Concurrent Zero Knowledge with Logarithmic Round-Complexity

Manoj Prabhakaran* Princeton University

Alon Rosen† Weizmann Institute of Science

Amit Sahai‡ Princeton University

Abstract

We show that every language in $\mathcal{NP}$ has a (black-box) concurrent zero-knowledge proof system using $\tilde{O}(\log n)$ rounds of interaction. The number of rounds in our protocol is optimal, in the sense that any language outside $\mathcal{BPP}$ requires at least $\tilde{\Omega}(\log n)$ rounds of interaction in order to be proved in black-box *concurrent zero-knowledge. The zero-knowledge property of our main protocol is proved under the assumption that there exists a collection of claw-free functions. Assuming only the existence of one-way functions, we show the existence of $\tilde{O}(\log n)$-round concurrent zero-knowledge arguments for all languages in $\mathcal{NP}$.*

1 Introduction

Zero-knowledge proof systems, introduced by Goldwasser, Micali and Rackoff [14] are efficient interactive proofs that have the remarkable property of yielding nothing beyond the validity of the assertion being proved. The generality of zero-knowledge proofs has been demonstrated by Goldreich, Micali and Wigderson [13], who showed that every NP-statement can be proved in zero-knowledge provided that one-way functions exist [16, 20]. Since then, zero-knowledge proofs have turned out to be an extremely useful tool in the design of various cryptographic protocols.

The original setting in which zero-knowledge proofs were investigated consisted of a single prover and verifier which execute only one instance of the protocol at a time. A more realistic setting, especially in the time of the Internet, is one which allows the concurrent execution of zero-knowledge protocols [8, 6]. In the concurrent setting, many protocols (sessions) are executed at the same time, involving many verifiers which may be talking with the same (or many) provers simultaneously (the so-called parallel composition considered in [12, 9, 10] is a special case). This setting presents the new risk of a coordinated attack in which an adversary controls many verifiers, interleaving the executions of the protocols and choosing verifiers' messages based on other partial executions of the protocol. Since it seems unrealistic (and certainly undesirable) for honest provers to coordinate their actions so that zero-knowledge is preserved, we must assume that in each prover-verifier pair the prover acts independently.

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

1.1 Previous Work

An informal argument concerning the difficulty of constructing round-efficient $c\mathcal{ZK}$ was given by Dwork, Naor, and Sahai in their paper introducing $c\mathcal{ZK}$ [6]. The first rigorous lower bound was given by Kilian, Petrank and Rackoff [19] who showed, building on the techniques of [12], that for every language outside $\mathcal{BPP}$ there is no 4-round protocol whose concurrent execution is simulatable in polynomial-time by a *black-box simulator*. (A black-box simulator is a simulator that has only black-box access to the adversarial verifier.) This lower bound was later improved by Rosen to seven rounds [22], and was further improved to $\Omega(\log n/\log\log n)$ rounds by Canetti, Kilian, Petrank and Rosen [5].

Even ignoring issues of round efficiency, it was not a-priori clear whether there exists $c\mathcal{ZK}$ protocols for languages outside of $\mathcal{BPP}$. Richardson and Kilian were the first to exhibit a family of $c\mathcal{ZK}$ protocols (parameterized by the number of rounds) for all languages in $\mathcal{NP}$ [21].

*Email: mp@cs.princeton.edu.

†Email: alon@wisdom.weizmann.ac.il.

‡Email: sahai@cs.princeton.edu.

The original analysis of the RK protocol showed how to simulate in polynomial-time $n^{O(1)}$ concurrent sessions only when the number of rounds in the protocol is at least n^ϵ (for some arbitrary $\epsilon > 0$). This analysis has been later improved by Kilian and Petrank [18], who 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., $\alpha(n) = \log\log n$).

In a recent breakthrough result, Barak [1] 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 that 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 size of the common input). This has been previously shown to be unachievable via black-box simulation [5] (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 [21, 18] (alas these protocols are not constant-round).

1.2 Our Results

In this work we close the gap between the known upper and lower bounds on the round-complexity of black-box $c\mathcal{ZK}$ [18, 5]. Specifically, assuming the existence of perfectly-hiding commitment schemes (which exist assuming the existence of a collection of claw-free functions [15]), we show that every language in $\mathcal{NP}$ can be proved in $c\mathcal{ZK}$ using only $\tilde{O}(\log n)$ rounds of interaction. Our main result is stated in the following theorem:

Theorem 1 (Main Theorem) *Assuming the existence of perfectly-hiding commitment schemes, there exists an $\tilde{O}(\log n)$-round black-box concurrent zero-knowledge proof system for every language $L \in \mathcal{NP}$ (that is, for every input x, the number of messages exchanged is at most $\tilde{O}(\log(|x|))$).*

We stress that our protocol retains its zero-knowledge property even under "full fledged" concurrent composition. That is, once the protocol is fixed it will remain zero-knowledge 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).

