

Lower Bounds for Zero Knowledge on the Internet

Joe Kilian *

Erez Petrank †

Charles Rackoff ‡

Abstract

We consider zero knowledge interactive proofs in a richer, more realistic communication environment. In this setting, one may simultaneously engage in many interactive proofs, and these proofs may take place in an asynchronous fashion. It is known that zero-knowledge is not necessarily preserved in such an environment; we show that for a large class of protocols, it cannot be preserved. Any 4 round (computational) zero-knowledge interactive proof (or argument) for a non-trivial language L is not black-box simulatable in the asynchronous setting.

1. Introduction

Zero knowledge [16] turned out to be a useful tool for many cryptographic applications. Many works have studied the numerous uses of zero knowledge proofs, and many other works have suggested how to improve the efficiency of these proofs. However, most of these works considered only the case where the proof stands alone, disconnected from the computing environment. An interesting question, which naturally arises these days, is how robust the notion of zero knowledge is in a broader setting. In particular, many computers today are connected through networks (from small local area networks to the entire Internet) in which connections are maintained in parallel asynchronous sessions. It is common to find several connections (such as FTP, Telnet, An internet browser, etc.) running together on a single workstation. Can zero knowledge protocols be trusted in such an environment?

The robustness of zero knowledge has been studied before in the “simple” case of parallel repetitions. It is often desirable to run a probabilistic protocol many times in parallel, usually in order to reduce the expected error of a single run. The alternative of running these protocols sequentially has a cost of increasing the number of rounds

and is considered inefficient. It had been noted by several researchers that even in the parallel repetitions case the zero knowledge property does not necessarily hold. Goldreich and Krawczyk [12] proved a general lower bound: Any language that has a three round (black-box) zero-knowledge interactive proof with a negligible error probability (as can be obtained by parallel repetitions) is in BPP. Thus, for example, unless Graph Isomorphism is in BPP, the protocol of [13] for Graph Isomorphism does not remain zero knowledge when run many times in parallel. Several papers have dealt with this problem, usually by letting the verifier commit on it’s (non-adaptive) questions in advance [1, 3, 11, 8].

Our initial feeling was that the protocols that keep their zero knowledge property when run in parallel should also remain zero knowledge even in a multi-session asynchronous environment. However, in this paper, we give some indications that this is not always the case.

Let us say a few words about what we need to show. In order to show that a protocol is zero knowledge in a modern networking environment, one must provide a proof that even within this complicated environment the protocol is still zero knowledge. But for us matters are simpler. We don’t need to cover all facets of a networking environment. We only have to show that zero knowledge may fail in a specific setting that is part of this environment. Namely, the environment may be more hostile to the protocol than the specific case we study, but since a protocol fails even with a benign setting, it is definitely not zero knowledge when we extend the power of the environment. In particular, a networking environment may have various protocols running, multiple sessions, more than two parties involved, asynchronous setting, etc.

We show that zero knowledge may fail for a large class of protocols, even if we only run a single protocol between only two parties. We exploit the asynchronicity and the existence of multiple sessions. Clearly, if we also add other features, such as additional parties and other protocols running in parallel, then the security problems can only be amplified.

We show that any four-round black-box zero-knowledge proof with perfect completeness is not zero knowledge even in a very benign setting: The setting in which the protocol is run many times in an asynchronous environment, and an ad-

*NEC Research Institute. E-mail: joe@research.nj.nec.com

†IBM Haifa Research Lab, MATAM, Haifa 31905, Israel. Email: erezp@haifa.vnet.ibm.com.

‡Dept. of Computer Science, University of Toronto, Toronto, Ontario, Canada M5B 3g4. Email: rackoff@cs.toronto.edu.

versary (or the verifier) gets to choose which message gets to be delivered next. Actually, the setting is even more benign: We set a specific schedule, known in advance to both prover and verifier. Still, if the protocol can be (black-box) simulated, then the language must be in BPP.

Theorem 1 *Suppose that (P, V) is a 4-message interactive proof or argument for L with perfect completeness and error at most $\frac{1}{2}$ and suppose that (P, V) can be black-box simulated in the asynchronous setting. Then $L \in BPP$.*

The $\frac{1}{2}$ in the above statement may be replaced by any constant less than 1, with essentially no change to our proofs; with slight care, any error bounded away from 1 by a nonnegligible factor may be accommodated.

Note that we get a separation between protocols that remain zero knowledge even under parallel repetition and protocols that remain zero knowledge in an asynchronous setting. Assuming the existence of one-to-one one-way functions, there exist 4-message (computational) zero knowledge arguments for all NP languages [8]. However, there are no 4-message zero knowledge arguments for languages outside of BPP that are black box simulatable in the asynchronous setting.

Some words on the terminology we are using. By zero knowledge we mean *computational* zero knowledge, i.e., the distribution output by the simulation is polynomial-time indistinguishable from the distribution of the views of the verifier in the original interaction. The prover may be infinitely powerful (i.e., an interactive proof) or it may be computationally bounded (i.e., an argument). We consider black box zero knowledge as defined by Goldreich and Oren [19, 14], and refined in [12].

1.1. Related work

There is a vast literature in the distributed computing community dealing with asynchronicity. Within the cryptography community, Beth and Desmedt [4] discuss such asynchronous attacks in the context of identification protocols, and proposed timing methods to defend against such attacks. They showed that by agreeing to perform steps of the protocol at certain precise times, and assuming that an adversary must take a certain amount of time to respond to or reroute a message it received, one can thwart its ability to schedule the message sequences in a detrimental fashion. Dwork, Naor and Sahai [6] consider the role of asynchronous attacks on zero-knowledge protocols. They give 4-round zero-knowledge protocols for NP in an asynchronous multi-session environment, assuming public keys and a much weaker constraint on the synchrony of honest players: there exist a pair (α, β) , where $\alpha \leq \beta$, such that when a good player has observed the passage of β units of

time, then every other good player has observed the passage of at least α units of time. Dwork and Sahai [7] reduce (but do not eliminate) the timing constraints required by their defense. Our result is complementary to theirs, illustrating why it is difficult to achieve zero-knowledge in the asynchronous setting without using such an augmented model. In another complementary result, Richardson and Kilian [20] show how to construct arguments with polynomially many rounds which are asynchronously simulatable against polynomially many simultaneous runs.

