$).
      • Output $\mathcal{T}_4$, $(p_1, v_1, \ldots, p_\ell, v_\ell)$.
```

Figure 4.2: The rewinding strategy of our simulator. We stress that the actual "work" is made at the bottom level of the recursion. Even though messages $(\tilde{p}_1, \tilde{v}_1, \ldots, \tilde{p}_\ell, \tilde{v}_\ell)$ do not explicitly appear in the output, some of them (i.e., the ones that are still "relevant") do appear in the table $\mathcal{T}_4$. Notice that the timing of the rewinds is *oblivious* of the scheduling.

The rewinding strategy: The rewinding strategy of our simulator is specified by the SOLVE procedure. The goal of the SOLVE procedure is to supply the simulator with V^*'s "challenges" before reaching the second stage in the protocol. As discussed above, this is done by rewinding the interaction with V^* while trying to achieve two "different" answers to some (Vj) message.

The timing of the rewinds performed by the SOLVE procedure depends only the number of first stage verifier messages received so far (and on the size of the schedule). For the sake of simplicity, we currently ignore second stage messages and refrain from specifying the way they are handled. On a very high level, the SOLVE procedure splits the (first stage) messages it is about to explore into two halves and invokes itself recursively twice for each half (completing the two runs of the first half before proceeding to the two runs of the second half).

At the top level of the recursion, the messages that are about to be explored consist of the entire schedule, whereas at the bottom level the procedure explores only a single message (at this level, the verifier message explored is stored in a special "data-structure", denoted $\mathcal{T}$). The solve procedure always outputs the sequence of "most recently explored" messages.

The input to the SOLVE procedure consists of a triplet $(\ell, \mathsf{hist}, \mathcal{T})$. The parameter ℓ corresponds to the number of verifier messages to be explored, the string hist consists of the messages in the "most recently visited" history of interaction, and $\mathcal{T}$ is a table containing the contents of all the messages explored so far (to be used whenever the second stage is reached in some session).[1]

The simulation is performed by invoking the SOLVE procedure with the appropriate parameters. Specifically, whenever the schedule contains $m = \text{poly}(n)$ sessions, the SOLVE procedure is invoked with input $(m(k+1), \phi, \phi)$ (where $m(k+1)$ is the total number of first stage verifier messages in a schedule of size m). The SOLVE procedure is depicted in Figure 4.2.

[1]The messages stored in $\mathcal{T}$ are used in order to determine the verifier's "challenge" according to "different" answers to (Vj). They are kept "relevant" by constantly keeping track of the sessions that are rewound past their initial commitment. That is, whenever the SOLVE procedure rewinds past the (V0) message of a session, all messages belonging to this session are deleted from $\mathcal{T}$ (since, once this happens, they become irrelevant to the rest of the simulation).

4.2 High Level Analysis of the Simulation

In order to prove the correctness of the simulation, it will be sufficient to show that for every adversary verifier V^*, the three conditions corresponding to the following subsections are satisfied.

4.2.1 The simulator runs in polynomial-time

Each invocation of the SOLVE procedure with parameter $\ell > 1$ involves four recursive invocations of the SOLVE procedure with parameter $\ell/2$. In addition, the work invested at the bottom of the recursion (i.e., when $\ell = 1$) is upper bounded by $\text{poly}(n)$. Thus, the recursive work $W(m \cdot (k+1))$, that is invested by the SOLVE procedure in order to handle $m \cdot (k+1)$ (first stage) verifier messages satisfies $W(m \cdot (k+1)) \leq (m \cdot (k+1))^2 \cdot \text{poly}(n) = \text{poly}(n)$ (see Section 4.5 for details).

4.2.2 The simulator's output is "correctly" distributed

Indistinguishability of the simulator's output from V^*'s view (of $m = \text{poly}(n)$ concurrent interactions with P) is shown assuming that the simulator does not get "stuck" during its execution (see below). Since the simulator S will get "stuck" only with negligible probability, indistinguishability will immediately follow. The key for proving the above lies in the following two properties:

- First stage messages output by S are *identically* distributed to first stage messages sent by P. This property is proved based on the definition of the simulator's actions. (We note that this property is easier to prove for our protocol than it is for the RK protocol.)

- Second stage messages output by S are *computationally indistinguishable* from second stage messages sent by P. This property is proved based on the zero-knowledge property of the underlying protocol (in our case, Blum's Hamiltonicity protocol).

4.2.3 The simulator (almost) never gets "stuck"

This is the most challenging part of the proof. What is required is to show that whenever a session (out of $m = \text{poly}(|x|)$ sessions in the schedule) reaches the second stage in the protocol, the simulator has already managed to obtain the value of the "challenge" corresponding to this session (at least with overwhelming probability). We assume, for simplicity of presentation, that the concurrent scheduling applied by V^* is fixed in advance (where by "fixed schedule" we mean a schedule that does not vary "dynamically" as a function of the messages that V^* has seen so far). The ideas for coping with "dynamic" schedulings are presented in the actual proof.

Partitioning the schedule into rewind intervals: The execution of the SOLVE procedure induces a partitioning of the $2 \cdot m \cdot (k+1)$ (prover and verifier) messages in the schedule into *disjoint* rewind intervals. At the top level of the recursion there are two disjoint intervals of length $m \cdot (k+1)$ and at the bottom of the recursion there are $m \cdot (k+1)$ disjoint intervals of length 2. In general, at the w^{th} level of the recursion (out of $d = \log_2(m \cdot (k+1))$ possible levels) there are 2^w disjoint intervals of $m(k+1)/2^{w+1}$ messages each.

Notice that rewind intervals may contain messages from all sessions. Also notice, that a rewind interval may be "visited" multiple times during the execution of the SOLVE procedure (in particular, a level-w interval is visited exactly 2^w times during the simulation). The fixed schedule assumption implies that each time an interval is "visited", it will contain the same scheduling of messages.

Minimal rewind intervals: We denote by $[a, b]$ an interval starting with prover message a and ending with verifier message b. Focusing on messages of a specific session, we note that for every pair of messages $(\mathrm{P}j), (\mathrm{V}j)$ in this session we can associate a level-w interval $[a_j, b_j]$ so that:

1. Both $(\mathrm{P}j)$ and $(\mathrm{V}j)$ are contained in $[a_j, b_j]$.

2. None of the level-$(w+1)$ sub-intervals of $[a_j, b_j]$ contains both $(\mathrm{P}j)$ and $(\mathrm{V}j)$.

We call such a rewind interval a j-minimal interval. Notice that for every $j \in \{1, \ldots, k\}$ there is only one j-minimal interval $[a_j, b_j]$ (and that for every $j \neq j'$ the interval $[a_j, b_j]$ is different from $[a_{j'}, b_{j'}]$).

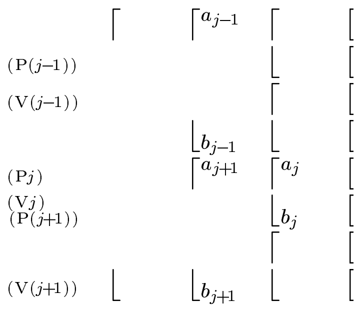


Figure 4.3: Demonstrates the way in which minimal intervals are determined. Also demonstrates possible containments between minimal intervals of different iterations. In this example, the intervals $[a_{j-1}, b_{j-1}]$ and $[a_{j+1}, b_{j+1}]$ are disjoint (as well as the intervals $[a_{j-1}, b_{j-1}]$ and $[a_j, b_j]$), whereas the interval $[a_{j+1}, b_{j+1}]$ contains $[a_j, b_j]$.

In some sense j-minimal intervals correspond to the shortest interval in which the simulator can rewind message $(\mathrm{V}j)$ (that is, while potentially changing the value of $(\mathrm{P}j)$). Intuitively, for such a rewinding to be useful, the interval should not contain message (V0). Otherwise, the value that was revealed in some run of the interval becomes irrelevant once the rewinding is performed (since all the relevant values in the $\mathcal{T}$ table are deleted whenever we rewind past (V0)). Likewise, the interval should not contain message (`p1`). Otherwise, the simulation faces the risk of getting "stuck" before it manages to reveal multiple $(\mathrm{P}j), (\mathrm{V}j)$ pairs of messages (by running the interval twice).

To rule out the above possibilities we focus on j-minimal intervals that do not contain neither (V0) nor (`p1`) (such intervals are said to be `good`). It can be seen that the number of minimal intervals that do not contain neither (V0) nor (`p1`) is at least $k - 2d$. This just follows from the fact that in every level the (V0) (resp. (`p1`)) message is contained in exactly one interval. In particular, the number of minimal intervals that are "spoiled" by (V0) (resp. (`p1`)) is at most d.

At this point, the simulator's task may seem easy to achieve. Indeed, if V^* acts according to the prescribed verifier strategy, then all that the simulator has to do is to run a good interval twice.[2] Since V^* is acting honestly, we are guaranteed that, with overwhelming probability, in each of the two runs the simulator obtains a "different" $(\mathrm{V}j)$ message. In such a case, it will be sufficient to require that there *exists* a good interval. By the above discussion this is guaranteed whenever $k > 2d$ (and since $d = O(\log n)$, setting $k = w(\log n)$ will do).

[2]Observe that whenever $[a_j, b_j]$ is reached during the simulation then it is run twice.

Dealing with ABORT messages: Unfortunately, the adversary verifier V^* may arbitrarily deviate from the prescribed strategy. In particular, it may be the case that throughout its interaction with the prover (simulator), V^* occasionally sends an ABORT message (in other words, V^* may potentially refuse to decommit to a previous commitment). Clearly, such an action on behalf of the verifier is considered illegal, and the interaction in the relevant session stops (i.e., there is no need to continue exchanging messages in this session). This may seem as good news (since, once this happens, the simulator does not really need to "invest" any more work in the corresponding session).

The problem is that V^* does not *always* refuse to decommit (but may refuse with some probability $0 \leq p \leq 1$, which is not known in advance by the simulator). Thus, if we focus on two consecutive runs of a specific interval, the simulator may find himself in a situation in which the first run is answered with ABORT whereas the second run of the interval is "properly answered". This means that the simulator has not managed to obtain the "challenge" from the two runs of this interval, and it thus faces the risk of getting "stuck" at a later stage of the interaction.

One naïve solution would be to let the simulator always output the run in which V^* has refused to decommit (that is, whenever it gets "stuck"). The problem with this solution is that it "skews" the distribution of transcripts outputted by the simulator towards transcripts that contain too many ill-formed messages. This may cause a too large deviation of the simulator's output distribution from the distribution of "real" interactions (between V^* and the honest prover P).

In our solution we have chosen to let the simulator always output the "most recently explored" run. This choice guarantees that the simulator indeed produces the "correct" distribution of first stage messages (in the sense discussed above). However it makes him face the risk that V^* aborts in the first run of an interval and "properly answers" in the second run.

Achieving "independent" rewinds: Let p_j denote the probability that V^* sends a "proper" (Vj) message. Using this notation, the probability that V^* aborts in the first run of $[a_j, b_j]$ but "properly answers" in the second run is equal to $(1 - p_j) \cdot p_j \leq 1/4$ (we call this a "bad" event). Let $k' < k$ be the number of good intervals in the simulation. At first glance, it may seem that the probability of the above "bad" event to occur in *all* good intervals is upper bounded by $(1/4)^{k'}$ (which means that the probability of getting "stuck" is negligible whenever $k' = \omega(\log n)$).

However, this reasoning applies *only* when all runs of the good intervals are *independent*. Unfortunately, very strong dependencies may exist between different good intervals. This will happen whenever one good interval contains another good interval. In such a case, aborting in the first run of one interval, may immediately imply abort in the first run of the other interval.

The solution is to focus on a set of *disjoint* intervals. Such intervals do not suffer from the dependencies described above, and can be shown to be "bad" independently from other (disjoint) intervals. The abundance of disjoint intervals can be easily guaranteed by taking k sufficiently large. Specifically, if $k = \omega(\log^2 n)$ (as in the Kilian-Petrank simulator [35]), then there must exist a level in the recursion (out of $d = \log(m \cdot (k+1)) = O(\log n)$ levels) that contains at least $k' = \omega(\log^2 n)/d = \omega(\log n)$ good intervals. Since same level intervals are all disjoint, then their runs are "independent". In particular, the probability that for all of them the "bad" event will occur is negligible.

Special intervals: Unfortunately, the argument establishing the abundance of disjoint intervals does not extend to the case when $k = \tilde{O}(\log n)$. Here we are not guaranteed that there exists a level with many good intervals. In fact, there may exists only few (i.e., $k' = o(\log n)$) disjoint intervals. To overcome this obstacle, we use a completely different approach. Rather than proving the existence of a *large* set of disjoint intervals (each being executed twice), we prove the existence of a (possibly small) set of disjoint intervals and guarantee that the *total* number of executions of these intervals is large. By doing so, we exploit the fact that, from the time that (V0) is visited until the time that (`p1`) is reached, the simulator *typically* visits each rewinding interval *many times* (and not just twice as we assumed before).

Specifically, for every scheduling applied by V^*, we define our set of intervals as the set of all minimal intervals that do not contain any other minimal interval (i.e., intervals $[a_j, b_j]$ that do not contain $[a_{j'}, b_{j'}]$ for any $j' \neq j$). We call such intervals special intervals. Notice that all special intervals are disjoint. We let $S \subseteq \{1, \ldots, k\}$ denote the set of all indices j for which $[a_j, b_j]$ is special. For simplicity, assume that $S = \{1, \ldots, |S|\}$.

Our goal will be to bound (from below) the total number of times that special intervals are visited. To do so, we introduce a notion of “distance” between consecutive special intervals. This “distance” is supposed to reflect the number of times that a certain special interval has been executed since the last time that the preceding special interval has been visited. For every $j \in S$, we let d_j denote the “distance” of $[a_j, b_j]$ from $[a_{j-1}, b_{j-1}]$.[3] Using this definition we show that, no matter what is the scheduling strategy applied by V^*, the following two conditions are always satisfied:

1. The number of “independent” runs of $[a_j, b_j]$ since $[a_{j-1}, b_{j-1}]$ has been last visited is 2^{d_j}.
2. $\sum_{j \in S} d_j \geq k - d$.

Loosely speaking, Item 1 follows from the definition of d_j and from the fact that $[a_j, b_j]$ and $[a_{j-1}, b_{j-1}]$ are disjoint. As for Item 2, this is a combinatorial statement on binary trees, that is proved by induction on the number of minimal intervals in the “recursion tree”.

Bounding the failure probability: Recall that we are interested in the probability that the “bad” event occurs during the simulation. Whereas in the previous analysis, this happened only if for all intervals the first run was aborted and the second was “properly answered”, in the current analysis the simulator will fail only if for *every* $j \in \{1, \ldots, |S|\}$, it holds that the first $2^{d_j} - 1$ runs of the interval $[a_j, b_j]$ are aborted and the last one is ”properly answered” (since otherwise the simulator has managed to obtain two “different” answers to (Vj)).

Let $\mathcal{R}$ be the set of all random tapes used by the simulator. A specific $\rho \in \mathcal{R}$ is said to be “bad” if the “bad” event occurs during a simulation that uses ρ as random tape (if the “bad” event does not occur during the simulation then ρ is called “good”). We shall show that the fraction of ”bad” tapes $\rho \in \mathcal{R}$ is negligible. To do this we will show that every “bad” random tape can be mapped into a set of *super-polynomially* many other “good” random tapes so that every two “bad” random tapes are mapped to two *disjoint* sets of “good” random tapes. This would imply that for every random tape that leads to the simulator’s failure there exist super-polynomially many other tapes that do not. Since the simulator picks a random tape uniformly amongst all possible random tapes, it follows that the simulator’s failure probability is negligible.

Mapping a “bad” random tape to many “good” ones: Let $u_1, \ldots, u_{|S|}$ (where for every $j \in \{1, \ldots, |S|\}$, the value of u_j is chosen in $\{1, \ldots, 2^{d_j}\}$). We map a random tape $\rho \in \mathcal{R}$ into another random tape $\rho' \in \mathcal{R}$ by swapping the portion of ρ used to produce prover messages in the u_j^{th} run of $[a_j, b_j]$ with the portion used in the $(2^{d_j})^{\text{th}}$ run (this is done for all $j \in S$). The swappings are made possible due to the following facts: (1) Prover messages in interval $[a_j, b_j]$ are produced using ”fresh” randomness each time it is visited. (2) If two intervals $[a_j, b_j]$ and $[a_{j'}, b_{j'}]$

[3]The value d_j is defined as the ”recursive depth” of $[a_j, b_j]$ relative to the “common ancestor” of $[a_j, b_j]$ and $[a_{j-1}, b_{j-1}]$ (i.e., relative to the smallest rewind interval containing both $[a_j, b_j]$ and $[a_{j-1}, b_{j-1}]$).

are disjoint then so is the randomness used to produce their corresponding prover messages (recall that all special intervals are disjoint).

We claim that if ρ is a "bad" random tape, then after the swappings have been applied, the resulting tape ρ' is "good". To see this, consider the smallest $j \in S$ for which $u_j \neq 2^{d_j}$ (i.e., for which the u_j^{th} run of $[a_j, b_j]$ and the $(2^{d_j})^{\text{th}}$ run have been actually swapped). The key observation is that, once the swappings have been applied to ρ, the last run of $[a_j, b_j]$ is aborted (and one of the first $2^{d_j} - 1$ runs is "properly answered").[4] In other words, there exists a $j \in S$ for which the "bad" event does not occur during the simulation (and so ρ' is "good").[5]

The above argument will apply as long as the sequence $u_1, \ldots, u_{|S|}$ causes the randomness of at least one special interval to be swapped. The number of possibilities to choose $u_1, \ldots, u_{|S|}$ so that this happens (i.e., the randomness of at least one special interval is swapped) is:

$$\prod_{j \in S} 2^{d_j} - 1 \quad = \quad 2^{\sum_{j \in S} d_j} - 1 \quad \geq \quad 2^{k-d} - 1$$

(the sequence $u_1, \ldots, u_{|S|} = 2^{d_1}, \ldots, 2^{d_{|S|}}$ being the only one that leaves the coin tosses intact). Overall, we get that a single "bad" random tape $\rho \in \mathcal{R}$ can be mapped to as many as $2^{k-d} - 1$ other "good" random tapes. As we show below, any two such "bad" tapes will be mapped to *disjoint* sets of "good" tapes and so the fraction of "bad" random tapes is at most $2^{k-d} = 2^{k-O(\log n)}$. Thus, whenever $k = \omega(\log n)$, the probability that the simulator gets "stuck" is negligible.

Defining an "inverse" mapping: To argue that any two "bad" random tapes are mapped to disjoint sets of "good" tapes we will define an "inverse" to the above mapping. To do this, we should be able, given a "good" random tape $\rho' \in \mathcal{R}$, to determine the value of u_j for every $j \in \{1, \ldots, |S|\}$ (that is, we should be able to determine with which run of $[a_j, b_j]$ the last run was swapped).

In order to to determine the value of the u_j's we will run the simulation with ρ' as random tape and examine for which special intervals one of the first $2^{d_j} - 1$ runs is "properly answered" and the last run is aborted by V^*. Once u_j is determined for some interval, we will swap back its randomness and continue to inspect and swap the next special interval.

If we take care of inspecting the intervals and reversing the swapping of their randomness "inductively", we are guaranteed that for every interval that we are examining exactly one of the runs is "properly answered" and the others are aborted. Loosely speaking, this follows from the fact that the "good" tape that we are trying to invert originates from a "bad" tape in which every interval is aborted in the first $2^{d_j} - 1$ runs and "properly answered" in the last run.

The reason for which the order of swapping is important is that V^*'s answer in a specific interval also depends on the randomness used to run the "most recent execution" of previous intervals (since, whenever we reach a specific interval, the outcome of these "recent" runs appears in the history of the interaction). In order to be able to say something meaningful about an interval's run we must make sure that, whenever we inspect the run of the simulator on this interval, the history of the interaction up to the starting point of the interval is consistent with the outcome of running the simulator with the "bad" tape that we are aiming to obtain.

As soon as we reach the last special interval we know that the resulting tape is the original "bad" random tape (since all along the way we preserve the "invariant" that the randomness used so far is consistent with the original "bad" random tape).

[4]Here we rely on the fact that the simulator's coin tosses completely determine the outcome of an interval's run (that is, modulo the history of the interaction up to the starting point of the interval).

[5]To see that the simulator does not get "stuck" when using ρ' as its random tape, notice that when reaching the second stage of the corresponding session, the simulator will not have to do anything in order to successfully produce a second stage transcript (since all second stage messages should appear as being aborted anyway).

4.3 A Detailed Description of our Protocol

4.3.1 Blum's protocol

We consider n parallel repetitions of the following basic proof system for the *Hamiltonian Cycle* (HC) problem which is $\mathcal{NP}$-complete (and thus get proof systems for any language in $\mathcal{NP}$) [6, 22]. We consider directed graphs (and the existence of directed Hamiltonian cycles).

Construction 4.3.1 (Basic proof system for HC):

- Common Input: *a directed graph* $G = (V, E)$ *with* $n \overset{\text{def}}{=} |V|$.
- Auxiliary Input to Prover: *a directed Hamiltonian Cycle,* $C \subset E$, *in* G.
- Prover's first step ($\widehat{\texttt{p1}}$): *Select a random permutation,* π, *of the vertices* V, *and commit (using a perfectly-binding commitment scheme) to the entries of the adjacency matrix of the resulting permuted graph. That is, send an* n*-by-*n *matrix of commitments so that the* $(\pi(i), \pi(j))^{\text{th}}$ *entry is a commitment to 1 if* $(i, j) \in E$, *and is a commitment to 0 otherwise.*
- Verifier's first step ($\widehat{\texttt{v1}}$): *Uniformly select* $\sigma \in \{0, 1\}$ *and send it to the prover.*
- Prover's second step ($\widehat{\texttt{p2}}$): *If* $\sigma = 0$, *send* π *to the verifier along with the revealing (i.e., preimages) of all commitments. Otherwise, reveal only the commitments to entries* $(\pi(i), \pi(j))$ *with* $(i, j) \in C$. *In both cases also supply the corresponding decommitments.*
- Verifier's second step ($\widehat{\texttt{v2}}$): *If* $\sigma = 0$, *check that the revealed graph is indeed isomorphic, via* π, *to* G. *Otherwise, just check that all revealed values are 1 and that the corresponding entries form a simple* n*-cycle. In both cases check that the decommitments are proper (i.e., that they fit the corresponding commitments). Accept if and only if the corresponding condition holds.*

A key propery of the above protocol (which is also satisfied by many other known protocols) is that if the prover knows the contents of verifier's "challenge" message σ (i.e., as determined in Step ($\widehat{\texttt{v1}}$)) prior to sending its own first message (i.e., as determined in Step ($\widehat{\texttt{p1}}$)), then it is able to convince the verifier that G contains an Hamiltionian cycle even without knowing such a cycle (actually, it will convince the verifier even if the graph does not contain an Hamiltionian cycle).

The reason for this is that in such a case, the prover can set up its first message according to σ in a way that will always make the verifier accept in Step ($\widehat{\texttt{v2}}$). Specifically, knowing in advance that $\sigma = 0$, the prover will commit to the entries of the adjacency matrix of the permuted graph (as specified in Step ($\widehat{\texttt{p1}}$) of Construction 4.3.1), thus being able to reveal a permutation π and the preimages of all commitments in Step ($\widehat{\texttt{p2}}$). On the other hand, knowing in advance that $\sigma = 1$, the prover will commit to the full graph K_n, thus being able to open an arbitrary cycle in the supposedly permuted graph.