Notice that the above theorem completes the classification of the round-complexity of black-box $c\mathcal{ZK}$. Namely, by combining Theorem 1 with the lower bound of Canetti et al. [5], we have:

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

By relaxing the soundness requirement of the protocol to hold only against computationally bounded provers (that is, by considering so-called zero-knowledge arguments [14, 3]), we are able to achieve a similar result assuming only the existence of one-way functions, namely:

Theorem 2 *Assuming the existence of one-way functions, there exists an $\tilde{O}(\log n)$-round black-box concurrent zero-knowledge* argument *system for every language $L \in \mathcal{NP}$.*

We note that the lower-bound by Canetti et al. [5] applies also in the case of arguments.

1.3 Techniques

The proof of Theorem 1 builds on the protocol by Richardson and Kilian [21] and on the simulator by Kilian and Petrank [18]. 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 [21] which is both simpler and more amenable to analysis than the original version. In the rest of this section, we briefly sketch the ideas we use to obtain our main result.

Constructing zero-knowledge proofs for $\mathcal{NP}$ involves resolving a tension between the *soundness* and *zero knowledge* conditions: In (black-box) zero-knowledge proofs, the simulator can be thought of as a party that interacts with the verifier, but unlike the prover, the simulator must be able to convince the verifier of both true and false statements. To enable this, the simulator is given a "super power," namely the ability to "rewind" the verifier to an earlier state, and thus base its messages on future verifier messages. Very roughly speaking, zero knowledge proofs for $\mathcal{NP}$ have been constructed by inserting "rewinding opportunities" into protocols, which allow the simulator to "win" if it can base one of its earlier messages to the verifier on a future message received from the verifier. We stress that in order to successfully "exploit" a "rewinding opportunity," the simulator must take care not to "rewind" too far back, otherwise the information it learned from the verifier will no longer be useful. It is precisely this problem which

[1] $f(n) = \tilde{\Theta}(h(n))$ if both $f(n) = \tilde{O}(h(n))$ and $f(n) = \tilde{\Omega}(h(n))$.

makes simulation so difficult in concurrent zero knowledge, because rewinding one verifier may cause another verifier to be rewound "too much," requiring re-simulation, as first pointed out by [6].

The Richardson-Kilian (RK) protocol and Kilian-Petrank simulation. The basic idea of the Richardson-Kilian $c\mathcal{ZK}$ protocol [21] is to have a protocol with *many* rewinding opportunities, so that even if the simulator has to miss one opportunity, it will still get many more. Kilian and Petrank then showed that in fact, there exists a simulator for the RK protocol which has a very natural "oblivious" rewinding strategy [18] – in other words, the simulator's decisions of when and how much to rewind do not depend on the behavior of the verifiers, but are predetermined.

At this point, we note that a simple technical calculation shows that a single chance to exploit a rewinding opportunity results in only a constant probability that the simulator will "win." Thus, the simulator needs a superlogarithmic number of (roughly independent) chances to exploit rewinding opportunities in order to reduce its failure probability to a negligible fraction. Kilian and Petrank showed that in their oblivious rewinding strategy, throughout the simulation, every time a session of the protocol completes, the simulator will have chances to exploit at least $\Omega(k/\log n)$ rewinding opportunities, where k is the total number of rewinding opportunities in the protocol (the number of rounds in the protocol would then be $O(k)$). This implies that $\tilde{O}(\log^2 n)$ rounds suffice for concurrent simulation of the RK protocol.

The new ideas underlying this work. Unfortunately, the Kilian-Petrank argument does not extend to the case when $k = \tilde{O}(\log n)$. In fact, in such a case there may exists only few (i.e., $o(\log n)$) rewinding opportunities that can be exploited by the simulator.

We overcome this limitation by shifting our focus from the number of "exposed" rewinding opportunities in the protocol, to the total number of chances to exploit rewinding opportunities *counted with multiplicity*, in case the rewinding schedule permits multiple chances to exploit a single rewinding opportunity in the protocol. In fact, we show that the Kilian-Petrank oblivious rewinding strategy itself always yields roughly $k - O(\log n)$ such chances in total. This allows us to conclude that $\tilde{O}(\log n)$ rounds suffice. Furthermore, rather than relying on a subtle manipulation of conditional probabilities as done in previous work [21, 18], building on a suggestion of [17] we employ a direct counting argument to prove our claim. We essentially show directly that there can only be very few random coins on which our simulation fails, by arguing that for every choice of random coins on which our simulation fails, there must be superpolynomially more other choices for the random coins on which it does not.

1.4 Conclusions and an open problem

Our result (together with [5]) essentially completes the classification of the round-complexity of black-box $c\mathcal{ZK}$ (Corollary 1). Still, in light of Barak's recent result [1], 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 $\mathcal{NP}$.

2 Definition of $c\mathcal{ZK}$

We use the standard definitions of interactive proofs (and interactive Turing machines) [14, 11] and arguments (a.k.a computationally-sound proofs) [3]. In defining concurrent zero knowledge, we follow the original definition of [6], using a refinement due to [5].