Feige and Shamir showed that witness indistinguishability is preserved also in the asynchronous setting [9].

The basic framework of our proof uses the ideas developed by Goldreich and Krawczyk [12]. Following their technique, we use a good simulator S of an interactive proof (or argument) (P, V) for a language L to create an efficient prover P_S that causes V to accept reasonably often on inputs in L .

1.2. Guide to the paper

In Section 2 we discuss black-box simulatability and the framework used by Goldreich and Krawczyk. In Section 3 we show how to convert an asynchronous simulator into an efficient prover. In Sections 4 and 5 we analyze the success probability of this prover. These sections assume a slightly restricted verifier: the verifier can decide whether or not to accept based on the conversation thus far, and not on its random coins. In Section 6 we describe how to eliminate this restriction.

2. Preliminaries

2.1. Black-Box Zero-Knowledge

The initial definition of zero-knowledge [15] required that for any probabilistic polynomial time verifier \hat{V} , a simulator $S_{\hat{V}}$ exists that could simulate \hat{V} 's view. Goldreich and Oren [19, 14] propose a seemingly stronger, "better behaved" notion of zero-knowledge, known as *black-box* zero-knowledge. The basic idea behind black box zero-knowledge is that instead of having a new simulator $S_{\hat{V}}$ for each possible verifier, we have a single probabilistic polynomial time simulator S that interacts with each possible \hat{V} . Furthermore, S is not allowed to examine the internals of \hat{V} , but must simply look at \hat{V} 's input/output behavior. That is, it can have conversations with \hat{V} and use these conversations to generate a simulation of \hat{V} 's view that is computationally indistinguishable from \hat{V} 's view of its interaction with P .

More formally, Goldreich and Krawczyk give the following version of this definition (notation changed for compat-

ibility with our own), which avoids certain trivial problems in the original expositions.

Definition 1 ([12], following [19, 14]) *An interactive proof $\langle P, V \rangle$ is called blackbox simulation zeroknowledge if for every polynomial p , there exists a probabilistic expected polynomial time oracle machine S_p such that for any polynomial size verifier \hat{V} that uses at most $p(n)$ random coins on inputs of length n , and for $x \in L$, the distributions $\langle P, \hat{V} \rangle(x)$ and $M_p^{\hat{V}}(x)$ are polynomially indistinguishable.*

Remarks: In the above definition, \hat{V} may be thought of as a circuit (pedantically, a circuit family) that has access to random bits. The size of \hat{V} refers to \hat{V} 's circuit size. It follows that V^* runs in polynomial time, which is necessary for a meaningful notion of computational zero knowledge, though not for statistical and perfect zero knowledge. By polynomially indistinguishability, we informally mean that no computationally bounded distinguisher can correctly guess whether a random sample came from $\langle P, \hat{V} \rangle(x)$ or $S_p^{\hat{V}}(x)$ with probability greater than $\frac{1}{2} + 1/|x|^c$ for any c , as $|x|$ grows sufficiently large. For statistical and perfect zero knowledge, we allow for unbounded distinguisher; equivalently, we require that the statistical difference between the distribution be negligible (less than $1/|x|^c$ for any c).

In our paper, as in [12], we only consider deterministic \hat{V} ; for this class of adversaries, the above definition requires the existence of a single universal simulator, S .

At first glance, the limitations on S may seem to force S to be as powerful as a prover. However, S has important advantages over a prover P , allowing it to perform simulations in probabilistic polynomial time. First, it may set \hat{V} 's coin tosses as it wishes, and even run \hat{V} on different sets of coin tosses. More importantly, S may conceptually “back up” \hat{V} to an earlier point in the conversation, and make different statements. This ability derives from S 's control of \hat{V} 's coin tosses; since \hat{V} otherwise operates deterministically, S can rerun it from the beginning, exploring different branches of the conversation tree.

Indeed, all known proofs of zero-knowledge construct black-box simulations. There is no way known to make use of a verifier's internal state, nor to customize simulators based on the description of \hat{V} other than by using it as a black box.¹ Thus, given the current state of the art, an impossibility result for black-box zero-knowledge seems to preclude a positive result for the older definitions of zero-knowledge.

¹As one slight exception, [17] proves security against space-bounded verifiers by considering the internal state of the verifiers. However, these techniques do not seem applicable to more standard classes of verifiers.

2.2. Black-box verifiers with private random functions

Following [12], we consider verifiers \hat{V} that have access to a private random hash function H , that is wired into them and is not directly accessible to the simulator (note that \hat{V} is deterministic in that it doesn't use an external source of randomness; its *construction* is randomized). That is, the simulator may gain only indirect access to H , by observing \hat{V} 's input/output behavior. For convenience, we assume that for any polynomially bounded n and m , H will take an n -bit input and return an m -bit output. In practice, H will be defined for big enough n and m , and its inputs (if short) will be padded to fit the length of H 's inputs. Pedantically, we can view H as a family $\{H_{m,n}\}$ of hash functions; we suppress these subscripts for clarity.

As in [12], we will think of H as being randomly chosen from a family of hash functions [5]. And as in [12] we do not use the standard pairwise independent family. Instead we use families of hash functions that achieve $p(n)$ -independence, for some sufficiently large polynomial p . A member H in this family can be described by a string of polynomial length, and it is this string that is wired into the verifier. The polynomial p is set to exceed the running time of the simulator times the length of V 's answers. Thus, even if the simulator poses to V a different query in each of its steps, and if for each query V generates its “random” coins R as the hash of the query, using H , then the simulator will face a verifier that uses a completely random string to answer each of its (different) queries. Of course, if the simulator repeats a query, then the “deterministic” V repeats the same response.

