Concurrent Zero Knowledge Proofs with Logarithmic Round-Complexity

Manoj Prabhakaran* Princeton University

Amit Sahai† Princeton University

May 6, 2002

Abstract

We consider the problem of constructing Concurrent Zero Knowledge Proofs [6], in which the fascinating and useful "zero knowledge" property is guaranteed even in situations where multiple concurrent proof sessions are executed with many colluding dishonest verifiers. Canetti et al. [3] show that black-box concurrent zero knowledge proofs for non-trivial languages require $\tilde{\Omega}(\log k)$ rounds where k is the security parameter. Till now the best known upper bound on the number of rounds for NP languages was $\omega(\log^2 k)$, due to Kilian and Petrank [16]. We establish an upper bound of $\omega(\log k)$ on the number of rounds for NP languages, thereby closing the gap between the upper and lower bounds, up to a $\omega(\log \log k)$ factor.

*Email: `mp@cs.princeton.edu`.

†Email: `sahai@cs.princeton.edu`.

1 Introduction

Zero Knowledge proofs, introduced in [14], are fascinating and important cryptographic primitives. Informally, a zero knowledge proof is a protocol between a prover and a verifier which yields nothing to the verifier beyond the validity of the assertion proved. However, more recently further refined notions of Zero Knowledge have been introduced to handle scenarios more complicated than the one considered in the original definition.

Zero knowledge proofs in settings involving many asynchronous protocol executions was first considered by Feige [8], who introduced relaxations of the zero knowledge property which were provably preserved under asynchronous protocol composition. Concurrent Zero Knowledge, introduced in [6], considers a setting in which the prover holds concurrent proof sessions with polynomially many colluding dishonest verifiers.

To show that a protocol is zero knowledge one must show that for every efficient verifier, everything it can do after taking part in the protocol can be simulated by an efficient *simulator* without taking part in the protocol. Black-box zero knowledge is a restricted version of zero knowledge in which we require that for all verifiers there exist a single simulator which has only black-box or oracle access to the verifier.

Previous Results A large body of fascinating research has dealt with concurrent zero knowledge and related concepts[6, 7, 12, 3, 17, 21, 13, 20, 16, 18, 2, 10, 4, 5, 1]. We review and focus on prior work dealing with black-box concurrent zero knowledge.

It turns out that black-box concurrent zero knowledge is indeed a stronger notion than black-box zero knowledge. Many of the known (black-box) zero knowledge proofs are not black-box concurrent zero knowledge. This is so because lower bounds have been established for the number of rounds in a black-box concurrent zero knowledge proof. Following a sequence of prior work [13, 17, 21], Canetti et al [3] showed that a black-box concurrent zero knowledge proof protocol to prove membership in a language not in BPP requires at least $\Omega\left(\frac{\log k}{\log\log k}\right)$ rounds, where k is a security parameter (number of concurrent sessions is polynomial in k).

Subsequently Kilian and Petrank [16], building on the work of [20], showed that under a standard complexity assumption, there exists a black-box concurrent zero knowledge proof system for proving membership in any NP language, with round-complexity $\omega(\log^2 k)$. There the question was raised whether the same proof system when reduced to $\omega(\log k)$ rounds remains concurrent zero knowledge. In this work we answer that question in the affirmative.

In a recent breakthrough Barak [1] has given the first non-black-box zero-knowledge proof system under standard complexity assumptions. He also presented a (non-black-box) proof system with only a constant number of rounds, which remains zero knowledge for a pre-determined (polynomial in k) number of concurrent sessions. The communication in the protocol is proportional to this pre-determined bound on the number of concurrent sessions. Compared to this scheme, our protocol requires $\omega(\log k)$ rounds, but can handle any polynomial number of concurrent session, and the communication in the protocol is independent of the actual number of sessions.

Our Results We give a black-box concurrent zero knowledge proof for languages in NP, with $\omega(\log k)$ rounds. This becomes the most efficient concurrent zero knowledge proof system which can handle any polynomial number of concurrent sessions. This round-complexity matches the currently known lower-bound for black-box proofs within a factor of $\omega(\log\log k)$. Also, as in [16] this implies a "resettable zero knowledge proof" system of similar round-complexity.

We use the same protocol (but with a smaller number of rounds) and the same simulator as in [16, 20], but provide a better analysis for that simulator. Apart from providing the tighter analysis, we present a novel counting argument, developing a suggestion by Kilian [15], to show that the simulation is indistinguishable from what the verifier sees in the proof. We also provide an alternate description of the simulator introduced in [20], and show that our analysis is asymptotically tight for that simulator.

As described in Section 2 the only quantity to be re-analyzed to establish our improvement is the probability that the simulator aborts in the middle of the simulation. As a warm-up, and as was done in [16, 20] we first analyze the simulator's abort probability, assuming that the adversarial verifier uses a "static scheduling strategy." This means that for all points in (protocol) time the verifier has to *a priori* decide the session from which the message is sent. It cannot adaptively change this schedule during the simulation. But it gets to decide at each point whether the message it sends over is well-formed or not. [16] shows that the probability of the simulator aborting, for the static case is $2^{-\Omega(m/\log k)} + \epsilon(k)$, for some negligible function ϵ. We improve on this to establish a probability bound of $2^{-(m-O(\log k))} + \epsilon(k)$. Thus with our new analysis, choosing $m = \omega(\log k)$ makes this probability negligible, where as previously $m = \omega(\log^2 k)$ was required.

As with [16, 20], our analysis for the static case can be carried over to imply concurrent zero knowledge for general adversaries. Previous analyses of concurrent zero knowledge in the general setting, however, have typically relied on delicate conditioning arguments, notoriously prone to subtle errors.

We develop and present an alternative to the conditioning-based analysis [1] of [20, 16], based on a suggestion to refine the analysis of [20, 16] due to Kilian [15]. The central idea is to essentially prove that for every set of random coins in the simulation which allows the adversary to make the simulator abort, there is a superpolynomially larger set of random coins all of which allow the simulator to succeed without aborting.

We present this argument using an analogy to decks of cards – we view the simulator's random coins as cards arranged in a sequence (a deck). We show that given any deck which allows the adversary to cause the simulator to abort, we can "shuffle" this deck to produce many new sets of random coins in which the simulator will provably succeed in the simulation without aborting. We then argue that each of the decks produced by our shuffling procedure is unique, by exhibiting a deterministic "unshuffling" procedure that allows us to reconstruct the original deck of cards in which the adversary causes the simulator to abort.

2 Preliminaries

Here we review the basic cryptographic concepts and assumptions we shall need, including black-box concurrent zero knowledge, commitment schemes and witness indistinguishability. Note, however, that as we build on the analysis of [20, 16, 18], we will be able to appeal to many of cryptographic arguments of [20, 16, 18] as a black box. Thus, even a reader unfamiliar with the details of cryptographic definitions should be able to follow our analysis.

Formal description and motivation of concepts described below are available in standard references on cryptography, and in particular in Goldreich [11]. [20, 16, 18] have the details and pointers specific to concurrent zero knowledge.

Zero Knowledge Proofs An interactive proof (P, V) for a language L is a protocol between a computationally unbounded prover P and a probabilistic polynomial time verifier V such that there exists a negligible

[1]Note that we do use some conditioning in our analysis: Typically proofs of concurrent zero knowledge have been given in two parts – first a (fairly standard cryptographic) proof that the simulator's output is indistinguishable from the real-world execution conditioned on the event that the simulator does not abort; then a (usually much more difficult) proof that the probability that the simulator aborts is negligible. Our proof will also have this structure, and thus we too make use of conditioning on the event that the simulator does not abort. We stress however, that in previous analyses [20, 16], the (difficult) proof that the simulator aborts with negligible probability has made use of delicate chains of conditionings. Our proof, however, does not.

function $\epsilon(k)$ such that [2] for every common input x (of length polynomial in k) (i) (completeness) if $x \in L$ $\mathbf{Pr}[(P, V)(x)] \geq 1 - \epsilon(k)$ and (ii) (soundness) if $x \notin L$, for every prover P^*, $\mathbf{Pr}[(P^*, V)(x)] \leq \epsilon(k)$.

An interactive proof system is said to be black-box (computational) zero knowledge if there is a probabilistic polynomial time oracle machine $\mathcal{S}$ such that for any probabilistic polynomial time verifier V^* and for all $x \in L$ the distribution of the output produced by $\mathcal{S}^{V^*}$ on input x is computationally indistinguishable from the view of the verifier at the end of the interaction $(P, V)(x)$.

In concurrent zero knowledge proofs the prover is involved in polynomially many (in k) sessions. We consider the verifiers of all the sessions to be co-ordinated by an adversary. It is up to the adversarial verifier to decide which messages it will send to the prover and when. But the prover should work for each session which behaves correctly as specified by the protocol, irrespective of messages in the other sessions, or their relative order. Proving the concurrent zero knowledge property involves showing that there is a simulator (a probabilistic polynomial time oracle machine) $\mathcal{S}$ whose output for every $x \in L$ is computationally indistinguishable from the view of this adversary for that x.