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).

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 $\mathrm{view}^P_{V^*}(x)$ a random variable describing the content of the random tape of V^* and the transcript of the concurrent interaction between P and V^*.

Following [5], we overcome subtle issues that arise in the context of black-box $c\mathcal{ZK}$ by allowing the existence of a different simulator S_q for every V^* that runs at most $q(|x|)$ concurrent sessions. (This is in contrast to the customary definition of "stand-alone" black-box $\mathcal{ZK}$ in which it is required that there exists a "universal" simulator that works for all potential verifiers V^*.)

Definition 1 (Black-Box $c\mathcal{ZK}$) *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 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 $\{\mathrm{view}^P_{V^*}(x)\}_{x \in L}$ and $\{S_q(x)\}_{x \in L}$ are computationally indistinguishable.*

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

In this section we present 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 RK protocol [21] and uses the well known 3-round protocol for Hamiltonicity by Blum [2] 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 "secret" 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 "secret" (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 simplicity of presentation). In fact, the above property is satisfied by many other known protocols. Any one of these protocols could have been used as a building block for our construction.

3.1 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, $\{\sigma_{i,j}^0\}_{i,j=1}^k$, and $\{\sigma_{i,j}^1\}_{i,j=1}^k$, each consisting of k^2 random n-bit strings (this first message employs a perfectly-hiding commitment scheme and is called the initial commitment of the protocol). The sequences are chosen under the constraint that for every i, j the value of $\sigma_{i,j}^0 \oplus \sigma_{i,j}^1$ equals σ. This is followed by k iterations so that in the j^{th} iteration the prover sends a random k-bit string, $b_j = b_{1,j}, \ldots, b_{k,j}$, and the verifier decommits to $\sigma_{1,j}^{b_{1,j}}, \ldots, \sigma_{k,j}^{b_{k,j}}$.

In the second stage, the prover and verifier engage in the 3-round protocol for Hamiltonicity, where the "secret" 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 $\sigma, \{\sigma_{i,j}^{1-b_{i,j}}\}_{i,j=1}^k$ that were not revealed in the first stage). The protocol is depicted in Figure 1.

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 negligible soundness error). However, knowing the value of σ in advance allows the simulation of the protocol: Whenever the simulator may cause the verifier to reveal both $\sigma_{i,j}^0$ and $\sigma_{i,j}^1$ for some i, j (this is done by the means of *rewinding* the verifier after the values $\sigma_{1,j}^{b_{1,j}}, \ldots, \sigma_{k,j}^{b_{k,j}}$ 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 $\sigma = \sigma_{i,j}^0 \oplus \sigma_{i,j}^1$ (which, as we said, is obtained before entering the second stage).

First stage:

$V \to P$: Commit to σ, $\{\sigma_{i,j}^0\}_{i,j=1}^k$, $\{\sigma_{i,j}^1\}_{i,j=1}^k$.
$\sigma_{i,j}^0 \oplus \sigma_{i,j}^1 = \sigma$ for every i, j.

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

$P \to V$: Send $b_{1,j}, \ldots, b_{k,j} \xleftarrow{r} \{0,1\}^k$.

$V \to P$: Decommit to $\sigma_{1,j}^{b_{1,j}}, \ldots, \sigma_{k,j}^{b_{k,j}}$.

Second stage:

$P \to V$: Send first message of Hamiltonicity protocol.

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

$P \to V$: Answer according to the value of σ.

Figure 1. Our $c\mathcal{ZK}$ protocol. The first stage is independent of the common input and consists of k iterations. The second stage consists of a 3-round proof of Hamiltonicity.

3.2 The Simulator

Let $(\mathrm{V}0), (\mathrm{P}1), (\mathrm{V}1), \ldots, (\mathrm{P}k), (\mathrm{V}k)$ denote the $2k+1$ first stage messages in our protocol and let $(\mathtt{p1}), (\mathtt{v1}), (\mathtt{p2})$ 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 $(\mathrm{V}j)$ message, it "goes back" to some point preceding the corresponding $(\mathrm{P}j)$ message and "re-executes" the relevant part of the interaction until $(\mathrm{V}j)$ is reached again.

Note that, if the simulator manages to receive $(\mathrm{V}j)$ as answer to two *different* $(\mathrm{P}j)$ messages (due to rewinding) the simulator has obtained both $\sigma_{i,j}^0$ and $\sigma_{i,j}^1$ for some $i \in \{1, \ldots, k\}$. If this happens in even one of the rounds j in the first stage, then it reveals the verifier's "secret" (which is equal to $\sigma_{i,j}^0 \oplus \sigma_{i,j}^1$). Once the secret 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 $(\mathrm{P}j)$'s uniformly at random. Since the length of the $(\mathrm{P}j)$ messages is super-logarithmic, the probability that *any* two $(\mathrm{P}j)$ messages sent during the simulation are equal is negligible.