As a side remark, we observe that the above property is in fact sufficient in order to prove that a single execution of Construction 4.3.1 is black-box zero-knowledge in the "stand alone" setting.[6] All that the simulator has to do is to try and "guess" the value of σ prior to determining the value of the prover's first message (and keep trying until it suceeds).

[6]This is in contrast to the protocol obtained by conducting n parallel repetitions of the basic Hamiltonicity proof system (from Construction 4.3.1), which cannot be proved to be black-box zero-knowldege (unless $\mathcal{NP} \subseteq \mathcal{BPP}$) [24].

4.3.2 The actual protocol

Using Construction 4.3.1 as a building block, we are now ready to present a concurrent zero-knowledge proof system for Hamiltonicity. Since Hamiltonicity is $\mathcal{NP}$-complete, it will follow that every language in $\mathcal{NP}$ can be proved in concurrent zero-knowledge.

Construction 4.3.2 (A $c\mathcal{ZK}$ proof system for HC):

- Common Input: *a directed graph* $G = (V, E)$ *with* $n \overset{\text{def}}{=} |V|$, *and a parameter* $k = k(n)$ *(determining the number of rounds).*
- Auxiliary Input to Prover: *a directed Hamiltonian Cycle,* $C \subset E$, *in* G.
- First stage: *This stage involves* $2k + 2$ *rounds and is independent of the common input* G.
  1. Prover's preliminary step (P0): *Uniformly select a first message for a (2-round) perfectly-hiding commitment scheme and send it to the verifier.*
  2. Verifier's preliminary step (V0): *Uniformly select* $\sigma \in \{0,1\}^n$, *and two sequences,* $\{\sigma^0_{i,j}\}^k_{i,j=1}$, $\{\sigma^1_{i,j}\}^k_{i,j=1}$, *each consisting of* k^2 *random* n*-bit strings. The sequences are chosen under the constraint that for every* i, j *the value of* $\sigma^0_{i,j} \oplus \sigma^1_{i,j}$ *equals* σ. *Commit (using the perfectly-hiding commitment scheme) to all* $2k^2 + 1$ *selected strings. The commitments are denoted* $\beta, \{\beta^0_{i,j}\}^k_{i,j=1}$, $\{\beta^1_{i,j}\}^k_{i,j=1}$.
  3. *For* $j = 1, \ldots, k$:
     (a) Prover's j^{th} step (Pj): *Uniformly select a* k*-bit string* $r_j = r_{1,j}, \ldots, r_{k,j} \in \{0,1\}^k$ *and send it to the verifier.*
     (b) Verifier's j^{th} step (Vj): *Reveal the values (preimages) of* $\beta^{r_{1,j}}_{1,j}, \ldots, \beta^{r_{k,j}}_{k,j}$.
  4. *The prover proceeds with the execution if and only if for every* $j \in \{1, \ldots, k\}$, *the verifier has properly decommited to the values of* $\sigma^{r_{1,j}}_{1,j}, \ldots, \sigma^{r_{k,j}}_{k,j}$ *(i.e., that for every* $i \in \{1, \ldots, k\}$, $\sigma^{r_{i,j}}_{i,j}$ *is a valid decommitment of* $\beta^{r_{i,j}}_{i,j}$*).*
- Second stage: *The prover and verifier engage in* n *(parallel) executions of a slightly modified version of the basic Hamiltonicity protocol (described in Construction 4.3.1):*
  1. Prover's first step (`p1`): *Send the first message in the Hamiltonicity proof system (i.e.,* n *parallel copies of Step* ($\widehat{\texttt{p1}}$) *in Construction 4.3.1).*
  2. Verifier's first step (`v1`): *Reveal the value (i.e., preimage) of* β *(which is supposed to be equal to* σ*). Also reveal the value of all* k^2 *commitments that have not been revealed in the first stage (i.e., the values of all* $\{\beta^{1-r_{i,j}}_{i,j}\}^k_{i,j=1}$*).*
  3. Prover's second step (`p2`): *Check that the verifier has properly decommited to the values of* σ *and* $\{\sigma^{1-r_{i,j}}_{i,j}\}^k_{i,j=1}$ *(in particular, check that* $\sigma^0_{i,j} \oplus \sigma^1_{i,j}$ *indeed equals* σ *for all* j*). If so, send the third message in the basic Hamiltonicity proof system (i.e.,* n *parallel copies of Step* ($\widehat{\texttt{p2}}$) *in Construction 4.3.1).*
  4. Verifier's second step (`v2`): *Conduct the verification of the prover's proofs (i.e., as described in Step* ($\widehat{\texttt{v2}}$) *of Construction 4.3.1), and accept if and only if all corresponding conditions hold.*

We next argue that Construction 4.3.2 indeed constitutes an interactive-proof for the language HC.

Completeness

Completeness of the above proof system (i.e., Construction 4.3.2) follows from the perfect completeness of the basic Hamiltonicity proof system (i.e., Construction 4.3.1). Suppose that the input graph G is indeed Hamiltonian. Then if the prover follows the prescribed program P, the verifier will always accept (i.e., accept with probability 1). Specifically, in order to successfully conduct the first stage, all that the prover has to do is to send an initialization message for the perfectly-hiding commitments scheme, and k uniformly and independently chosen k-bit strings, one string per each round in the first stage (this can be done even wihtout knowing an Hamiltonian cycle in G). As for the second stage, since the prover knows an Hamiltionian cycle $C \subset E$ in G, then no matter what is the "challenge" sent by the verifier in Step (`v1`), the perfect completeness of Construction 4.3.1 guarantees that the prover will be always able to answer properly in Step (`p2`) (thus making the verifier accept).

Soundness

Soundness of the above proof system (i.e. Construction 4.3.2) follows from soundness of the basic Hamiltonicity proof system (i.e., Construction 4.3.1), and from the perfectly-hiding property of the commitment sent by the verifier in Step (V0). Suppose that the input graph G is not Hamiltonian. Then no matter what the prover does, the k^2 values, $\{\sigma_{i,1}^{r_{i,1}}\}_{i=1}^{k}, \ldots, \{\sigma_{i,k}^{r_{i,k}}\}_{i=1}^{k}$, that are revealed by the verifier in the first stage are uniformly and independently chosen (and so reveal no information about the actual value of σ). Since the commitment scheme used by the verifier in Step (V0) is perfectly-hiding, we deduce that when reaching Step (`p1`) the prover has "no idea" about the value of the "challenge" σ that is about to be revealed in Step (`v2`) (i.e., as far as the information available to the prover is concerned each possiblity is almost equally likely). In other words, even though the cheating prover reaches the second stage (i.e., Step (`p1`)) after seeing all messages in the first stage, the messages in the second stage are (almost) statistically independent of the verifier's messages in the first stage. A standard argument can be then used to demonstrate how a cheating prover for Construction 4.3.2 is transformed into an (all-powerful) cheating prover for Construction 4.3.1 (with only a negligible difference in the cheating probability), in contradiction to the soundness property of Construction 4.3.1. Hence, we get:

Proposition 4.3.3 *Construction 4.3.2 constitutes an interactive proof system for Hamiltonicity.*

4.4 Zero-Knowledge

In order to demonstrate the zero-knowledge property of Construction 4.3.2, we will show that for every polynomial $p(\cdot)$ there exists a "universal" black-box simulator, S_p, so that for every $G = (V, E) \in \mathrm{HC}$ and concurrent adversary verifier V^* (running at most $p(|V|)$ concurrent sessions), $S_p(G)$ runs in time $\mathrm{poly}(n)$ (where $n = |V|$), and satisfies that the ensemble $\{\mathrm{view}_{V^*}^{P}(G)\}_{G \in HC}$ is computationally indistinguishable from the ensemble $\{S_p^{V^*}(G)\}_{G \in HC}$.

4.4.1 The simulator's strategy

We assume that the number of sessions that are run by the concurrent adversary verifier V^* is fixed in advance and known to everybody. We denote it by m ($= \mathrm{poly}(n)$). The simulator S_m starts by selecting and fixing a random tape r for V^*. It then proceeds by exploring various prefixes of possible interactions between P and V^*. This is done while having only "black-box" access to V^*'s strategy (as described in Section 2.6). For simplicity of presentation, we partition the description of the simulator's strategy into two disjoint (but not independent) procedures. The first procedure handles the messages that are exchanged in the first stage of the protocol. This is done while completely ignoring the messages of the second stage. The second procedure handles the messages in the second stage while using auxiliary information produced by the first procedure. This information is located in some "global" data-structure that is dynamically updated (by the first procedure) as the simulation proceeds. To complete the simulator's description we describe how the two procedures can be merged into one super-procedure that with overwhelming probability outputs a "legal" transcript (representing a concurrent interaction between P and V^*). The analysis of the simulator's running time and output distribution are then presented in Sections 4.5, 4.6 and 4.7.

Handling first stage messages

First stage messages are handled by the SOLVE procedure. The goal of this procedure is to supply the simulator with the values of V^*'s "challenges" before it reaches the second stage in the protocol (where by "challenges" we refer to messages that correspond to Step (`v1`) of Construction 4.3.2). To this end, the SOLVE procedure tries to make sure that for every session (out of m concurrent sessions) there exists $i, j \in \{1, \ldots, k\} \times \{1, \ldots, k\}$ for which the verifier V^* has properly revealed the values of both $\sigma^0_{i,j}$ and $\sigma^1_{i,j}$ (during the simulation process). This should always take place prior to reaching the second stage of the corresponding session (or otherwise, the simulator will get "stuck"). Once both $\sigma^0_{i,j}$ and $\sigma^1_{i,j}$ are revealed, the value of V^*'s challenge (which should be equal to $\sigma^0_{i,j} \oplus \sigma^1_{i,j}$) can be easily determined, and the required goal is indeed achieved.

In order to receive both $\sigma^0_{i,j}$ and $\sigma^1_{i,j}$ (i.e., in some (Vj) message of a specific session), the simulator must explore at least two different interaction prefixes in which the corresponding (Pj) message is different. The way this is done is by means of "rewinding" the interaction with V^* to a point in the schedule that precedes the (Pj) message (while hoping that the (Pj) message is indeed modified in the process).[7]

The rewinding strategy of the SOLVE procedure is recursive and is essentially identical to the simulation strategy suggested by Kilian-Petrank [35]. The key idea underlying this simulation strategy is that the order and timing of the simulator's rewinds are determined *obliviously* of the concurrent scheduling (which is determined "on the fly" by the adversary verifier V^*). Specifically, the order and timing of the rewinds depend only on m, the number of concurrent sessions (which, by definition, is determined prior to the simulation process), and on the length of the execution prefix explored so far.[8]

The "global" data-structure: To store the values it has discovered about the verifier's "challenge" in session $s \in \{1, \ldots, m\}$, the SOLVE procedure will write information into a table denoted $\mathcal{T}$. This table will contain the (first stage) verifier messages that have been revealed so far (such

[7]Note that great care should be taken in planning the rewinding strategy. As we have previously mentioned, rewinding a specific session in the concurrent setting may result in loss of work done for other sessions, and cause the simulator to do the same amount of work again. In particular, all simulation work done for sessions starting after the point to which to rewind may be lost (since the revealed values of $\sigma^0_{i,j}$ and $\sigma^1_{i,j}$ become irrelevant once we rewind to a point preceding the verifier's preliminary commitment in Step (V0)). Conducting a "wasteful" rewinding strategy may cause the work done by the simulator to accumulate to too much (thus causing the simulator to run in super-polynomial time).

[8]This is in contrast to the rewinding strategy of the Richardson-Kilian simulator [42] which heavily depends on the schedule as it is being revealed (remember that the scheduling is dynamically determined by the adversary verifier and is not necessarily known to the simulator in advance).

messages may consist of the opened values of some $\{\sigma_{i,j}^{r_j}\}_{i=1}^{k}$'s or, alternatively, of an ABORT message). As we have already mentioned, "rewinds" that take place during the simulation process may render part of the data stored in the $\mathcal{T}$ table irrelevant. In particular, whenever the interaction is rewound to a point that precedes the verifier's preliminary commitment in session s (i.e., a (V0) message), all the values corresponding to session s in the $\mathcal{T}$ table are not relevant any more. In such cases, these values should be deleted from the table and the accumulation of information for session s should restart from scratch.

The input of the SOLVE procedure: The SOLVE procedure is given three arguments as input. The first argument, denoted ℓ, is a parameter determining the total number of verifier messages that the SOLVE procedure should handle. At the top level of the recursion, the argument ℓ equals $m \cdot (k+1)$, which is the total number of (first stage) verifier messages in a schedule of m sessions (that is, including the verifier's preliminary step, (V0)). At the bottom level of the recursion, the SOLVE procedure should handle a single (first stage) verifier message (that is, ℓ equals 1).

The second argument given to the SOLVE procedure, denoted hist, is a sequence of alternating prover and verifier messages which corresponds to the "most recently visited" history of the interaction (as induced by the simulator's queries). In accordance with our conventions, all queries made by the relevant invocation of the SOLVE procedure will have hist as their prefix. At the top level of the recursion, the hist argument is initialized as an empty list and becomes increasingly longer as the simulation proceeds (its eventual length being $2m \cdot (k+1)$). In intermediate stages of the recursion the hist argument may initially be of arbitrary length and is eventually augmented with a suffix containing a total of 2ℓ (prover and verifier) messages.

The third argument of the SOLVE procedure is the table $\mathcal{T}$. As mentioned above, this argument is used in order to store the first stage messages revealed so far. In order to keep these messages relevant, the SOLVE procedure will inspect the hist argument to see for which sessions the (V0) message does not appear in the history of the interaction. Since for such sessions, any value that is possibly stored in the $\mathcal{T}$ table is not relevant for the rest of the simulation (see above discussion about the "global" data-structure), the SOLVE procedure will delete this value from the $\mathcal{T}$ table and will restart the accumulation of information for these sessions from scratch.

The SOLVE procedure: We are now ready to proceed with the description of the SOLVE procedure. Given ℓ, hist and $\mathcal{T}$ as inputs, the SOLVE procedure acts as follows (see also Figure 4.2):

If $\ell = 1$ (i.e., we are at the bottom level of the recursion):

1. If (V0) message of session s does not appear in hist, delete all session s messages from $\mathcal{T}$.
2. Uniformly choose a first stage prover message p, and feed V^* with $\overline{q} = (\mathsf{hist}, \mathrm{p})$.
3. Store V^*'s answer v, in $\mathcal{T}$.[9]
4. Output (p, v), $\mathcal{T}$.

If $\ell > 1$ (i.e., we are at some intermediate level of the recursion):

1. Invoke the SOLVE procedure recursively with parameters $\ell/2$, hist and $\mathcal{T}$. The recursive invocation outputs a table $\mathcal{T}_1$, as well as a transcript of ℓ (first stage) messages denoted $(\tilde{\mathrm{p}}_1, \tilde{\mathrm{v}}_1, \ldots, \tilde{\mathrm{p}}_{\ell/2}, \tilde{\mathrm{v}}_{\ell/2})$.

[9]The message $\mathrm{v} = V^*(\overline{q})$ consists of a first stage verifier message in session $s \in \{1, \ldots, m\}$. It is either of the form (V0) or (Vj) for some $j \in \{1, \ldots, k\}$ (supposedly containing the "legal" openings of $\sigma_{1,j}^{r_{1,j}}, \ldots, \sigma_{k,j}^{r_{k,j}}$).

2. "Rewind" the interaction and perform Step 1 again. That is, invoke the SOLVE procedure recursively with parameters $\ell/2$, hist and $\mathcal{T}$. The recursive invocation outputs a table $\mathcal{T}_2$, as well as a transcript of ℓ (first stage) messages denoted $(\mathtt{p}_1, \mathtt{v}_1, \ldots, \mathtt{p}_{\ell/2}, \mathtt{v}_{\ell/2})$.[10]
3. Augment the hist argument with the "most recently visited" transcript (that is, the transcript $(\mathtt{p}_1, \mathtt{v}_1, \ldots, \mathtt{p}_{\ell/2}, \mathtt{v}_{\ell/2})$ computed in Step 2) and invoke recursively the SOLVE procedure with parameters $\ell/2$, $(\mathsf{hist}, \mathtt{p}_1, \mathtt{v}_1, \ldots, \mathtt{p}_{\ell/2}, \mathtt{v}_{\ell/2})$ and $\mathcal{T}_2$. The recursive invocation outputs a table $\mathcal{T}_3$, as well as a transcript of the ℓ subsequent (first stage) messages denoted $(\tilde{\mathtt{p}}_{\ell/2+1}, \tilde{\mathtt{v}}_{\ell/2+1}, \ldots, \tilde{\mathtt{p}}_{\ell}, \tilde{\mathtt{v}}_{\ell})$.
4. "Rewind" the interaction and perform Step 3 again. That is, invoke the SOLVE procedure recursively with parameters $\ell/2$, $(\mathsf{hist}, \mathtt{p}_1, \mathtt{v}_1, \ldots, \mathtt{p}_{\ell/2}, \mathtt{v}_{\ell/2})$ and $\mathcal{T}_2$. The recursive invocation outputs a table $\mathcal{T}_4$, as well as a transcript of the ℓ subsequent (first stage) messages denoted $(\mathtt{p}_{\ell/2+1}, \mathtt{v}_{\ell/2+1}, \ldots, \mathtt{p}_{\ell}, \mathtt{v}_{\ell})$.[11]
5. Output $\mathcal{T}_4$ and the "most recently visited" transcript, which consists of the messages $(\mathtt{p}_1, \mathtt{v}_1, \ldots, \mathtt{p}_{\ell}, \mathtt{v}_{\ell})$.

Some comments: Notice that the order and timing of the "rewinds" performed by the SOLVE procedure are determined obliviously of the concurrent schedule (whereas the order in which the $\mathcal{T}$ table is updated *does* depend on the scheduling of the various messages in the various sessions). Also note that, as opposed to the [42, 35] simulation strategies, the values of the (first stage) prover messages (i.e., of (Pj) messages) do not depend on the values revealed by the verifier in the corresponding answers (i.e., in the (Vj) messages), but are rather chosen uniformly and independently each time. Since the transcript output by the simulator consists of the prover/verifier messages that were "most recently visited" by the SOLVE procedure, the first stage messages that eventually appear in the simulator's output are *identically* distributed to "real" first stage messages (i.e., messages that are actually exchanged between an honest prover P and the verifier V^*).

Updating the $\mathcal{T}$ table: The $\mathcal{T}$ table is updated only when visiting the bottom level of the recursion. Given a first stage verifier message $\mathtt{v}$, the SOLVE procedure determines the session number, $s \in \{1, \ldots, m\}$, of the corresponding (Vj) message (according to the session identifiers that appear in $\mathtt{v}$) and stores (Vj) in $\mathcal{T}$. The (Vj) message may either contain a sequence $\sigma_{1,j}^{r_{1,j}}, \ldots, \sigma_{1,j}^{r_{1,j}}$ of n-bit strings or an `ABORT` message. Since the (Pj) message to which (Vj) is given as answer may occur in the schedule much earlier than (Vj) does, the simulator may perform rewinds that do not reach (Pj) (and so do not change its value), but repeatedly obtain different (Vj)'s as answer. In such cases, the SOLVE procedure will always store the "recently obtained" (Vj) message instead of previous ones (that were given as answer to the same (Pj)).

Note that since the schedule may vary "dynamically" as a function of the history of the interaction, it may be the case that not all messages in a specific session $s \in \{1, \ldots, m\}$ are "visited" the same number of times by the SOLVE procedure. In particular, the number of verifier messages that appear in $\mathcal{T}$ may differ from session to session and from iteration to iteration (within a specific session). A detailed analysis of the contents of the $\mathcal{T}$ table whenever the simulator reaches the second stage in session s appears in Section 4.7.

[10] We stress that corresponding messages in the $(\tilde{\mathtt{p}}_1, \tilde{\mathtt{v}}_1, \ldots, \tilde{\mathtt{p}}_{\ell/2}, \tilde{\mathtt{v}}_{\ell/2})$ and the $(\mathtt{p}_1, \mathtt{v}_1, \ldots, \mathtt{p}_{\ell/2}, \mathtt{v}_{\ell/2})$ sequences do not necessarily belong to the same sessions $s \in \{1, \ldots, m\}$. This is because the concurrent schedule may be "dynamically" determined by V^* as a function of the history of the interaction (in particular, different values of $\tilde{\mathtt{p}}_1, \ldots, \tilde{\mathtt{p}}_{\ell/2}$ and $\mathtt{p}_1, \ldots, \mathtt{p}_{\ell/2}$ may cause the corresponding answers of V^* to belong to different sessions).

[11] Again, notice that corresponding messages in the $(\tilde{\mathtt{p}}_{\ell/2+1}, \tilde{\mathtt{v}}_{\ell/2+1}, \ldots, \tilde{\mathtt{p}}_{\ell}, \tilde{\mathtt{v}}_{\ell})$ and the $(\mathtt{p}_{\ell/2+1}, \mathtt{v}_{\ell/2+1}, \ldots, \mathtt{p}_{\ell}, \mathtt{v}_{\ell})$ sequences do not necessarily belong to the same session.

Handling second stage messages

Second stage messages are handled by the PROVE procedure. The goal of this procedure is to produce a second stage transcript that is indistinguishable from actual second stage transcripts (that is, between P and V^*). This should be done while avoiding a situation in which the basic Hamiltonicity proof system that is conducted in the second stage of the protocol is rejected by the verifier V^* (since in such cases the simulator may get "stuck"). The key for the success of the PROVE procedure lies in the success of the SOLVE procedure to discover the "challenge" sent by V^* already during the first stage of the protocol. Given that the SOLVE procedure has indeed succeeded in discovering the "challenge", the task of the PROVE procedure is trivial (whereas if the SOLVE procedure did not succeed to discover the "challenge" then the PROVE procedure is bound to fail). One other case in which the task of the PROVE procedure is trivial is when the "current history" of the interaction contains an `ABORT` message on behalf of the verifier V^* (that is, in the relevant session). In such cases the interaction in the relevant session stops and the PROVE procedure does not need to do anything in order to produce a "legal" second stage transcript.

The PROVE procedure: The PROVE procedure is invoked either when the concurrent schedule reaches the first prover message in the second stage of session $s \in \{1, \ldots, m\}$ (that is, a (`p1`) message) or when it reaches the second prover message in the second stage (that is, a (`p2`) message). Note that this may happen many times during the simulation process (as induced by the adversary verifier's scheduling strategy and the "rewinds" of the SOLVE procedure). On input $s \in \{1, \ldots, m\}$ and a partial execution transcript (denoted hist), the PROVE procedure acts as follows:

1. Start by checking whether the hist argument contains an `ABORT` message on behalf of the verifier (in session s). Specifically, for every $j \in \{1, \ldots, k\}$, check whether the (Vj) message of session s (as it appears in hist) consists of an `ABORT` message. If it does (for some j), abort session s (just as an honest prover P would have done in such a case).

2. Otherwise (i.e., the hist argument does not contain an `ABORT` message in session s), search the $\mathcal{T}$ table for a pair $\sigma^0_{i,j}, \sigma^1_{i,j}$ belonging to session s:

    (a) If the $\mathcal{T}$ table does not contain such a pair (that is, if for every i, j the $\mathcal{T}$ table contains only $\sigma^b_{i,j}$ for some fixed $b \in \{0, 1\}$, and possibly some additional `ABORT` messages), output $\perp$ (indicating failure of the simulation).