Commitment Schemes A commitment protocol involves two parties– the sender and the receiver, and two phases– the commit and reveal. In the commit phase the sender commits to a bit (or a string), by sending a commitment to the receiver. But we require the commitment to be secret: it is infeasible for the receiver to find out anything about the committed string. Later in the reveal phase the sender sends over the committed string and possibly more information so that the receiver can verify that the revealed value is identical to the committed value. We require that the commitment is binding on the sender, i.e., it cannot reveal a value other than what it committed to (with overwhelming probability over the randomness chosen by the receiver). This kind of commitment scheme is said to have statistical binding and computational secrecy. The Zero knowledge protocol we analyze will employ such a scheme from [19] in which the receiver initiates the commitment by sending a random string to the sender and the commit phase has a single message from the sender.

The above mentioned commitment is used when the all powerful prover is the sender. We will also need a commitment scheme when the prover is the receiver. For this we require a scheme with information theoretic secrecy and computational binding. [11] describes one, again with a single initiation message from the receiver, a single message from the sender for each commitment, and for each reveal.

Witness Indistinguishable proofs Witness Indistinguishable proofs, introduced in [9], is a notion similar to, but weaker than zero-knowledge. A witness indistinguishable proof for a language in NP is a protocol such that the prover uses some witness to carry out the proof, but the view of the verifier when the prover uses a witness w_1 and that when it uses a different witness w_2 are computationally indistinguishable. This notion is weak enough to let the security be preserved under concurrent composition.

The concurrent zero-knowledge protocol we are analyzing uses the proof-system for NP languages by Goldreich and Kahan [12]. The proof system involves five messages, the first one from the prover to the verifier. Though the prover is allowed to be computationally unbounded, given a witness w for the membership of the input x the prover can run in polynomial time. This allows us to construct simulators which run in polynomial time, and can carry out the prover's part in this protocol.

In the concurrent zero knowledge protocol, the witness indistinguishable proof is used with respect to the language L' (in NP) defined as follows: $(x,\texttt{preamble}) \in L'$ iff either $x \in L$, or `preamble` is the transcript of a preamble in which the session was "solved" by the prover (see Section 3 for details). The two witnesses we shall consider for $x' \in L$ are (i) a witness w for $x \in L$ or (ii) a witness w' for `preamble` containing a solution.

[2] a function $\nu(k)$ is negligible in k if, as k grows $\nu(k)$ eventually becomes less than $1/p(k)$ for all polynomials p.

Cryptographic Assumptions The cryptographic assumptions we need are the ones on which constructions of the above primitives are based. Assuming the existence of a *collection of claw-free permutations* suffices for this purpose [11].

Concurrent Sessions In the concurrent setting that we are interested in there are up to $\ell = poly\ k$ sessions that run concurrently, using one single prover. In session s the prover is trying to prove that $x_s \in L$. The prover responds to each verifier message in the order in which they come; but it is upto the (adversarial) verifier to choose the session from which the next message sent to the prover comes from.

The simulator and previous analysis We analyze the same protocol as given by [16, 20], except for reducing the number of rounds. The soundness and completeness properties of this system are proved there and the proof holds for the reduced number of rounds too. For proving the zero-knowledge property one needs to demonstrate that for every efficient, but possibly corrupt verifier co-ordinating the verifiers in the polynomially many sessions, there is a simulator such that, for each set of inputs x_s, the output of the simulator and the view of the verifier at the end of the protocol are computationally indistinguishable from each other. [16, 18] gives a black-box simulator, and analyzes the simulator.

They show that to prove this indistinguishability *it is enough to show that the probability their simulator aborts in the middle of the simulation is negligible in k.*

So we need only analyze the probability that the simulator aborts. We give a better analysis of the same simulator, and show that if $m = \omega(\log k)$ the probability of the simulator aborting is negligible.

3 The Protocol

We provide a brief review of the concurrent zero-knowledge proof protocol as described in [16, 18], which in turn is a slight modification of the protocol introduced in [20].

The protocol employs a statistically binding commitment scheme (used by the prover to commit), a statistically hiding commitment scheme (used by the verifier to commit), and a witness indistinguishable proof (interactive proof) system. In Section 2 we briefly reviewed these schemes.

The protocol has two phases– a preamble and a main proof body. The outline of the preamble is provided below:

$V \to P$: Commit to $v_0, \ldots, v_m \in \{0,1\}^k$
$P \to V$: Commit to p_0
$V \to P$: Reveal v_0
$P \to V$: Commit to p_1
$\vdots$
$V \to P$: Reveal v_i
$P \to V$: Commit to p_{i+1}
$\vdots$
$V \to P$: Reveal v_m
$P \to V$: Start the main body of the proof

When the verifier initiates a session, the prover in response initiates the commitment scheme for the verifier. Then the preamble starts, in which the first message from the verifier is a commitment to $m+1$ random strings $v_0, \ldots, v_m \in \{0,1\}^k$, k being the security parameter (it also initiates the prover's commitment scheme). In response the prover commits a random string $p_0 \in \{0,1\}^k$. In subsequent steps, the verifier reveals v_i, and the prover commits to p_{i+1}, for i from 0 to $m-1$. In the last step in the preamble the verifier reveals v_m and the prover starts the witness indistinguishable proof. In all there are $m+2$ messages from the verifier in the preamble.

The witness indistinguishable proof is used with respect to the language L' (in NP) defined as follows: $(x,\texttt{preamble}) \in L'$ iff either $x \in L$, or `preamble` is the transcript of a valid preamble such that there is some i such that $p_i = v_i$.

The protocol continues with the witness indistinguishable proof, in which the prover uses the witness of $x \in L$. The verifier accepts or rejects as in that proof system. At any point during the protocol if an invalid message from the verifier arrives, the protocol is terminated.

In the concurrent setting the prover runs the different sessions independent of each other. If a session is terminated further messages from that session are ignored; but it does not affect the other sessions.

We shall show that $m = \omega(\log k)$ suffices for the zero-knowledge property of the proof system to hold in the concurrent setting.

4 The Simulator

In this section we describe the simulator of [16], which is based on the simulator of [20]. Our description differs from previously given descriptions, as we identify features of the simulator which we will use to achieve our stronger result.

The simulator does not have access to the witness of $x_s \in L$ for any x_s. So in the simulated proof it tries to get a preamble `preamble` such that $(x_s,\texttt{preamble}) \in L'$, and use information on this preamble as the witness.

The simulator $\mathcal{S}$ has black-box access to the verifier. It randomly sets up the random coins for the verifier in the beginning, and then starts running the verifier and a modified prover on the common input x. But every now and then the simulator will rewind the verifier. For each session $\mathcal{S}$ hopes to find out the value of some string v_i before committing to p_i, so that it can commit to $p_i = v_i$. For this $\mathcal{S}$ should wait till the verifier reveals some v_i and then rewind the execution beyond the point where it committed to p_i. But $\mathcal{S}$ cannot afford to do too many rewinds as it must finish running in $poly(k)$ time.

[16] proposes an efficient rewind strategy, which is essentially the same as the rewind strategy of [20]. The rewind strategy is specified with respect to the at most $N := (m+2)\ell$ preamble messages, numbered from 0 to $N-1$. We can assume that N is a power of two by adding empty "dummy" messages at the end, if necessary. The points 0 to $N-1$ when a preamble message arrives, are referred to as the *protocol points* or the *protocol time*.

Protocol Tree $\mathcal{T}$ Consider the complete balanced binary tree with N leaves. We shall call this the *protocol tree* and denote it by $\mathcal{T}$. Height of $\mathcal{T}$ is $h = \log N = O(\log k)$ (as we will consider only $m = O(poly(k))$).

It will be convenient later to identify each node in $\mathcal{T}$ by the path to that node from the root; the path is specified as a string of L's and R's, referred to as the "left-right" or L/R path, in the natural way.

To describe the rewind strategy consider the directed acyclic graph obtained by doubling the edges of $\mathcal{T}$ except the ones at the leaf level (see figure 1). The schedule is essentially a depth-first traversal of this DAG, with the slot numbers appearing at the leaves. Informally, the simulator traverses the DAG, and at a node, after returning from the first edge in a double edge, it rewinds the verifier, and continues the traversal by descending the second edge.

Simulation Tree $\hat{\mathcal{T}}$ The above DAG can be written out as a 4-ary tree by duplicating the nodes. We call this 4-ary tree the *simulation tree*, and denote it by $\hat{\mathcal{T}}$ (see figure 2). Again we shall identify each node in $\hat{\mathcal{T}}$ by the path to it from the root. The path is specified by a string of edge-labels from the set $\{$L0, L1, R0, R1$\}$.