Motivating discussion. The binding property of the initial commitment guarantees us that, once $\sigma_{i,j}^0$ and $\sigma_{i,j}^1$ have been revealed, the verifier cannot "change his mind" and decommit to $\sigma \neq \sigma_{i,j}^0 \oplus \sigma_{i,j}^1$ on a later stage. However, this remains true *only* if we have not rewound past the initial commitment. As observed by Dwork et al. [6], 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

Input: $(\ell, \mathsf{hist}, \mathcal{T})$

Bottom level ($\ell = 1$)**:**

- Uniformly choose a first stage prover message p, and feed V^* with $(\mathsf{hist}, \mathrm{p})$.
- Store V^*'s answer v, in $\mathcal{T}$.
- Output (p, v), $\mathcal{T}$.

Recursive step ($\ell > 1$)**:**

- Set $(\tilde{\mathrm{p}}_1, \tilde{\mathrm{v}}_1, \ldots, \tilde{\mathrm{p}}_{\ell/2}, \tilde{\mathrm{v}}_{\ell/2})$, $\mathcal{T}_1 \leftarrow$SIMULATE($\ell/2$, hist, $\mathcal{T}$).
- Set $(\mathrm{p}_1, \mathrm{v}_1, \ldots, \mathrm{p}_{\ell/2}, \mathrm{v}_{\ell/2})$, $\mathcal{T}_2 \leftarrow$ SIMULATE($\ell/2$, hist, $\mathcal{T}_1$).
- Set $(\tilde{\mathrm{p}}_{\ell/2+1}, \tilde{\mathrm{v}}_{\ell/2+1}, \ldots, \tilde{\mathrm{p}}_{\ell}, \tilde{\mathrm{v}}_{\ell})$, $\mathcal{T}_3 \leftarrow$SIMULATE($\ell/2$, $(\mathsf{hist}, \mathrm{p}_1, \mathrm{v}_1, \ldots, \mathrm{p}_{\ell/2}, \mathrm{v}_{\ell/2})$, $\mathcal{T}_2$).
- Set $(\mathrm{p}_{\ell/2+1}, \mathrm{v}_{\ell/2+1}, \ldots, \mathrm{p}_{\ell}, \mathrm{v}_{\ell})$, $\mathcal{T}_4 \leftarrow$ SIMULATE($\ell/2$, $(\mathsf{hist}, \mathrm{p}_1, \mathrm{v}_1, \ldots, \mathrm{p}_{\ell/2}, \mathrm{v}_{\ell/2})$, $\mathcal{T}_3$).
- Output $(\mathrm{p}_1, \mathrm{v}_1, \ldots, \mathrm{p}_{\ell}, \mathrm{v}_{\ell})$, $\mathcal{T}_4$.

Figure 2. The rewinding strategy of the simulator. Even though messages $(\tilde{\mathrm{p}}_1, \tilde{\mathrm{v}}_1, \ldots, \tilde{\mathrm{p}}_{\ell}, \tilde{\mathrm{v}}_{\ell})$ do not explicitly appear in the output, some of them do appear in the table $\mathcal{T}_4$.

session all $\sigma_{i,j}^{b_{i,j}}$ values that we have gathered in this session become irrelevant). Consequently, the simulator may find himself doing the same amount of "work" again.

The rewinding strategy. The big question is how to design a simulation strategy that will manage to overcome the above difficulty. In this work we follow the approach taken by Kilian and Petrank [18] and let the simulator determine the order and timing of its rewindings *obliviously* of the concurrent scheduling.

The rewinding strategy of our simulator is specified by the SIMULATE procedure. The goal of the SIMULATE procedure is to supply the simulator with V^*'s "secret" for each session before reaching the second stage in the protocol. As discussed above, this is done by rewinding the interaction with V^* while trying to make the verifier answer two different challenges $(\mathrm{P}j)$.

The timing of the rewindings performed by the SIMULATE procedure depends only on *the number of 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 SIMULATE 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 SIMULATE 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 is a transcript of the *current* thread 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).[2]

The simulation is performed by invoking the SIMULATE procedure with the appropriate parameters. Specifically, whenever the schedule contains $m = \mathrm{poly}(n)$ sessions, the SIMULATE procedure is invoked with input $(m(k+2), \phi, \phi)$ (where $m(k+2)$ is the total number of verifier messages in a schedule of m sessions). The SIMULATE procedure is depicted in Figure 2.

If the simulation reaches the second stage (the main $\mathcal{ZK}$ proof part) in the protocol at any time, without the secret having been extracted, the simulator commits to a random string.[3] But if subsequently the verifier sends the message (`v1`) to reveal a secret consistent with its earlier messages, the simulator "gets stuck," i.e., it cannot continue the proof as in the original protocol and keep it indistinguishable from an actual proof. Then it gives up the entire simulation and outputs $\perp$.