    (b) If the $\mathcal{T}$ table indeed contains a pair $\sigma^0_{i,j}$ and $\sigma^1_{i,j}$ belonging to session s, compute the value of V^*'s "challenge", $\sigma = \sigma^0_{i,j} \oplus \sigma^1_{i,j}$ and invoke the CONVINCE subroutine with input (σ, hist). The CONVINCE subroutine handles the execution of second stage messages in the protocol (and is described below).

    (c) Let `p` denote the output of the CONVINCE subroutine (where `p` is either of the form (`p1`) or (`p2`), depending on our location in the schedule). Output `p`.

The CONVINCE subroutine: Given the value of $\sigma = \sigma_1 \sigma_2 \ldots \sigma_n$ and hist, the CONVINCE subroutine handles the ℓ^{th} (parallel) execution in the second stage of session s in the following way:

- Prover's first step (p1): *If $\sigma_\ell = 0$, act according to Step $(\widehat{\mathtt{p1}})$ in Construction 4.3.1. Specifically, select a random permutation, π, of the vertices V, and commit (using a perfectly-binding commitment scheme) to the entries of the adjacency matrix of the resulting permuted graph. That is, output an n-by-n matrix of commitments so that the $(\pi(i), \pi(j))^{\text{th}}$ entry is a commitment to 1 if $(i,j) \in E$, and is a commitment to 0 otherwise.*

  Otherwise (i.e., if $\sigma_\ell = 1$), commit to the entries of the adjacency matrix of the full graph K_n. That is, output an n-by-n matrix of commitments so that for every $(i,j) \in \{1, \ldots, n\}$, the $(i,j)^{\text{th}}$ entry is a commitment to 1.

- Prover's second step (p2): *Check (in* hist*) that V^* has properly decommited to all relevant values (in particular, check that the ℓ^{th} bit of $\sigma_j^0 \oplus \sigma_j^1$ indeed equals σ_ℓ for all j) and abort otherwise.*

  If $\sigma_\ell = 0$, output π along with the revealing (i.e., preimages) of all commitments.

  Otherwise (i.e., if $\sigma_\ell = 1$), output only the openings of commitments to entries $(\pi(i), \pi(j))$ with $(i,j) \in C$ where C is an arbitrary Hamiltonian cycle in K_n. In both cases also supply the corresponding decommitments.

Some comments: Note that the CONVINCE subroutine never causes the verifier V^* to reject in the second stage (that is, unless V^* sends an ABORT message instead of the corresponding (v1) message). The reason for this is that it is always invoked with the correct value of σ (which was previously revealed by the SOLVE procedure). In particular, once the PROVE procedure has "safely" reached Step 2b the success of the PROVE procedure is guaranteed.

The actions taken by the CONVINCE subroutine are identical to the actions taken by the simulator of Blum's basic Hamiltonicity protocol. As a consequence, the distribution of the simulated second stages in our protocol are identical to the distribution produced by Blum's simulator. This fact will be used later in order to reduce the indistiguishability property of our simulator's output to the indistiguishability property of Blum's simulator's output.

4.4.2 "Gluing" it all together

The SIMULATE procedure which merges the SOLVE and PROVE procedures together handles all messages sent by V^* during the simulation process (that is, both first stage and second stage messages). In general, the SIMULATE procedure is obtained by incorporating the PROVE procedure into the SOLVE procedure in a way that enables the SOLVE procedure to handle also second stage messages (see Figure 4.4 for a "pseudocode" description of the SIMULATE procedure).

The two main modifications applied to the SOLVE procedure (in order to obtain the SIMULATE procedure) are the following: (1) If $\ell = 1$ (that is, at the bottom level of the recursion), the SIMULATE procedure will keep exchanging messages until it reaches a first stage verifier message. This is done while augmenting the hist argument with the corresponding outcomes of the PROVE procedure (according to the schedule that is being revealed by V^*). Once a first stage message is reached, the SIMULATE procedure acts exactly as the SOLVE procedure. (2) Similarly to the SOLVE procedure, the output of the SIMULATE procedure is a partial execution transcript. However, unlike the SOLVE procedure, the ouptut length of the SIMULATE procedure is greater than 2ℓ (since, besides 2ℓ first stage messages, it will also contain second stage messages).

```
Procedure SIMULATE($\ell$, hist, $\mathcal{T}$):
  1. If $\ell = 1$,
     (a) If session $s$ does not appear in hist, delete all session $s$ messages in $\mathcal{T}$.
     (b) As long as no first stage verifier message has been reached, do:
          i. If next scheduled message, $\mathtt{p}_u$, is a first stage prover message:
             Uniformly choose $\mathtt{p}_u$, and set $\mathtt{v}_u \leftarrow V^*(\mathsf{hist}, \mathtt{p}_1, \mathtt{v}_1, \ldots, \mathtt{p}_{u-1}, \mathtt{v}_{u-1}, \mathtt{p}_u)$.
         ii. If next scheduled message, $\mathtt{p}_u$, is a second stage prover message:
             Set $\mathtt{p}_u \leftarrow$ PROVE$(i, (\mathsf{hist}, \mathtt{p}_1, \mathtt{v}_1, \ldots, \mathtt{p}_{u-1}, \mathtt{v}_{u-1}))$, and $\mathtt{v}_u \leftarrow V^*(\mathsf{hist}, \mathtt{p}_1, \mathtt{v}_1, \ldots, \mathtt{v}_{u-1}, \mathtt{p}_u)$.
     (c) As soon as a first stage message $\mathtt{v}_\alpha$ has been reached store $\mathtt{v}_\alpha$ in $\mathcal{T}$.
     (d) Output $\mathcal{T}$ and $(\mathtt{p}_1, \mathtt{v}_1, \ldots, \mathtt{p}_\alpha, \mathtt{v}_\alpha)$.
     Remark: Note that $\mathtt{v}_\alpha$ is the only first stage verifier message in $(\mathtt{p}_1, \mathtt{v}_1, \ldots, \mathtt{p}_\alpha, \mathtt{v}_\alpha)$.
  2. Otherwise (i.e., if $\ell > 1$),
     (a) Set $\mathcal{T}_1$, $(\tilde{\mathtt{p}}_1, \tilde{\mathtt{v}}_1, \ldots, \tilde{\mathtt{p}}_{\tilde{\alpha}}, \tilde{\mathtt{v}}_{\tilde{\alpha}}) \leftarrow$ SIMULATE$(\ell/2, \mathsf{hist}, \mathcal{T})$.
     (b) Set $\mathcal{T}_2$, $(\mathtt{p}_1, \mathtt{v}_1, \ldots, \mathtt{p}_\alpha, \mathtt{v}_\alpha) \leftarrow$ SIMULATE$(\ell/2, \mathsf{hist}, \mathcal{T}_1)$.
     (c) Set $\mathcal{T}_3$, $(\tilde{\mathtt{p}}_{\alpha+1}, \tilde{\mathtt{v}}_{\alpha+1}, \ldots, \tilde{\mathtt{p}}_{\alpha+\tilde{\beta}}, \tilde{\mathtt{v}}_{\alpha+\tilde{\beta}}) \leftarrow$ SIMULATE$(\ell/2, (\mathsf{hist}, \mathtt{p}_1, \mathtt{v}_1, \ldots, \mathtt{p}_\alpha, \mathtt{v}_\alpha), \mathcal{T}_2)$.
     (d) Set $\mathcal{T}_4$, $(\mathtt{p}_{\alpha+1}, \mathtt{v}_{\alpha+1}, \ldots, \mathtt{p}_{\alpha+\beta}, \mathtt{v}_{\alpha+\beta}) \leftarrow$ SIMULATE$(\ell/2, (\mathsf{hist}, \mathtt{p}_1, \mathtt{v}_1, \ldots, \mathtt{p}_\alpha, \mathtt{v}_\alpha), \mathcal{T}_3)$.
     (e) Output $\mathcal{T}_4$ and $(\mathtt{p}_1, \mathtt{v}_1, \ldots, \mathtt{p}_{\alpha+\beta}, \mathtt{v}_{\alpha+\beta})$.
     Remark: (1) The value of $\tilde{\alpha}$ (resp. $\tilde{\beta}$) is not necessarily equal to the value of $\alpha$ (resp. $\beta$). (2)
     The sequence $(\mathtt{p}_1, \mathtt{v}_1, \ldots, \mathtt{p}_{\alpha+\beta}, \mathtt{v}_{\alpha+\beta})$ contains exactly $2\ell$ first stage prover and verifier messages
     (as well as arbitrarily many second stage messages). In particular, $\alpha + \beta \geq \ell$.
```

Figure 4.4: The SIMULATE procedure - Handles both first and second stage messages. It is obtained by merging the SOLVE and PROVE procedures (with the help of the table $\mathcal{T}$).

4.5 The Simulator's Running Time

We start by showing that the simulator's running time is polynomial both in m and in $n = |V|$. Since $m = \text{poly}(n)$ it will follow that the simulator runs in polynomial-time in n.

Using the fact that the total number of sessions run by the adversary verifier V^* is at most m, we infer that the number of invocations of the PROVE procedure at the bottom level of the recursion (i.e., when $\ell = 1$) is upper bounded by m. In particular, the work invested by the SIMULATE procedure at the bottom level of the recursion is upper bounded by $\text{poly}(n) \cdot m = \text{poly}(n)$ (where the $\text{poly}(n)$ factor in the $\text{poly}(n) \cdot m$ term results from the polynomial amount of work invested in each invocation of the PROVE procedure). Since each invocation of the SIMULATE procedure with parameter $\ell > 1$ involves four recursive invocations of the SIMULATE procedure with parameter $\ell/2$, we have that the work $W(\ell)$, that is invested by the SIMULATE procedure in order to handle ℓ (first stage) verifier messages satisfies:

$$W(\ell) \leq \begin{cases} \text{poly}(n) & \text{If } \ell = 1 \\ 4 \cdot W(\ell/2) & \text{If } \ell > 1 \end{cases} \tag{4.1}$$

Since the total number of first stage verifier messages in the m sessions of the concurrent schedule equals $m \cdot (k+1)$, the total running time of the simulation process (which consists of a single invocation of the SIMULATE procedure with parameter $m \cdot (k+1)$) equals $W(m \cdot (k+1))$. A straightforward solution of the recursive formula in Eq. (4.1) establishes that $W(m \cdot (k+1))$ is upper bounded by:

$$4^{\log_2(m\cdot(k+1))} \cdot \mathrm{poly}(n) = (m \cdot (k+1))^2 \cdot \mathrm{poly}(n) = \mathrm{poly}(n)$$

Hence, we have:

Proposition 4.5.1 *For every $m = \mathrm{poly}(n)$, the simulator S_m runs in (strict) polynomial-time in n.*

4.6 The Simulator's Output Distribution

We now turn to show that for every $G \in HC$, the simulator's output distribution is computationally indistinguishable from V^*'s view of interactions with the honest prover P. Specifically,

Proposition 4.6.1 *The ensemble $\{S_m^{V^*}(G)\}_{G\in HC}$ is computationally indistinguishable from the ensemble $\{\mathrm{view}_{V^*}^{P}(G)\}_{G\in HC}$.*

Proof: As a hybrid experiment, consider what happens to the output distribution of the simulator S_m if we (slightly) modify its simulation strategy in the following way: Suppose that on input $G = (V, E) \in HC$, the simulator S_m obtains a directed Hamiltonian Cycle $C \subset E$ in G (as auxiliary input) and uses it in order to produce real prover messages whenever it reaches the second stage of the protocol. Specifically, whenever it reaches the second stage of session $s \in \{1, \ldots, m\}$, the hybrid simulator inspects the $\mathcal{T}$ table and checks whether the PROVE procedure should output $\perp$ (in which case it also does). If the PROVE procedure does not have to output $\perp$, the hybrid simulator follows the prescribed prover strategy and generates prover messages for the corresponding second stage (by using the cycle it possesses rather than invoking the PROVE procedure). We claim that the ensemble consisting of the resulting output (which we denote by $\widehat{S}_m^{V^*}(G, C)$) is computationally indistinguishable from $\{S_m^{V^*}(G)\}_{G\in HC}$. Namely,

Claim 4.6.2 *The ensemble $\{S_m^{V^*}(G)\}_{G\in HC}$ is computationally indistinguishable from the ensemble $\{\widehat{S}_m^{V^*}(G, C)\}_{G\in HC}$.*

Proof Sketch: The claim is proved using a standard hybrid argument. It reduces the indistinguishability of two neighbouring hybrids to the indistinguishability of Blum's simulator's output (that is, if the output of Blum's simulator [6] is computationally indistinguishable from the view of real executions of the basic Hamiltonicity proof system, then so are neighbouring hybrids). The latter is proved to hold based on the computational-secrecy property of the commitment scheme that is used by the prover in Step ($\widehat{\mathrm{p1}}$) of Construction 4.3.1 (see [6, 22] for further details).

We consider $m + 1$ hybrid distributions that are induced by the output of the following hybrid simulation procedure. For $s \in \{0, \ldots, m\}$, given $G \in HC$, a Hamiltonian cycle C in G and black-box access to V^*, the s^{th} hybrid simulation procedure (which we denote by H_s), handles first stage messages exactly as the "original" simulator S_m would have handled. For every session index $s' \leq s$, the hybrid simulator H_s handles also second stage messages exactly as S_m does (that is, by invoking the PROVE procedure), whereas for every session index $s' > s$, the hybrid simulator H_s handles the relevant second stage messages exactly as the "modified" simulator $\widehat{S}_m$ does (that is, by using the cycle it possesses in order to produce real prover messages). Note that the output of $H_m^{V^*}$ is identically distributed to the output of $S_m^{V^*}$, whereas the output of $H_0^{V^*}$ is identically distributed to the output of $\widehat{S}_m^{V^*}$. Also note that for every $s \in \{0, \ldots, m\}$, the distribution $H_s^{V^*}(G, C)$ is efficiently constructible (specifically, given a Hamiltonian cycle C in G, it is easy to follow $S_m^{V^*}(G)$'s strategy, while producing real prover messages whenever necessary). Thus, indistinguishability of the ensemble $\{S_m^{V^*}(G)\}_{G\in HC}$ from the ensemble $\{\widehat{S}_m^{V^*}(G)\}_{G\in HC}$ follows from indistinguishability of $\{H_{s-1}(G,C)\}_{G\in HC}$ and $\{H_s(G,C)\}_{G\in HC}$.

Claim 4.6.3 *For all $s \in \{1,\ldots,m\}$, the ensembles $\{H_{s-1}(G,C)\}_{G\in HC}$ and $\{H_s(G,C)\}_{G\in HC}$ are computationally indistinguishable.*

Proof Sketch: Follows from indistinguishability of Blum's simulator's output (by applying an additional hybrid argument). Uses the extra property that the output is indistinguishable even if the distinguisher has a-priori knowledge of a Hamiltonian Cycle C in G. ■

This completes the proof of Claim 4.6.2. ■

We next consider what happens to the output distribution of the hybrid simulator $\widehat{S}_m$ if we assume that it does not output $\perp$ (i.e., does not get "stuck"). It turns out that in such a case, the resulting output distribution is *identical* to the distribution of $\{\text{view}_{V^*}^P(G)\}_{G\in HC}$. Namely,

Claim 4.6.4 *The ensemble $\{\widehat{S}_m^{V^*}(G,C)\}_{G\in HC}$ conditioned on it not being $\perp$, is identically distributed to the ensemble $\{\text{view}_{V^*}^P(G)\}_{G\in HC}$.*

Proof: Notice that the first stage messages that appear in the output of the "original" simulator (that is, S_m) are identically distributed to the first stage messages that are produced by an honest prover P (since they are uniformly and independently chosen). Since the first stage messages that appear in the output of the "modified" simulator (that is, $\widehat{S}_m$) are identical to the ones appearing in the output of S_m, we infer that they are identically distributed to the first stage messages that are produced by an honest prover P. Using the fact that the second stage messages that appear in the output of the "modified" simulator are (by definition) identically distributed to the second stage messages that are produced by an honest prover P, we infer that the ensemble $\{\widehat{S}_m^{V^*}(G,C)\}_{G\in HC}$ is identically distributed to $\{\text{view}_{V^*}^P(G)\}_{G\in HC}$. ■

As we show in Proposition 4.7.1 (see next section), $\widehat{S}_m$ outputs $\perp$ only with negligible probability. In particular, the ensemble $\{\widehat{S}_m^{V^*}(G,C)\}_{G\in HC}$ is computationally indistinguishable from (and in fact statistically close to) the ensemble $\{\widehat{S}_m^{V^*}(G,C)\}_{G\in HC}$, conditioned on it not being $\perp$. Namely,

Claim 4.6.5 *The ensemble $\{\widehat{S}_m^{V^*}(G,C)\}_{G\in HC}$ is computationally indistinguishable from the ensemble $\{\widehat{S}_m^{V^*}(G,C)\}_{G\in HC}$ conditioned on it not being $\perp$.*

It can be seen that Claims 4.6.2, 4.6.4 and 4.6.5 imply the correctness of Proposition 4.6.1. ■

4.7 The Probability of Getting "stuck"

We next analyze the probability that the SIMULATE procedure gets "stuck" during its execution. We are particularly interested in the probability that any specific invocation of the PROVE procedure returns $\perp$ during the simulation process (note that this is the only reason for which the simulator may get "stuck"). As will turn out from our analysis, any specific invocation of the PROVE procedure will return $\perp$ with probability at most $1/2^{\Omega(k)}$. Since the number of invocations of the PROVE procedure is polynomial in n, it follows that the SIMULATE procedure outputs $\perp$ with probability

$\text{poly}(n) \cdot 1/2^{\Omega(k)}$. By setting the number of rounds in the protocol to be $k(n) = \alpha(n) \cdot \log n$, where $\alpha(\cdot)$ is any super-constant function (e.g., $\alpha(n) = \log\log n$), we are guaranteed that the SIMULATE procedure outputs $\perp$ with negligible probability. Specifically:

Proposition 4.7.1 *Let $\alpha : N \to N$ be any super-constant function, let $k(n) = \alpha(n) \cdot \log n$, and consider any instantiation of Construction 4.3.2 with parameter $k = k(n)$. Then the probability of getting "stuck" during the simulation is negligible. Specifically, for every sufficiently large $G = (V, E) \in HC$:*

$$\Pr\left[\widehat{S}_m^{V^*}(G) = \perp\right] < \frac{1}{n^{\alpha(n)/8}}$$

where $n = |V|$ and the probability above is taken over the simulator's coin tosses.

Proof: We consider executions of the hybrid simulator $\widehat{S}_m$, given input $G = (V, E)$, random coins ρ, and black-box access to V^* (we let $\widehat{S}_{m,\rho}^{V^*}(G)$ denote the resulting output).

Let $q_S(n)$ be a (polynomial) bound on the total number of invocations of the PROVE procedure during an execution of the simulator (note that $q_S(n)$ is upper bounded by the simulator's running time). As we have mentioned before, the (hybrid) simulator $\widehat{S}_m$ gets "stuck" (i.e., outputs $\perp$) if and only if there exists a session $s \in \{1, \ldots, m\}$ and an index $\ell \in \{1, \ldots, q_S(n)\}$ so that the ℓ^{th} invocation of the PROVE procedure (for session s) outputs $\perp$. Let $\mathsf{hist}_{s,\ell} = \mathsf{hist}_{s,\ell}(\rho)$ be a random variable describing the contents of the hist argument at the moment that the PROVE procedure is invoked for the ℓ^{th} time (with s as its first argument). Using the union-bound we have:

$$\Pr_\rho\left[\widehat{S}_{m,\rho}^{V^*}(G) = \perp\right] \leq \sum_{i=1}^{m} \sum_{\ell=1}^{q_S(n)} \Pr_\rho\left[\text{PROVE}(s, \mathsf{hist}_{s,\ell}(\rho)) = \perp\right] \tag{4.2}$$

Eq. (4.2) will be bounded using the following lemma. This lemma, which in some sense is the crux of the proof (of the zero-knowledge property), establishes an upper bound on the probability that a specific invocation of the PROVE procedure outputs $\perp$.

Lemma 4.7.2 *For every $(s, \ell) \in \{1, \ldots, m\} \times \{1, \ldots, q_S(n)\}$ and all sufficiently large $G \in HC$:*

$$\Pr_\rho\left[\text{PROVE}(s, \mathsf{hist}_{s,\ell}(\rho)) = \perp\right] < \frac{1}{2^{k/4}}$$

Combining Lemma 4.7.2, Eq. (4.2) and the hypothesis of Proposition 4.7.1 we infer that for all sufficiently large $n = |V|$:

$$\begin{aligned} \Pr_\rho\left[\widehat{S}_{m,\rho}^{V^*}(G) = \perp\right] &\leq m \cdot q_S(n) \cdot \frac{1}{2^{k/4}} \\ &= \frac{m \cdot q_S(n)}{2^{k/8}} \cdot \frac{1}{2^{k/8}} \\ &< \frac{1}{n^{\alpha(n)/8}} \end{aligned} \tag{4.3}$$

Where Eq. (4.3) holds whenever $m \cdot q_S(n) < 2^{k/8}$ (which is satisfied whenever $k(n)$ is equal to $\alpha(n) \cdot \log n$, and n is sufficiently large). This completes the proof of Proposition 4.7.1. ■

Proof of Lemma 4.7.2: Let $s \in \{1,\dots,m\}$ and $\ell \in \{1,\dots,q_S(n)\}$. We next show that the probability that the ℓ^{th} invocation of the PROVE procedure outputs $\perp$ is upper bounded by $1/2^{k/4}$.

Throughout the analysis we will assume that the simulator never sends the same prover message twice during its execution. Such an assumption is justified by the fact that prover messages in the protocol are k-bits long. In particular, the probability that any two uniformly chosen $(\mathrm{P}j)$ messages are equal is at most $1/2^k$. Since for every session, the number of prover messages sent is at most $\mathrm{poly}(n)$, then the overall probability of sending the same prover message twice is at most $\mathrm{poly}(n)/2^k$. By taking n to be sufficiently large this probability is upper bounded $1/2^{k/2}$.

We thus set our goal to bound the probability that the PROVE procedure outputs $\perp$ assuming the simulator never sends the same prover message twice. By the above discussion, it will be sufficient to bound this probability by $1/2^{k/2}$. The probability that the PROVE procedure outputs $\perp$ would be then upper bounded by $1/2^{k/2} + 1/2^{k/2} = 2/2^{k/2} < 1/2^{k/4}$.

From now on, we focus on messages that belong to session s and ignore messages from other sessions (unless otherwise specified). For every choice ρ of the simulator's randomness we focus on an invocation of the PROVE procedure with input $(s, \mathsf{hist}_{s,\ell}) = (s, \mathsf{hist}_{s,\ell}(\rho))$. We associate the invocation of the PROVE procedure with the value of the (V0) message that appears in the $\mathsf{hist}_{s,\ell}(\rho)$ argument. We will analyze the execution of the simulator from the time (V0) has been last visited until the time that $\text{PROVE}(s, \mathsf{hist}_{s,\ell})$ is invoked.

The contents of the $\mathcal{T}$ table: For every $j \in \{1,\dots,k\}$, we consider the sequence of (first stage) verifier messages, $(\mathrm{V}j)$, that appear in the $\mathcal{T}$ table at the moment that $\text{PROVE}(s, \mathsf{hist}_{s,\ell})$ is invoked. Let α_j denote the length of this sequence. The value α_j actually corresponds to the number of times that the $(\mathrm{V}j)$ message has been visited since all session s messages have been last deleted from $\mathcal{T}$. (Recall that this happens whenever a (V0) message is visited by the SIMULATE procedure.) Note that α_j is not necessarily equal for all $j \in \{1,\dots,k\}$.