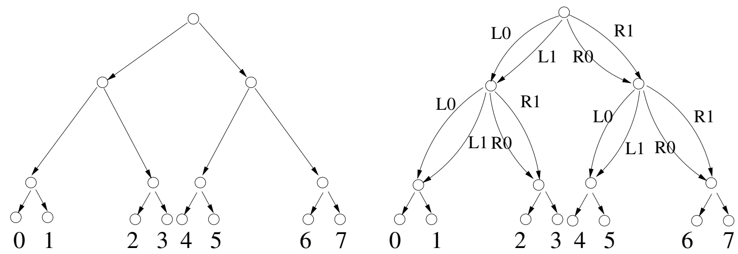

Figure 1: $\mathcal{T}$ *and the edge-doubled DAG for* $N = 8$

We separate this string, referred to as the *composite path* into two strings– the L/R path, and the 0/1 path, in the natural way.

Instances in $\hat{\mathcal{T}}$ Each node a in $\mathcal{T}$ has many *instances* in $\hat{\mathcal{T}}$, which are the nodes with the same L/R path as a has in $\mathcal{T}$. There is one instance for each 0/1 path. Below, we shall usually denote nodes in $\mathcal{T}$ by a, X etc. and those in $\hat{\mathcal{T}}$ by $\hat{a}, \hat{X}$ etc., possibly with some subscripts or superscripts.

The $N^2/2$ leaves of $\hat{\mathcal{T}}$ correspond to the points at which a preamble message arrives during the simulation. The simulator goes through the leaves of the tree from left to right, and we visualize it as an in-order traversal of $\hat{\mathcal{T}}$. We refer to these $N^2/2$ points in the execution of the simulator as *simulation points* or *simulation-time* (as opposed to protocol-time).

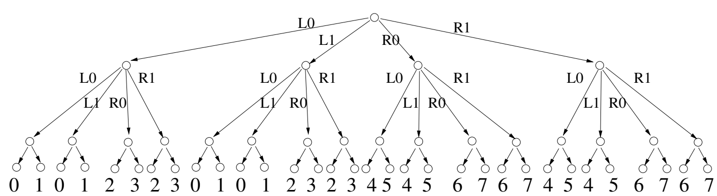

Figure 2: *Simulation tree* $\hat{\mathcal{T}}$ *for* $N = 8$

Sessions A *session* during the run of the simulator is identified by the point in simulation time (leaf of $\hat{\mathcal{T}}$) where the first preamble message of the session, namely the initial commit message from the verifier, arrives. Note that many sessions may simply disappear from the views of the prover and the verifier as the simulator rewinds beyond the start of the session.

The Runs Suppose x is a node in $\mathcal{T}$ and $\hat{x}$ is an instance of x in $\hat{\mathcal{T}}$. The *run* of x associated with $\hat{x}$ is defined as the execution of the simulator from the point at which it traverses down the node $\hat{x}$ to the point at which it returns from $\hat{x}$. The run of $\hat{x}$ can be identified with the interval in simulation-time containing all the descendant-leaves of $\hat{x}$.

Consider a node x in $\mathcal{T}$ with children y and z. Let $\hat{x}$ be an instance of x in $\hat{\mathcal{T}}$, with children $\hat{y}_0$, $\hat{y}_1$, $\hat{z}_0$ and $\hat{z}_1$. The run of x at $\hat{x}$ consists of two runs of y and two runs of z. After the first run of y, $\hat{y}_0$ the simulator *rewinds* – i.e., sets the state of the verifier and the prover to that before the run $\hat{y}_0$. Then it does another run of y, $\hat{y}_1$. Then it goes on to do two runs of z, with a rewind between them.

Suppose during the run $\hat{y}_0$, two properly revealed preamble messages v_i and v_{i+1} are received from a session s that starts *before* that run. At the end of the run $\hat{y}_0$, the simulator knows v_{i+1}. Now the simulator rewinds to the state before the start of the run $\hat{y}_0$. Note that the prover's message p_{i+1} is in response to v_i, which has not yet arrived at the point after the rewind. When the simulator continues its run and at some point the reveal of v_i arrives, it responds by committing not to a random value as p_{i+1} but to $p_{i+1} := v_{i+1}$. This gives a valid preamble with $p_{i+1} = v_{i+1}$ which the prover can use as a witness in the main body of the proof. When the simulator rewinds the run $\hat{y}_0$, we say that the simulator has *solved* the session s.

Once a session s is solved as above, the simulator records this in an auxiliary table called the Solution Table. The Solution Table has entries of the form (s, i, v_i), for $0 \leq s < N$ and $0 \leq i \leq m$. Whenever the modified prover has to respond with a commitment to p_i for a session s it checks if an entry (s, i, v_i) is available in the Solution Table. If it is, it commits to $p_i = v_i$ and notes down this fact and the random coins used in making the commitment; later when the session s enters the main body of the proof the prover can use this information as a witness that $(x_s,\texttt{preamble}) \in L'$ without knowing the witness for $x_s \in L$. Since the main body of the proof is a witness indistinguishable system, using this witness is indistinguishable from what an actual prover will do, and the simulation remains indistinguishable from the real thing.

Note that during the run of the simulator, a session s may reach the i-th message many times; each time the solution from the table (if available) is used to commit to $p_i = v_i$. Also the session may enter the main body of the proof many times; again each time a witness will be available from the last commitment of p_i.

Aborting the simulation If at any point in the simulation, a session reaches the main body, i.e., the reveal for v_m arrives, and no solution is available for the session, the simulator cannot successfully simulate the witness indistinguishable proof. If this happens the simulator *aborts* the entire simulation.

If the simulator does not abort till all the sessions are over (or the verifier terminates), it outputs the view of the verifier at that point. As shown in [20, 16, 18], conditioned on the simulator not aborting, the simulated view it outputs is distributed indistinguishably from the distribution of the view of the verifier after the interaction with an honest prover with witnesses for $x_s \in L$ for all the sessions.

States of the Simulator We go on to give a more formal description of the simulator in terms of its states during the execution. (This may be skipped without much loss of continuity.)

To describe the rewind schedule formally, we will consider a snapshot of the simulator, when the simulator is at a node in its traversal of $\hat{\mathfrak{T}}$.[3] (see below for details). We define the *state* of the simulator, as this snapshot consisting of

(i) The (current) View: The verifier-view consists of a transcript between the prover and the verifier, and the state (work tapes) of the verifier. The simulator also maintains the state of a modified prover (described later). Collectively all this is referred to as the current view.

(ii) Solution Table: the internal table to store the solutions to the solved sessions,

(iii) Book-keeping: A stack of views, called the *view-stack*, to do the rewinds; a counter to indicate the depth of the current node in the simulation tree $\hat{\mathfrak{T}}$, and a stack of 5-ary values, called the *traversal-stack* to traverse $\hat{\mathfrak{T}}$.

We visualize the operation of the simulator as an in-order traversal of $\hat{\mathfrak{T}}$, as described below. The simulator starts from the root, traverses down the tree to the left most leaf of $\hat{\mathfrak{T}}$, and waits there for the first preamble message to arrive; at any time the simulator is at some leaf of $\hat{\mathfrak{T}}$ and when a preamble message, arrives it continues the depth-first traversal until it reaches the next leaf. The preamble message is indexed by the leaf at which the simulator was when the message arrived.

[3]Given such a snapshot we will be able to start the simulator from the point where the snapshot was taken.

A state describes the simulator at a node in $\hat{\mathfrak{T}}$. The top of the traversal stack holds one of the values L0, L1, R0, R1 indicating the next child to descend into in the traversal, or a special value `return`. Below we describe the traversal formally by how one state is updated to the next.

Suppose the depth counter indicates that we are not yet at a node just above the leaves. To move to the next state the simulator checks the top of this stack. If the top value of the traversal-stack is `return` it pops the value from the stack and decrements the depth counter. If it is L0 or R0, the current view is saved by pushing it into the view stack, and the top of the traversal-stack is incremented to L1 or R1, resp. Else if the top of the traversal-stack is L1 or R1, the simulator does a *rewind* by popping the view-stack and replacing the current view with the popped value; also it increments L1 to R0, or R1 to `return`. Finally it moves down in the traversal by incrementing the depth counter and pushing an R0 into the traversal stack, so that the traversal of the child node starts with its first child.

If the depth counter indicates that we are just one level above the leaves, then the simulator has to wait for the next two preamble messages, i.e., it has to move through the two leaves, and then return. For this the simulator keeps modifying the current view by letting the (modified) prover and the verifier run, until two preamble messages arrive. When the second one arrives the simulator moves into the next state by popping the traversal-stack and decrementing the counter.

The Solution Table is a data structure maintained by the simulator and used by the modified prover. While the simulator is running the prover and verifier, the solution table is updated as follows: whenever a properly revealed preamble message v_i from session s comes along, the simulator records the revealed value as the (s, i)-th entry in the solution table.

Whenever the modified prover has to make the commitment p_i for session s, it checks if the (s, i)-th entry in the table is available. If so it commits that value, marks the session as *solved* and records the details of this commitment as a *solution* in the table. If this happens we say the session was solved. Else the prover commits an arbitrary string (say, the zero string).

When the modified prover reaches the main body of proof in a session, instead of entering the witness indistinguishable proof with the witness for the membership of $x_s \in L$, it looks at the solution table to see if the session was ever marked as solved. If it was solved, the solution gives a witness in terms of an i such that $p_i = v_i$, and the modified prover uses this witness. If the session was not solved till this point, then the prover makes the entire simulation *abort*.