A **slot** in the simulation consists of two messages: a prover message and the next verifier message. The two messages of a slot may be from different sessions; but for each verifier message, the next message in the simulation is the simulated prover's reply to it (in the same session). The simulator will rewind to points *between* slots. A **session**

[2]The messages stored in $\mathcal{T}$ are used in order to determine the verifier's "secret" according to "different" answers to $(\mathrm{V}j)$.

[3]If the secret has been extracted, it is used to manufacture a message which helps the simulator complete the proof later.

during the run of the simulator is identified by the slot in the simulation where the first message of the session, namely the initial commit message (V0) from the verifier, arrives (thereby ending that slot).

3.3 Blocks

We define a **block** as the part of execution of the simulator within an invocation of the SIMULATE procedure. The smallest block is a single slot, corresponding to the base of recursion. The other blocks are composed of four blocks of the next lower level.

Figure 3 illustrates one block. The way in which the history is passed to the lower level invocations tie them together as shown. The invocation of the (lower level) block called $1'$ in the top thread corresponds to the first (look-ahead) call. It is truncated immediately (i.e., its history is not continued further) as the simulator rewinds when the call returns; the second call (block marked 1) starts off with the same history as the first one, as indicated by the first fork in the thread; the resulting thread continues, as the SIMULATE procedure goes to the next *half* in the recursion. Again the first call is truncated by rewinding, and the history from the second call is passed to the outside of the block.

Of these four blocks, the first one (in Figure 3, $1'$) is called the **look-ahead block** of the second one (1). Similarly the third block ($2'$) is the look-ahead block of the fourth one (2). Every block except the one at the top-most level either *is* a look-ahead block or *has* a look-ahead block. A block may *contain* another block of a lower level, but no two blocks can ever overlap otherwise.

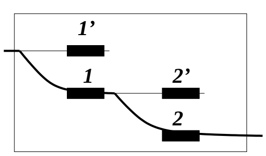

Figure 3. The Threads of execution of the simulator. The shaded blocks hide the threads in the recursive calls. The block returns the messages from blocks 1 and 2.

Figure 4 illustrates the "threads" in the simulation. A thread refers to a path from left to right in such a figure. A thread from the initial point of simulation, up to a slot x corresponds to the transcript of the simulated protocol when the simulation reaches x.

4 High Level Analysis of the Simulator

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.

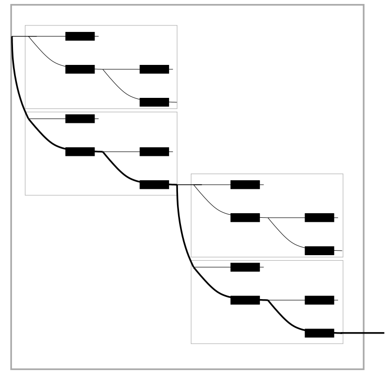

Figure 4. A block in the execution of the simulator. The shaded boxes correspond to blocks two levels below the block shown. The lines indicate the different "threads" of execution taken by the simulator.

4.1 The simulator runs in polynomial-time

Each invocation of the SIMULATE procedure with parameter $\ell > 1$ involves four recursive invocations of the SIMULATE 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 $\mathrm{poly}(n)$. Thus, the recursive work $W(m\cdot(k+1))$, that is invested by the SIMULATE procedure in order to handle $m \cdot (k+1)$ (first stage) verifier messages satisfies $W(m \cdot (k+1)) \leq (m \cdot (k+1))^2 \cdot \mathrm{poly}(n) = \mathrm{poly}(n)$.

4.2 The simulator's output is "correctly" distributed

Indistinguishability of the simulator's output from V^*'s view (of $m = \mathrm{poly}(n)$ concurrent interactions with P) is shown assuming that the simulator does not "get stuck" and output $\perp$ during its execution (see the next section). 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 is proved based on the definition of the simulator's actions. (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 is proved based on the fact that the verifier cannot feasibly distinguish between the prover using a real witness and the prover cheating by knowing the secret string used by the verifier. This follows from the security of the commitment scheme used by the prover inside the ZK proof system employed in the second stage of the protocol.

A formal proof can be given using a hybrid simulator which differs from the original simulator only in that for each session s, it knows the witness for $x_s \in L$, and uses that for the second stage. Though the hybrid simulator does not use the entries in the Solution Table, it also fails if it reaches the last message in an unsolved session. In the sequel, we shall analyse this hybrid simulator.

4.3 The simulator (almost) never "gets stuck"

This is the most involved part of the proof. What is required is to show that whenever a session in the simulation reaches the second stage of the protocol, the simulator has already **solved** it – i.e., managed to obtain the value of the verifier's "secret" corresponding to that session (if there is a valid secret for the session) with overwhelming probability.

The adversarial verifier is said to **succeed** on a random tape of the simulator, if the simulator gets stuck in some session s. Recall that a session is specifed by the "start-slot." In contrast, the simulator is said to **secure** a session if it does not get stuck in that session (but the simulator may still get stuck in some other session). We would like to bound the probability that the adversary succeeds in any session.