For $u \in \{1,\dots,\alpha_j\}$, let $(\mathrm{P}j)_u, (\mathrm{V}j)_u$ denote the u^{th} pair of $(\mathrm{P}j), (\mathrm{V}j)$ messages that was visited by the SIMULATE procedure since (V0) has been last visited.[12] (Using this notation, the j^{th} above sequence can be written as $(\mathrm{V}j)_1, (\mathrm{V}j)_2, \dots, (\mathrm{V}j)_{\alpha_j}$.) We now have the following claim.

Claim 4.7.3 *Suppose that* $\text{PROVE}(s, \mathsf{hist}_{s,\ell}) = \perp$. *Then for all* $j \in \{1,\dots,k\}$*:*

1. $(\mathrm{V}j)_u = \mathtt{ABORT}$ *for all* $u < \alpha_j$.
2. $(\mathrm{V}j)_{\alpha_j} \neq \mathtt{ABORT}$.

Proof: Going back to the description of the PROVE procedure we observe that the only reason for which it outputs $\perp$ is that it has reached Step (2a). Put in other words, the PROVE procedure will output $\perp$ if and only if:

1. The $\mathsf{hist}_{s,\ell}$ argument does not contain an ABORT message in session s.
2. The $\mathcal{T}$ table does not contain a pair $\sigma^0_{i,j}, \sigma^1_{i,j}$ belonging to session s.

We start by showing that for all $j \in \{1,\dots,k\}$, it holds that $(\mathrm{V}j)_{\alpha_j} \neq \mathtt{ABORT}$. Consider the sequence of first stage verifier messages that appear in $\mathsf{hist}_{s,\ell}$ and belong to session s. Notice that

[12]Note that the $(\mathrm{P}j)$ message may occur in the schedule much earlier than $(\mathrm{V}j)$ does. In particular, the simulator may perform rewinds that do not reach $(\mathrm{P}j)$ (and so do not change its value), but repeatedly obtain different $(\mathrm{V}j)$'s as answer. In such cases, the $(\mathrm{V}j)_u$ message stored in $\mathcal{T}$ as answer to $(\mathrm{P}j)_u$ will always correspond to the "most recently obtained" $(\mathrm{V}j)$ message that was given as answer to $(\mathrm{P}j)_u$ (see discussion in Page 73).

this sequence contains all $k+1$ first stage messages in session s (since PROVE$(s, \mathsf{hist}_{s,\ell})$ is always invoked only after the first stage of session s has been completed). Using the fact that the $\mathsf{hist}_{s,\ell}$ argument consists of the "most recently visited" execution transcript in the simulation, we have that the sequence of first stage verifier messages that appear in $\mathsf{hist}_{s,\ell}$ and belong to session s can be written as $(\mathrm{V}0), (\mathrm{V}1)_{\alpha_1}, (\mathrm{V}2)_{\alpha_2}, \ldots, (\mathrm{V}k)_{\alpha_k}$. Since, by Condition (1) above, the $\mathsf{hist}_{s,\ell}$ argument does not contain an ABORT message in session s it immediately follows that for all $j \in \{1, \ldots, k\}$, it holds that $(\mathrm{V}j)_{\alpha_j} \neq \texttt{ABORT}$.

Suppose now for contradiction that there exists a $j \in \{1, \ldots, k\}$ and a $u \in \{1, \ldots, \alpha_j - 1\}$ so that $(\mathrm{V}j)_u \neq \texttt{ABORT}$. Since we are assuming that all $(\mathrm{P}j)$'s in the simulation are different, then so are $(\mathrm{P}j)_u$ and $(\mathrm{P}j)_{\alpha_j}$. Since both $(\mathrm{V}j)_u$ and $(\mathrm{V}j)_{\alpha_j}$ are not equal to ABORT it immediately follows that the table contains a pair $\sigma^0_{i,j}, \sigma^1_{i,j}$ belonging to session s.[13] This is in contradiction to Condition (2) above and thus to our hypothesis that the PROVE procedure outputs $\perp$. ■

Definition 4.7.4 (Bad random tapes) *Let $\mathcal{R}$ be the set of all random tapes used by the simulator and let $\rho \in \mathcal{R}$. For any $j \in \{1, \ldots, k\}$ define a Boolean indicator $\mathsf{bad}_j(\rho) = \mathsf{bad}_{s,\ell,j}(\rho)$ to be true if and only if when S uses ρ as random tape it holds that:*

1. $(\mathrm{V}j)_u = \texttt{ABORT}$ *for all $u < \alpha_j$.*
2. $(\mathrm{V}j)_{\alpha_j} \neq \texttt{ABORT}$.

By Claim 4.7.3, we have:

$$\Pr_\rho\left[\text{PROVE}(s, \mathsf{hist}_{s,\ell}(\rho)) = \perp\right] \quad \leq \quad \Pr_\rho\left[\bigwedge_{j=1}^{k} \mathsf{bad}_j(\rho)\right] \tag{4.4}$$

We shall show that for all sufficiently large n, the fraction of "bad" tapes $\rho \in \mathcal{R}$ for which $\bigwedge_{j=1}^{k} \mathsf{bad}_j(\rho)$ holds is at most $1/2^{k-3d}$ where d $(= \log_2(m \cdot (k+1)))$ is the depth of the simulator's recursion. To do this we will show that every "bad" random tape can be mapped into a set of $2^{k-3d} - 1$ random tapes for which $\mathsf{bad}_j(\rho)$ does not hold for some j. Moreover, this will be done so that every two "bad" random tapes are mapped to two *disjoint* sets of "good" random tapes. Put in other words, for every random tape that causes $\bigwedge_{j=1}^{k} \mathsf{bad}_j(\rho)$ to hold, there exist $2^{k-3d} - 1$ other tapes that do not. Since the simulator picks a random tape uniformly amongst all possible random tapes, it will then follow that the probability that $\bigwedge_{j=1}^{k} \mathsf{bad}_j(\rho)$ holds is at most $1/2^{k-3d}$.

Lemma 4.7.5 (Counting bad random tapes) *Let $\mathcal{B} \subseteq \mathcal{R}$ be the set of all $\rho \in R$ for which $\bigwedge_{j=1}^{k} \mathsf{bad}_j(\rho)$ holds. Then, there exists a mapping $f : \mathcal{R} \longrightarrow 2^{\mathcal{R}}$ such that for every $\rho \in \mathcal{B}$:*

1. $|f(\rho)| \geq 2^{k-3d}$.
2. *For all $\rho' \in \mathcal{B} \setminus \{\rho\}$, the sets $f(\rho)$ and $f(\rho')$ are disjoint.*
3. *The sets $f(\rho) \setminus \{\rho\}$ and $\mathcal{B}$ are disjoint.*

The proof of Lemma 4.7.5 is the most involved part in the simulator's analysis. Before we prove it (in Section 4.7.1), we show how it can be used in order to complete the proof of Lemma 4.7.2. We start with the following corollary of Lemma 4.7.5.

[13]To see this notice that, if $(\mathrm{P}j)_u = r_{1,j}, \ldots, r_{k,j}$ and $(\mathrm{P}j)_{\alpha_j} = s_{1,j}, \ldots, s_{k,j}$ are different, then there must exist $i \in \{1, \ldots, k\}$ so that $r_{i,j} \neq s_{i,j}$. Since both $(\mathrm{V}j)_u$ and $(\mathrm{V}j)_{\alpha_j}$ are not equal to ABORT, then the values of both $\sigma^{r_{i,j}}_{i,j}$ and $\sigma^{s_{i,j}}_{i,j}$ must have been revealed by V^*.

Corollary 4.7.6 *Let $\mathcal{B} \subseteq \mathcal{R}$ be as above. Then* $|\mathcal{B}|/|\mathcal{R}| \leq \frac{1}{2^{k-3d}}$.

Proof: Consider the set:

$$\mathcal{G} \stackrel{\text{def}}{=} \bigcup_{\rho \in \mathcal{B}} \Big(f(\rho) \setminus \{\rho\} \Big)$$

By Condition (3) in Lemma 4.7.5 it holds that $\mathcal{G} \subseteq \mathcal{R} \setminus \mathcal{B}$. We thus have:

$$\begin{aligned} |\mathcal{R}| - |\mathcal{B}| &= |\mathcal{R} \setminus \mathcal{B}| \\ &\geq |\mathcal{G}| \\ &= \sum_{\rho \in \mathcal{B}} |f(\rho) \setminus \{\rho\}| \qquad (4.5) \\ &\geq |\mathcal{B}| \cdot \Big(2^{k-3d} - 1\Big) \qquad (4.6) \end{aligned}$$

Where Eq. (4.5) follows from Condition (2) in Lemma 4.7.5 and Eq. (4.6) follows from Condition (1) in Lemma 4.7.5. ■

Using Corollary 4.7.6 we are now able to complete the proof of Lemma 4.7.2:

$$\begin{aligned} \Pr_\rho \left[\bigwedge_{j=1}^{k} \mathsf{bad}_j(\rho) \right] &= \Pr_\rho \left[\rho \in \mathcal{B} \right] \\ &= \frac{|\mathcal{B}|}{|\mathcal{R}|} \\ &\leq \frac{1}{2^{k-3d}} \qquad (4.7) \end{aligned}$$

Since $k = \omega(\log n)$ and $d = \log m \cdot (k+1) = O(\log n)$ then for all sufficiently large n's it holds that $1/2^{k-3d} < 1/2^{k/2}$. By combining Eq. (4.7) with Eq. (4.4) we infer that for all sufficiently large n's:

$$\Pr_\rho \left[\text{PROVE}(s, \mathsf{hist}_{s,\ell}(\rho)) = \perp \right] < \frac{1}{2^{k/2}}$$

This completes the proof of Lemma 4.7.2. ■

4.7.1 Proof of Lemma 4.7.5 (counting bad random tapes)

The proof of Lemma 4.7.5 will proceed as follows. We first define the notion of rewind intervals. Loosely speaking, these are segments of the concurrent schedule that are induced by the various rewindings of the simulator and are executied multiple times during the simulation. We then focus on a subset of "special" intervals. These intervals satify some useful properties that enable us to use them in order to define the desired mapping $f : \mathcal{R} \rightarrow 2^{\mathcal{R}}$. Using the properties of the "special" intervals we can then argue that the mapping f indeed satisfies the required properties.

Throughout the proof of Lemma 4.7.5, we consider the actions taken during the execution of the SOLVE procedure (rather than considering the full execution of the SIMULATE procedure). This renders our analysis much "cleaner" since we only have to refer only to first stage messages (namely, $(\text{P}0), (\text{V}0), (\text{P}1), (\text{V}1), \ldots, (\text{P}k), (\text{V}k)$), and can ignore second stage messages (namely $(\texttt{p1}), (\texttt{v1}), (\texttt{p2})$). Extension of the analysis to the SIMULATE procedure case can be then achieved in a straightforward way (the reason this is possible is that the timing of the simulator's "rewinds" depends only on the number of first stage messages exchanged so far).

Partitioning the schedule into rewind intervals: The execution of the SOLVE procedure induces a partitioning of the $2 \cdot m \cdot (k+1)$ (prover and verifier) messages in the schedule into *disjoint* rewind intervals. At the top level of the recursion there are two disjoint intervals of length $m \cdot (k+1)$ and at the bottom of the recursion there are $m \cdot (k+1)$ disjoint intervals of length 2. In general, at the w^{th} level of the recursion (out of $d = \log_2(m \cdot (k+1))$ possible levels) there are 2^w disjoint intervals of $m(k+1)/2^{w+1}$ messages each.

Notice that rewind intervals may contain messages from all sessions. Also notice, that a rewind interval may be “visited” multiple times during the execution of the SOLVE procedure (in particular, a level-w interval is visited exactly 2^w times during the simulation). Since the scheduling of messages may vary ”dynamically” with the history of the interaction, a specific interval may contain a different scheduling of messages each time it is visited.

Minimal rewind intervals: We denote by $[a, b]$ an interval starting with prover message a and ending with verifier message b. Consider the scheduling of messages as they appear in the $\mathsf{hist}_{s,\ell}$ argument (i.e., at the moment that PROVE$(s, \mathsf{hist}_{s,\ell})$ is invoked). Focusing on messages of session s, we note that for every pair of messages (Pj), (Vj) in this session we can associate a level-w interval $[a_j, b_j]$ so that:

1. Both (Pj) and (Vj) are contained in $[a_j, b_j]$.
2. None of the level-$(w+1)$ sub-intervals of $[a_j, b_j]$ contains both (Pj) and (Vj).

We call such a rewind interval a j-minimal interval. Notice that for every $j \in \{1, \ldots, k\}$ there is only one j-minimal interval $[a_j, b_j]$ and that for every $j \neq j'$ the interval $[a_j, b_j]$ is different from $[a_{j'}, b_{j'}]$.

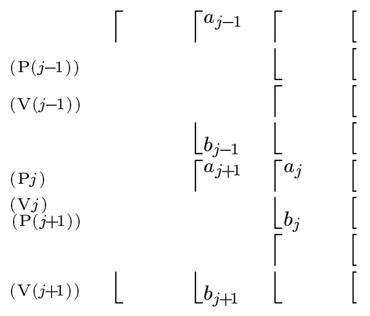


Figure 4.5: Demonstrates the way in which minimal intervals are determined. Also demonstrates possible containments between minimal intervals of different iterations. In this example, the intervals $[a_{j-1}, b_{j-1}]$ and $[a_{j+1}, b_{j+1}]$ are disjoint (as well as the intervals $[a_{j-1}, b_{j-1}]$ and $[a_j, b_j]$), whereas the interval $[a_{j+1}, b_{j+1}]$ contains $[a_j, b_j]$.

In some sense j-minimal intervals correspond to the shortest interval in which the simulator can rewind message (Vj) (that is, while potentially changing the value of (Pj)). Intuitively, for such a rewinding to be useful, the interval should not contain message (V0). Otherwise, the values that were revealed in some run of the interval become irrelevant once rewinds are performed (since all the relevant values in the $\mathcal{T}$ table are deleted whenever we rewind past (V0)). Likewise, the interval should not contain message (`p1`). Otherwise, the simulation faces the risk of getting “stuck” before it manages to reveal multiple (Pj), (Vj) pairs of messages (by running the interval multiple times).

It can be seen that the number of minimal intervals that do not contain neither (V0) nor (`p1`) is at least $k' = k - 2d$ (for simplicity, these intervals will be indexed by $\{1, \ldots, k'\}$). The reason for this is that in every level of the recursion the (V0) (resp. (`p1`)) message is contained in exactly one interval. In particular, the number of minimal intervals that are "spoiled" by (V0) (resp. (`p1`)) is at most d. This guarantees that (V0) and (`p1`) are not visited during single invocations of $[a_j, b_j]$. For the sake of our analysis, however, we will want to make sure that (V0) and (`p1`) are not visited also during *multiple* invocations of $[a_j, b_j]$. In such a case, requiring that $[a_j, b_j]$ does not contain neither (V0) nor (`p1`) may not be sufficient.[14] Jumping ahead, we remark that what we will have to require is that for some intervals, even intervals containing them do not contain neither (V0) nor (`p1`).

Special rewind intervals: In order to define the mapping $f : \mathcal{R} \rightarrow 2^{\mathcal{R}}$, we will need to focus on a specific set of *disjoint* minimal intervals (called special intervals). An important fact that we will extensively use is that if two intervals are disjoint then so is the portion of the random tape that used to run them (i.e., in order to produce uniformly chosen (Pj) messages for the corresponding interval). Another important fact is that in each run of the interval, the SOLVE procedure makes use of "fresh" randomness (i.e., randomness used in one run is never used in a later run).

Definition 4.7.7 (Special intervals) *A minimal interval $[a_j, b_j]$ is said to be* special *if it does not contain any other minimal interval (i.e., if $[a_j, b_j]$ does not contain $[a_{j'}, b_{j'}]$ for any $j' \neq j$).*

Notice that all special intervals are disjoint. We let $S \subseteq \{1, \ldots, k'\}$ denote the set of all indices j for which $[a_j, b_j]$ is special. For simplicity, assume that $S = \{1, \ldots, |S|\}$.

For $j \in S$, let δ_j be the number of times $[a_j, b_j]$ is run since $[a_{j-1}, b_{j-1}]$ is "last" visited (where by "last" we mean during the time (V0) is visited until (`p1`) is reached). A trivial upper bound on δ_j is 2^w, where w is the recursive depth of interval $[a_j, b_j]$. However, since we restrict ourselves to the time between (V0) is visited until (`p1`) is reached, the value of δ_j is typically smaller that 2^w and is in fact upper bounded by α_j (recall that α_j denotes the number of times (Vj) has been visited since the $\mathcal{T}$ table was initialized). Notice that δ_j may be actually smaller than α_j since we are counting only the runs of $[a_j, b_j]$ that have occurred after $[a_{j-1}, b_{j-1}]$ was "last" visited.

The mapping f: We are finally ready to define the mapping $f : \mathcal{R} \rightarrow 2^{\mathcal{R}}$ (Figure 4.6). This mapping makes use of another mapping $h_S : \mathcal{R} \times [\delta_1] \times \ldots \times [\delta_{|S|}] \rightarrow \mathcal{R}$ (Figure 4.7) that depends on the set S (as determined by the schedule at the moment that PROVE$(s, \mathsf{hist}_{s,\ell})$ is invoked).

At a high level, given input ρ and $u_1, \ldots, u_{|S|}$, the mapping h_S takes the portion of the random tape ρ that corresponds to the u_j^{th} run of interval $[a_j, b_j]$ and swaps it with the portion that corresponds to the "last" (i.e., δ_j^{th}) run of this interval (in case $u_j = \delta_j$ then h_S leaves the runs of $[a_j, b_j]$ intact). This is done for all $j \in S$. As we have observed above, different runs of a specific interval use disjoint portions of the random tape. In particular, swapping the randomness of two runs of $[a_j, b_j]$ is an operation that makes sense. Moreover, since disjoint intervals use disjoint portions of the random tape, for every $j \neq j'$ swapping two runs of $[a_j, b_j]$ will not interfere with swapping two runs of $[a_{j'}, b_{j'}]$.

The mapping f will be obtained by invoking $h_S(\rho, u_1, \ldots, u_{|S|})$ with all possible values of $u_1, \ldots, u_{|S|} \in [\delta_1] \times \ldots \times [\delta_{|S|}]$ as input. The set S and the values $\delta_1, \ldots, \delta_{|S|}$ used in order to define the mapping h_S are determined by the mapping f. This is done by running and monitoring

[14]For example, if the interval containing $[a_j, b_j]$ contains either (V0) or (`p1`), then, in some cases, the number of "safe" invocations of $[a_j, b_j]$ is not more than two (even though $[a_j, b_j]$ itself does not contain (V0) or (`p1`)).

the simulation with random tape ρ and black-box access to V^*. Once PROVE$(s, \mathsf{hist}_{s,\ell})$ is reached, f can inspect the scheduling of messages as it appears in $\mathsf{hist}_{s,\ell}$ and determine the set S.[15]

Notice that the mapping f can be computed efficiently. However, this fact is immaterial for the correctness of the analysis since all we have to do is to estasblish the *existence* of such a mapping (regardless of its efficiency).

```
Mapping $f : \mathcal{R} \longrightarrow 2^{\mathcal{R}}$
  **Input:** A random tape $\rho \in \mathcal{R}$
  **Output:** A set of random tapes $\mathcal{G} \subseteq 2^{\mathcal{R}}$
    1. Determine the set of special indices $S \subseteq \{1, \ldots, k'\}$:
        (a) Run the simulator given random tape $\rho$ and black-box access to $V^*$.
        (b) Check for which $j$, interval $[a_j, b_j]$ is special (as induced by $V^*$'s scheduling).
    2. For $j \in S$, let $\delta_j$ be the number of times $[a_j, b_j]$ is run since $[a_{j-1}, b_{j-1}]$ is "last" visited.
    3. Let $\overline{u} = u_1, \ldots, u_{|S|}$ denote a sequence in $\Delta \overset{\text{def}}{=} [\delta_1] \times \ldots \times [\delta_{|S|}]$. Set
           $\mathcal{G} = \bigcup_{\overline{u} \in \Delta} \{h_S(\rho, \overline{u})\}$
    4. Output $\mathcal{G}$.
```

Figure 4.6: Mapping a single "bad" random tape to a set of "good" random tapes.

```
Mapping $h_S : \mathcal{R} \times [\delta_1] \times \ldots \times [\delta_{|S|}] \longrightarrow \mathcal{R}$
  **Input:** A random tape $\rho \in \mathcal{R}$ and a sequence $\overline{u} = u_1, \ldots, u_{|S|} \in [\delta_1] \times \ldots \times [\delta_{|S|}]$
  **Output:** A random tape $\rho_{u_1, \ldots, u_{|S|}} \in \mathcal{R}$
    1. Set $\rho_{u_0} \leftarrow \rho$.
    2. For $j = 1, \ldots, |S|$:
        (a) Let $\rho_w$ denote the portion of $\rho_{u_1, \ldots, u_{j-1}}$ that is used in $w^{\text{th}}$ run of $[a_j, b_j]$.
        (b) Swap the locations of $\rho_{u_j}$ and $\rho_{\delta_j}$ within $\rho_{u_1, \ldots, u_{j-1}}$.
        (c) Denote by $\rho_{u_1, \ldots, u_j}$ the resulting string.
    3. Output $\rho_{u_1, \ldots, u_{|S|}}$.
```

Figure 4.7: Mapping a "bad" random tape to a "good" random tape.

The following Claim will establish Item (3) of Lemma 4.7.5.

Claim 4.7.8 *Let $\rho \in \mathcal{B}$ be a bad random tape. Then the sets $f(\rho) \setminus \{\rho\}$ and $\mathcal{B}$ are disjoint.*

[15]Here we implicitly assume that all invocations of the PROVE procedure prior to the ℓ^{th} invocations did not return $\perp$. This issue can be handled by conducting the analysis inductively while assuming that all previous invocations of the PROVE procedure did not return $\perp$.

Proof: It will be sufficient to show that for every $u_1, \ldots, u_{|S|} \neq \delta_1, \ldots, \delta_{|S|}$, the random tape $\rho_{u_1,\ldots,u_{|S|}} = h_S(\rho, u_1, \ldots, u_{|S|})$ does not belong to $\mathcal{B}$ (notice that $h_S(\rho, \delta_1, \ldots, \delta_{|S|}) = \rho$).

Consider the smallest $j \in S$ for which $u_j \neq \delta_j$. We start by observing that, up to the point in which $[a_j, b_j]$ is run for the first time (after the "last" run of interval $[a_{j-1}, b_{j-1}]$), the randomness used by the simulator when running with $\rho_{u_1,\ldots,u_{|S|}}$ is equal to the randomness used by the simulator when running with ρ. This means that all runs of $[a_j, b_j]$ that occur after $[a_{j-1}, b_{j-1}]$ has been "last" visited will have the same "history" of interaction regardless of whether $\rho_{u_1,\ldots,u_{|S|}}$ or ρ is used.

The key observation for proving the claim is that, modulo the history of the interaction at the starting point of an interval, the randomness used in a specific run of an interval completely determines its outcome (remember that V^*'s random tape is fixed in advance). Since the last occurrence of (Vj) in $\mathcal{T}$ corresponds to the "last" time $[a_j, b_j]$ is visited, then the portion of the random tape used for the δ_j^{th} run of $[a_j, b_j]$ completely determines the value of $(\mathrm{V}j)_{\alpha_j}$ (which is the last occurrence of (Vj) in $\mathcal{T}$).