3. Creating an efficient prover

To prove our main theorem, we construct a particular malicious verifier (pedantically a family of closely related malicious verifiers), with a fixed scheduling strategy (a very similar strategy is used in [6]). We show that a simulator that successfully simulates the multi-conversation on input x with high probability can be converted to a probabilistic polynomial prover P_S for the original protocol. This prover will cause V to accept x with probability strictly greater than $1/2$. Thus, we can use this prover to probabilistically decide whether $x \in L$, implying that $L \in BPP$.

3.1. The attack

Let the original protocol consist of an initial challenge q , followed by a reply, r and a second challenge, s and a final reply t . The original verifier V generates q as a function $q(x, R)$ of the input and its random coin flips, R . V generates s as a function $s(x, R, r)$ of the input, R and r

(q is implicit given x and R). Finally, V computes a predicate $accept(x, q, r, s, t)$ to determine whether to accept or reject. Note that this restricts the acceptance predicate to being “conversation-based,” in which one can tell whether V will accept based on its conversation, without looking at its random coins. In Section 6 we sketch how to eliminate this restriction.

Let k and M be parameters that will be chosen later. (Both polynomial in the length of the input.) We consider the protocol obtained by performing M proofs in parallel. Thus, we denote the initial challenge by

$$\vec{q} = (q_1, \dots, q_m) = (q(x, R_1), \dots, q(x, R_m)),$$

where $\vec{R} = (R_1, \dots, R_m)$. We define \vec{r}, \vec{s} and \vec{t} analogously. Finally, we define $accept(x, \vec{q}, \vec{r}, \vec{s}, \vec{t})$ to be true iff $accept(x, q_i, r_i, s_i, t_i)$ for all $i, 1 \leq i \leq M$. We call such a parallel set of proofs an M -block.

Note that parallel repetition is a special case of scheduling in an asynchronous environment. From now and on, we always use parallel repetitions, i.e., M -blocks proofs, instead of running a single message in each round.

Our attacking verifier \hat{V} is defined by the value of its private random hash function, H and parameters k and M . \hat{V} runs a total of k blocks with the prover. We use subscripts to denote the version, e.g., \vec{q}_i denotes the first question in the i th run of the protocol. The verifier interleaves its challenges so that the sequence of messages appears as

$$\vec{q}_1, \vec{r}_1, \vec{q}_2, \vec{r}_2, \dots, \vec{q}_k, \vec{r}_k, \vec{s}_k, \vec{t}_k, \vec{s}_{k-1}, \vec{t}_{k-1}, \dots, \vec{s}_1, \vec{t}_1.$$

The “random” \vec{R}_i are determined using the hash function, by

$$\vec{R}_i = H(x, \vec{q}_1, \vec{r}_1, \dots, \vec{q}_{i-1}, \vec{r}_{i-1}). \quad (1)$$

We assume that H returns the correct number of random bits used by V . The questions \vec{q}_i and \vec{s}_i are defined by

$\vec{q}_i = \vec{q}(x, \vec{R}_i)$ and $\vec{s}_i = \vec{s}(x, \vec{R}_i, \vec{r}_i)$. However, if for any i , $accept(x, \vec{q}_i, \vec{r}_i, \vec{s}_i, \vec{t}_i)$, then \hat{V} aborts immediately, without sending $\vec{s}_{i-1}, \dots, \vec{s}_1$ to the prover.

We can thus view a conversation as consisting of two phases. The generation of

$$\vec{q}_1, \vec{r}_1, \vec{q}_2, \vec{r}_2, \dots, \vec{q}_k$$

constitutes the *creation* phase, in which new blocks (values for \vec{R}_i) are created by \hat{V} , and the generation of

$$\vec{r}_k, \vec{s}_k, \vec{t}_k, \vec{s}_{k-1}, \vec{t}_{k-1}, \dots, \vec{s}_1, \vec{t}_1$$

constitutes the *resolution* phase, in which these proofs run their course. We treat these phases quite differently when discussing the simulator.

Note that all the randomness used by \hat{V} comes from H . In particular, \hat{V} doesn’t use its random input, to some extent limiting the simulator S ’s power over it.

3.2. The simulator

Our proof is by contradiction: assuming a simulator S we construct an efficient prover P_S . We make some assumptions (without loss of generality) about the simulation, and give a convenient way of looking at the workings of the simulator.

First, we assume that whenever S generates a transcript it runs it through \hat{V} . That is, before returning

$$(\vec{q}_1, \vec{r}_1, \vec{q}_2, \vec{r}_2, \dots, \vec{s}_2, \vec{t}_2, \vec{s}_1, \vec{t}_1),$$

S runs \hat{V} to obtain \vec{q}_1 , sends \vec{r}_1 to \hat{V} , receiving \vec{q}_2 , and so on. Clearly, any simulator can be modified to perform in this manner without changing the quality of its simulation. Since our protocols are dialog based, we also assume without loss of generality that the simulator never sends \hat{V} a value for \vec{t}_i that would cause it to reject, since it could simply compute for itself that \hat{V} would reject.

3.2.1 The proof forest

As it interacts with \hat{V} , S implicitly generates many partial conversations, not all of which are successfully finished (only one need be). In generating these partial conversations, S typically starts many proofs that may or may not be completed as the simulation proceeds. For our proof, we organize these proofs using a levelled forest, which we call the *proof forest*. Each vertex v of the graph corresponds to a new M -block that has been initiated between S and \hat{V} . A vertex v has parameters $\vec{R}, \vec{q}, \vec{r}$, corresponding to the beginning of a conversation. A vertex on level $i < k$ may have zero or more children on level $i + 1$; the proof forest has at most k levels. We adopt the convention that Level 1 of the graph is the “top” level and Level k is the bottom level.

Whenever S runs a partial conversation

$$\vec{q}_1, \vec{r}_1, \dots, \vec{q}_i, \vec{r}_i, \dots$$

through \hat{V} , it may be thought of as traversing/creating the proof forest as follows. First, S implicitly generates $\vec{R}_1, \dots, \vec{R}_i$. S will visit or create a sequence of vertices v_1, \dots, v_i , where v_j is on level j . First, S visits or creates the top level vertex v_1 with parameters $(\vec{R}_1, \vec{q}_1, \vec{r}_1)$. Then, for $1 < j \leq i$, after visiting/creating v_{j-1} , S visits/creates v_j , the unique child of v_{j-1} with parameters $(\vec{R}_j, \vec{q}_j, \vec{r}_j)$.