We shall bound this probability for each setting of the coin flips of the verifier. So now onwards we fix the coin flips of the verifier and consider the probability with respect to the coin-flips of the simulated prover only. So given the *random tape* of the simulator (i.e., the randmoness used to generate the prover messages), the entire execution of the simulator is determined.

To bound the probability that the adversary succeeds we have to bound the number of random tapes on which the adversary succeeds. We shall show that for every random tape on which the adversary succeeds with respect to a particular session s, there are many other tapes with which that is not the case (taking care not to double-count the tapes). In the sequel we restrict ourselves to random tapes which cause the simulator to never pick two identical challenges; this does not affect the probabilities by more than a negligible fraction.

Lemma 1 *Let $\mathcal{R}$ be the set of all random tapes used by the simulator. There exists a mapping $f : \mathcal{R} \to 2^{\mathcal{R}}$ such that for every $R \in \mathcal{R}$, if the adversary succeeds on R for a session s, then*

1. $\forall R' \in \mathcal{R}\backslash\{R\}$, $f(R) \cap f(R') = \phi$
2. $|f(R)| \geq 2^{k-O(h)}$, *where h is the maximum depth of recursion of the simulator.*
3. $\forall R' \in f(R)\backslash\{R\}$, *the simulator secures s on random tape R'.*

We shall sketch the proof of this lemma in the next section, but before that note that it achieves our goal. Since all the random tapes are equally probable, the next lemma follows immediately from Lemma 1.

Lemma 2 *The probability that the adversary succeeds for a given session s is at most $2^{-(k-O(h))}$.*

Now we prove the assertion of this section:

The number of possible sessions is at most the number of slots, and therefore $poly(n)$. (When the simulator is simulating a concurrent session involving at most ℓ messages, the number of slots in the simulation is at most ℓ^2.) Thus by union bound, Lemma 2 implies that the probability of the simulator getting stuck (i.e., that of the adversary succeeding with respect to some session) is at most $poly(n)2^{-(k-O(h))}$. This is negligible in n as we take $k = \omega(\log n)$, and $h = O(\log n)$. The latter follows, because h, the depth of the recursion, is logarithmic in the number of slots.

5 Proof Sketch of Lemma 1

Here we sketch the proof of Lemma 1. (A more complete proof is included in the full version of this paper).

5.1 Overview

Let the adversary succeed on the random tape (deck) R, in session s which starts at a slot `start` (when the message $(\mathrm{V}0)$ arrives) and ends at the slot `stop` (at whose beginning the simulator gets stuck unable to send (`p2`)). The map f is established by demonstrating a procedure which takes R and outputs at least $2^{k-O(h)}$ distinct tapes in which the simulator secures s. To show that $f(R) \cap (R') = \phi$ we will demonstrate an inverse procedure which takes any tape in $f(R)$ and gives back R.

The random-tape of the simulator can be considered a concatenation of the random strings used at each slot (normalized to the same length). Imagine that each such random string is a *card* drawn from a large universe, and the random tape is a *deck* of such cards. Then, each tape output by the procedure is obtained by shuffling the input deck. That is, the order in which the different random strings are used is changed, but the random strings themselves are not altered.

Spans. Suppose the verifier sends a correct message $(\mathrm{V}j)$ in session s in response to prover's challenge in a message $(\mathrm{P}j)$. The prover's challenge $(\mathrm{P}j)$ starts a slot `x` and the subsequent verifier's answer ends a slot `y` (the two slots may be the same). The set of slots along the `x`-`y` thread, between (and inclusive of) `x` and `y` is called a **span**.

Let us call the segment of the thread between, but not including, the slots `start` and `stop` the `start-stop` segment. Since the simulator reaches (`p2`) at the slot `stop`, within the `start-stop` segment the prover (simulator) must send the k challenges $(\mathrm{P}1), \ldots, (\mathrm{P}k)$, and the verifier must properly answer in messages $(\mathrm{V}1), \ldots, (\mathrm{V}k)$. Thus the `start-stop` segment is partitioned into k spans.

A span is called **good** if the challenge at the beginning of the span is correctly answered in the verifier message at the end of the span. With the random tape R all the k spans in the `start-stop` segment are good, and there are no other good spans.

Shuffling Threads. The random strings (cards) in all the slots along a thread fixes the execution of that thread[4] So if we move the randomness in a thread (or in a segment thereof) to some other thread (or its segment), the execution in the latter will be identical to that of the former before the change, as long as the two threads or segments in question fork off from the same point.

Suppose that there is a look-ahead thread that starts after the slot `start`, but is not as long as the `start-stop` segment, and that the execution in the `start-stop` segment were to be advanced to that thread. Then if the latter thread is long enough, at least one of the k good spans originally in the `start-stop` segment, with messages $(\mathrm{P}j)$ and $(\mathrm{V}j)$ say, will appear in that thread. If that happens the simulator would have secured the session (i.e., it will not get stuck in that session) by the time it rewinds out of that thread, because if the verifier answers a later challenge $(\mathrm{P}j)$ correctly (since we are assuming that no two challenges are the same), it can successfully extract the secret σ for session s.