Notice that, when using $\rho_{u_1,\ldots,u_{|S|}}$ as random tape, the randomness used in ρ in order to perform the δ_j^{th} run of $[a_j, b_j]$ is instead used for the u_j^{th} run of interval $[a_j, b_j]$. Since the randomness used in a specific run of an interval completely determines its outcome, the value of (Vj) in the u_j^{th} run of $[a_j, b_j]$ is now equal to $(\mathrm{V}j)_{\alpha_j}$. Recall that $\rho \in \mathcal{B}$. This in particular means that, when using ρ as random tape, it holds that $(\mathrm{V}j)_{\alpha_j} \neq \texttt{ABORT}$ (by Condition (2) in Definition 4.7.4). Denoting the (Vj) message that appears in the u_j^{th} run of interval $[a_j, b_j]$ by $(\mathrm{V}j)_u$ we then have that, when the simulator uses $\rho_{u_1,\ldots,u_{|S|}}$ as random tape, $(\mathrm{V}j)_u = (\mathrm{V}j)_{\alpha_j} \neq \texttt{ABORT}$.

Since $u_j < \delta_j$, then $(\mathrm{V}j)_u$ does not appear in $\mathsf{hist}_{s,\ell}$ (since it appears in the outcome of the u_j^{th} run of $[a_j, b_j]$ and the "most recently visited" run when $\textsc{prove}(s, \mathsf{hist}_{s,\ell})$ is invoked is the δ_j^{th} run). In addition since whenever $\textsc{prove}(s, \mathsf{hist}_{s,\ell})$ is invoked, some (Vj) message must appear in $\mathsf{hist}_{s,\ell}$, we infer that there exist a (Vj) that occurs after $(\mathrm{V}j)_u$ does. This message corresponds to $(\mathrm{V}j)_{\alpha'_j}$ where α'_j is the number of occurrences of (Vj) in $\mathcal{T}$ when using $\rho_{u_1,\ldots,u_{|S|}}$ as random tape.

We thus have that, when using $\rho_{u_1,\ldots,u_{|S|}}$ as random tape, there must exist a $u < \alpha'_j$ for which $(\mathrm{V}j)_u \neq \texttt{ABORT}$. By Condition (1) in Definition 4.7.4 this implies that $\rho_{u_1,\ldots,u_{|S|}} \notin \mathcal{B}$. ■

We now turn to establish Item (2) of Lemma 4.7.5. Let $g : \mathcal{B} \times \{0,1\}^k \times [\delta_1] \times \ldots \times [\delta_{|S|}] \to \mathcal{R}$ be a mapping defined as:

$$g(\rho, S, \overline{u}) \overset{\text{def}}{=} h_S(\rho, \overline{u})$$

where $\rho \in \mathcal{B}$. To show that for all $\rho \neq \rho' \in \mathcal{B}$, the sets $f(\rho)$ and $f(\rho')$ are disjoint it will be sufficient to show that g is one-to-one. In such a case we would have that for any two $S \neq S' \subseteq \{1, \ldots, k'\}$, it holds that $h_S(\rho, \overline{u}) \neq h_{S'}(\rho', \overline{u}')$ (regardless of the values of $\rho, \overline{u}$ and $\rho', \overline{u}'$) and so the sets $f(\rho) = \bigcup_{\overline{u}} \{h_S(\rho, \overline{u})\}$ and $f(\rho') = \bigcup_{\overline{u}} \{h_{S'}(\rho', \overline{u})\}$ are disjoint.

Claim 4.7.9 *Let $g : \mathcal{B} \times \{0,1\}^k \times [\delta_1] \times \ldots \times [\delta_{|S|}] \to \mathcal{R}$ be as above. Then, g is one-to-one.*

Proof: To argue that g is one-to-one we will define an inverse mapping g^{-1} so that for every random tape $\rho' \in \mathsf{range}(g)$, the value of $g^{-1}(\rho') = (\rho, S, \overline{u})$ satisfies $g(\rho, S, \overline{u}) = \rho'$.

Given $\rho' \in \mathsf{range}(g)$, the basic idea for defining g^{-1} is to recognize the subset of intervals whose randomness was swapped by f (while "producing" ρ' from some $\rho \in \mathcal{B}$) and to reverse the swapping (i.e. to swap back the randomness of these intervals). The main difficulty in doing so lies in the task of recognizing which are these intervals whose randomness is to be swapped (i.e., to recognize what is the set S that corresponds to a run of the simulator with $\rho \in \mathcal{B}$ as random tape).

The solution to this problem will be to inspect the intervals and reverse the swapping of their randomness "inductively". The reason for which the order of swapping is important is that V^*'s answer in a specific interval also depends on the randomness used to run the "most recent execution" of previous intervals (since, whenever we reach a specific interval, the outcome of these "recent" runs appears in the history of the interaction). In order to be able to say something meaningful about an interval's run we must make sure that, whenever we inspect the run of the simulator on this interval, the history of the interaction up to the starting point of the interval is consistent with the outcome of running the simulator with the bad tape $\rho \in \mathcal{B}$ that ρ' "originates" from. The process describing the mapping g^{-1} is depicted in Figure 4.8.

```
Mapping $g^{-1} : \mathcal{R} \longrightarrow \mathcal{B} \times \{0,1\}^k \times [\delta_1] \times \ldots \times [\delta_{|S|}]$

  **Input:** A random tape $\rho \in \mathsf{range}(g) \subseteq \mathcal{R}$.
  **Output:** A random tape $\rho' \in \mathcal{B}$, a set $S \subseteq \{1, \ldots, k'\}$ and a sequence $\overline{u} \in [\delta_1] \times \ldots \times [\delta_{|S|}]$.

    1. Set $\rho_{u_0} \leftarrow \rho$ and $S = \phi$.
    2. For $j = 1, \ldots, k'$:
        (a) Run the simulator given random tape $\rho_{u_1,\ldots,u_{j-1}}$ and black-box access to $V^*$.
        (b) Find *unique* $u_j$ so that $u_j^{\text{th}}$ run of $[a_j, b_j]$ is "properly answered".
        (c) If $u_j < \delta_j$:
            i. Set $S \leftarrow S \cup \{j\}$.
            ii. Let $\rho_w$ denote the portion of $\rho_{u_1,\ldots,u_{j-1}}$ that is used in $w^{\text{th}}$ run of $[a_j, b_j]$.
            iii. Swap the locations of $\rho_{u_j}$ and $\rho_{\delta_j}$ within $\rho_{u_1,\ldots,u_{j-1}}$.
            iv. Denote by $\rho_{u_1,\ldots,u_j}$ the resulting string.
        (d) Otherwise, continue to next $j \in \{1, \ldots, k'\}$.
    3. Output $\rho' = \rho_{u_1,\ldots,u_{|S|}}$, $S$ and $\overline{u} = u_1, \ldots, u_{|S|}$.
```

Figure 4.8: Mapping a "good" tape back to the original "bad" tape.

Since the tape $\rho' \in \mathsf{range}(g)$ that we are trying to invert originates from a bad tape $\rho \in \mathcal{B}$ then for every $j \in \{1, \ldots, k'\}$, when using ρ as random tape, the interval $[a_j, b_j]$ is aborted in all but the last runs of $[a_j, b_j]$, where by last run we mean the last time $[a_j, b_j]$ is executed prior to the invocation of PROVE$(s, \mathsf{hist}_{s,\ell})$. Notice that, once PROVE$(s, \mathsf{hist}_{s,\ell})$ is invoked, we can determine the value of δ_j by simply counting the number of times $[a_j, b_j]$ has been visited from the time (V0) was visited until (p1) is reached. If it happens to be the case that when using ρ' as random tape the last (i.e., δ_j^{th}) run of the currently inspected interval $[a_j, b_j]$ is not properly answered, then we know that the randomness of $[a_j, b_j]$ has been swapped by f and should be swapped back.

If along the way we preserve the "invariant" that the randomness used so far is consistent with the original bad random tape $\rho \in \mathcal{B}$ then it must be the case that, for the above interval, there exists a *unique* $u_j < \delta_j$ so that the u_j^{th} run of $[a_j, b_j]$ is properly answered. We can thus swap the randomness used for the u_j^{th} run with the randomness used for the δ_j^{th} run. As soon as we reach the last special interval we know that the resulting tape is the original "bad" random tape (since all along the way we have preserved the "invariant" that the randomness used so far is consistent with the original $\rho \in \mathcal{B}$). ■

All that remains in order to complete the proof, is to establish Item (1) of Lemma 4.7.5. To do so, we will need to argue that for all $\rho \in \mathcal{B}$ it holds that $|f(\rho)| \geq 2^{k-3d}$. This will be achieved by proving the following lemma.

Lemma 4.7.10 *Let $d = log_2(m \cdot (k+1))$. Then, there exist values $d_1, \ldots, d_{|S|} \in \{1, \ldots, d\}$ so that:*

1. *For all $j \in S$, it holds that $\delta_j = 2^{d_j}$.*
2. *$\sum_{j \in S} d_j \ \geq \ k' - d$.*

Corollary 4.7.11 *Let $\rho \in \mathcal{B}$ be a bad random tape. Then $|f(\rho)| \geq 2^{k-3d}$.*

Proof: By the definition of $f : \mathcal{R} \longrightarrow 2^{\mathcal{R}}$ and by Claim 4.7.9, we have:

$$|f(\rho)| \ = \ \left| \bigcup_{\overline{u} \in \Delta} \{h_S(\rho, \overline{u})\} \right| \ = \ \sum_{\overline{u} \in \Delta} |\{h_S(\rho, \overline{u})\}|$$

Since $|\{h_S(\rho, \overline{u})\}| = 1$, then the size of $f(\rho)$ is in fact equal to the number of $\overline{u}$'s in Δ. The size of $\Delta \stackrel{\text{def}}{=} [\delta_1] \times \ldots \times [\delta_{|S|}]$ is precisely $\prod_{j \in S} \delta_j$, and so:

$$\begin{aligned} |f(\rho)| &= \prod_{j \in S} \delta_j \\ &= \prod_{j \in S} 2^{d_j} && (4.8) \\ &= 2^{\sum_{j \in S} d_j} \\ &\geq 2^{k'-d} && (4.9) \end{aligned}$$

where Eq. (4.8) and Eq. (4.9) follow from Items (1) and (2) of Lemma 4.7.10 respectively. Finally, since $k' = k - 2d$, we get that $|f(\rho)| \geq 2^{k-3d}$, as required. ■

4.7.2 Proof of Lemma 4.7.10 (special intervals are visited many times)

A central tool in the proof of Lemma 4.7.10 will be the notion of the recursion tree. This is a full binary tree whose nodes correspond to the rewind intervals as induced by the recursive calls of the SOLVE procedure. Every node $[a, b]$ in the recursion tree has two descendants. Each one of the descendants corresponds to one of the recursive calls made during some visit to $[a, b]$. The root of the tree corresponds to a rewind interval of size $m \cdot (k+1)$. At the bottom level of the recursion tree there are $m \cdot (k+1)$ nodes each corresponding to distinct interval of length 2. In general, at the w^{th} level of the tree (out of $d = \log_2(m \cdot (k+1))$ possible levels) there are 2^w nodes, each corresponding to a distinct interval of length $m(k+1)/2^{w+1}$.

It can be seen that, for any two nodes labeled $[a, b]$ and $[a', b']$ in the recursion tree, $[a, b]$ is a descendant of $[a', b']$ if and only if interval $[a, b]$ is contained in $[a', b']$. The distance of $[a, b]$ from $[a', b']$ is determined in the straightforward manner by considering the distance between these nodes in the binary tree. Recall that we are focusing on the scheduling as it appears in $\mathsf{hist}_{s,\ell}$ (i.e., at the moment that PROVE$(s, \mathsf{hist}_{s,\ell})$ is invoked). This scheduling induces a specific labeling of the tree's nodes acording to the messages that appear at each one of the rewind intervals at that time. It also determines the identity of the nodes that correspond to minimal intervals, as well as the nodes that correspond to special intervals. By Definition 4.7.7, nodes that correspond to a special interval do not have any descendant that corresponds to a minimal interval.

Let $S \subseteq \{1, \ldots, k'\}$ be the set of all indices j for which interval $[a_j, b_j]$ is special. Let $j \in S$ and let $[\mathsf{A}_j, \mathsf{B}_j]$ be the common ancestor of $[a_{j-1}, b_{j-1}]$ and $[a_j, b_j]$ in the recursion tree. That is, $[\mathsf{A}_j, \mathsf{B}_j]$ is the "deepest" node in the tree that has both $[a_{j-1}, b_{j-1}]$ and $[a_j, b_j]$ as descendants (this corresponds to the smallest rewind interval that contains both $[a_{j-1}, b_{j-1}]$ and $[a_j, b_j]$).

Defining the d_j's - first step: We are now redy to define the value of the d_j's. This will proceed in two steps. We first define a sequence of values $c_1, \ldots, c_{|S|}$. For any $j \in S$, the value of c_j will reflect the overall number of times that interval $[a_j, b_j]$ is visited after $[a_{j'}, b_{j'}]$ is last visited. We then turn to show how to "correct" the values of the c_j's so to take into consideration only those visits that have occurred *before* (p1) has been reached. The resulting sequence of values $d_1, \ldots, d_{|S|}$ will then faithfully reflect the number of times that $[a_j, b_j]$ is visited after $[a_{j'}, b_{j'}]$ is last visited (as required by the definition of the δ_j's). The values $c_1, \ldots, c_{|S|}$ are defined as follows:

- If $j = 1$, then $c_j = 1$.
- If $j > 1$, then c_j equals to the distance of $[a_j, b_j]$ from $[\mathsf{A}_j, \mathsf{B}_j]$.

Notice that for all $j \in S$, it holds that $c_j \geq 1$. Figure 4.9 demonstrates the way in which $c_1, \ldots, c_{|S|}$ are defined.

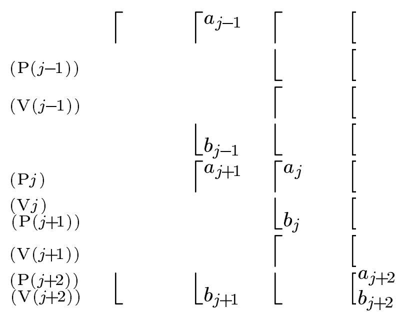


Figure 4.9: Demonstrates the definition of the c_j's. In this example the special intervals are $[a_{j-1}, b_{j-1}]$, $[a_j, b_j]$ and $[a_{j+2}, b_{j+2}]$ (and $j+1 \notin S$). Notice that the distance of $[a_{j+2}, b_{j+2}]$ from its common ancestor with $[a_j, b_j]$ is 2, and so $c_{j+2} = 2$ (the common ancestor being $[a_{j+1}, b_{j+1}]$). Similarly, the distance of $[a_j, b_j]$ from its common ancestor with $[a_{j-1}, b_{j-1}]$ is also 2 and so $c_j = 2$.

Claim 4.7.12 *Let $j \in S$. Then, for every invocation of the common ancestor of $[a_{j-1}, b_{j-1}]$ and $[a_j, b_j]$, the number of times that $[a_j, b_j]$ is visited after $[a_{j-1}, b_{j-1}]$ is last visited is precisely 2^{c_j}.*

Proof: Let $j \in S$ and let $[\mathsf{A}_j, \mathsf{B}_j]$ be the common ancestor of $[a_{j-1}, b_{j-1}]$ and $[a_j, b_j]$. By definition, the value of c_j equals the recursive depth of $[a_j, b_j]$ relative to $[\mathsf{A}_j, \mathsf{B}_j]$. We thus know that for every invocation of interval $[\mathsf{A}_j, \mathsf{B}_j]$, the interval $[a_j, b_j]$ is invoked precisely 2^{c_j} times. To see that all 2^{c_j} invocations of $[a_j, b_j]$ occur after the last invocation of $[a_{j-1}, b_{j-1}]$, we recall that $[a_{j-1}, b_{j-1}]$ and $[a_j, b_j]$ are contained in different halves of the common ancestor $[\mathsf{A}_j, \mathsf{B}_j]$. By definition of the SOLVE procedure, the two invocations of the second half of an interval occur only after the two invocations of the first half have occured. Thus all 2^{c_j} invocations of $[a_j, b_j]$ (which occur as a result of the two recursive invocations of the second half of $[\mathsf{A}_j, \mathsf{B}_j]$) occur after all invocations of $[a_{j-1}, b_{j-1}]$ (which occur as a result of the two recursive invocations of the first half of $[\mathsf{A}_j, \mathsf{B}_j]$). ■

Interfering intervals: Consider any run of the simulator from the time that message (V0) was visited until message (p1) is reached. Since this run involves the exchange of messages $(\mathrm{P}0), (\mathrm{V}0), (\mathrm{P}1), (\mathrm{V}1), \ldots, (\mathrm{P}k), (\mathrm{V}k)$, then it must have been caused by some invocation of an interval $[\mathsf{A}, \mathsf{B}]$ that contains $[a_j, b_j]$ for all $j \in \{1, \ldots, k'\}$. Notice that for all $j \in S$ the interval $[\mathsf{A}, \mathsf{B}]$ contains $[\mathsf{A}_j, \mathsf{B}_j]$. In particular, for every $j \in S$, by the time that (p1) is reached, the interval $[\mathsf{A}_j, \mathsf{B}_j]$ is invoked at least once. By, Claim 4.7.12, this implies that for all $j \in S$, the number of times that $[a_j, b_j]$ is visited after the last visit of $[a_{j-1}, b_{j-1}]$ is precisely 2^{c_j}.

At first glance this seems to establish that $\delta_j = 2^{c_j}$. However, this is not necessarily true. The reason for this is that, by definition, the value of δ_j reflects only the number of visits to $[a_j, b_j]$ *before* (p1) is reached. It might very well be the case that not all of the 2^{c_j} runs of $[a_j, b_j]$ have occurred before (p1) is reached.

Specifically, whenever the second half of the common ancestor $[\mathsf{A}_j, \mathsf{B}_j]$ contains the message (p1), only one of its invocations will occur prior to reaching (p1). This already cuts the number of visits to $[a_j, b_j]$ by a factor of two. The situation is made even worse by the fact that *every* interval that lies “in between” $[\mathsf{A}_j, \mathsf{B}_j]$ and $[a_j, b_j]$ and that contains (p1) can be invoked at most once before reaching (p1) (such intervals are said to be interfering to $[a_j, b_j]$). Thus, the number of invocations of $[a_j, b_j]$ before (p1) is reached decreases exponentially with the number of interfering intervals. For every $j \in S$, let e_j denote the number of intervals interfering to $[a_j, b_j]$. Notice that for all $j \in S$, it holds that $c_j > e_j$ (since for all $j \in \{1, \ldots, k'\}$ interval $[a_j, b_j]$ does not contain (p1)).

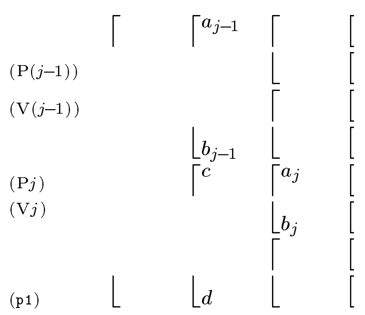


Figure 4.10: Demonstrates the definition of interfering intervals. In this example the special intervals are $[a_{j-1}, b_{j-1}]$ and $[a_j, b_j]$. Notice that $[c, d]$ lies “in between” $[a_j, b_j]$ and its common ancestor with $[a_{j-1}, b_{j-1}]$. Since the interval $[c, d]$ contains (p1), then it is interfering to $[a_j, b_j]$. This means that e_j equals 1 (whereas $c_j = 2$), and that the number of invocations of $[a_j, b_j]$ prior to reaching (p1) (and after visiting $[a_{j-1}, b_{j-1}]$ for the last time) is equal to $2^{c_j - e_j} = 2$ (whereas, without taking interference into account, it would have been $2^{c_j} = 4$).

Claim 4.7.13 *Let $j \in S$. Then, for every invocation of the common ancestor of $[a_{j-1}, b_{j-1}]$ and $[a_j, b_j]$, the number of times that $[a_j, b_j]$ is visited after $[a_{j-1}, b_{j-1}]$ is last visited and* before (p1) is reached *is precisely $2^{c_j - e_j}$.*

Proof Sketch: Let $j \in S$ and let $[\mathsf{A}_j, \mathsf{B}_j]$ be the common ancestor of $[a_{j-1}, b_{j-1}]$ and $[a_j, b_j]$. By definition, the number of “non-interfering” intervals that: (1) are contained in $[\mathsf{A}_j, \mathsf{B}_j]$, (2) contain

$[a_j, b_j]$ but, (3) do not contain (`p1`), is exactly $c_j - e_j$. The key observation is that no such "non-interfering" interval contains an interfering interval (since otherwise it would have contained (`p1`) as well). Thus, prior to reaching (`p1`), all these intervals are invoked at least twice by the interval containing them. This means that the total number of invocations of $[a_j, b_j]$ (which is contained in all of these intervals) is exactly $2^{c_j - e_j}$. ■

We are finally ready to define $d_1, \ldots, d_{|S|}$. For any $j \in S$, let

$$d_j \overset{\text{def}}{=} c_j - e_j$$

To complete the proof of Lemma 4.7.10 we need to prove the following claim.

Claim 4.7.14 *Let $d_1, \ldots, d_{|S|}$ be defined as above. Then,*

$$\sum_{j \in S} d_j \quad \geq \quad k' - d$$

Proof: The proof is by induction on k'. For any choice of k', let $S \subseteq \{1, \ldots, k'\}$, $\{(c_j, d_j, e_j)\}_{j \in S}$ be as above. We will show that for every $k' \geq d$, it holds that that $\sum_{j \in S}(c_j - e_j) \geq k' - d$. We stress that throughout the proof, we do not make use of any property of the schedule (besides using the "binary-tree structure" and the depth, d, of the simulator's execution).

Base case ($k' = d + 1$): Since $|S| \geq 1$, and for all $j \in S$, it holds that $c_j - e_j > 0$, we have:

$$\sum_{j \in S}(c_j - e_j) \quad \geq \quad 1 \quad = \quad k' - d$$

Induction step ($k' > d$): Consider the $k' - 1$ intervals that are obtained by removing the index $|S|$ (i.e., the index corresponding to the "latest" special interval $[a_{|S|}, b_{|S|}]$). Let $S' \subseteq \{1, \ldots, k'\} \setminus \{|S|\}$ denote the set of special intervals after the removal of the index $|S|$. Notice that $S \subseteq S'$. This is because any interval that was special before the removal of $|S|$ will remain special after the removal. Moreover, for all $j \in S \cap S'$, the value of $c_j - e_j$ has not been changed by the removal of $|S|$ (since it is always defined relative to the "preceding" element in $|S|$). We now have two cases.