When all the sessions are over (or the verifier terminates), the simulator outputs the current view of the verifier.

Modified Simulators We use a couple of modified simulators $\mathcal{S}^*$ and $\mathcal{S}^\dagger$ for purposes of analysis. They differ from the original simulator $\mathcal{S}$ only in the behaviour of the prover. The simulator $\mathcal{S}^*$ has for each session s, the witness for $x_s \in L$, and its prover uses that for the body of the proof. In $\mathcal{S}^\dagger$, in addition to using these witnesses, the prover always commits to the zero string in the preamble. Though the provers of $\mathcal{S}^*$ and $\mathcal{S}^\dagger$ do not use the entries in the Solution Table, they also abort the simulation if it reaches the main body of proof in an unsolved session. Note that $\mathcal{S}^*$ and $\mathcal{S}^\dagger$ are also efficient simulators because in the witness indistinguishable proof system used the prover can run efficiently given a witness for $x_s \in L$.

We would like to show that the distribution of the view output by the simulator $\mathcal{S}$ is computationally indistinguishable from that of the view obtained by the verifier as a result of the interaction with the prover. As shown in [16, 20] it is enough to show that the probability that the $\mathcal{S}$ aborts is negligible in the security parameter k. The following lets us show this only for $\mathcal{S}^\dagger$.

Lemma 1 *The difference between the probability of $\mathcal{S}^\dagger$ aborting, and the probability of $\mathcal{S}$ aborting is negligible in k.*

Proof: This follows from the guarantees of the commitment scheme used by the prover, and the witness indistinguishable proof employed in the main body. $\mathcal{S}^*$ serves as a hybrid between $\mathcal{S}^\dagger$ and the original simulator $\mathcal{S}$. The difference in abort probabilities of $\mathcal{S}$ and $\mathcal{S}^*$ is negligible by the guarantee on the witness indistinguishable scheme, and that of $\mathcal{S}^*$ and $\mathcal{S}^\dagger$ is negligible by the guarantee on the commitment scheme.
$\mathcal{S}$ and $\mathcal{S}^*$ are indistinguishable: First we construct hybrid simulators $\mathcal{S} = \mathcal{S}_0^*, \mathcal{S}_1^*, \ldots, \mathcal{S}_N^* = \mathcal{S}^*$, where $\mathcal{S}_i^*$ uses the witness for $x_s \in L$ only for the sessions $s = 0, \ldots, i-1$. By the hybrid argument it is enough to show that the abort probabilities of $\mathcal{S}_i^*$ and $\mathcal{S}_{i+1}^*$ differ negligibly for all i. We shall construct an adversary for the witness indistinguishable proof, which has an advantage in distinguishing the witness equal to the difference between the abort probabilities of $\mathcal{S}_i^*$ and $\mathcal{S}_{i+1}^*$. This is achieved by introducing the proof to be identified into the simulator's prover for session $i+1$. More formally, the adversary is a modification of $\mathcal{S}_i^*$. It outputs 1 if the (modified) simulator aborts, and 0 otherwise. The adversary starts $\mathcal{S}_i^*$ and if a session gets started at the simulation point $i+1$, then it engages with the given prover as follows: when the simulator reaches the main body of proof in session $i+1$ (if it does at all), the messages from the verifier are directed to the prover. The prover enters the proof with one of the two witnesses– the witness for $x_{i+1} \in L$ or the witness in terms of the preamble. If it uses the former, the execution of the adversary is identical to that of $\mathcal{S}_{i+1}^*$ and else to that of $\mathcal{S}_i^*$. Thus the difference in the abort probabilities of $\mathcal{S}_i^*$ and $\mathcal{S}_{i+1}^*$ translates into an advantage in distinguishing the witnesses.
$\mathcal{S}^*$ and $\mathcal{S}^\dagger$ are indistinguishable: This can be shown in a fashion similar to the above. But this time we are attacking the commitment scheme. The simulator's prover makes many commitments for each session. So this time we introduce one more level of hybrids to take care of this. Define $\mathcal{S}_{i,j}^\dagger$ as a simulator which commits to the zero string in all sessions $s \leq i$ for the first j commitments. Then $\mathcal{S}^* = \mathcal{S}_{0,0}^\dagger$ and $\mathcal{S}^\dagger = \mathcal{S}_{N,m+1}^\dagger$. Rest of the argument is standard.

■

In the rest of the paper we analyze the simulator $\mathcal{S}^\dagger$.

Adversary's success on (start,stop) The adversarial verifier is said to succeed for a pair of simulation points (start,stop), if the session starting at start point reaches the main proof body at the point stop without $\mathcal{S}^\dagger$ having solved the session, thereby forcing $\mathcal{S}^\dagger$ to abort at that point.

Note that for the session to be alive at the point stop, the point start is never rewound beyond, within the interval (start,stop). Formally this means that the current view when the simulator reaches the stop leaf, is obtained by letting the prover and verifier run on the view at the start leaf.

There are only $O(N^2)$ (start,stop) pairs, which is polynomial in k. We shall show that for any given pair the probability that the adversary succeeds with respect to that pair is negligible in k. Then the probability of the adversary succeeding is negligible by union bound.

The points corresponding to start and stop in the protocol-time are called *proto-start* and *proto-stop*.

The forests We define a few structures which we shall be referring to frequently throughout the rest of the paper. The *protocol forest* $\mathcal{F}$ and the *simulation forest* $\hat{\mathcal{F}}$ are subgraphs of $\mathcal{T}$ and $\hat{\mathcal{T}}$ respectively, and are determined by the pair (start,stop). Later on we also define the *good forest* $\mathcal{G}$ which is a subgraph of $\mathcal{F}$ and is determined by a set of m protocol points.

Figures 4 and 4 illustrate these structures.

1. **The protocol forest** An edge in $\mathcal{T}$ is retained in $\mathcal{F}$ if it points to a node all of whose descendant leaves occur strictly within the protocol interval (proto-start,proto-stop). $\mathcal{F}$ is the subgraph of $\mathcal{T}$ induced by these edges.

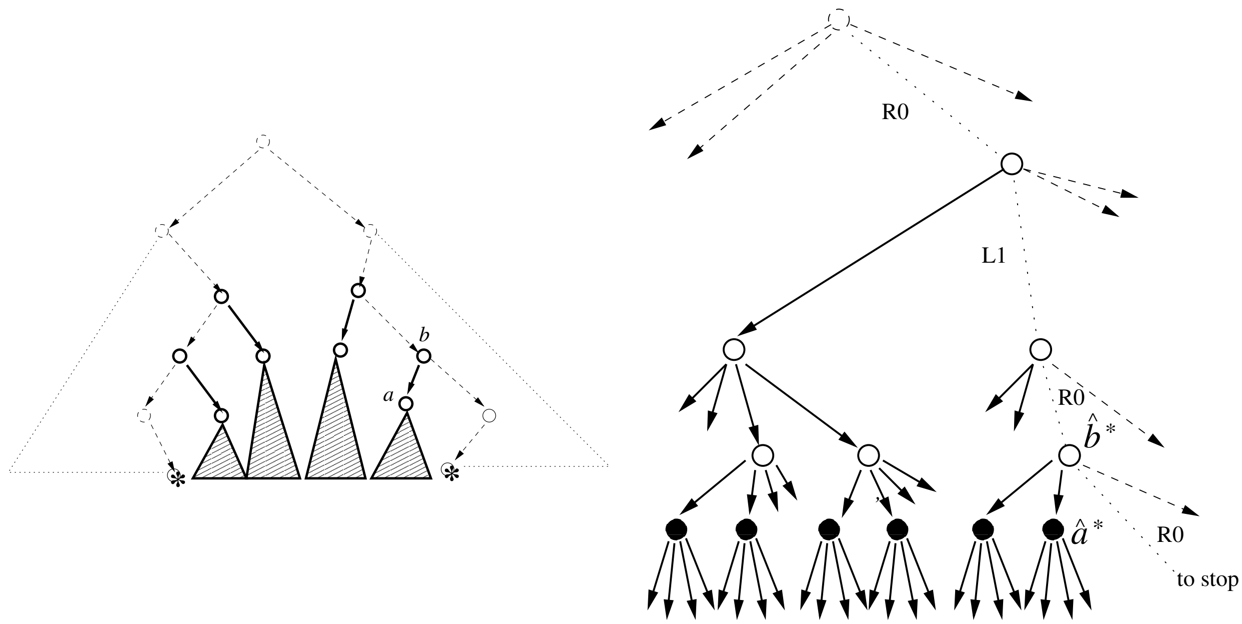

Figure 3: Left, the Protocol forest $\mathcal{F}$ as a subgraph of $\mathcal{T}$: the elements in thick outline are part of $\mathcal{F}$. The *'s indicate the points proto-start and proto-stop. Also, node a is labeled. Right, a portion of the corresponding Simulation forest $\hat{\mathcal{F}}$: the 0/1 path of the point stop begins as 0100..; the corresponding composite path is labeled. This path along with the corresponding path to the start point *cut out* $\hat{\mathcal{F}}$ from the tree $\hat{\mathcal{T}}$. Filled nodes are all the instances of the node a. The final instances of nodes a and b in $\mathcal{F}$ are marked in $\hat{\mathcal{F}}$ as $\hat{a}^*$ and $\hat{b}^*$.