Note for example, that all siblings vertices will have the same values for \vec{R} and \vec{q} , as will all the top level vertices. To avoid special cases, we adopt the convention that the top level vertices are siblings.

A simulated partial conversation of the form

$$\vec{q}_1, \vec{r}_1, \dots, \vec{q}_k, \vec{r}_k, \vec{s}_k, \vec{t}_k, \dots, \vec{s}_i, \vec{t}_i, \dots$$

may be thought of as follows. First, the simulator traverses/creates a path from the top level to some bottom-level vertex v , with parameters $(\vec{R}_k, \vec{q}_k, \vec{r}_k)$, of the proof forest. At this point, it has simulated $\vec{q}_1, \vec{r}_1, \dots, \vec{q}_k, \vec{r}_k$. When it runs \hat{V} further, receiving $\vec{s}_k = \vec{s}(x, \vec{R}_k, \vec{r}_k)$, it is said to *activate* v . When it sends an acceptable \vec{t}_k to \hat{V} , it is said to *resolve* v . When it receives \vec{s}_{k-1} it conceptually activates v 's parent, and so on. S thus may be thought of as retracing its path back up the proof forest. The simulation may, without loss of generality be viewed as a series of such bounces. In general, S may retrace partially, then continue down another path - but insisting that S "start over" from the top level of the forest does not impair S 's efficiency (by more than a polynomial factor) or correctness (since \hat{V} is deterministic).

3.3. Turning a simulator into a prover: Overview

We give a high-level overview of how to convert a good simulator S into an efficient prover P_S . We can view our (S, \hat{V}) interaction as generating and playing (not always to completion) many different proofs of the original protocol. On a very high level, our efficient prover P_S "splices in" its interaction with V and uses the answers created by the simulator as its own. When V chooses a random R and send its first challenge, q , to P_S , P_S conceptually alters the random hash function H so that for some set of siblings in the proof forest, with parameters

$$(\vec{R} = (R_1, \dots, R_M), \vec{q} = q_1, \dots, q_M, \cdot),$$

$R_j = R$ for some j , $1 \leq j \leq M$, and hence $q_j = q$. One of these siblings v in particular, with parameter $(\vec{R}, \vec{q}, \vec{r})$ will be chosen. Under the right circumstances, P_S will send V the value of $r = r_j$; great care must be taken to choose when to do so.

Later in the simulation, v may be activated, generating \vec{s} ; P_S will send r to V , receive s and splice in $s_j = s$. Hopefully, S will eventually respond with an acceptable \vec{t} ; allowing P_S to forward $t = t_j$, causing V to accept.

The splicing attempt will have three possible outcomes:

- P_S can succeed in making V accept,
- P_S can fail (get stuck), losing its chance to make V accept, or
- P_S can abort, without causing V to accept, but giving it the chance to try again.

Nearly all the time, P_S will abort. With some small probability, P_S will succeed and with some (hopefully much smaller) probability, P_S will fail. We show that for most R , P_S will succeed more often than it will fail, and can

therefore try this splicing procedure repeatedly, ultimately succeeding with probability at least $2/3$.

There are a number of difficulties with the above approach. First, S doesn't know R , so it can't really alter H as described. We observe that, barring certain bad events, S can usually simulate the behavior of the spliced \hat{V} . A more technically difficult problem is that the simulator may generate many proofs that it never completes. Indeed, the ratio of completed proofs to uncompleted proofs may be quite small (though nonnegligible). If P_S sends \vec{r}_v to V , and S fails to generate an acceptable \vec{t}_v , P_S will not be able to cause V to accept.

To get around the problem of incomplete proofs, we have P_S use a strategy that allows it to abort a splicing attempt before it has responded to V . We also have P_S choose where to insert the real proof into the proof forest according to a particular distribution. We show that using these two techniques, P_S may often abort but will only rarely fail.

Remark: If the original proof system has negligible error, then it suffices for P_S to succeed with nonnegligible probability. Given such an efficient P_S , one can run the proof many times to determine whether an input x is in the language. Given an interactive proof with error bounded away from 1, one can run it in parallel to obtain a proof with negligible error. This gives a simpler proof of our theorem for this case, and can be used to extend the theorem to work for 5 message interactive proofs [18]. Unfortunately, in the argument model, parallel amplification doesn't always work [2], so we can't use this trick to obtain a general theorem.

3.4. Splicing in the proof

Let us first state more explicitly how the splicing operation works. In the proof system, V flips coins to generate R , and generates an initial challenge $q = q(x, R)$. P_S conceptually chooses a random hash function H , defining \hat{V} . P_S can simply randomly generate new values of H as needed on the fly. Thus, P_S can simulate \hat{V} . P_S then runs (S, \hat{V}) , which starts generating the proof forest. Recall that if S runs through the partial conversation $\vec{q}_1, \vec{r}_1, \dots, \vec{r}_{i-1}$ and "asks" \hat{V} for the next question \vec{q}_i , \hat{V} then computes its parameters \vec{r} and \vec{q} by

$$\begin{aligned} \vec{R} &= H(x, \vec{q}_1, \vec{r}_1, \dots, \vec{q}_{i-1}, \vec{r}_{i-1}), \text{ and} \\ \vec{q} &= \vec{q}(x, \vec{R}). \end{aligned}$$

Writing $\vec{R} = (R_1, \dots, R_M)$, Define $splice(\vec{R}, j, R)$ to be the M -tuple equal to \vec{R} at every coordinate except the j th, which is equal to R . The splice operation takes place when P_S randomly chooses j , $1 \leq j \leq m$ and conceptually defines

$$H(x, \vec{q}_1, \vec{r}_1, \dots, \vec{q}_{i-1}, \vec{r}_{i-1}) = splice(\vec{R}, j, R),$$

and hence $R_j = R$. Then, conceptually, P_S just lets S continue with the simulation (we show below the mechanics of how P_S can do this).