Case 1: There exists $J \in S' \setminus S$ so that the interval $[a_J, b_J]$ is special. That is, by removing $[a_{|S|}, b_{|S|}]$ we have caused $[a_J, b_J]$ to be special (even though it was not special before). This could have happened only if the *unique* interval previously contained by $[a_J, b_J]$ was $[a_{|S|}, b_{|S|}]$ (otherwise, $[a_J, b_J]$ would have not become special after removing $[a_{|S|}, b_{|S|}]$). In particular, $[a_J, b_J]$ does not contain the intervals $[a_{|S|-1}, b_{|S|-1}]$ (i.e., the special interval preceding $[a_{|S|}, b_{|S|}]$) and $[\mathsf{A}_{|S|}, \mathsf{B}_{|S|}]$ (i.e., the common ancestor of $[a_{|S|-1}, b_{|S|-1}]$ and $[a_{|S|}, b_{|S|}]$). This means that both $[a_J, b_J]$ and $[a_{|S|}, b_{|S|}]$ have the same common ancestor with $[a_{|S|-1}, b_{|S|-1}]$. Since $[a_J, b_J]$ contains $[a_{|S|}, b_{|S|}]$ then $c_{|S|} > c_J$. In addition, since the set of intervals interfering to $[a_J, b_J]$ is equal to the set of intervals interfering to $[a_{|S|}, b_{|S|}]$ then $e_{|S|} = e_J$. As a consequence, $c_{|S|} - e_{|S|} > c_J - e_J$. Using the induction hypothesis (for $k' - 1$), we get:

$$\begin{aligned}
\sum_{j \in S}(c_j - e_j) \quad = \quad \sum_{j \in S' \setminus \{J\}}(c_j - e_j) + (c_{|S|} - e_{|S|}) \quad &\geq \quad \sum_{j \in S' \setminus \{J\}}(c_j - e_j) + (c_J - e_J + 1) \\
&\geq \quad \sum_{j \in S'}(c_j - e_j) + 1 \\
&\geq \quad (k' - 1) - d + 1 \\
&= \quad k' - d
\end{aligned}$$

Case 2: $S' = S$. Using the induction hypothesis, and the fact that $c_{|S|} - e_{|S|} > 0$, we get:

$$\begin{aligned}
\sum_{j\in S}(c_j - e_j) &= \sum_{j\in S'}(c_j - e_j) + (c_{|S|} - e_{|S|}) \\
&\geq \sum_{j\in S'}(c_j - e_j) + 1 \\
&\geq (k'-1) - d + 1 \\
&= k' - d
\end{aligned}$$

In both cases, we obtain the desired result. This completes the proof of Claim 4.7.14. ■

Using, Claims 4.7.13 and 4.7.14, we have:

1. For all $j \in S$, it holds that $\delta_j = 2^{d_j}$.
2. $\sum_{j\in S} d_j \;\geq\; k' - d$.

This completes the proof of Lemma 4.7.10.

4.8 Extensions

4.8.1 Applicability to other protocols

Theorem 4.1 is proved by adding an $O(\alpha(n)\cdot\log n)$-round "preamble" to the well known 3-round protocol for Hamiltonicity by Blum [6]. The crucial property of Blum's protocol that we need in order to prove concurrent zero-knowledge is that it is a "challenge–response" type of protocol so that the simulation task becomes trivial if the verifier's "challenge" is known in advance. Using our mehtodology, it is possible to transform *any* such protocol into concurrent zero-knowledge, while paying only a logarithmic cost in the round complexity.

Denote by $\mathcal{CRZK}(r(\cdot))$ the class of all languages $L \subseteq \{0,1\}^*$ having an $r(\cdot)$-round "challenge–response" interactive proof (resp. argument) system, so that the simulation task becomes "trivial" if the verifier's "challenges" are known in advance. We now have the following theorem.

Theorem 4.2 (A generic transformation for $\mathcal{CRZK}$) *Let $\alpha : N \rightarrow N$ be any super-constant interger function, and let $r : N \rightarrow N$ be any integer function. Then, assuming the existence of perfectly-hiding commitment schemes (resp. one-way functions), every language $L \in \mathcal{CRZK}(r(\cdot))$ has an $(r(n) + O(\alpha(n)\cdot\log n))$-round concurrent zero-knowledge proof (resp. argument) system.*

In light of Theorem 4.2, Construction 4.3.2 may be viewed as a generic transformation that enhances such protocols and makes them secure in the concurrent setting with only a logarithmic increase in the round complexity. Examples for protocols satsfying the above property are the well known protocols for graph 3-coloring [25], for proving the knowledge of a square root modulo a composite [20], as well as the protocol for proving knowledge of discrete logarithms modulo a prime [44].

4.8.2 $c\mathcal{ZK}$ arguments based on any one-way function

Using Construction 4.3.1 as a building block, we are able to present a $c\mathcal{ZK}$ argument system for Hamiltonicity, while assuming only the existence of one-way functions. Since Hamiltonicity is $\mathcal{NP}$-complete, it will follow that every language in $\mathcal{NP}$ can be argued in $c\mathcal{ZK}$.

Construction 4.8.1 (A $c\mathcal{ZK}$ argument system for HC):

- Common Input: *a directed graph* $G = (V, E)$ *with* $n \stackrel{\text{def}}{=} |V|$, *and a parameter* $k = k(n)$ (*determining the number of rounds*).
- Auxiliary Input to Prover: *a directed Hamiltonian Cycle,* $C \subset E$, *in* G.
- First stage: *This stage involves* $2k + 2$ *rounds and is independent of the common input* G.
  1. Prover's preliminary step (P0): *Uniformly select a first message for a* (*2-round*) *perfectly-hiding commitment scheme and send it to the verifier.*
  2. Verifier's preliminary step (V0): *Uniformly select* $\sigma \in \{0,1\}^n$, *and two sequences,* $\{\sigma^0_{i,j}\}^k_{i,j=1}$, $\{\sigma^1_{i,j}\}^k_{i,j=1}$, *each consisting of* k^2 *random* n*-bit strings. The sequences are chosen under the constraint that for every* i, j *the value of* $\sigma^0_{i,j} \oplus \sigma^1_{i,j}$ *equals* σ. *Commit* (*using the perfectly-hiding commitment scheme*) *to all* $2k^2+1$ *selected strings. The commitments are denoted* $\beta, \{\beta^0_{i,j}\}^k_{i,j=1}$, $\{\beta^1_{i,j}\}^k_{i,j=1}$.
  3. *For* $j = 1, \ldots, k$:
     (a) Prover's j^{th} step (Pj): *Uniformly select a* k*-bit string* $r_j = r_{1,j}, \ldots, r_{k,j} \in \{0,1\}^k$ *and send it to the verifier.*
     (b) Verifier's j^{th} step (Vj): *Reveal the values* (*preimages*) *of* $\beta^{r_{1,j}}_{1,j}, \ldots, \beta^{r_{k,j}}_{k,j}$.
  4. *The prover proceeds with the execution if and only if for every* $j \in \{1, \ldots, k\}$, *the verifier has properly decommited to the values of* $\sigma^{r_{1,j}}_{1,j}, \ldots, \sigma^{r_{k,j}}_{k,j}$ (*i.e., that for every* $i \in \{1, \ldots, k\}$, $\sigma^{r_{i,j}}_{i,j}$ *is a valid decommitment of* $\beta^{r_{i,j}}_{i,j}$).
- Second stage: *The prover and verifier engage in* n (*parallel*) *executions of a slightly modified version of the basic Hamiltonicity protocol* (*described in Construction 4.3.1*):
  1. Prover's first step (`p1`): *Send the first message in the Hamiltonicity proof system* (*i.e.,* n *parallel copies of Step* ($\widehat{\texttt{p1}}$) *in Construction 4.3.1*).
  2. Verifier's first step (`v1`): *Send the value of* σ, *as well as the value of all* k^2 *commitments that have not been revealed in the first stage* (*i.e.,* $\{\sigma^{1-r_{i,j}}_{i,j}\}^k_{i,j=1}$). *In addition prove* (*using an ordinary zero-knowledge argument of knowledge*) *the knowledge of* $k+1$ *strings,* $s, s_1, \ldots, s_k$, *so that* $C_s(\sigma) = \beta$ *and* $C_{s_j}(\sigma^{1-r_j}_j) = \beta^{1-r_j}_j$ *for all* j.
  3. Prover's second step (`p2`): *Check that the verifier has properly decommited to the values of* σ *and* $\{\sigma^{1-r_{i,j}}_{i,j}\}^k_{i,j=1}$ (*in particular, check that* $\sigma^0_{i,j} \oplus \sigma^1_{i,j}$ *indeed equals* σ *for all* j). *If so, send the third message in the basic Hamiltonicity proof system* (*i.e.,* n *parallel copies of Step* ($\widehat{\texttt{p2}}$) *in Construction 4.3.1*).
  4. Verifier's second step (`v2`): *Conduct the verification of the prover's proofs* (*i.e., as described in Step* ($\widehat{\texttt{v2}}$) *of Construction 4.3.1*), *and accept if and only if all corresponding conditions hold.*

Completeness and soundness of Construction 4.8.1 are proved in a similar way to Construction 4.3.2. The main difference is in the proof of soundness. This time, rather than using the perfect secrecy of the commitments used in Step (V0) of Construction 4.3.2, we use the zero-knowedge property of the argument used in Step (`v1`), as well as the computational secrecy of the commitments used in Step (V0) of Construction 4.8.1. We thus have:

Proposition 4.8.2 *Suppose there exist one-way functions. Then Construction 4.8.1 constitutes an interactive argument system for Hamiltonicity.*

Using the same simulator as the one used for Construction 4.3.2 and with some more work on the analysis of its success probability and output distribution (building on the soundness of the $\mathcal{ZK}$ argument used in Step (`v1`)), we obtain.

Theorem 4.3 ($c\mathcal{ZK}$ argument) *Suppose there exist one-way functions. Let $\alpha : N \to N$ be any super-constant function, and let $k(n) = \alpha(n) \cdot \log n$. Then, any instantiation of Construction 4.8.1 with parameter $k = k(n)$ is concurrent zero-knowledge.*

4.8.3 Applicability to resettable zero-knowledge

Our result also enables improvement in the round complexity of *resettable* zero-knowledge [8]. Specifically, using a general transformation of (certain) concurrent zero-knowledge protocols into resettable zero-knowledge [8], we obtain:

Theorem 4.4 (Resettable $\mathcal{ZK}$) *Assuming the existence of perfectly-hiding commitment schemes (resp. one-way functions), there exists an $\tilde{O}(\log n)$-round resettable zero-knowledge proof (resp. argument) system for every language $L \in \mathcal{NP}$.*

Theorem 4.4 is proven by employing a general transformation (by Canetti et al. [8]) that applies to a subclass of $c\mathcal{ZK}$ protocols. When applied to the $c\mathcal{ZK}$ proof system presented in Construction 4.3.2 (as well as Construction 4.8.1), the transformation yields a resettable $\mathcal{ZK}$ proof (resp. argument) system. The class of protocols to which the [8] transformation applies is the class of admissible protocols. Loosely speaking, the class of admissible protocols consists of all $c\mathcal{ZK}$ protocols in which the first verifier message "essentially determines" all its subsequent messages. What we mean by "essentially determines" is that the only freedom retained by the verifier is either to abort (or act so that the prover aborts) or to send a practically predetermined message. Recall that, in our case, the first verifier message is a sequence of commitments that are revealed (i.e., decommited) in subsequent verifier steps. In such a case, the verifier's freedom in subsequent steps is confined to either send an illegal decommitment (which is viewed as aborting and actually causes the prover to abort) or properly decommit to the predetermined value. It follows that our $c\mathcal{ZK}$ protocol satisfies the "admissibility" property required by [8], and can be thus transformed into resettable $\mathcal{ZK}$. For more details, see [8].

4.8.4 Concurrent $\mathcal{ZK}$ arguments with poly-logarithmic efficiency

Another application of our work is the existence of concurrent zero-knowledge arguments with *poly-logarithmic efficiency.* Denote by $c\mathcal{ZK}(r(\cdot), m(\cdot))$ the class of all languages $L \subseteq \{0,1\}^*$ having a zero-knowledge argument system, so that on common input $x \in \{0,1\}^*$, the number of messages exchanged is at most $r(|x|)$, and the total length of the messages exchanged is at most $m(|x|)$. In case that $m(n) = \text{polylog}(n)$, the argument system is said to have poly-logarithmic efficiency. Zero-knowledge arguments with poly-logarithmic efficiency have been constructed by Kilian [33], while assuming the existence of strong collision resistant hash functions (i.e., so that for some $\epsilon > 0$ forming collisions with probability greater than 2^{-k^ϵ} requires at least 2^{k^ϵ} time). We now have the following theorem.

Theorem 4.5 ($c\mathcal{ZK}$ with poly-logarithmic efficiency) *Assuming the existence of strong collision resistant hash functions, $\mathcal{NP}$ is contained in $c\mathcal{ZK}(\tilde{O}(\log), \text{polylog})$. That is, for every language $L \in \mathcal{NP}$, there exists an $\tilde{O}(\log n)$-round black-box concurrent zero-knowledge* argument *system with poly-logarithmic efficiency.*

Theorem 4.5 is proved by applying the transformation referred to in Section 4.8.1 to the protocol of Kilian [33]. The theorem will follow by noting that the preamble of Construction 4.3.2 can be constructed with polylogarithmic efficiency, and that Kilian's arguments satisfy the property required by Theorem 4.2.

Chapter 5

$c\mathcal{ZK}$ Without Aborts

The lower bound presented in Chapter 4 heavily relies on the fact that the adversary verifier is allowed to occasionally abort the concurrent interaction. In this chapter, we consider concurrent composition of $\mathcal{ZK}$ in which the verifier is never allowed to abort during the interaction (we call such executions *non-aborting concurrent executions*). This restriction on the verifier strategy is quite reasonable and it is not inconceivable that it might enable the construction of constant-round $c\mathcal{ZK}$ protocols (in particular, the lower-bound presented in Chapter 4 does not apply in such a setting). Determining the round-complexity of $c\mathcal{ZK}$ protocols with non-aborting concurrent executions thus seems to be an interesting question (see also Chapter 7). In this Chapter we make a preliminary step towards achieving this task by showing that even in the non-aborting case the task of concurrent composition is "non-trivial".

Theorem 5.1 *Suppose that (P, V) is a 7-round proof system for a language L, and that non-aborting concurrent executions of P can be simulated in polynomial-time using black-box simulation. Then $L \in \mathcal{BPP}$. This 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).*

The proof of Theorem 5.1 builds on the works of Goldreich and Krawczyk [24] and Kilian, Petrank and Rackoff [36]. It utilizes a fixed scheduling of the concurrent executions. This scheduling is defined recursively and is more sophisticated than the one proposed by [15] and used by [36]. It also exploits a special property of the first message sent by the verifier. At the end of this chapter (Section 5.2) we show hot to extend Theorem 5.1 so to prove that the 9-round version of the Richardson-Kilian protocol [42] (i.e., with parameter $k = 2$) cannot be black-box simulated under non-aborting concurrent executions.

5.1 Proof of Theorem 5.1

The high level structure of the proof roughly follows the one the proof presented in Chapter 3. That is, we construct a concurrent schedule of sessions, and demonstrate 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 rewinds will be super-polynomial unless the number of rounds used by the protocol obeys the bound, or $L \in \mathcal{BPP}$. The main difference lies in the fact that the adversary verifier considered in the current proof never aborts. This obviously facilitates the simulator's task. Still, since the bound we are proving here is considerably more modest than the one proved in Chapter 3, the resulting proof ends up being much simpler.

5.1.1 The schedule, aversary verifiers and decision procedure

The schedule

For each $x \in \{0,1\}^n$, we consider the following concurrent scheduling of n sessions all run on common input x. The scheduling is defined recursively, where the scheduling of m sessions (denoted $\mathcal{R}_m$) proceeds in 3 phases:

First phase: Each of the first $m/\log m$ sessions exchanges three messages (i.e., $\mathtt{p}_1, \mathtt{v}_1, \mathtt{p}_2$), this is followed by a recursive application of the scheduling on the next $m/\log m$ sessions.

Second phase: Each of the first $m/\log m$ sessions exchanges two additional messages (i.e., $\mathtt{v}_2, \mathtt{p}_3$), this is followed by a recursive application of the scheduling on the last $m - 2 \cdot \frac{m}{\log m}$ sessions.

Third phase: Each of the first $m/\log m$ sessions exchanges the remaining messages (i.e.,$\mathtt{v}_3$,$\mathtt{p}_4$,$\mathtt{v}_4$).

The schedule is depicted in Figure 5.1. We stress that the verifier typically postpones its answer (i.e., $\mathtt{v}_j^{(i)}$) to the last prover's message (i.e., $\mathtt{p}_j^{(i)}$) till after a recursive sub-schedule is executed, and that it is crucial that in the first phase each session will finish exchanging its messages before the next sessions begins (whereas the order in which the messages are exchanged in the second and third phases is immaterial).

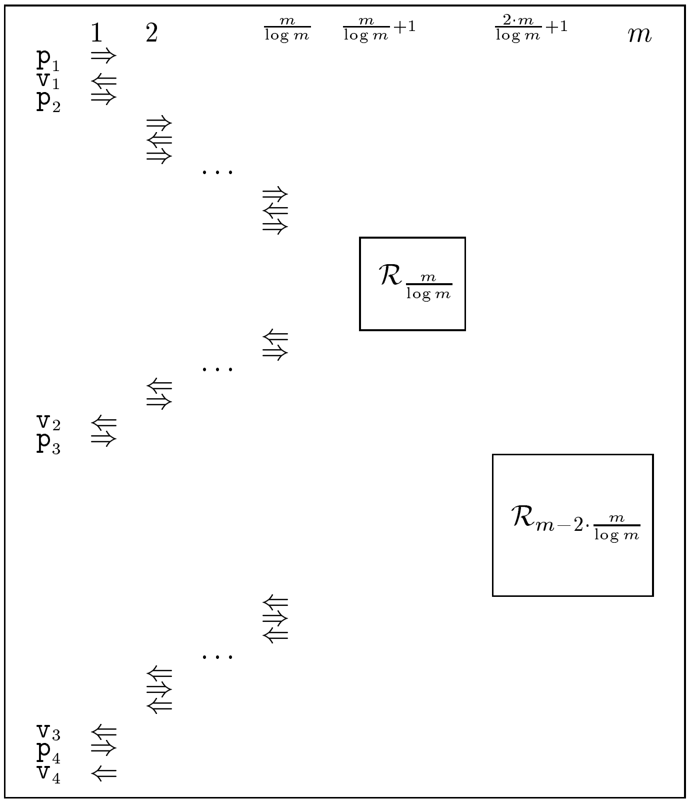


Figure 5.1: The fixed schedule – recursive structure for m sessions.

Definition 5.1.1 (Identifiers of next message) *The fixed 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* $\overline{q} = (a_1, b_1, ..., a_t, b_t, a_{t+1})$*, we let* $\pi_{\rm sn}(\overline{q}) \in \{1, ..., n\}$ *denote the session to which the next verifier message belongs, and by* $\pi_{\rm msg}(\overline{q}) \in \{1, ..., 4\}$ *its index within the verifier's messages in this session.*

Definition 5.1.2 (Initiation-prefix) *The* initiation-prefix $\overline{ip}$ of a query $\overline{q}$ *is the prefix of* $\overline{q}$ *ending with the prover's initiation message of session* $\pi_{\rm sn}(\overline{q})$*. More formally,* $\overline{ip} = a_1, b_1, ..., a_\ell, b_\ell, a_{\ell+1}$*, is the initiation-prefix of* $\overline{q} = (a_1, b_1, ..., a_t, b_t, a_{t+1})$ *if* $a_{\ell+1}$ *is of the form* $\mathtt{p}_1^{(i)}$ *for* $i = \pi_{\rm sn}(\overline{q})$*.* (Note that $\pi_{\rm msg}(\overline{q})$ may be any index in $\{1, ..., 4\}$, and that a_{t+1} need not belong to session i.)

Definition 5.1.3 (Prover-sequence) *The* prover-sequence *of a query* $\overline{q}$ *is the sequence of all prover messages in session* $\pi_{\rm sn}(\overline{q})$ *that appear in the query* $\overline{q}$*. The length of such a sequence is* $\pi_{\rm msg}(\overline{q}) \in \{1, \ldots, 4\}$*. In case the length of the prover-sequence equals* 4*, both query* $\overline{q}$ *and its prover-sequence are said to be* terminating (*otherwise, they are called* non-terminating)*. The prover-sequence is said to correspond to the initiation-prefix* $\overline{ip}$ *of the query* $\overline{q}$*.* (Note that all queries having the same initiation-prefix agree on the first element of their prover-sequence, since this message is part of the initiation-prefix.)

We consider what happens when a black-box simulator (for the above schedule) is given oracle access to a verifier strategy V_h defined as follows (depending on a hash function h and the input x).

The verifier strategy V_h

On query $\overline{q} = (a_1, b_1, ..., a_t, b_t, a_{t+1})$, where the a's are prover messages (and x is implicit in V_h), the verifier answers as follows:

1. First, V_h checks if the execution transcript given by the query is legal (i.e., consistent with V_h's prior answers), and answers with an error message if the query is not legal. (In fact this is not necessary since by our convention the simulator only makes legal queries. From this point on we ignore this case.)

2. More importantly, V_h checks whether the query contains the transcript of a session in which the last verifier message indicates rejecting the input. In case such a session exists, V_h refuses to answer (i.e., answers with some special "refuse" symbol).

3. Next, V_h determines the initiation-prefix, denoted $a_1, b_1, ..., a_\ell, b_\ell, a_{\ell+1}$, of query $\overline{q}$. It also determines $i = \pi_{\rm sn}(\overline{q})$, $j = \pi_{\rm msg}(\overline{q})$, and the prover-sequence of query $\overline{q}$, denoted $\mathtt{p}_1^{(i)}, ..., \mathtt{p}_j^{(i)}$.

4. Finally, V_h determines $r_i = h(a_1, b_1, ..., a_\ell, b_\ell, a_{\ell+1})$ (as coins to be used by V), and answers with the message $V(x, r_i; \mathtt{p}_1^{(i)}, ..., \mathtt{p}_j^{(i)})$ that would have been sent by the honest verifier on common input x, random-pad r_i, and prover's messages $\mathtt{p}_1^{(i)}, ..., \mathtt{p}_j^{(i)}$.

Assuming towards the contradiction that a black-box simulator, denoted S, contradicting Theorem 5.1 exists, we now descibe a probabilistic polynomial-time decision procedure for L, based on S. Recall that we may assume that S runs in strict polynomial time: we denote such time bound by $t_S(\cdot)$. On input $x \in L \cap \{0,1\}^n$ and oracle access to any (probabilistic polynomial-time) V^*, the simulator S must output transcipts with distribution having computational deviation of at most $1/6$ from the distribution of transcripts in the actual concurrent executions of V^* with P.

A slight modification of the simulator: Before presenting the procedure, we slightly modify the simulator so that it never makes a query that is refused by a verifier V_h. Note that this condition can be easily checked by the simulator, 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.

Decision procedure for L

On input $x \in \{0,1\}^n$, proceed as follows:

1. Uniformly select a function h out of a small family of $t_S(n)$-wise independent hash functions mapping poly(n)-bit long sequences to $\rho_V(n)$-bit sequences, where $\rho_V(n)$ is the number of random bits used by V on an input $x \in \{0,1\}^n$.
2. Invoke S on input x providing it black-box access to V_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_h.
3. Accept if and only if all sessions in the transcript output by S are accepting.

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

Lemma 5.1.4 (Performance on YES-instances) *For all but finitely many $x \in L$, the above procedure acccepts x with probability at least* $2/3$.

Proof Sketch: The key observation is that for uniformly selected h, the behavior of V_h in actual (concurrent) interactions with P is identical to the behavior of V in such interactions. The reason is that, in such actual interactions, a randomly selected h determines uniformly and independently distributed random-pads for all n sessions. Since with high probability (say at least $5/6$), V accepts in all n concurrent sessions, the same must be true for V_h, when h is uniformly selected. Since the simulation deviation of S is at most $1/6$, it follows that for every h the probability that $S^{V_h}(x)$ is a transcript in which all sessions accept is lower bounded by $p_h - 1/6$, where p_h denotes the probability that V_h accepts x (in all sessions) when interacting with P. Taking expectation over all possible h's, the lemma follows. ■

Lemma 5.1.5 (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 polynomial p and all but finitely many $x \notin L$, the above procedure accepts x with probability at most $1/p(|x|)$. Assuming towards the contradiction that this is not the case, we will construct a (probabilistic polynomial-time) strategy for a cheating prover that fools the honest verifier V with success probability at least $1/\text{poly}(n)$ (in contradiction to the computational-soundness of the proof system). Loosely speaking, the argument capitalizes on the fact that rewinding of a session requires the simulator to work on a new simulation sub-problem (one level down in the recursive construction). New work is required since each different message for the rewinded session forms an unrelated instance of the simulation sub-problem (by virtue of definition of V_h). The schedule causes work involved in such rewinding to accumulate to too much, and so it must be the case that the simulator does not rewind some (full instance of some) session. In this case the cheating prover may use such a session in order to fool the verifier.