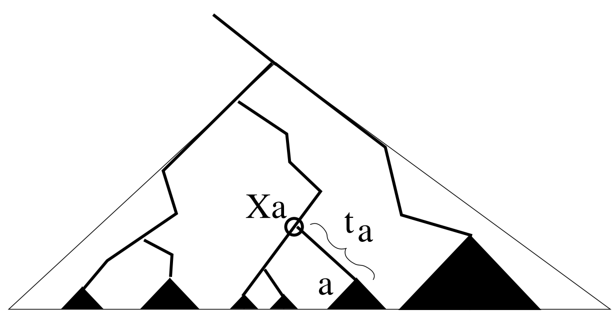

Figure 4: *A tree from the good forest $\mathcal{G}$. The thick edges are part of $\mathcal{G}$. The roots of the shaded subtrees (which have two schedule points, one in each child) are the leaves of $\mathcal{G}$. A leaf a and its pivot X_a are marked. $\mathcal{G}$ consists of many trees like this.*

2. **The simulation forest** An edge in $\hat{\mathcal{T}}$ is retained in $\hat{\mathcal{F}}$ if it points to a node all of whose descendant leaves occur strictly within the simulation interval (start,stop). $\hat{\mathcal{F}}$ is the subgraph of $\hat{\mathcal{T}}$ induced by these edges.

We shall focus on these forests $\hat{\mathcal{F}}$ and $\mathcal{F}$ rather than the entire trees $\hat{\mathcal{T}}$ and $\mathcal{T}$. Figure 4 illustrates portions of $\mathcal{F}$ and $\hat{\mathcal{F}}$, for a given (start,stop) pair. Note that all nodes except possibly the roots of the trees in $\mathcal{F}$ has all its descendant leaves strictly in the interval (start,stop). If a node a in $\mathcal{F}$ is at depth d in its tree in the forest $\mathcal{F}$, a has at least 2^d instances in $\hat{\mathcal{F}}$. (The precise number of instances in $\hat{\mathcal{F}}$ of a node at depth d in $\mathcal{F}$, is $2^d \times$ (number of instances in $\hat{\mathcal{F}}$ of the root of that node's tree in $\mathcal{F}$).) However $\hat{\mathcal{F}}$ can have nodes which are instances of nodes in $\mathcal{T}$ outside $\mathcal{F}$.

Now suppose we are given a run of $\mathcal{S}^\dagger$ in which the adversary succeeds. Then the (current) transcript when the simulator reaches the point stop will show m preamble messages of the session, where the verifier properly reveals $v_0, \ldots, v_{m-1}$, at some m points in the interval (proto-start,proto-stop). We call these m protocol points the *schedule points* of the session. Given such a transcript with m schedule points, we define the following forest:

3. **The Good Forest** $\mathcal{G}$ is the subgraph of $\mathcal{F}$ induced by all the nodes in $\mathcal{F}$ which have at least two schedule points as descendants (see Figure 4).

The leaves of $\mathcal{G}$ essentially correspond to the *may-solve* intervals as defined in [16, 18] (where it is shown that there are $\Omega(m/h)$ such leaves, but we will not need this). They cover *disjoint* intervals in protocol-time. We order these leaves from left to right according to their intervals, in the natural way.

Pivot X_a and t_a For each leaf of $\mathcal{G}$ a, let the *pivot* of a, denoted by X_a, be the node in $\mathcal{G}$ defined as follows: X_a is the least common ancestor of a with the *previous* leaf in $\mathcal{G}$, or if no such node exists, the root of a's tree in $\mathcal{G}$. Define t_a as the distance of a to X_a.

Note that $\sum_a t_a =$ number of edges in $\mathcal{G}$, where the summation is over all the leaves of $\mathcal{G}$. This is because each edge in $\mathcal{G}$ is counted in the t_a when a is the first leaf (in the order defined above) among its descendants.

Lemma 2 *Number of edges in the good forest $\mathcal{G}$ is at least $m - O(h)$, where h is the height of the protocol tree $\mathcal{T}$.*

Then by the above note, we have $\sum_a t_a = m - O(h)$ where the summation is over all the leaves of $\mathcal{G}$.

Proof: Define the good tree $\mathcal{G}'$ as a subgraph of the protocol tree $\mathcal{T}$, of nodes with at least two schedule points among descendants. Note that $\mathcal{G}$ is a subgraph of $\mathcal{G}'$ obtained by removing all the edges which are in the path to the leaves proto-start or proto-stop. Now map each schedule point to its closest ancestor in $\mathcal{G}'$. Then each leaf in $\mathcal{G}'$ has exactly two points mapped to it and each other node in $\mathcal{G}'$ has at most one point mapped to it (or none if it has degree 2 in $\mathcal{G}'$). So there are $\Theta(m)$ nodes with something mapped on to it. So $\mathcal{G}'$ has at least that many nodes. In fact $\mathcal{G}'$ has at least $m-1$ nodes. To see this note that the nodes of $\mathcal{G}'$ with one point mapped to it are internal nodes with only one child in $\mathcal{G}'$, and those with two are leaves of $\mathcal{G}'$; if there n_1 and n_2 of them respectively, $n_1 + 2n_2 = m$. To have n_2 leaves, $\mathcal{G}'$ must have at least $n_2 - 1$ internal nodes of out-degree 2; thus in all $\mathcal{G}'$ has at least $n_1 + n_2 + n_2 - 1 = m - 1$ nodes. Thus $\mathcal{G}'$ has at least $m - 2$ edges.

There are at most $2h$ edges in $\mathcal{G}'$ which point to nodes with proto-start or proto-stop among descendants, namely the edges in the paths up to the root from those points. We delete these edges to get $\mathcal{G}$. Thus $\mathcal{G}$ has at least $m - O(h)$ edges as claimed. ■

We make a few definitions regarding the nodes in $\hat{\mathfrak{F}}$.

- *Final instance:* Consider any node x in $\mathcal{G}$. There are many instances of this node in $\hat{\mathfrak{F}}$ (and many in $\hat{\mathfrak{T}}$ outside $\hat{\mathfrak{F}}$, which we do not consider [4]). These instances can be ordered by the time the simulator starts their run. We define the *final instance* of x to be the last instance in this order *within* $\hat{\mathfrak{F}}$. We denote the final instance of x by $\hat{x}^*$.

- *Critical runs:* Consider a leaf of $\mathcal{G}$ a, and its pivot X_a. There are many instances of X_a in $\hat{\mathfrak{F}}$. Each run of X_a contains 2^{t_a} runs of a. Consider the last such run, $\hat{X}_a^*$. We define the 2^{t_a} runs of a in $\hat{X}_a^*$ as the *critical runs* of a. Note that, if b is a leaf of $\mathcal{G}$ before a, the critical runs of a are all after the run $\hat{b}^*$.

- *$\hat{a}^{\langle i \rangle}$ and $\hat{a}^{\langle * \rangle}$:* The 2^{t_a} critical instances of a leaf a in $\mathcal{G}$ appearing in $\hat{\mathfrak{F}}$ are numbered from left to right. They are denoted by $\hat{a}^{\langle i \rangle}$ for $0 \leq i < 2^{t_a}$, where i is a t_a digit binary number. Note that $\hat{a}^{\langle 2^{t_a}-1 \rangle}$ is the same as $\hat{a}^*$. We denote this by $\hat{a}^{\langle * \rangle}$ to emphasize that this is a critical run.

5 Analysis: Static Case

In the static case the adversarial verifier schedules the messages from the various sessions at pre-determined slots. The only choice the adversary gets to make is as to whether the message is revealed properly or not. There are m points in the protocol time interval (proto-start,proto-stop), where the adversary has non-zero probability of ever scheduling a message from the session. These points are called the potential points. Thus if the adversary succeeds in a run, the schedule points of that run are exactly the potential points.

In the static case we consider the $\mathcal{G}$ defined with respect to the potential points as the schedule points.

- Recall that a leaf of $\mathcal{G}$ has two potential points below it. A run of a leaf of $\mathcal{G}$ is *good* if the transcript at the end of the run has both the potential messages properly revealed. Else it is called bad.

- A leaf of $\mathcal{G}$ a is said to be *won* (by the adversarial verifier) if all the 2^{t_a} critical runs of a are bad except the final one which is good.

- For each leaf a of $\mathcal{G}$, we define $p_a^{\langle i \rangle}$ as the probability that $\hat{a}^{\langle i \rangle}$ is good given that $\hat{a}^{\langle j \rangle}$ for all $j < i$ were bad and all previous leaves of $\mathcal{G}$ were won. The probability is taken over the coin-flips of the simulator (which includes the coin-flips of the verifier and the modified prover).