The splice operations assures that a set of siblings will have the above values of \vec{R} and \vec{q} ; we call these *critical vertices*. At most one of these siblings, v , will be chosen to generate the conversation with V .

If v is later activated, \hat{V} generates $\vec{s} = s_1, \dots, s_M$. To simulate the spliced \hat{V} (\hat{V} using the spliced H), P_S sends $r = r_j$ to V , receives s and sets $s_j = s$. Here is where P_S suffers from not knowing R , which would allow P_S to compute s_j by itself. It uses V to perform this computation for it, but note that this trick may be performed only once (and at great risk) and P_S cannot waste it on a sibling of v . So, if a sibling of v is activated before v is activated, P_S aborts. Note that in this case, P_S has a chance to try again, since no r has been sent to V . If a sibling of v is activated after v is activated, we assume that P_S fails (though it may have actually solved the protocol).

If S resolves v , generating a value of $\vec{t} = t_1, \dots, t_M$ that will cause the spliced \hat{V} to accept, then P_S sends t to V . At this point, P_S succeeds (Here we use the fact that S only sends an acceptable \vec{t}).

Except for choosing where to splice in the proof, we have specified P_S . It can be verified that, at least up to the point where P_S aborts, P_S correctly implements the behavior of the simulator with the spliced H .

3.5. Choosing v

We define the *height* of a vertex at level i to be $h = k - i + 1$. Our method for choosing which special vertex v to splice in the conversation with V obeys the following design criteria:

- The probability that a generated vertex v is the special one depends only on its height, and is completely independent of any other aspect of the entire run of S .
- If v has height h , it's probability will be proportional to $f(h)$, for some carefully chosen f .

First, we note that S must run in an expected number of steps at most $(nkM)^\alpha$, where $n = |x|$ and α is some constant. Since P always causes \hat{V} to accept, S must cause \hat{V} to accept (that is, generate a path going all the way down and then all the way up the proof forest) with probability close to 1. By Markoff's inequality, if we only allow S to run for $N = 100(nkM)^\alpha$ steps, it will still succeed with probability greater than .9 (though its simulation will no longer be close to the actual one). However, our analysis will only consider whether S causes \hat{V} to accept. For the rest of the analysis, S will only run for N steps.

With S as above, every level has at most N vertices and every vertex has at most N children. We use the following

addressing scheme for the vertices of the proof forest. For uniformity, imagine a dummy vertex at Level 0 that has all the Level 1 vertices as children. As each vertex v level i we keep track of how many vertices have been generated on the same level as v and how many siblings of v have been generated. To each vertex v we assign an address (i, a, b) , denoting that v is on level i and is the b th child of the a th level $i-1$ vertex generated (top level vertices have addresses of the form $(1, 1, b)$). We choose the address of v by first picking a level i with probability $cf(h)$, where $h = k - i + 1$ and c is the normalizing constant defined by $\sum_{h=1}^k cf(h) = 1$, and choosing a and b uniformly subject to $1 \leq a, b \leq M$. Thus, any address of height h is selected with probability $cf(h)/N^2$.

The precise function f is a polynomial chosen to make the analysis work out; we defer its determination to that section.

4. Preliminary analysis of the splicing operation

We now bound below the probability that P_S succeeds and the ratio of the probability that P_S succeeds to the probability that P_S fails when R is chosen uniformly. As discussed later, this is *not* sufficient to prove our theorem, but is a very good start. That is, we consider the following experiment.

EXPERIMENT 1: Execute the following steps:

1. Choose a random address (i, a, b) as above.
2. Choose R uniformly (over strings of the appropriate length) and run V with R .
3. Generate and traverse the proof forest by running P_S , choosing (i, a, b) as the address of the special vertex v , and splice accordingly.

We assume that the representation of the proof forest generation/traversal keeps track of the order in which vertices are generated/visited. Given this information, we can determine whether a vertex has been activated or resolved. We also assume, for the sake of the analysis, that the simulation continues even if P_S fails. That is, once we splice the game in, we allow P_S to query V multiple times. Of course, we have to take into account the fact that P_S really failed in this case.

Definition 2 We say that an address (i, a, b) is good iff the vertex at that address is resolved and no sibling vertex (with address (i, a, b')) is ever activated. We say that (i, a, b) is bad iff the vertex at that address is activated but not resolved or if any sibling vertex is ever activated. We say that (i, a, b) is interesting if it is either good or bad. We

say that a vertex is good/bad/interesting if its address is good/bad/interesting.

Now, we observe that when R is chosen uniformly, all the splicing operation does is conceptually replace a uniformly chosen value (of the hash function) with another uniformly chosen value. It follows that any value of (i, a, b) yields the same distribution on how the forest is generated and traversed, and this distribution is the same as if we never spliced in a game. We can therefore reorder the steps of the experiment as follows.

EXPERIMENT 2: Execute the following steps:

1. Generate and traverse the proof forest by running S .
2. Choose a random address (i, a, b) as above.

Note that the notion of being activated and resolved is simply a property of the forest generation/traversal, and is therefore still well defined. Lemma 2 follows from the above discussion and a straightforward application of Bayes theorem.

Lemma 2 *The probability that P_S succeeds is bounded below by the probability that (i, a, b) is good after performing Experiment 2. The probability that P_S fails is bounded above by the probability that (i, a, b) is bad after performing Experiment 2.*

The event that (i, a, b) is good is equal to the sum over all good addresses (i', a', b') of the probability that $(i, a, b) = (i', a', b')$ (and similarly for the probability that (i, a, b) is bad. We can thus recast Lemma 2 as the following calculation. Consider the random variables SUCCEED, FAIL and INTERESTING, generated as follows. S generates and traverses the proof forest, and then computes

$$\begin{aligned} \text{SUCCEED} &= \sum_{(i, a, b) \text{ good}} cf(k - i + 1)/N^2, \\ \text{FAIL} &= \sum_{(i, a, b) \text{ bad}} cf(k - i + 1)/N^2 \text{ and} \\ \text{INTERESTING} &= \sum_{(i, a, b) \text{ interesting}} cf(k - i + 1)/N^2. \end{aligned}$$

Note that $\text{INTERESTING} = \text{SUCCEED} + \text{FAIL}$. We will bastardize terminology slightly, and speak of these variables after a given generation/traversal of a proof forest.