The above observation suggests that from a random tape in which the adversary succeeds for a session s, just by swapping the randomness of the "crashing thread" with that of many other appropriate threads, we get random tapes in which the session s is secured. But the resulting mapping is not invertible. For our counting argument to go through smoothly, we do a slightly more sophisticated mapping, as explained in Section 5.2.

5.2 Shuffling by Swapping Blocks

The aim of the shuffling procedure is to establish that there are some non-overlapping "swappable" segments (each containing one good span) in the `start-stop` thread, and for each segment there are many segments with which it can be swapped. Further each of these swappings can be carried out independently one after the other, and still the entire swapping remains invertible. We shall show that there are at least $2^{k-O(h)}$ distinct tapes that can be produced by these swappings, all of which will allow the simulator to secure session s.

The shuffling procedure can be described in terms of the block structure of the execution of the simulator as described in Section 3.3.

A block is said to be **swappable** if it is the smallest block containing a good span, it does not properly contain any other good span, and it does not contain the slots `start` or `stop`. Note that the minimal block containing a good span, as long as it does not contain `start` or `stop`, either is a swappable block, or contains a swappable block. The part of the `start-stop` segment inside a swappable block is called a swappable segment. A swappable segment will be swapped with some other segments as described shortly.

The swappable blocks are ordered according to the order of their associated spans. A block is called an **allied block** of a swappable block B if (a) it contains (or is) B, but does not contain the previous swappable block, and (b) does not contain the `start` or `stop` slots.

A swappable block B is an allied block of itself. At each higher level, there is one allied block of B, namely the one containing the allied block of the lower level, up to the level at which the block containing B also contains its previous swappable block or `start` or `stop`. The number of allied blocks of B will be denoted by t_B.

Note that since an allied block cannot contain the `start` or `stop` slots, the `start-stop` segment enters the block and leaves it. Such a block cannot be a look-ahead block (as defined in Section 3.3), because a look-ahead block cannot have any thread continuing out of it. Thus every allied block *has* a look-ahead block. We are now ready to outline the shuffling strategy.

Basic-Shuffle. The entire shuffling of a thread is composed of many *basic-shuffles*, each of which works on a swappable block. The basic-shuffle is a hierarchical procedure involving the allied blocks of a swappable block. We illustrate this through an example. A formal description is available in the full version of this paper.

Figure 5 shows how the swappable block marked 1 in the lower thread APQ is shuffled up to the upper thread ABC. The block marked 1^* (in thread ABC) is called the *target*. The allied blocks of 1 are blocks marked $1, 2$ and 3, and the blocks containing 1^* at the corresponding levels are marked $1^*, 2^*$ and 3^*. First, block 1 is swapped with its look-ahead block $1'$, as 1^* is a look-ahead block. But blocks 2 and $2'$ are not swapped, because 2^* is not a look-ahead block. Finally blocks 3 and $3'$ are swapped with each other as 3^* is a look-ahead block, completing the basic-shuffle.

For a swappable block B, by choosing at each of the t_B levels whether to swap the allied block with its look-ahead block or not, the above strategy specifies 2^{t_B} targets with which B can be shuffled (one of them being itself). (In our example this number is 2^3.)

[4]Recall that the hybrid simulator that we are analysing does not make use of the table $\mathcal{T}$. Also, the verifier is assumed to be deterministic.

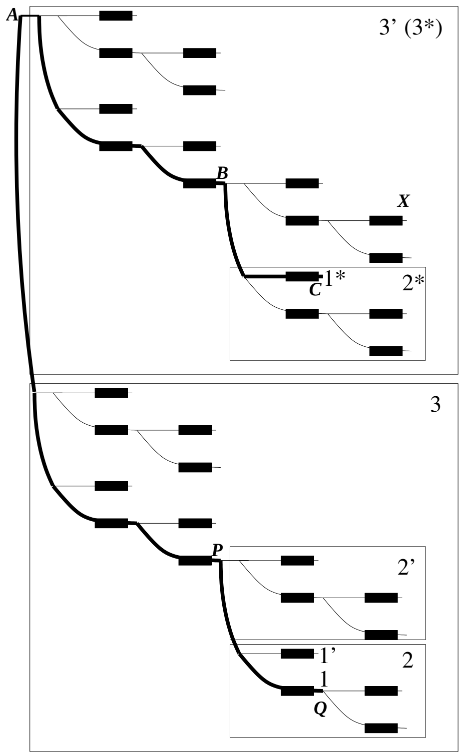

Figure 5. A basic-shuffle can move the swappable block 1 to the block 1^*, one of its 8 target blocks.

Inverting a Basic-Shuffle. Suppose the r-th swappable block (ordered according to the order of the spans associated with the swappable blocks) with the original random tape R is B, and it was shuffled to a target block B^* to get the tape R'. Inverting this basic-shuffle involves recovering R from R', as well as identifying the target block B^*. The latter ensures that the tapes obtained by shuffling B with the different targets of B are indeed distinct.