Lemma 3 $p_a^{\langle i \rangle} = p_a^{\langle j \rangle}$ *for all* i, j, $0 \leq i < j < 2^{t_a}$.

Proof: It is enough to prove this for the case when the binary representation of i and j have a hamming distance of one. There is some node $\hat{b}$ in $\hat{\mathfrak{F}}$ which is the least common ancestor of the two critical instances of a, $\hat{a}^{\langle i \rangle}$ and $\hat{a}^{\langle j \rangle}$. Let the children of $\hat{b}$, $\hat{c}_0$ and $\hat{c}_1$ be the ancestors of $\hat{a}^{\langle i \rangle}$ and $\hat{a}^{\langle j \rangle}$ respectively, where $\hat{c}_0, \hat{c}_1$ are either the L0 and L1 children (resp.) of $\hat{b}$ or the R0 and R1 children (resp.) of $\hat{b}$.

Suppose $p_a^{\langle i \rangle} \neq p_a^{\langle j \rangle}$. Consider a state of the simulator (as defined earlier) at the start of the run of $\hat{c}_0$, and define ${p'}_a^{\langle i \rangle}$ (resp ${p'}_a^{\langle j \rangle}$) by modifying the definition of $p_a^{\langle i \rangle}$ (resp $p_a^{\langle j \rangle}$) by further conditioning on that state. Then $p_a^{\langle i \rangle}$ (resp $p_a^{\langle j \rangle}$) is a convex combination of ${p'}_a^{\langle i \rangle}$ (resp ${p'}_a^{\langle j \rangle}$) defined with respect to the various states. Then there is one such state τ, such that ${p'}_a^{\langle i \rangle}$ and ${p'}_a^{\langle j \rangle}$ defined with respect to τ differ.

[4] also there are nodes in $\hat{\mathfrak{F}}$ which do not have any corresponding nodes in $\mathcal{G}$ or even $\mathcal{F}$, but we will not consider them

Reasoning similarly, there must exist some state τ' at the start of the run of $\hat{c}_1$ such that (i) τ' is an extension of τ (because we are now considering events conditioned on τ) and, (ii) ${p''}_a^{\langle j \rangle}$ defined by modifying ${p'}_a^{\langle j \rangle}$ by further conditioning on τ', differs from ${p'}_a^{\langle i \rangle}$.

We note the following regarding τ and τ':

(i) The view of the verifier is identical in τ and τ' because τ' is an extension of τ (by which we mean that the simulator can reach τ' starting from τ) and at the point of starting the run of $\hat{c}_1$ the simulator rewinds the verifier's view to the point before it started the run of $\hat{c}_0$, to get the view in τ.

(ii) They can differ in the solution tables. But note that $\mathcal{S}^\dagger$ runs independent of the solution table.

(iii) The depth counter of τ and τ' are the same. But the stacks in τ and τ' are different: view-stack of τ is a prefix of that of τ', and the tree-traversal-stack of τ and τ' differ in the top value. But during the run of $\hat{c}_0$ or $\hat{c}_1$ both are equivalent; that is, if one is replaced by the other, the simulator will still behave identically. This is because during the run of $\hat{c}_0$ or $\hat{c}_1$ (or any run for that matter) the simulator does not make use of the records already in the stacks before the run starts.

Thus we see that the run of $\hat{c}_0$ starting from the state τ and the run of $\hat{c}_1$ starting from the state τ' are identical and this contradicts the two probabilities being different.

■

This lets us write p_a for $p_a^{\langle i \rangle}$ for all i.

Bounding the probability

We consider an adversarial verifier. When the simulator runs, if the adversary has to succeed in taking the session started at the point *start* to the point *stop* with out the simulator solving the session, each leaf a in $\mathcal{G}$ must be won (recall that it means all the 2^{t_a} runs of a are bad except the final one which is good). We seek to bound the probability of this event by a negligible function.

Theorem 1 *(Static Case) If $m = \omega(\log k)$ the probability that at the point* stop*, the adversary succeeds in a session starting at* start*, is negligible.*

Proof: In the products below, a ranges over all the leaves of $\mathcal{G}$.

$$
\begin{aligned}
&\mathbf{Pr}\,[\text{adversary succeeds}] \le \mathbf{Pr}\,[\text{ all the critical runs of all the leaves of } \mathcal{G} \text{ are won }] \\
&= \prod_a \mathbf{Pr}[\text{ all the critical runs of } a \text{ are won } | \\
&\qquad\qquad \text{all the critical runs of all the previous leaves of } \mathcal{G} \text{ are won }] \\
&= \prod_a \prod_{i=0}^{2^{t_a}-2} \mathbf{Pr}[\ \hat{a}^{\langle i \rangle} \text{ is bad } | \ \hat{a}^{\langle j \rangle} \text{ is bad for all } j < i, \\
&\qquad\qquad \text{and all the previous leaves of } \mathcal{G} \text{ are won }] \\
&\qquad \times \mathbf{Pr}[\text{ last critical run of } a \text{ is good } | \ \hat{a}^{\langle j \rangle} \text{ is bad for all } j < i, \\
&\qquad\qquad \text{and all the previous leaves of } \mathcal{G} \text{ are won }] \\
&= \prod_a \left(\prod_{i=0}^{2^{t_a}-2} (1 - p_a^{\langle i \rangle}) \right) p_a^{\langle 2^{t_a}-1 \rangle} \le \prod_a (1 - p_a)^{2^{t_a}-1} p_a
\end{aligned}
$$

But for all values of p_a in the range $[0,1]$ and all $T \geq 1$ we have $(1-p_a)^T p_a \leq \left(\frac{T}{T+1}\right)^T \frac{1}{T+1} < \frac{1}{T+1}$. So $(1-p_a)^{2^{t_a}-1} p_a \leq \left(\frac{1}{2}\right)^{t_a}$ (this is true for $t_a = 0$ also). Thus,

$$\mathbf{Pr}\,[\text{adversary succeeds}] \leq \prod_a 1/2^{t_a} = \left(\frac{1}{2}\right)^{\sum_a t_a}$$

But $\sum_a t_a$ (where the summation is over all the leafs in $\mathcal{G}$) is exactly the number of edges in $\mathcal{G}$, which by Lemma 2 is $m - O(h) = m - O(\log k)$. If we choose $m = \omega(\log k)$, $\sum_a t_a = \omega(\log k)$ and the probability of adversary succeeding is bounded by $1/2^{\omega(\log k)}$, which is negligible. ■

6 Analysis: General Case

Now we present the analysis for the general case.

Recall that the adversarial verifier is said to succeed on a run of $\mathcal{S}^\dagger$ for a (start,stop) pair, if the session starting at the point start reaches the main proof body at the point stop without $\mathcal{S}^\dagger$ having solved it. We would like to bound the probability that the adversary succeeds. We shall bound this probability for each setting of the coin flips of the verifier. Now onwards we assume the coin flips of the verifier to be fixed. Thus we consider the probability with respect to the coin-flips of the simulated prover only.

Cards and Decks To represent the coin-flips of the simulator, we augment the simulation tree $\hat{\mathfrak{T}}$ by installing long enough random strings at each leaf. The number of random bits needed by the simulator at each leaf is bounded by a polynomial in k, say $p(k)$, which we let to be the length of the random string at each leaf. Each such random string is called a *card*, drawn uniformly from a universe of size $2^{p(k)}$. All the N cards in the leaves of $\hat{\mathfrak{T}}$ will be collectively referred to as a *deck*. When $\mathcal{S}^\dagger$ is at a leaf it uses the random string from the card at that leaf to do its commitments and non-preamble proof steps.

Since we have already fixed the coins of the verifier, given a deck the entire execution of the simulator is determined. Now, to bound the probability that the adversary succeeds we have to bound the number of decks for which the adversary succeeds. We shall show that for every deck for which the adversary succeeds, there are many other decks with which the adversary fails (taking care not to double-count the decks).

Good forest, good nodes and pivot-instance The (start,stop) pair defines the protocol forest $\mathfrak{F}$ and the simulation forest $\hat{\mathfrak{F}}$ as before, and also fixes the session we will be considering, namely the session with the initial verifier commitment at the start point. The only variable then is the deck. Given a deck with which the adversary succeeds, we can define the *good tree* $\mathcal{G}$ from the transcript at the stop point, as described earlier.

Further for each leaf a of $\mathcal{G}$ we can define the pivot X_a (and its final instance in $\hat{\mathfrak{F}}$, $\hat{X}_a^*$), the length t_a and the 2^{t_a} critical runs of a as described before.

A node in $\hat{\mathfrak{F}}$ is *good* if its run adds exactly two properly revealed preamble messages of the session to the transcript, and each of its descendant nodes' run adds at most one. Note that the good instance is defined with no reference to "potential-points" or $\mathcal{G}$. Given a deck (with which the adversary may or may not succeed), it is possible to check if an instance is good or not.

If the adversary succeeds with a deck, the good instances are exactly the *final* instances of the leaves of $\mathcal{G}$. Having a good instance which is not final allows the simulator to solve the session.