Lemma 3 *The probability that P_S will succeed is at least $E(\text{SUCCEED})$ and the probability that P_S will fail is at most $E(\text{FAIL})$.*

To bound these expectations, we need to consider the structure of good and bad addresses.

4.1. The structure of bad and good addresses

We note that \hat{V} will not ask the second question for a level- i proof for $i < k$ unless one of its children (in the proof forest) has been completed and \hat{V} accepts. As a consequence, any interesting vertex has to have a child vertex that is resolved. This implies some structure to the set of good and bad addresses.

Definition 3 *A snake σ is a series of vertices*

$$v_i, v_{i+1}, \dots, v_k$$

such that v_j is on level j and v_{j+1} is a child of v_j for $i \leq j < k$. We call v_i the head of σ and v_{i+1}, \dots, v_k the body of σ . We define the height $h(\sigma)$ to be $k - i + 1$ (the height of σ 's head).

Lemma 4 *After any generation/traversal of a proof forest, the set of interesting vertices can be canonically decomposed into disjoint snakes such that*

- Any bad vertex with no bad siblings is at the head of a snake.
- Given a set of bad siblings, at most one is in the body of a snake.

Proof: (Sketch) From each interesting vertex v without an interesting parent, we start a snake, with v as the head. If v is on level k we're done, else v must have one or more interesting children. In some canonical fashion, choose one to recursively continue the snake, and start new snakes at the other siblings. The properties required by Lemma 4 are easily verified. \square

Given a canonical snake decomposition of the interesting vertices of a graph, we can bound FAIL and INTERESTING as follows. Let $F(i) = \sum_{j=1}^i f(j)$.

Lemma 5 *After any generation/traversal of a proof forest,*

$$\begin{aligned} \text{FAIL} &\leq 2 \sum_{\sigma} cf(h(\sigma))/N^2 \text{ and} \\ \text{INTERESTING} &\geq \sum_{\sigma} cF(h(\sigma))/N^2. \end{aligned}$$

Proof: (Sketch) To establish the bound for FAIL, note that $cf(h(\sigma))/N^2$ is simply $cf(k - i + 1)/N^2$, where i is the level of σ 's head. A lone bad vertex (without a bad sibling) is therefore counted in the summation. Given a set of bad siblings, one of them may not be counted; the worst case is when there are exactly two bad siblings of which one is the head of the snake but the other isn't. However, this undercounting is compensated by the 2 in front of the summation. To establish the bound for INTERESTING, note that $cF(h(\sigma))/N^2$ simply sums the contribution of each vertex in the snake. \square

4.2. Setting the parameters

We set $f(h) = h^\beta$, for a sufficiently large β , to be determined. It is easy to verify that $F(h) \geq h^{\beta+1}/\beta$, and hence $F(h)/f(h) \geq h/\beta$.

Definition 4 We say that a snake σ is short if $h(\sigma) < 10\beta$ and long otherwise.

Using Lemma 5 and the fact that $F(i)/f(i) > 10$ for long σ , we have

$$\begin{aligned} \text{INTERESTING} &\geq 5 \left(\text{FAIL} - \sum_{\text{short } \sigma} cf(h(\sigma))/N^2 \right) \\ &\geq 5 \left(\text{FAIL} - \sum_{\text{short } \sigma} c(10\beta)^\beta/N^2 \right) \\ &\quad \text{[Since } \sigma \text{ is short]} \\ &\geq 5 \left(\text{FAIL} - c(10\beta)^\beta/N \right) \end{aligned}$$

The last inequality follows because there are at most N snakes in the decomposition. By the linearity of expectation and simple algebra, we have

$$E(\text{FAIL}) \leq E(\text{INTERESTING})/5 + c(10\beta)^\beta/5N. \quad (2)$$

Now, if the constant term were 0, this would imply that we succeed much more than we fail, since $E(\text{INTERESTING}) = E(\text{SUCCEED}) + E(\text{FAIL})$. To deal with the constant term, we bound $E(\text{INTERESTING})$ from below.

Lemma 6

$$E(\text{INTERESTING}) \geq .9cF(k)/N^2 \geq .9ck^{\beta+1}/\beta N^2.$$

Proof: Whenever S succeeds in finishing all k games, there is a snake of height k (from the bottom to the top); the term in the statement of the lemma is the contribution of this snake. S succeeds with probability at least .9. \square

We now set the parameters k, M and β . Let $M = k^3$ and $k = n = |x|$. Thus, $N = 100k^{5\alpha}$ and $c = 1/k^{O(1)}$. We want to make small the ratio

$$\frac{c(10\beta)^\beta/5N}{.9ck^{\beta+1}/\beta N^2} = \frac{20k^{5\alpha}(10\beta)^\beta\beta}{.9k^{\beta+1}}.$$

We can set β to be a sufficiently large integer so that for sufficiently large n (and hence sufficiently large k) this ratio is less than .01.

Finally, Lemma 7 follows from Lemma 6, Equation 2 and the setting of the parameters.

Lemma 7 P_S succeeds with probability $\Omega(1/k^\gamma)$ for some constant γ . Furthermore, P_S succeeds at least 3 times as often as P_S fails.

5. Showing success for most R

Naively, one might suppose that Lemma 7 would imply that we are done. Given an input $x \in L$, P_S keeps on trying the splicing strategy (with the parameters determined above) until it succeeds or fails. It will conclude in expected polynomial time and it will succeed with probability at least 3/4, implying that $L \in BPP$. However, this analysis would only hold if V chose R independently for each of P_S attempts; in reality, V chooses R once. The problem is that the previous section bounds the expected success rates over all R . However, it might be that for a very small fraction of R , P_S succeeds much more often than it fails, yet for the rest of the R , P_S fails more often than it succeeds.