5.1.2 Proof of Lemma 5.1.5 (performance on NO-instances)

Let us fix an $x \in \{0,1\}^n \setminus L$ as above.[1] Define by $\mathtt{AC} = \mathtt{AC}_x$ the set of pairs (σ, h) so that on input x, coins σ and oracle access to V_h, the simulator outputs a transcript, denoted $S_\sigma^{V_h}(x)$, in which all n sessions accept. Recall that our contradiction assumption is that $\Pr_{\sigma,h}[(\sigma,h) \in \mathtt{AC}] > 1/p(n)$, for some fixed polynomial $p(\cdot)$.

The cheating prover

The cheating prover starts by uniformly selecting a pair (σ, h) and hoping that (σ, h) is in $\mathtt{AC}$. It next selects uniformly two elements ℓ and ζ in $\{1, ..., q_S(n)\}$, where $q_S(n) < t_S(n)$ is a bound on the number of queries made by S on input $x \in \{0,1\}^n$. The prover next emulates an execution of $S_\sigma^{V_{h'}}(x)$ (where h', which is essentially equivalent to h, will be defined below), while interacting with the honest verifier V. The prover handles the simulator's queries as well as the communication with the verifier as follows: Suppose that the simulator makes query $\overline{q} = (a_1, b_1, ..., a_t, b_t, a_{t+1})$, where the a's are prover messages.

1. Operating as V_h, the cheating prover first determines the initiation-prefix, $\overline{ip}$ corresponding to the current query $\overline{q}$. Let $\overline{ip} = a_1, b_1, ..., a_\ell, b_\ell, a_{\ell+1}$, (Note that by our convention and the modification of the simulator there is no need to perform Steps 1 and 2 of V_h.)

2. If $\overline{ip}$ is the ℓ^{th} distinct initiation-prefix resulting from the simulator's queries so far then the cheating prover operates as follows:

   (a) The cheating prover determines $i = \pi_{\text{sn}}(\overline{q})$, $j = \pi_{\text{msg}}(\overline{q})$, and the prover-sequence of $\overline{q}$, denoted $\mathtt{p}_1^{(i)}, ..., \mathtt{p}_j^{(i)}$ (as done by V_h in Step 3).

   (b) If the query $\overline{q}$ is non-terminating (i.e., $j \leq 3$), and the cheating prover has only sent $j-1$ messages to the actual verifier then it forwards $\mathtt{p}_j^{(i)}$ to the verifier, and feeds the simulator with the verifier's response (i.e., which is of the form $\mathtt{v}_j^{(i)}$).[2]

   (c) If the query $\overline{q}$ is non-terminating (i.e., $j \leq 3$), and the cheating prover has already sent j messages to the actual verifier, the prover retrieves the j^{th} message it has received and feeds it to the simulator.[3]

   (d) Whenever the query $\overline{q}$ is terminating (i.e., $j = 4$), the cheating prover operates as follows:

[1] In a formal proof we need to consider infinitely many such x's.

[2] We comment that by our conventions regarding the simulator, it cannot be the case that the cheating prover has sent less than $j-1$ messages to the actual verifier: The prefixes of the current query dictate $j-1$ such messages.

[3] We comment that the cheating prover may fail to conduct Step 2c. This will happen whenever the simulator makes two queries with the same initiation-prefix and the same number of prover messages in the corresponding session, but with a different sequence of such messages. Whereas this will never happen when $j = 1$ (as once the initiation-prefix is fixed then so is the value of $\mathtt{p}_1^{(i)}$), it may very well be the case that for $j \in \{2,3\}$ a previous query regarding initiation-prefix $\overline{ip}$ had a different $\mathtt{p}_j^{(i)}$ message. In such a case the cheating prover will indeed fail. The punchline of the analysis is that with noticeable probability this will not happen.

i. As long as the ζ^{th} terminating query corresponding to the above initiation-prefix has not been made, the cheating prover feeds the simulator with $\mathtt{v}_4^{(i)} = 0$ (i.e., session rejected).

ii. Otherwise, the cheating prover operates as in Step 2b (i.e., it forwards $\mathtt{p}_4^{(i)}$ to the verifier, and feeds the simulator with the verifier's response – some $\mathtt{v}_4^{(i)}$ message).[4]

3. If $\overline{ip}$ is NOT the ℓ^{th} distinct initiation-prefix resulting from the queries so far then the prover emulates V_h in the obvious manner (i.e., as in Step 4 of V_h): It first determines $r_i = h(a_1, b_1, ..., a_\ell, b_\ell, a_{\ell+1})$, and then answers with $V(x, r_i; \mathtt{p}_1^{(i)}, ..., \mathtt{p}_j^{(i)})$, where all notations are as above.

Defining h' (mentioned above): Let (σ, h) and ℓ be the initial choices made by the cheating prover, and suppose that the honest verifier uses coins r. Then, the function h' is defined to be uniformly distributed among the functions h'' which satisfy the following conditions: The value of h'' on the ℓ^{th} initiation-prefix equals r, whereas for every $\ell' \neq \ell$, the value of h'' on the ℓ'^{th} initiation-prefix equals the value of h on this prefix. (Here we use 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' does not effect Step 3 of the cheating prover, and that the prover does not know h'.)

The probability that the cheating prover makes the honest verifier accept is lower bounded by the probability that both $(\sigma, h') \in \mathtt{AC}$ and the messages forwarded by the cheating prover in Step 2 are consistent with an accepting conversation with $V_{h'}$. For the latter event to occur, it is necessary that the ℓ^{th} distinct initiation-prefix will be useful (in the sense hinted above and defined now). It is also necessary that ζ was "successfully" chosen (i.e., the ζ^{th} terminating query which corresponds to the ℓ^{th} initiation-prefix is accepted by $V_{h'}$).

Definition 5.1.6 (Accepting query) *A terminating query $\overline{q} = (a_1, b_1, ..., a_t, b_t, a_{t+1})$ (i.e., for which $\pi_{\text{msg}}(\overline{q}) = 4$) is said to be* accepting *if $V_{h'}(a_1, b_1, ..., a_t, b_t, a_{t+1})$ equals 1 (i.e., session $\pi_{\text{sn}}(\overline{q})$ is accepted by $V_{h'}$).*

Definition 5.1.7 (Useful initiation-prefix) *A specific initiation-prefix $\overline{ip}$ in an execution of $S_\sigma^{V_{h'}}(x)$ is called* useful *if the following conditions hold:*

1. *During its execution, $S_\sigma^{V_{h'}}(x)$ made at least one accepting query that corresponds to the initiation-prefix $\overline{ip}$.*

2. *As long as no accepting query corresponding to the initiation-prefix $\overline{ip}$ was made during the execution of $S_\sigma^{V_{h'}}(x)$, the number of (non-terminating) different prover-sequences that correspond to $\overline{ip}$ is at most 3, and these prover-sequences are prefixes of one another.*[5]

Otherwise, the prefix is called unuseful.

[4] We note that once the cheating prover arrives to this point, then it either succeds in the cheating task or completely fails (depending on the verifier's response). As a consequence, it is not essential to define the cheating prover's actions from this point on (as in both cases the algorithm will be terminated).

[5] In other words, we allow for many different terminating queries to occur (as long as they are not accepting). On the other hand, for $j \in \{1, 2, 3\}$ only a single query that has a prover sequence of length j is allowed. This requirement will enable us to avoid situations in which the cheating prover will fail (as described in Footnote 3).

The success probability

Define a Boolean indicator $\chi(\sigma, h', \ell)$ to be true if and only if the ℓ^{th} distinct initiation-prefix in an execution of $S_\sigma^{V_{h'}}(x)$ is useful. Define an additional Boolean indicator $\psi(\sigma, h', \ell, \zeta)$ to be true if and only if the ζ^{th} terminating query among all terminating queries that correspond to the ℓ^{th} distinct initiation-prefix (in an execution of $S_\sigma^{V_{h'}}(x)$) is the first one to be accepting. It follows that if the cheating prover happens to select (σ, h, ℓ, ζ) so that both $\chi(\sigma, h', \ell)$ and $\psi(\sigma, h', \ell, \zeta)$ hold then it convinces $V(x, r)$; the first reason being that the ζ^{th} such query is answered by an accept message[6], and the second reason being that the emulation does not get into trouble (in Steps 2c and 2d). To see this, notice that all first $(\zeta - 1)$ queries having the ℓ^{th} distinct initiation-prefix satisfy exactly one of the following conditions:

1. They have non-terminating prover-sequences that are prefixes of one another (which implies that the cheating prover never has to forward such queries to the verifier twice).

2. They have terminating prover-sequences which should be rejected (recall that as long as the ζ^{th} terminating query has not been asked by $S_\sigma^{V_{h'}}(x)$, the cheating prover automatically rejects any terminating query).

Thus, the probability that when selecting (σ, h, ℓ, ζ) the cheating prover convinces $V(x, r)$ is at least:

$$\begin{aligned} & \Pr\left[\psi(\sigma, h', \ell, \zeta) \;\&\; \chi(\sigma, h', \ell)\right] \\ = \; & \Pr\left[\psi(\sigma, h', \ell, \zeta) \mid \chi(\sigma, h', \ell)\right] \cdot \Pr\left[\chi(\sigma, h', \ell)\right] \\ \geq \; & \Pr\left[\psi(\sigma, h', \ell, \zeta) \mid \chi(\sigma, h', \ell)\right] \cdot \Pr\left[(\sigma, h') \in \mathtt{AC} \;\&\; \chi(\sigma, h', \ell)\right] \end{aligned} \tag{5.1}$$

Note that if the ℓ^{th} distinct initiation-prefix is useful, and ζ is uniformly (and independently) selected in $\{1, ..., q_S(n)\}$, the probability that the ζ^{th} query corresponding to the ℓ^{th} distinct initiation–prefix is the first to be accepting is at least $1/q_S(n)$. Thus, Eq. (5.1) is lower bounded by:

$$\frac{\Pr\left[(\sigma, h') \in \mathtt{AC} \;\&\; \chi(\sigma, h', \ell)\right]}{q_S(n)} \tag{5.2}$$

Using the fact that, for every value of ℓ and σ, when h and r are uniformly selected the function h' is uniformly distributed, we infer that ℓ is distributed independently of (σ, h'). Thus, Eq. (5.2) is lower bounded by

$$\Pr[(\sigma, h') \in \mathtt{AC}] \cdot \frac{\Pr[\exists i \text{ s.t. } \chi(\sigma, h', i) \,|\, (\sigma, h') \in \mathtt{AC}]}{q_S(n)^2} \tag{5.3}$$

Thus, Eq. (5.3) is noticeable (i.e., at least $1/\text{poly}(n)$) provided that so is the value of

$$\Pr[\exists i \text{ s.t. } \chi(\sigma, h', i) \,|\, (\sigma, h') \in \mathtt{AC}]$$

The rest of the proof is devoted to establishing the last hypothesis. In fact we prove a much stronger statement:

Lemma 5.1.8 *For every $(\sigma, h') \in \mathtt{AC}$, the execution of $S_\sigma^{V_{h'}}(x)$ contains a useful initiation-prefix (that is, there exists an i s.t. $\chi(\sigma, h', i)$ holds).*

[6]We use the fact that $V(x, r)$ behaves exactly as $V_{h'}(x)$ behaves on queries for the ℓ^{th} distinct initiation-prefix.

5.1.3 Proof of Lemma 5.1.8 (existence of useful initiation prefixes)

The proof of Lemma 5.1.8 is by contradiction. We assume the existence of a pair $(\sigma, h') \in \mathtt{AC}$ so that all initiation-prefixes in the execution of $S_\sigma^{V_{h'}}(x)$ are unuseful and show that this implies that $S_\sigma^{V_{h'}}(x)$ made at least $n^{\Omega\left(\frac{\log n}{\log\log n}\right)} \gg \mathrm{poly}(n)$ queries which contradicts the assumption that it runs in polynomial-time.

The query–and–answer tree

Throughout the rest of the proof, we fix an arbitrary $(\sigma, h') \in \mathtt{AC}$ so that all initiation-prefixes in the execution of $S_\sigma^{V_{h'}}(x)$ are unuseful, and study this execution. A key vehicle in this study is the notion of a query–and–answer tree introduced in [36]. This is a rooted tree in which vertices are labeled with verifier messages and edges are labeled by prover's messages. The root is labeled by the empty string, and it has outgoing edges corresponding to the possible prover's messages initializing the first 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 in the order traversed. We stress that each vertex in the tree corresponds to a query actually made by the simulator.

Satisfied sub-path: A sub-path from one node in the tree to some of its descendants is said to satisfy session i if the sub-path contains edges (resp., vertices) for each of the messages sent by the prover (resp., verifier) in session i, and if the last such message (i.e., $\mathtt{v}_4^{(i)}$) indicates that the verifier accepts session i. A sub-path is called satisfied if it satisfies all sessions for which the first prover's message appears on the sub-path.

Forking sub-tree: For any i and $j \in \{2, 3, 4\}$, we say that a sub-tree (i, j)-forks if it contains two sub-paths, $\overline{p}$ and $\overline{r}$, having the same initiation-prefix, so that

1. $\overline{p}$ and $\overline{r}$ differ in the edge representing the j^{th} prover message for session i (i.e., a $\mathtt{p}_j^{(i)}$ message).
2. Each of $\overline{p}$ and $\overline{r}$ reaches a vertex representing the j^{th} verifier message (i.e., some $\mathtt{v}_j^{(i)}$).

In such a case, we may also say that the sub-tree (i, j)-forks on $\overline{p}$ (or on $\overline{r}$).

Good sub-tree: Consider an arbitrary sub-tree rooted at a vertex corresponding to the first message in some session so that this session is the first at some level of the recursive construction of the schedule. The full tree is indeed such a tree, but we will need to consider sub-trees which correspond to m sessions in the recursive schedule construction. We call such a sub-tree m-good if it contains a sub-path satisfying all m sessions for which the prover's first message appears in the sub-tree (all these first messages are in particular contained in the sub-path). Since $(\sigma, h') \in \mathtt{AC}$ it follows that the full tree contains a path from the root to a leaf representing an accepting transcript. The path from the root to this leaf thus satisfies all sessions (i.e., 1 through n) which implies that the full tree is n-good.

The structure of good sub-trees: The crux of the entire proof is given in the following lemma.

Lemma 5.1.9 *Let T be an m-good sub-tree. Then, at least one of the following holds:*

1. *T contains at least two different $\left(m - 2\cdot\frac{m}{\log m}\right)$-good sub-trees.*

2. *T contains at least $\frac{m}{\log m}$ different $\left(\frac{m}{\log m}\right)$-good sub-trees.*

Denote by $W(m)$ the size of an m-good sub-tree (where $W(m)$ stands for the work actually performed by the simulator on m concurrent sessions in our fixed scheduling). It follows (from Lemma 5.1.9) that any m-good sub-tree must satisfy

$$W(m) \geq \min\left\{\frac{m}{\log m}\cdot W\left(\frac{m}{\log m}\right), 2\cdot W\left(m - 2\cdot\frac{m}{\log m}\right)\right\} \tag{5.4}$$

Since Eq. (5.4) solves to $n^{\Omega\left(\frac{\log n}{\log\log n}\right)}$ (proof omitted), and since every vertex in the query–and–answer tree corresponds to a query actually made by the simulator, then the assumption that the simulator runs in poly(n)-time (and hence the tree is of poly(n) size) is contradicted. Thus, Lemma 5.1.8 follows from Lemma 5.1.9.

5.1.4 Proof of Lemma 5.1.9 (the structure of good sub-trees)

Considering the m sessions corresponding to an m-good sub-tree, we focus on the $m/\log m$ sessions dealt explicitly at this level of the recursive construction (i.e., the first $m/\log m$ sessions, which we denote by $\mathcal{F} \overset{\text{def}}{=} \{1, ..., m/\log m\}$).

Claim 5.1.10 *Let T be an m-good sub-tree. Then for any session $i \in \mathcal{F}$, there exists $j \in \{2,3\}$ such that the sub-tree (i,j)-forks.*

Proof: Consider some $i \in \mathcal{F}$, and let $\overline{p}_i$ be the first sub-path reached during the execution of $S_\sigma^{V_{h'}}(x)$ which satisfies session i (since the sub-tree is m-good such a sub-path must exist, and since $i \in \mathcal{F}$ every such sub-path must be contained in the sub-tree). Recall that by the contradiction assumption for the proof of Lemma 5.1.8, all initiation-prefixes in the execution of $S_\sigma^{V_{h'}}(x)$ are unuseful. In particular, the initiation-prefix corresponding to sub-path $\overline{p}_i$ is unuseful. Still, path $\overline{p}_i$ contains vertices for each prover message in session i and contains an accepting message by the verifier. So the only thing which may prevent the above initiation-prefix from being useful is having two (non-terminating) queries with the very same initiation-prefix (non-terminating) prover-sequences of the same length. Say that these sequences first differ at their j^{th} element, and note that $j \in \{2,3\}$ (as the prover-sequences are non-terminating and the first prover message, $\mathtt{p}_1^{(i)}$, is constant once the initiation-prefix is fixed). Also note that the two (non-terminating) queries were answered by the verifier (rather than refused), since the (modified) simulator avoids queries which will be refused. By associating a sub-path to each one of the above queries we obtain two different sub-paths (having the same initiation-prefix), that differ in some $\mathtt{p}_j^{(i)}$ edge and eventually reach a $\mathtt{v}_j^{(i)}$ vertex (for $j \in \{2,3\}$). The required (i,j)-forking follows. ■

Claim 5.1.11 *If there exists a session $i \in \mathcal{F}$ such that the sub-tree $(i,3)$-forks, then the sub-tree contains two different $\left(m-2\cdot\frac{m}{\log m}\right)$-good sub-trees.*

Proof: Let $i \in \mathcal{F}$ such that the sub-tree $(i,3)$-forks. That is, there exist two sub-paths, $\overline{p}_i$ and $\overline{r}_i$, that differ in the edge representing a $\mathtt{p}_3^{(i)}$ message, and that eventually reach some $\mathtt{v}_3^{(i)}$ vertex.

In particular, paths $\overline{p}_i$ and $\overline{r}_i$ split from each other before the edge which corresponds to the $\mathtt{p}_3^{(i)}$ message occurs along these paths (as otherwise the $\mathtt{p}_3^{(i)}$ edge would have been identical in both paths). By nature of the fixed scheduling, the vertex in which the above splitting occurs precedes the first message of all (nested) sessions in the second recursive construction (that is, sessions $2\cdot\frac{m}{\log m}+1,\ldots,m$). It follows that both $\overline{p}_i$ and $\overline{r}_i$ contain the first and last messages of each of these (nested) sessions (as they both reach a $\mathtt{v}_3^{(i)}$ vertex). Therefore, by definition of V_h, all these sessions must be satisfied by both these paths (or else V_h would have not answered with a $\mathtt{v}_3^{(i)}$ message but rather with a "refuse" symbol). Consider now the corresponding sub-paths of $\overline{p}_i$ and $\overline{r}_i$ which begin at edge $\mathtt{p}_1^{(k)}$ where $k = 2\cdot\frac{m}{\log m}+1$ (i.e., $\mathtt{p}_1^{(k)}$ is the edge which represents the first message of the first session in the second recursive construction). Each of these new sub-paths is contained in a disjoint sub-tree corresponding to the recursive construction, and satisfies all of its $(m-2\cdot\frac{m}{\log m})$ sessions. It follows that the (original) sub-tree contains two different $(m-2\cdot\frac{m}{\log m})$-good sub-trees and the claim follows. ■

Claim 5.1.12 *If for every session $i \in \mathcal{F}$ the sub-tree $(i,2)$-forks, then the sub-tree contains at least $|\mathcal{F}| = \frac{m}{\log m}$ different $(\frac{m}{\log m})$-good sub-trees.*

In the proof of Claim 5.1.12 we use a special property of $(i,2)$-forking: The only location in which the splitting of path $\overline{r}_i$ from path $\overline{p}_i$ may occur, is a vertex which represents a $\mathtt{v}_1^{(i)}$ message. Any splitting which has occured at a vertex which precedes the $\mathtt{v}_1^{(i)}$ vertex would have caused the initiation-prefixes of (session i along) paths $\overline{p}_i$ and $\overline{r}_i$ to be different (by virtue of the definition of V_h, and since all vertices preceding $\mathtt{v}_1^{(i)}$ are part of the initiation-prefix of session i).

Proof: Since for all sessions $i \in \mathcal{F}$ the sub-tree $(i,2)$-forks, then for every such i there exist two sub-paths, $\overline{p}_i$ and $\overline{r}_i$, that split from each other in a $\mathtt{v}_1^{(i)}$ vertex and that eventually reach some $\mathtt{v}_2^{(i)}$ vertex. Similarly to the proof of Claim 5.1.11, we can claim that each one of the above paths contains a "special" sub-path (denoted $\overline{\overline{p}}_i$ and $\overline{\overline{r}}_i$ respectively), that starts at a $\mathtt{v}_1^{(i)}$ vertex, ends at a $\mathtt{v}_2^{(i)}$ vertex, and satisfies all $\frac{m}{\log m}$ sessions in the first recursive construction (that is, sessions $\frac{m}{\log m}+1,\ldots,2\cdot\frac{m}{\log m}$). Note that paths $\overline{\overline{p}}_i$ and $\overline{\overline{r}}_i$ are completely disjoint. Let i_1, i_2 be two different sesions in $\mathcal{F}$ (without loss of generality $i_1 < i_2$), and let $\overline{\overline{p}}_{i_1}, \overline{\overline{r}}_{i_1}, \overline{\overline{p}}_{i_2}, \overline{\overline{r}}_{i_2}$ be their corresponding "special" sub-paths. The key point is that for every i_1, i_2 as above, it cannot be the case that both "special" sub-paths corresponding to session i_2 are contained in the sub-paths corresponding to session i_1 (to justify this, we use the fact that $\overline{\overline{p}}_{i_2}$ and $\overline{\overline{r}}_{i_2}$ split from each other in a $\mathtt{v}_1^{(i_2)}$ vertex and that for every $i \in \{i_1, i_2\}$, paths $\overline{\overline{p}}_i$ and $\overline{\overline{r}}_i$ are disjoint).

This enables us to associate a distinct $(\frac{m}{\log m})$-good sub-tree to every $i \in \mathcal{F}$ (i.e., which either corresponds to path $\overline{\overline{p}}_i$, or to path $\overline{\overline{r}}_i$). Which in particular means that the tree contains at least $|\mathcal{F}|$ different $(\frac{m}{\log m})$-good sub-trees. ■

We are finally ready to analyze the structure of the sub-tree T. Since for every $i \in \mathcal{F}$ there must exist $j \in \{2,3\}$ such that the sub-tree (i,j)-forks (Claim 5.1.10), then it must be the case that either T contains two distinct $(m-2\cdot\frac{m}{\log m})$-good sub-trees (Claim 5.1.11), or T contains at least $\frac{m}{\log m}$ distinct $(\frac{m}{\log m})$-good sub-trees (Claim 5.1.12). This completes the proof of Lemma 5.1.9 which in turn implies Lemmata 5.1.8 and 5.1.5. The proof of Theorem 5.1 is complete.