Define the *pivot-instance* of a node $\hat{a}$ as a node $\hat{X}_{\hat{a}}$ in $\hat{\mathfrak{F}}$ as follows: if the last good node before the start of the run of $\hat{a}$ (if it exists) is in the same tree in $\hat{\mathfrak{F}}$ as $\hat{a}$, then $\hat{X}_{\hat{a}}$ is the least common ancestor of that node and $\hat{a}$; otherwise $\hat{X}_{\hat{a}}$ is the root of the tree in $\hat{\mathfrak{F}}$ containing $\hat{a}$. For a deck with which the adversary succeeds, if a is a leaf of $\mathcal{G}$ and $\hat{a}$ is any critical instance of a in $\hat{\mathfrak{F}}$, then $\hat{X}_{\hat{a}^*}$ is the same as the final instance of the pivot of a, $\hat{X}^*_a$.

Lemma 4 *There exists a procedure with the following behaviour.*
Input: *A deck D^* such that the adversary succeeds in D^*.*
Output: *At least $2^{m-O(h)}$ distinct decks, such that in all but one of them, $\mathcal{S}^\dagger$ solves the session. There exists a procedure* UNSHUFFLE *such that it unshuffles each of these distinct decks back to the original deck D^*.*

We shall prove this lemma shortly, but before that note that it achieves our goal.

Theorem 2 *The probability that the adversary succeeds for a given (start,stop) pair is at most $2^{-(m-O(h))}$.*

Proof: All the decks are equally probable. Lemma 4 says that for every deck with which the adversary succeeds there are $2^{m-O(h)} - 1$ other decks for which it doesn't. Being able to unshuffle these decks to the original one guarantees us that we do not double-count any of them for different decks given by the adversary. Thus the existence of a procedure as described in Lemma 4 establishes an upper bound of $2^{-(m-O(h))}$ on the probability that the adversary succeeds. ■

Shuffle-Unshuffle In order to prove Lemma 4, first we give a procedure SHUFFLE and an inverse procedure UNSHUFFLE.

Let the leaves of $\mathcal{G}$, from left to right, be $a_1, a_2, \ldots, a_q$. The shuffle procedure SHUFFLE consists of a sequence of basic shuffles, one associated with each leaf of $\mathcal{G}$. The leaves are processed right to left, i.e. in the order $a_q, \ldots, a_1$.

SHUFFLE takes a deck D^* in which the adversary succeeds. With this deck there are many good instances as defined above, but all of them have to be final. The aim of SHUFFLE is to produce a new deck such that these final good instances get moved around and occur as non-final good instances, allowing the simulator to solve the session.

The algorithm is described in Figure 6. Note that we refer to the L0 child of a node $\hat{X}$ in $\hat{\mathfrak{F}}$ as $(L:0)$-child of X and so on.

SWAP does an atomic shuffle operation, exchanging the cards at the leaves of a run $\hat{Z}_0$ with that of a run $\hat{Z}_1$; informally, this advances the run of $\hat{Z}_1$ ahead of that of $\hat{Z}_0$, as the runs of these nodes are essentially determined by the cards at the leaves. BASIC-SHUFFLE (D, j, α_j) will shuffle the deck so that the final and good run of α_j, $\hat{a}_j^{\langle * \rangle}$ is advanced to the node $\hat{a}_j^{\langle \alpha_j \rangle}$. If α_j is the all-ones string this does not change anything; but otherwise $\hat{a}_j^{\langle \alpha_j \rangle}$ is a non-final run and after the shuffle is a good run, allowing the simulator to solve the session.

Lemma 5 *If the first j good instances with the deck D are $\hat{a}_1^{\langle * \rangle}, \ldots, \hat{a}_{j-1}^{\langle * \rangle}, \hat{a}_j^{\langle * \rangle}$, then the deck $D' =$* BASIC-SHUFFLE(D, j, α_j) *is such that* (I) *the first j good instances with D' are $\hat{a}_1^{\langle * \rangle}, \ldots, \hat{a}_{j-1}^{\langle * \rangle}, \hat{a}_j^{\langle \alpha_j \rangle}$, and* (II) BASIC-UNSHUFFLE (D', j) *gives (D, α_j).*

Proof: (I) We examine the steps during the shuffle. Let α_j/r denote the string α_j but with the last $t_{a_j} - r$ bits replaced by all ones. We shall establish that after r iterations of the loop in BASIC-SHUFFLE, with the resulting deck $\hat{a}_j^{\langle \alpha_j/r \rangle}$ is the j-th good instance; then, since at the end of the subroutine $r = t_{a_j}$ we have that $\hat{a}_j^{\langle \alpha_j/r \rangle} = \hat{a}_j^{\langle \alpha_j \rangle}$ is the j-th good instance as claimed. We shall also show that the first $j-1$ good nodes unchanged.

SHUFFLE $(D^*, (\alpha_1; \ldots; \alpha_q))$
  $D = D^*$
  for $j := q$ to 1 **do**
    $D :=$BASIC-SHUFFLE (D, j, α_j)
  output D

BASIC-SHUFFLE (D, j, α_j)
  $\beta :=$ L/R path from X_{a_j} to a_j in $\mathcal{T}$
  $\hat{Z} := \hat{X}^*_{a_j}$
  for $r := 1$ to t_j **do**
    $b := \beta[r]$ $\{\beta[r]$ is the r-th bit of $\beta\}$
    $\hat{Z}_0 := (b : 0)$-child of $\hat{Z}$
    $\hat{Z}_1 := (b : 1)$-child of $\hat{Z}$
    if $\alpha_j[r] = 0$ **then**
      $D :=$ SWAP $(D, \hat{Z}_0, \hat{Z}_1)$
      $\hat{Z} := \hat{Z}_0$
    else
      $\hat{Z} := \hat{Z}_1$
  output D

SWAP $(D, \hat{Z}_0, \hat{Z}_1)$
  for each composite path γ from $\hat{Z}_0$ **do**
    Take γ from $\hat{Z}_0$ to reach leaf τ_0
    Take γ from $\hat{Z}_1$ to reach leaf τ_1
    Exchange cards at τ_0 and τ_1
  output D

UNSHUFFLE (D)
  for $j := 1$ to q **do** $\{q$ is not known *a priori* but determined when adversary succeeds in D $\}$
    $(D, \eta) :=$BASIC-UNSHUFFLE (D, j)
    $\alpha := \alpha; \eta$
  output (D, α)

BASIC-UNSHUFFLE (D, j)
  $\hat{a} := j$-th good node
  $\hat{X} := \hat{X}_{\hat{a}}$, the pivot-instance of $\hat{a}$
  $\beta :=$ L/R path from $\hat{X}$ to $\hat{a}$ (of length t_j)
  $\alpha :=$ 0/1 path from $\hat{X}$ to $\hat{a}$
  $\hat{Z} := \hat{a}$
  for $r := t_j$ to 1 **do**
    $\hat{Z} :=$ parent of $\hat{Z}$
    if $\alpha[r] = 0$ **then**
      $\hat{Z}_0 := (\beta[r] : 0)$-child of $\hat{Z}$
      $\hat{Z}_1 := (\beta[r] : 1)$-child of $\hat{Z}$
      $D :=$ SWAP $(D, \hat{Z}_0, \hat{Z}_1)$
  output (D, α)

Figure 6: The SHUFFLE and UNSHUFFLE procedures for modifying the decks.

We use induction on r. When $r = 0$, $\alpha_j/0$ is the all-ones string, and $\hat{a}_j^{\langle \alpha_j/0 \rangle} = \hat{a}_j^{\langle * \rangle}$. The claim is true by the assumption on the input. Now we consider $r > 0$. At the beginning of the r-th iteration $\hat{Z}$ is at a distance $t_{a_j} - r + 1$ from $\hat{a}_j^{\langle \alpha_j \rangle}$, and $\hat{a}_j^{\langle \alpha_j \rangle}$ is under the $(\beta[r] : \alpha_j[r])$-child of $\hat{Z}$. If $\alpha_j[r] = 1$, then in the r-th iteration of the loop in BASIC-SHUFFLE the deck stays unchanged and $\alpha_j/r = \alpha_j/(r-1)$; so the conclusion follows trivially. But if it is 0 a swap takes place, in which the cards on the leaves of the subtree under $\hat{Z}_1$ are moved to the leaves of the subtree under $\hat{Z}_0$. The subtree under $\hat{Z}_1$ contains $\hat{a}_j^{\langle \alpha_j/r-1 \rangle}$ which by induction hypothesis, is the j-th good instance before the swap. Also at that point the subtree under $\hat{Z}_0$ does not have any good instance because the $j-1$-st good instance is $\hat{a}_{j-1}^{\langle * \rangle}$ which occurs before $\hat{Z}_0$.

The simulator has the same state when it starts the run of $\hat{Z}_0$ and the run of $\hat{Z}_1$ except for the contents already in its stacks, which are not examined during the runs, and the contents of the Solution Table, which are also ignored by the simulator $\mathcal{S}^\dagger$. The cards at the leaves of $\hat{Z}_1$ before the swap appear at the leaves of $\hat{Z}_0$ after the swap. So the run of $\hat{Z}_0$ with the deck after the swap is identical to the run of $\hat{Z}_1$ with the deck before the swap. (But the execution *after* $\hat{Z}_1$ may now be totally different from anything before the shuffle.)