However, note that each vertex in the proof forest corresponds to $M = k^3$ original proofs, of which at most one is altered by the splicing operation. We exploit this fact to show that for nearly all R , the success and failure probabilities are multiplicatively close to the expected values.

Lemma 8 For all but measure .01 of the random strings \mathcal{R} , P_S succeeds with probability $\Omega(1/k^\gamma)$ for some constant γ . Furthermore, for all but measure .01 of the random strings \mathcal{R} , P_S succeeds at least 2 times as often as P_S fails.

Remark: By setting parameters correctly, the 2 can be replaced by anything less than the corresponding value in Lemma 7. In turn, this value (set arbitrarily at 3) can be set to any constant (or indeed, can be polynomially large, with care). Similarly, the .01 may be made arbitrarily (though nonnegligibly) small.

Let us explain why Lemma 8 is sufficient for proving our result. If V chooses a “bad” random string, that doesn’t behave like most random strings, then we give up and assume that P_S fails. But this only happens with probability .02. The rest of the time, the naive argument sketched above does indeed hold; and hence P_S will succeed with probability close to 2/3.

Let \mathcal{R} denote the set of V ’s possible coin tosses and \mathcal{R}' denote an arbitrary arbitrary subset of \mathcal{R} of measure at least .01 (that is, a uniformly chosen R is in \mathcal{R}' with probability at least .01). Let D denote the uniform probability measure on \mathcal{R}^M and let D' denote probability measure obtained by choosing (R_1, \dots, R_M) uniformly, choosing j , $1 \leq j \leq M$ uniformly, and then replacing R_j by a uniformly chosen element of \mathcal{R}' . Lemma 9 says that these measures are multiplicatively close over most elements of their domain.

Lemma 9

$(1 - \frac{1}{k})Pr_D(\vec{R}) \leq Pr_{D'}(\vec{R}) \leq (1 + \frac{1}{k})Pr_D(\vec{R})$ for all but a negligibly small measure (over both D and D').

Proof: (Sketch) Suppose that \vec{R} has l elements in \mathcal{R}' and let ρ denote the measure of \mathcal{R}' under the uniform measure.

By an elementary probability argument, we can explicitly derive the relevant probabilities, obtaining

$$\begin{aligned} Pr_D(\vec{R}) &= \frac{\binom{M}{l} \rho^l (1-\rho)^{M-l}}{|\mathcal{R}'|^l |\mathcal{R}|^{M-l}} \text{ and} \\ Pr_{D'}(\vec{R}) &= \frac{l \binom{M}{l} \rho^{l-1} (1-\rho)^{M-l}}{M |\mathcal{R}'|^l |\mathcal{R}|^{M-l}} \end{aligned}$$

Thus, we have

$$\frac{Pr_{D'}(\vec{R})}{Pr_D(\vec{R})} = \frac{l}{\rho M}.$$

Thus, the inequality of Lemma 9 holds as long as

$$(1 - \frac{1}{k}) \rho M < l < (1 + \frac{1}{k}) \rho M.$$

For $M = k^3$, standard bounds on the tail of binomial distributions can be used to show that holds with all but negligible probability under both D and D' (here we use the fact that ρ is reasonably large). \square

We use Lemma 9 to argue that the bounds in Lemma 7 hold for most random R .

Proof of Lemma 8: (Sketch) Our proof is by contradiction. Let \mathcal{R}' be a set of measure $> .01$ such that P_S succeeds with less than half the probability implied by Lemma 7 whenever $R \in \mathcal{R}'$. Then if one chooses R uniformly from \mathcal{R}' P_S must succeed with abnormally low probability. We show that this is impossible.

We can view the splicing operation in the following perverse way. H is initially chosen uniformly; for our analysis it is irrelevant whether H is uniform or has (sufficiently) high independence, so we adopt the cleaner view. P_S chooses (i, a, b) as before and V chooses R uniformly from \mathcal{R}' . Then, P_S runs S , waiting for the moment to splice H to enter R into the proof. Now that the final (post-splicing) value of H has been chosen, the actions of S are completely determined. Conceptually, S just completes the simulation and we check whether (i, a, b) is good or bad.

Now, by a Bayesian analysis, once we determine H , we can allow P_S to forget (i, a, b) and V to forget R , and then choose them with the correct probabilities, conditioned on H and their *a priori* distributions. If H were uniformly distributed, and for all H , the conditional distribution on (i, a, b) were the same as the *a priori* distribution on (i, a, b) , then we could simply use the analysis of Experiment 2 of the previous section. Now, of course, H is not uniform and the conditional distribution on (i, a, b) is not the same as the *a priori* distribution. However, using Lemma 9, it follows that for all but a negligible measure of functions H , the probability that H is chosen is within a multiplicative $(1 \pm 1/k)$ factor of the uniform probability, for all (i, a, b) ; we call such an H *balanced*. We can safely neglect the case where H is not balanced. Using

Bayes rule, the conditional probability of (i, a, b) given a balanced H is within a $(1 \pm O(1/k))$ multiplicative factor of its *a priori* distribution. It follows that the probability that (i, a, b) is good or bad is within a $(1 + O(1/k))$ multiplicative factor of the corresponding probability in the ideal case (plus or minus a negligible additive factor we can safely ignore). Hence, the success and failure probabilities of P_S differ from the ideal case by only (essentially) $(1 + O(1/k))$ multiplicative factors. \square

6. Generalizing the interactive proofs

The proof as we have written it required that S never give \hat{V} answers that would cause it to reject. We need this, because otherwise S might give many final answers for the spliced proof, only one of which is actually good, but P_S would have no way of knowing which of these answers to forward back to V . If it is apparent from the conversation whether V would accept, this is not a problem, since S need not send a bad answer and P_S could filter them anyway. Every one-sided protocol we know of has this property, but for completeness we would like to eliminate this condition.