5.2 Extending the proof for the Richardson-Kilian protocol

Recall that the Richardson-Kilian protocol [42] consists of two stages. We will treat the first stage of the RK protocol (which consists of 6 rounds) as if it were the first 6 rounds of any 7-round protocol, and the second stage (which consists of a 3-round WI proof) as if it were the remaining 7th message. An important property which is satisfied by the RK protocol is that the coin tosses used by the verifier in the second stage are independent of the coins used by the verifier in the first stage. We can therefore define and take advantage of two (different) types of initiation-prefixes. A first-stage initiation prefix and a second-stage initiation prefix (which is well defined only given the first one). These initiation-prefixes will determine the coin tosses to be used by V_h in each corresponding stage of the protocol (analogously to the proof of Theorem 5.1).

The cheating prover will pick a random index for each of the above types of initiation-prefixes (corresponding to ℓ and ζ in the proof of Theorem 3.1). The first index (i.e., ℓ) is treated exactly as in the proof of Theorem 3.1, whereas the second index (i.e., ζ) will determine which of the WI session corresponding to the second-phase initiation-prefix (and which also correspond to the very same ℓ^{th} first-phase initiation-prefix) will be actually executed between the cheating prover and the verifier. As long as the ζ^{th} second-stage initiation prefix will not be encountered, the cheating prover will be able to impersonate V_h while always deciding correctly whether to reject or to accept the corresponding "dummy" WI session (as the second-stage initiation-prefix completely determines the coins to be used by V_h in the second stage of the protocol). As in the proof of Theorem 3.1, the probability that the ζ^{th} second-stage initiation prefix (that corresponds to the ℓ^{th} first-phase initiation-prefix) will make the verifier accept is non-negligible. The existence of a useful pair of initiation-prefixes (i.e., ℓ and ζ) is proved essentially in the same way as in the proof of Theorem 3.1.

Chapter 6

Constant-Round $\mathcal{ZK}$ proofs for $\mathcal{NP}$ with a Simpler Proof of Security

In this chapter we consider the task of constructing a constant-round $\mathcal{ZK}$ proof system for all languages in $\mathcal{NP}$. This problem has been previously addressed by Goldreich and Kahan [23], who constructed a 5-round proof system assuming the existence of a collection of claw-free functions. We show how to use a variant of the $c\mathcal{ZK}$ protocol presented in Chapter 4 in order to construct a 7-round $\mathcal{ZK}$ proof system for $\mathcal{NP}$. The advantage of the new proof system over the one of [23] is that it admits a considerably simpler proof of security. Consider the following protocol, which is a variant of Construction 4.3.2 in which the preamble has only one iteration (rather than k iterations as in Construction 4.3.2).

A 7-round $\mathcal{ZK}$ Proof System for $\mathcal{NP}$

Common Input: A directed graph $G = (V, E)$ with $n \stackrel{\text{def}}{=} |V|$.

Auxiliary Input to Prover: A directed Hamiltonian Cycle, $C \subset E$, in G.

Additional parameter: A super-logarithmic function $k(n)$.

Stage 1: Commitment to challenge $\sigma \in \{0,1\}^n$ (independent of common input):

- $P \rightarrow V$ (P1): Send first message for perfectly hiding commitment scheme.
- $V \rightarrow P$ (V1): Commit to random $\sigma, \{\sigma_i^0\}_{i=1}^k, \{\sigma_i^1\}_{i=1}^k$ s.t. $\sigma_i^0 \oplus \sigma_i^1 = \sigma$ for all i.
- $P \rightarrow V$ (P2): Send a random k-bit string $r = r_1, \ldots, r_k$.
- $V \rightarrow P$ (V2): Decommit to $\sigma_1^{r_1}, \ldots, \sigma_k^{r_k}$.

Stage 2: Engage in Blum's 3-round Hamiltonicity protocol using σ as challenge:

- $P \rightarrow V$ (`p1`): Use C to produce first prover message of Hamiltonicity protocol.
- $V \rightarrow P$ (`v1`): Decommit to σ and to $\{\sigma_i^{1-r_i}\}_{i=1}^k$.
- $P \rightarrow V$ (`p2`): Answer σ with second prover message of Hamiltonicity protocol.

Figure 6.1: A 7-round $\mathcal{ZK}$ proof for $\mathcal{NP}$.

Using the same arguments as in Section 4.3.2, it can be seen that the resulting protocol is both complete and sound. In particular, the construction above is an interactive proof system for HC. The following theorem states that it is also $\mathcal{ZK}$.

Theorem 6.1 (Constant-round $\mathcal{ZK}$ proof for $\mathcal{NP}$) *Assume the existence of perfectly-hiding commitment schemes. Then, the protocol described in Figure 6.1 is a $\mathcal{ZK}$ proof system for HC.*

6.1 Zero-Knowledge

In order to demonstrate the $\mathcal{ZK}$ property of the protocol, we will show that there exists a "universal" black-box simulator, S, so that for every $G = (V, E) \in HC$ and adversary verifier V^* that runs in polynomial time (in $n = |V|$), $S(G)$ runs in expected time poly(n), and satisfies that the ensemble $\{\text{view}^P_{V^*}(G)\}_{G \in HC}$ is computationally indistinguishable from the ensemble $\{S^{V^*}(G)\}_{G \in HC}$.

6.1.1 The Simulator

On input $G = (V, E)$ with $n = |V|$, the simulator S starts by selecting and fixing a random tape $s \in \{0,1\}^{\text{poly}(n)}$ for V^*. It then proceeds by exploring various prefixes of possible interactions between P and V^*. This is done while having only black-box access to V^*.

```
Step (S1): Randomly generate a (P1) message and obtain $(\text{V1}) = V^*(G, (\text{P1}); s)$.
Step (S2): Randomly generate a (P2) message and obtain $(\text{V2}) = V^*(G, (\text{P1}), (\text{P2}); s)$.
    1. If $(\text{V2}) \neq \texttt{ABORT}$, proceed to Step (S3).
    2. If $(\text{V2}) = \texttt{ABORT}$, output $\langle(\text{P1}), (\text{V1}), \texttt{ABORT}\rangle$ and stop.
Step (S3): For $j = 1, 2, \ldots$
    1. Randomly generate $(\text{P2})_j$ and obtain $(\text{V2})_j = V^*(G, (\text{P1}), (\text{P2})_j; s)$.
    2. If $(\text{V2})_j \neq \texttt{ABORT}$, proceed to Step (S4).
    3. If $(\text{V2})_j = \texttt{ABORT}$ continue.
  end(for)
Step (S4): Let $(\text{P2}) = r_1, \ldots, r_k$ be the prover message generated in Step (S2) of the simulation
  and let $(\text{P2})_j = r'_1 \ldots, r'_k$ be the last prover message generated in Step (S3):
    1. If $(\text{P2}) = (\text{P2})_j$, output $\perp$ and stop.
    2. If $(\text{P2}) \neq (\text{P2})_j$, there exists $i \in \{1, \ldots, k\}$ so that $r_i \neq r'_i$. Let $\sigma = \sigma_i^{r_i} \oplus \sigma_i^{r'_i}$.
    3. Use $\sigma$ to produce an accepting transcript $(\texttt{p1}), (\texttt{v1}), (\texttt{p2})$ for $G \in HC$ (as in Page 74).
    4. Output $\langle(\text{P1}), (\text{V1}), (\text{P2}), (\text{V2}), (\texttt{p1}), (\texttt{v1}), (\texttt{p2})\rangle$ and stop.
```

Figure 6.2: The black-box simulator S.

Notice that simulator always picks the $(\text{P2})_j$ messages uniformly at random. Since the length of the (P2)'s is super-logarithmic, the probability that *any* two (P1) messages sent during the simulation are equal is negligible (see Section 6.1.3 for further details). We note that in previous simulators (cf. [23]), the values of the (Pj) messages depended on the values revealed by the verifier in the corresponding (V2) answers, and were *not* chosen uniformly and independently each time. This is the main reason in the complication of previous analysises of the simulator's output distribution.

6.1.2 The simulator's running time

For any $G \in HC$, for any choice of s and of (P1), let $\zeta = \zeta(G, (\text{P1}), s)$ denote the probability that the verifier V^* does not send an `ABORT` message in message (V2). The probability ζ is taken over the random choices of message (P2). (Or, in other words, over the coin-tosses used by the simulator to generate (P2) during the simulation (both in Steps (S2) and (S3).1).)

Using this notation, the simulator proceeds to Step (S3) with probability ζ and is then expected to reach Step (S4) after repeatedly rewinding in Step (S3).1 for $1/\zeta$ times (since the probability of successfully rewinding in each one of the rewinds is precisely ζ, independently of other rewinds). For $i \in \{1,2,3,4\}$, let $p_i(\cdot)$ be a polynomial bound on the work required in order to perform Step (Si) of the simulation (where in Step (S3), the value $p_3(\cdot)$ represents the work of a single execution of Step (S3).1). The expected running time of the simulator is then:

$$\begin{aligned} p_1(n) + (1-\zeta)\cdot p_2(n) + \zeta \cdot \left(p_2(n) + \frac{1}{\zeta}\cdot p_3(n) + p_4(n)\right) &\leq p_1(n) + p_2(n) + p_3(n) + p_4(n) \\ &= \text{poly}(n) \end{aligned}$$

Since the above holds for any choice of s and (P1), then it is also true for randomly chosen s and (P1) (and offcourse for any $G \in HC$). We thus have,

Proposition 6.1.1 *The simulator S runs in expected polynomial-time (in $n = |V|$).*

6.1.3 The simulator's output distribution

We now turn to show that for every $G \in HC$, the simulator's output distribution is computationally indistinguishable from V^*'s view of interactions with the honest prover P. Specifically,

Proposition 6.1.2 *Suppose that the commitment used in Step* (`p1`) *is hiding. Then, the ensemble $\{S^{V^*}(G)\}_{G\in HC}$ is computationally indistinguishable from the ensemble $\{\text{view}^P_{V^*}(G)\}_{G\in HC}$.*

Proof: As a hybrid experiment, consider what happens to the output distribution of the simulator S if we (slightly) modify its simulation strategy in the following way: Suppose that on input $G = (V, E) \in HC$, the simulator S obtains a directed Hamiltonian Cycle $C \subset E$ in G (as auxiliary input) and uses it in order to produce real prover messages whenever it reaches the second stage of the protocol. Specifically, when it reaches the second stage, the hybrid simulator checks whether the original simulator S should output $\perp$ (in which case it also does). If S does not have to output $\perp$, the hybrid simulator follows the prescribed prover strategy and generates prover messages for the corresponding second stage (by using the cycle it possesses rather than its prior knowledge of σ). We claim that the ensemble consisting of the resulting output (which we denote by $\widehat{S}^{V^*}(G, C)$) is computationally indistinguishable from $\{S^{V^*}(G)\}_{G\in HC}$. Namely,

Claim 6.1.3 *Suppose that the commitment used in Step* (`p1`) *is hiding. Then, the ensemble $\{S^{V^*}(G)\}_{G\in HC}$ is computationally indistinguishable from the ensemble $\{\widehat{S}^{V^*}(G, C)\}_{G\in HC}$.*

Proof Sketch: The claim is proved by reducing the proof to the indistinguishability of Blum's simulator's output (that is, if the output of Blum's simulator [6] is computationally indistinguishable from the view of real executions of the basic Hamiltonicity proof system, then $\{S^{V^*}(G)\}_{G\in HC}$ and $\{\widehat{S}^{V^*}(G, C)\}_{G\in HC}$ are indistinguishable as well). The latter is proved to hold based on the computational-hiding property of the commitment scheme that is used by the prover in Step ($\widehat{\texttt{p1}}$) of Construction 4.3.1 (see [6, 22] for further details). Here we also use the extra property that the output of Blum's simulator is indistinguishable from true interactions even if the distinguisher has a-priori knowledge of a Hamiltonian Cycle C in G. ■

We next consider what happens to the output distribution of the hybrid simulator $\widehat{S}$ if we assume that it does not output $\perp$. It turns out that in such a case, the resulting output distribution is *identical* to the distribution of $\{\mathrm{view}^P_{V^*}(G)\}_{G\in HC}$. Namely,

Claim 6.1.4 *The ensemble $\{\widehat{S}^{V^*}(G,C)\}_{G\in HC}$ conditioned on it not being $\perp$, is identically distributed to the ensemble $\{\mathrm{view}^P_{V^*}(G)\}_{G\in HC}$.*

Proof: Notice that the first stage messages that appear in the output of the "original" simulator (that is, S) are identically distributed to the first stage messages that are produced by an honest prover P (since they are uniformly and independently chosen). Since the first stage messages that appear in the output of the "modified" simulator (that is, $\widehat{S}$) are identical to the ones appearing in the output of S, we infer that they are identically distributed to the first stage messages that are produced by an honest prover P. Using the fact that the second stage messages that appear in the output of the "modified" simulator are (by definition) identically distributed to the second stage messages that are produced by an honest prover P, we infer that the ensemble $\{\widehat{S}^{V^*}(G,C)\}_{G\in HC}$ is identically distributed to $\{\mathrm{view}^P_{V^*}(G)\}_{G\in HC}$. ■

As we will show in Proposition 6.1.6 below, $\widehat{S}$ outputs $\perp$ only with negligible probability. In particular, the ensemble $\{\widehat{S}^{V^*}(G,C)\}_{G\in HC}$ is computationally indistinguishable from (and in fact statistically close to) the ensemble $\{\widehat{S}^{V^*}(G,C)\}_{G\in HC}$, conditioned on it not being $\perp$. Namely,

Claim 6.1.5 *The ensemble $\{\widehat{S}^{V^*}(G,C)\}_{G\in HC}$ is computationally indistinguishable from the ensemble $\{\widehat{S}^{V^*}(G,C)\}_{G\in HC}$ conditioned on it not being $\perp$.*

As we have mentioned above, Claim 6.1.5 follows by establishing the following claim.

Claim 6.1.6 *For any $G=(V,E)\in HC$, the probability that $\widehat{S}^{V^*}(G,C)=\perp$ is negligible (in $|V|$).*

Proof: Let $G\in HC$ with $n=|V|$. We will show that for any choice of $s\in\{0,1\}^{\mathrm{poly}(n)}$ and (P1) the probability of $\widehat{S}$ outputting $\perp$ (over random choices of (P2) $= r\in\{0,1\}^k$) is precisely $1/2^k$. Since k is super-logarithmic it will immediately follow that the probability that $\widehat{S}^{V^*}(G,C)=\perp$ is negligible. Let $\widetilde{V}^*=\widetilde{V}^*((\mathrm{P1}),s)$ denote the "residual" strategy of V^* when $\langle(\mathrm{P1}),s\rangle$ are fixed (i.e., $\widetilde{V}^*(G,r)\stackrel{\mathrm{def}}{=}V^*(G,(P1),r;s)$), and let ζ be as in Section 6.1.2. We then have:

$$\begin{aligned}
\Pr_r\left[\widehat{S}^{\widetilde{V}^*}(G,C)=\perp\right] &= \Pr_r\left[\widehat{S}^{\widetilde{V}^*}(G,C)=\perp \;\middle|\; \widehat{S}\text{ reaches (S3)}\right]\cdot\Pr_r\left[\widehat{S}\text{ reaches (S3)}\right]\\
&= \Pr_r\left[\widehat{S}^{\widetilde{V}^*}(G,C)=\perp \;\middle|\; \widehat{S}\text{ reaches (S3)}\right]\cdot\zeta\\
&= \Pr_r\left[(\mathrm{P2})=(\mathrm{P2})_j\right]\cdot\zeta \qquad (6.1)
\end{aligned}$$

Now, since (P2) and $(\mathrm{P2})_j$ are uniformly and independently chosen in $\{0,1\}^k$, and since the number of $r\in\{0,1\}^k$ for which $\widetilde{V}^*(G,r)$ is not equal to ABORT is precisely $2^k\cdot\zeta$, then it holds that $\Pr[(\mathrm{P2})=(\mathrm{P2})_j]=1/(2^k\cdot\zeta)$. Using Eq. (6.1) we infer that:

$$\Pr_r\left[\widehat{S}^{\widetilde{V}^*}(G)=\perp\right] = \frac{1}{2^k\cdot\zeta}\cdot\zeta = \frac{1}{2^k}$$

as required. ■

It can be seen that Claims 6.1.3, 6.1.4 and 6.1.5 imply the correctness of Proposition 6.1.2. ■

Chapter 7

Conclusions and Open Problems

7.1 Avoiding the Lower Bounds of Chapter 3

The lower bound presented in Chapter 3 of this thesis draws severe limitations on the ability of black-box simulators to cope with the standard concurrent zero-knowledge setting. This suggests two main directions for further research in the area.

Alternative models: One first possibility that comes into mind would be 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 [15, 16]. Damgård uses a common reference string [13], and Canetti et.al. use a public registry file [8].

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* [17, 19], which is preserved under concurrent composition.

Beyond black-box simulation: 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 Chapter 3, 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 [30] 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 [24]). 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.

As mentioned before, 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. It should be noted, however, that Barak's new techniques are still not known to yield a satisfactory solution to the problem of "full-fledged" concurrent composition (even when allowing arbitrarily many rounds in the protocol).

7.2 Open problems

The main conclusion of this work is that the round-complexity of black-box $c\mathcal{ZK}$ is essentially logarithmic. Specifically, by combining Theorem 3.1 with Theorem 4.1, we have:

Corollary *The round-complexity of black-box concurrent zero-knowledge is $\tilde{\Theta}(\log n)$ rounds.*[1]

Still, in light of Barak's recent result [2], constant-round $c\mathcal{ZK}$ protocols (with non black-box simulators) do not seem out of reach. A natural open question is whether there exists a constant-round (non black-box) $c\mathcal{ZK}$ protocol for all languages in $\mathcal{NP}$.

Open Problem 1 *Is there a $c\mathcal{ZK}$ protocol for $\mathcal{NP}$ with a constant number of rounds?*

As a first step, it would be interesting to determine whether non black-box simulation techniques can at all improve over black-box simulation techniques in the context of concurrent composition.

Open Problem 2 *Is there a $c\mathcal{ZK}$ protocol for $\mathcal{NP}$ with a sublogarithmic number of rounds?*

It would be in fact interesting to see whether Barak's non black-box simulation techniques can at all be extended to handle unbounded concurrency (regardless of the number of rounds).

Regarding $c\mathcal{ZK}$ without aborts. Here the situation is not resolved as well. In particular, assuming that the verifier never aborts is not known to enable any improvement in the round-complexity of $c\mathcal{ZK}$ protocols. On the other hand, the best lower bound to date shows that 7 round are not sufficient for black-box simulation. It would be interesting to close the gap between the currently known upper and lower bounds (presented in Chapter 4 and Chapter 5 respectively).

Open Problem 3 *Determine the exact round-complexity of $c\mathcal{ZK}$ without aborts.*

The latter question mainly refers to black-box simulation, though it is also interesting (and open) in the context of non black-box simulation.

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

Chapter 8

Appendix

8.1 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 $i_1, \ldots, i_m$.)

8.2 Solving the Recursion

Claim 8.2.1 *Suppose that Eq. (3.6) holds. Then for all sufficiently large n, $W(n^2) > n^c$.*

Proof: By applying Eq. (3.6) iteratively $\log_k(n-1)$ times, we get:

$$\begin{aligned} W(n^2) &\geq \left(k^{c+1}\right)^{\log_k(n-1)} \cdot W(n) \\ &\geq \left(k^{c+1}\right)^{\log_k(n-1)} \cdot 1 \\ &= (n-1)^{c+1} \\ &> n^c \qquad (8.1) \end{aligned}$$

where Eq. (8.1) holds for all sufficiently large n. ■

Bibliography

[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] B. Barak and Y. Lindell. Strict Polynomial-Time in Simulation and Extraction. In *34th STOC*, pages 484–493, 2002.

[4] M. Bellare, O. Goldreich. On Defining Proofs of Knowledge. In *CRYPTO92*. Springer LNCS 0740. Pages 390-420, 1992.

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

[6] M. Blum. How to prove a Theorem So No One Else Can Claim It. *Proc. of the International Congress of Mathematicians,* Berekeley, California, USA, pages 1444-1451, 1986.

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

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

[9] 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.

[10] R. Canetti, J. Kilian, E. Petrank and A. Rosen. Black-Box Concurrent Zero-Knowledge Requires (almost) Logarithmically many Rounds. In *SIAM Jour. on Computing,* , Vol. 32, No. 1, pages 1–47, 2002.

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

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

[13] I. Damgard. Eficient Concurrent Zero-Knowledge in the Auxiliary String Model. In *Euro-Crypt2000*, LNCS 1807, pages 418–430, 2000.

[14] I. Damgard, T. Pedersen and B. Pfitzmann. On the Existence of Statistically Hiding Bit Commitment Schemes and Fail-Stop Signatures. In *Crypto93*, Springer-Verlag LNCS Vol. 773, pages 250–265, 1993.

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

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

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

[18] U. Feige, A. Fiat and A. Shamir. Zero Knowledge Proofs of Identity. *Journal of Cryptology*, Vol. 1(2), pages 77-94,1988.

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

[20] A. Fiat and A. Shamir. How to Prove Yourself: Practical Solution to Identification and Signature Problems. In *Crypto86*, Springer LNCS 1233, pages 186-189, 1987.

[21] O. Goldreich. Concurrent Zero-Knowledge with Timing – Revisited. In *34th STOC*, pages 332-340, 2002.

[22] O. Goldreich. *Foundation of Cryptography – Basic Tools*. Cambridge University Press, 2001.

[23] 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.

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

[25] 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.

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

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

[28] 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.

[29] S. Goldwasser, S. Micali and R.L. Rivest. A Digital Signature Scheme Secure Against Adaptive Chosen Message Attacks. *SIAM J. Comput.*, Vol. 17, No. 2, pp. 281–308, 1988.

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

[31] D. Harnik, M. Naor, O. Reingold and A. Rosen. Completeness in Two-Party Secure Computation Revisited. Unpublished manuscript.

[32] 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.

[33] J. Kilian. A Note on Efficient Zero-Knowledge Proofs and Arguments. In *24th STOC*, pages 723–732, 1992.

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

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

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

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

[38] M. Naor, R. Ostrovsky, R. Venkatesan and M. Yung. Zero-Knowledge Arguments for NP can be Based on General Assumptions. *Jour. of Cryptology*, Vol. 11, pages 87–108, 1998.

[39] M. Naor and M. Yung. Universal One-Way Hash Functions and their Cryptographic Applications. In *21st STOC*, pages 33–43, 1989.

[40] R. Pass and A. Rosen. Bounded-Concurrent Secure Two-Party Computation in a Constant Number of Rounds. To appear in *FOCS 2003*.

[41] M. Prabhakaran, A. Rosen and A. Sahai. *Concurrent Zero-Knowledge with Logarithmic Round Complexity.* Proceedings of the 43rd annual IEEE symposium on Foundations of Computer Science (FOCS 2002), 2002.

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

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

[44] C.P. Schnorr. Efficient Signature Generation by Smart Cards. *Jour. of Cryptology* Vol. 4 (3), pages 161-174, 1991.

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