Further the cards at the leaves of all the runs completing before the start of the run of $\hat{Z}_0$ remain unchanged. Hence with the new deck, the good nodes encountered before starting the run of $\hat{Z}_0$ are exactly the ones encountered till then with the old deck. Thus after the r-th iteration, $\hat{a}_j^{\langle \alpha_j/r \rangle}$ is the j-th good node for the new deck and the first $j-1$ good nodes remain unchanged.

(II) By the above, the j-th good node with D' is $\hat{a}_j^{\langle \alpha_j \rangle}$ (denoted in BASIC-UNSHUFFLE as $\hat{a}$) and the one before that is $\hat{a}_{j-1}^{\langle * \rangle}$. So the pivot-instance of $\hat{a}_j^{\langle \alpha_j \rangle}$ is the same as that of $\hat{a}_j^{\langle * \rangle}$ with D, and is denoted as $\hat{X}$. Then the 0/1 path from $\hat{X}$ to $\hat{a}$ (denoted as α) is α_j. Also β, the L/R path from $\hat{X}$ to $\hat{a}$ is the same as the L/R path from X_{a_j} to a_j in $\mathcal{T}$. Then it is straight-forward to verify that BASIC-UNSHUFFLE reverses the shuffle done by BASIC-SHUFFLE (note that SWAP is its own inverse operation). ■

Proof of Lemma 4: The complete shuffling procedure takes q shuffle-strings $\alpha_1 \dots \alpha_q$, and the deck given by the adversary D^* (such that the only good nodes with D_q are the q final instances $\hat{a}_1^{\langle * \rangle} \dots \hat{a}_q^{\langle * \rangle}$). Let $D_q := D^*$. It then applies the shuffle subroutine repeatedly to produce $D_{j-1} :=$ BASIC-SHUFFLE (D_j, j, α_j), for $j = q, \dots, 1$. The final deck D_0 is output.

UNSHUFFLE applies the BASIC-UNSHUFFLE repeatedly to this deck. By the above claim we have $(D_j, \alpha_j) :=$ BASIC-UNSHUFFLE (D_{j-1}, j), for $j = 1, 2, \dots, q$ (q can be found out once it reaches a deck with which the adversary succeeds). Thus UNSHUFFLE indeed recovers D^* and $\alpha_1 \dots \alpha_q$ from D_0.

α_j is a t_{a_j} long bit string, and therefore there are $2^{\sum_{a_j} t_{a_j}}$ strings $(\alpha_1; \dots; \alpha_q)$. Consider a procedure which calls SHUFFLE with all of them. Since UNSHUFFLE can recover $(\alpha_1; \dots; \alpha_q)$ from the shuffled deck, this way we get $2^{\sum_{a_j} t_{a_j}}$ *distinct* decks. But $\sum_{a_j} t_{a_j}$ is equal to the number of edges in $\mathcal{G}$ and by Lemma 2 is at least $m - O(h)$.

To complete the proof we observe the following: when all the q shuffle-strings are all-ones strings, the resulting deck is the original deck. For any other collection of shuffle-strings, the resulting deck lets the simulator solve the session: suppose α_j is the first not-all-ones shuffle-string. Then $\hat{a}_j^{\langle \alpha_j \rangle}$ is a non-final instance which is good. This allows $\mathcal{S}^\dagger$ with this deck to solve the session. ■

In Theorem 2 if we set $m = \omega(\log k)$, the probability of the simulator aborting becomes negligible (as h the height of the simulation tree is $O(\log k)$). By the analysis in [16, 18] this establishes that this protocol of round complexity $\omega(\log k)$ is a concurrent zero knowledge for languages in NP, yielding the improvement that we promised.

7 Tightness of the analysis

Our analysis of the simulator is asymptotically tight. We demonstrate that if $m < h$, h being the height of the protocol tree, then a simple deterministic schedule by the verifier can make the simulator $\mathcal{S}$ abort with probability one. Note that with $\Theta(k)$ sessions $h = \Theta(\log k)$. So $m = O(\log k)$ rounds is not sufficient for $\mathcal{S}$ not to abort the simulation.

The first preamble message (verifier's commit) is scheduled at the first protocol point, 0 and the last one in response to which the prover has to enter the main body of the proof is scheduled at the last point $N - 1$, where $N = 2^h$. The other $m < h$ preamble messages are scheduled at the $h - 1$ protocol points numbered $N/2, N/2 + N/4, \dots, N - 2$. The verifier deterministically reveals all the messages properly.

Then it is not hard to verify that the first time the simulator reaches the point $N - 1$, the session wouldn't have been solved.

8 Conclusion

We have shown concurrent zero knowledge proofs for languages in NP with round complexity $\omega(\log k)$. In [3] it is established that if the round-complexity of a concurrent zero-knowledge proof system for a language L is $o\left(\frac{\log k}{\log \log k}\right)$ then L is in BPP. Our upperbound on round-complexity, on the other hand is $\omega(\log k)$. It will be interesting to close this gap. But in Section 7 we saw that it is not possible to bring down the upperbound using this simulator.

Acknowledgments

We gratefully thank Joe Kilian for sharing his thoughts with us and generously giving us his permission to use and build up on his suggestion [15] for a general-case analysis which avoids conditioning. This suggestion came out of discussions between the authors and Kilian regarding the dangers and subtleties involved in conditioning-based analyses.

References

[1] B. Barak. How to Go Beyond the Black-Box Simulation Barrier. In *42nd IEEE Symposium on Foundations of Computer Science*, pages 106–115, 2001.

[2] R. Canetti, O. Goldreich, S. Goldwasser, and S. Micali. Resettable Zero-Knowledge. (revised version available from `http://www.wisdom.weizmann.ac.il/~oded/p_cggm.html`). In *ACM Symposium on Theory of Computing*, 2000.

[3] R. Canetti, J. Kilian, E. Petrank, and A. Rosen. Black-box Concurrent Zero-Knowledge Requires Omega (log n) Rounds. In *ACM Symposium on Theory of Computing*, pages 570–579, 2001.

[4] G. D. Crescenzo and R. Ostrovsky. On Concurrent Zero-Knowledge with Pre-processing. In *CRYPTO*, pages 485–502, 1999.

[5] I. Dåmgard. Efficient Concurrent Zero-Knowledge in the Auxiliary String Model. In *EUROCRYPT*, 2000.

[6] C. Dwork, M. Naor, and A. Sahai. Concurrent Zero-Knowledge. In *ACM Symposium on Theory of Computing*, pages 409–418, 1998.

[7] C. Dwork and A. Sahai. Concurrent Zero-Knowledge: Reducing the Need for Timing Constraints. In *CRYPTO*, pages 442–457, 1998.

[8] U. Feige. PhD thesis, Weizmann Institute of Science, 1990.

[9] U. Feige and A. Shamir. Witness Indistinguishability and Witness Hiding Protocols. In *22nd ACM Symposium on the Theory of Computing*, 1990.

[10] O. Goldreich. Concurrent Zero-Knowledge With Timing, Revisited. (available from `http://www.wisdom.weizmann.ac.il/~oded/p_conc-zk.html`), 2001.

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

[12] O. Goldreich and A. Kahan. How to Construct Constant-Round Zero-Knowledge Proof Systems for NP. *Journal of Cryptology*, 9(2):167–189, 1996.

[13] O. Goldreich and H. Krawczyk. On the Composition of Zero-Knowledge Proof Systems. *SIAM Journal on Computing*, 25(1):169–192, 1996.

[14] S. Goldwasser, S. Micali, and C. Rackoff. The Knowledge Complexity of Interactive Proof-Systems. *SIAM Journal on Computing*, 18:186–208, 1989.

[15] J. Kilian. Personal Communication.

[16] J. Kilian and E. Petrank. Concurrent and resettable zero-knowledge in poly-logorithmic rounds. In *ACM Symposium on Theory of Computing*, pages 560–569, 2001.

[17] J. Kilian, E. Petrank, and C. Rackoff. Lower Bounds for Zero Knowledge on the Internet. In *IEEE Symposium on Foundations of Computer Science*, pages 484–492, 1998.

[18] J. Kilian, E. Petrank, and R. Richardson. Concurrent Zero-Knowledge Proofs for NP. (available at `http://www.cs.technion.ac.il/~erez/publications.html` from the public webpage of Erez Petrank), 2001.

[19] M. Naor. Bit Commitment Using Pseudorandomness. *Journal of Cryptology: the journal of the International Association for Cryptologic Research*, 4(2):151–158, 1991.

[20] R. Richardson and J. Kilian. On the Concurrent Composition of Zero-Knowledge Proofs. In *EUROCRYPT*, pages 415–431, 1999.

[21] A. Rosen. A Note on the Round-Complexity of Concurrent Zero-Knowledge. In *CRYPTO*, pages 451–468, 2000.