We note that shuffling B does not change anything outside the outer-most allied block and its look-ahead block. In particular, all the previous $r-1$ swappable blocks in the simulation remain unchanged. Also, the shuffling makes the execution of B^* identical to that of B before the shuffle. Further, the execution of every block till B^* with R', is identical to that of some block before B, with the tape R. Thus B^* becomes the r-th swappable block after the shuffle. This makes it possible to identify the target block of the shuffle by inspecting R'. This is crucially used for inverting the mapping.[5]

[5] Note that if a simpler shuffling strategy of exchanging the randomness in the two threads to be shuffled is used, this may no longer be true. In our illustration, if we just swap the randomness in the two threads ABC and APQ, the execution in the segment BX for instance, will be unpredictable (and in particular may introduce a good span in BX and introduce an associated swappable block). This is because, in the original random tape there was no thread with the randomness same as in the thread ABX after the swap. But when the swap is carried out systematically as illustrated above, every thread before ABC was already present in the original setting, and none of them had a good span in them.

Having identified B^*, we are ready to start our unshuffling. We set B^* as the **current block**. Next we check if it is a look-ahead block or not. If it is, then it means it reached there due to a swap. So it is swapped to become a non-look-ahead block, and the current-block is also changed to the resulting block. Then we check if the block containing the current-block is an allied block (i.e., we check if it contains the $r-1$-th swappable block or the `start` or `stop` slots). If it is, we make it the current-block and repeat by checking if it is a look-ahead block, and if necessary swapping. We continue this way until the current block becomes the maximal allied block. It is not hard to see that this operation undoes the basic-shuffle which takes B to B^*.

Shuffling the entire thread. To shuffle the entire thread, the above basic-shuffle procedure is carried out on each of the swappable blocks. This is done from right to left, i.e., the last (in simulation order) swappable block is shuffled first, then the previous one, and so on. The first basic-shuffle does not change the execution of any of the previous swappable blocks (as all the segments involved in a swapping occur *after* the previous swappable segments). Then the next swappable block is swapped and so on. This ensures that the unswapping can be done, in the reverse order, first swapping back the earliest swappable segment, then the next and so on.

5.3 Counting Swaps

By the above, the random tape R can be invertibly mapped to $\Pi_B 2^{t_B} = 2^{\sum_B t_B}$ tapes, where the summation is over all swappable segments B. So to prove condition (2) of Lemma 1 we need to count the total number of allied blocks of all swappable blocks for the random tape R.

If B is a block which does not contain `start` or `stop`, then we have the following: (1) For every good span q, if B is the smallest block containing q, then B is either a swappable block or contains a swappable block. (2) B is an allied block of the first swappable block that it contains, if it contains at least one swappable block. (3) Therefore, B is an allied block (of the first swappable block that it contains) if it contains at least one good span.

Suppose we map each of the k good spans in the `start`-`stop` segment to the smallest block containing it. Then, a block B can have at most one span mapped to it; this is because a span mapped to B must include slots in both halves of the B, and the k spans are all disjoint. Thus there are at least k blocks which contain at least one good span. Of these, at most h blocks contain the slot `start`, and similarly for `stop`. Thus by Observation 3 above, at least $k-2h$

blocks are allied blocks, there by proving condition (2) of Lemma 1.

5.4 Securing the session

Out of all the new random tapes obtained by the strategy above, there is one which is identical to the original tape R. In any other tape $R' \in f(R)$, there is one good span outside the `start-stop` segment, in a look-ahead block, namely the target of the left-most swappable block swapped. As described earlier, if the call to SIMULATE returns from that look-ahead segment to a point after the `start` slot, the simulator will be able to find the secret of the verifier (conditional on all the challenges of the simulator being distinct) the next time it goes through the same round in that session. But we know that the call to SIMULATE will return because the block swapped did not contain the slot `stop`, and that it will return to a point after the `start` slot because it did not contain the slot `start`. Thus on all random strings obtained above except for the original adversarially given one, the simulator indeed secures the session which began at `start`. (The simulation may still get stuck, but only for a different session. The union bound argument given in Section 4.3 shows that this can't happen too often, and the proof goes through.) This completes the proof of Lemma 1

6 Acknowledgements

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 [17] for an analysis which avoids the dangers involved in earlier similar analyses involving subtle arguments based on conditional probability.

We are grateful to Oded Goldreich for his support, for enlightening conversations and for giving many useful remarks on previous manuscripts. We are also grateful to Moni Naor for discussions leading to the new $c\mathcal{ZK}$ protocol. Thanks also to Uri Feige, Ronen Shaltiel and Erez Petrank for helpful discussions.

References

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

[2] 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.

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

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

[5] 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.

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

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

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

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

[10] O. Goldreich. Concurrent Zero-Knowledge with Timing – Revisited. To appear, in *34th STOC*, 2002.

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] J. Kilian. Personal Communication

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

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

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

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

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