Suppose that S is giving the answers for the M -block of proofs of which the real one is buried. Now, P_S can evaluate these answers for all but the one proof it cares about, since it knows the random string of the $M - 1$ simulated verifiers. If it detects a single mistake, it knows that the simulated \hat{V} will reject (it requires that all the proofs in an M -block accept before it accepts). Hence, it can continue the simulation of \hat{V} without making any calls to V .

Now, given that the chosen vertex v of the game forest is been visited, it could turn out to be either good or bad (depending on whether it is resolved or its siblings are activated). When v is bad, we already assume the worst case (P_S fails). When v is good, then in the previous analysis, P_S succeeded in getting V to accept with probability 1. We can't achieve this success rate, but if P_S can succeed with probability at least .99 (this can be made arbitrarily close to 1) the previous analysis won't be materially affected.

By the definition of a good vertex, S eventually gives a set of M correct answers, since it causes \hat{V} to accept. Since P_S is running subsimulations of all but at most one of the verifiers (the one it cares about), it can check all but at most one of the answers (the one it cares about) to see if they make the verifier accept. A naive strategy is for P_S to wait until all the $M - 1$ answers it can check are correct (cause the simulated V to accept), and then use the remaining answer. However, this strategy is not good. Suppose that before giving a set of all correct answers, S gives M sets of answers such that in the i th set all but the i th answer is correct. In this case, P_S will always detect this mistake, and not use the remaining (correct) answer, until the one set occurs in which the answer it cares about is wrong. It will

then use this bad answer. By a similar argument, any sharp threshold criteria is subject to this problem.

We instead employ a randomized strategy. If m out of the $M - 1$ checkable answers are incorrect, we use the remaining answer with probability γ^{-m} , where γ is a constant slightly smaller than 1 ($\gamma = .999$ suffices).

By the definition of a good vertex, eventually P_S will see a set of answers with no detected mistakes and use the remaining answer. Thus, it will eventually try an answer. Lemma 10 states that this method works.

Lemma 10 *For a set of all but measure .01 of the random coin tosses R , conditioned on the (i, a, b) chosen by P_S being a good address, P_S will go on to cause V to accept with probability at least .99.*

(proof omitted)

References

- [1] C. Brassard, C. Crepeau and M. Yung, "Constant-Round Perfect Zero-Knowledge Computationally Convincing Protocols", *Theoretical Computer Science*, Vol. 84, 1991, pp. 23-52.
- [2] M. Bellare, R. Impagliazzo, and M. Naor. Does Parallel Repetition Lower the Error in Computationally Sound Protocols? In *Proceedings of 38th Annual Symposium on Foundations of Computer Science*, IEEE, 1997.
- [3] M. Bellare, S. Micali, and R. Ostrovsky. Perfect zero-knowledge in constant rounds. In *Proc. 22nd Ann. ACM Symp. on Theory of Computing*, pages 482-493, 1990.
- [4] T. Beth and Y. Desmedt. Identification tokens - or: Solving the chess grandmaster problem. In A. J. Menezes and S. A. Vanstone, editors, *Proc. CRYPTO 90*, pages 169-177. Springer-Verlag, 1991. Lecture Notes in Computer Science No. 537.
- [5] I. L. Carter and M. N. Wegman. Universal classes of hash functions. In *Conference Record of the Ninth Annual ACM Symposium on Theory of Computing: Papers Presented at the Symposium, Boulder, Colorado, May 2-4, 1977*, pages 106-112, New York, NY 10036, USA, 1977. ACM Press.
- [6] C. Dwork, M. Naor and A. Sahai. Concurrent Zero-Knowledge. *Proceedings, 30th Symposium on Theory of Computing*, pp. 409-428, 1998.
- [7] C. Dwork and A. Sahai. Concurrent Zero-Knowledge: Reducing the Need for Timing Constraints. *Proceedings, Advances in Cryptology - Crypto '98*.
- [8] U. Feige and A. Shamir, "Zero Knowledge Proofs of Knowledge in Two Rounds", *Advances in Cryptology - Crypto 89 proceedings*, pp. 526-544, 1990.
- [9] U. Feige and A. Shamir. Witness indistinguishable and witness hiding protocols. In Baruch Awerbuch, editor, *Proceedings of the 22nd Annual ACM Symposium on the Theory of Computing*, pages 416-426, Baltimore, MY, May 1990. ACM Press.
- [10] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. In *25th Annual Symposium on Foundations of Computer Science*, pages 464-479, Los Angeles, Ca., USA, October 1984. IEEE Computer Society Press.
- [11] O. Goldreich and A. Kahan, "How to Construct Constant-Round Zero-Knowledge Proof Systems for NP", *Journal of Cryptology*, Vol. 9, No. 2, 1996, pp. 167-189.
- [12] O. Goldreich, H. Krawczyk. On the Composition of Zero-Knowledge Proof Systems. *SIAM J. on Computing*, Vol. 25, No.1, pp. 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", *Jour. of ACM.*, Vol. 38, 1991, pp. 691-729.
- [14] Oded Goldreich and Yair Oren. Definitions and properties of zero-knowledge proof systems. *Journal of Cryptology*, 7(1):1-32, Winter 1994.
- [15] S. Goldwasser, S. Micali, C. Rackoff. The Knowledge Complexity of Interactive Proofs. *Proc. 17th STOC*, 1985, pp. 291-304.
- [16] S. Goldwasser, S. Micali, and C. Rackoff. "The Knowledge Complexity of Interactive Proof Systems", *SIAM J. Comput.*, 18 (1):186-208, 1989.
- [17] J. Kilian. Zero-Knowledge with Log-Space Verifiers *Proceedings, 29th annual IEEE Symposium on the Foundations of Computer Science*.
- [18] J. Kilian and E. Petrank. Manuscript in preparation.
- [19] Y. Oren. On the cunning powers of cheating verifiers: Some observations about zero knowledge proofs. In Ashok K. Chandra, editor, *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, pages 462-471, Los Angeles, CA, October 1987. IEEE Computer Society Press.
- [20] R. Richardson and J. Kilian. Non-Synchronized Composition of Zero-Knowledge Proofs. Manuscript.