Incrementally Verifiable Computation or Proofs of Knowledge Imply Time/Space Efficiency

Paul Valiant

pvaliant@mit.edu, Massachusetts Institute of Technology

Abstract. A probabilistically checkable proof (PCP) system enables proofs to be verified in time polylogarithmic in the length of a classical proof. Computationally sound (CS) proofs improve upon PCPs by additionally shortening the length of the transmitted proof to be polylogarithmic in the length of the classical proof.
In this paper we explore the ultimate limits of non-interactive proof systems with respect to time and space efficiency. We present a proof system where the prover uses space polynomial in the space of a classical prover and time essentially linear in the time of a classical prover, while the verifier uses time and space that are essentially constant. Further, this proof system is *composable*: there is an algorithm for merging two proofs of length k into a proof of the conjunction of the original two theorems in time polynomial in k, yielding a proof of length *exactly* k.
We deduce the existence of our proposed proof system by way of a natural new assumption about proofs of knowledge. In fact, a main contribution of our result is showing that knowledge can be "traded" for time and space efficiency in noninteractive proof systems. We motivate this result with an explicit construction of noninteractive CS proofs of knowledge in the random oracle model.

1 Introduction

Perhaps the simplest way to introduce the computational problem we address is by means of the following.

Human motivation. Suppose humanity needs to conduct a very long computation which will span super-polynomially many generations. Each generation runs the computation until their deaths when they pass on the computational configuration to the next generation. This computation is so important that they also pass on a proof that the current configuration is correct, for fear that the following generations, without such a guarantee, might abandon the project. Can this be done?

Computational setting. In a more computational context, this problem becomes:
How can we compile a machine M into a new machine M' that frequently outputs pairs (c_i, π_i) where the ith output consists of the ith memory state c_i of machine M, and a proof π_i of its correctness, while keeping the resources of M intact?[1]

1.1 A new problem

We motivate our problem by way of a few examples of how current techniques fail to achieve our goal. Suppose we are given a computation M that takes time t and space $k \ll t$.

A natural approach is have the compiled machine M' keep a complete record of all the memory states of M it has simulated so far; every time it simulates a new state of M, it uses this record to output a proof that its simulation of M is thus far correct. However, this approach has the clear drawbacks that the compiled machine M' uses space tk to store the records, and the proofs it outputs consist simply of this record of

[1] More generally one might consider a machine that, instead of outputting proofs π_i, engages in some interactive proof protocol.

size tk; this requires the verifier of the proofs to also use time tk and space tk to verify *each* proof. If t is polynomial in k, then all these parameters are polynomial in k and this simple system is in fact "optimal up to polynomial factors in k." We concern ourselves here with the much more interesting case where the running time t is much larger than k —exponentially larger, even— in which case this naive system is not at all efficient. What we need is a more efficient proof system.

We note that the problems of improving the efficiency of the construction, transmission, and verification of proofs have been important themes in our field, and have fueled a long line of research. One major milestone on this path was the discovery of *probabilistically checkable proofs* (PCPs) (see [1, 2, 5, 10] and the references therein). Under a PCP proof system statements with classical proofs of exponential length could now be verified in polynomial time, via randomized sampling of an encoded version of the classical proof. A PCP system still uses exponential resources to construct and transmit the proof, but verification is now polynomial time.

The second milestone we note is the theory of *computationally sound* (CS) proofs as formalized by Kilian and Micali [12, 13]. This notion improves on the PCP system by keeping verification polynomial time while shortening the length of the transmitted proof from exponential to polynomial in k. If we instruct the compiled machine M' to output (noninteractive) CS proofs, then the length of the transmitted proofs, and the time and space required by the verifier are now polynomial in k, but the compiler still requires memory at least t, and a time interval of at least t between consecutive proofs.[2]

1.2 Intuitive idea of our solution

The ideal way to achieve incrementally verifiable computation consists of *efficiently merging two CS proofs of equal length into a single CS proof* which is as short and easy to verify as each of the original ones. Letting $c_0, c_1, \ldots$ be the sequence of configurations of machine M, and for $i < j$, intuitively denote by $(M : c_i \xrightarrow{t} c_j)$ the assertion that configuration c_j is correctly obtained from configuration c_i by running M for t steps. After running M for 1 step from the initial configuration c_0 so as to reach configuration c_1 one could easily produce a CS proof of $(M : c_0 \xrightarrow{1} c_1)$. Running M for another step from configuration c_1, one can easily produce a CS proof that $(M : c_1 \xrightarrow{1} c_2)$. At this point, if CS proofs can be easily merged as hypothesized above, one could obtain a CS proof that $(M : c_0 \xrightarrow{2} c_2)$. And so on, until a final configuration c_f is obtained, together with a CS proof that $(M : c_0 \xrightarrow{t} c_f)$

Unfortunately, we have no idea of how to achieve such efficient and length preserving merging of CS proofs. However, if a variant of CS proofs —which we call CS *proofs of knowledge*— exist, we show a sufficient approximation of this ideal strategy. The main idea is to construct recursively embedded CS proofs: to merge proofs π_1 and π_2 I prove that "I have seen convincing π_1 and π_2." In a nutshell, the CS proof methodology enables us to work with very short proofs, and proofs of knowledge enable the soundness of the proof system to persist across many levels of recursion.

1.3 A new role for a new type of proof of knowledge

Proofs of knowledge may be seen as a restricted form of classical proofs. While classically, proofs of a statement "There exists w such that $R(x, w) = 1$"[3] can take a wide variety of non-constructive forms, the proof of knowledge form asserts essentially "I have seen a w such that $R(x, w) = 1$." We note here that the inapplicability of classical proofs to our setting results from the combination of two circumstances: we

[2] A third major approach for improving the efficiency of proofs, arguably the most historically successful, is that of adding interaction between the prover and verifier[11, 15, 3]. Unfortunately, this approach does not help us here: our prover has only k memory so he may transfer his entire knowledge to the verifier at the start of their interaction; any further correspondence between the prover and verifier may be simulated by the verifier with no loss of efficiency.

[3] We remind the reader that since classical proofs are verifiable in polynomial time, we may consider any classical theorem as being a statement of membership in an NP-language of the form "There exists a proof w such that the verifier R accepts the pair consisting of the theorem x and proof w."

require our proofs to be embeddable in other proofs, and we must work in merely computationally sound proof systems where deceptive proofs— while almost impossible to find— exist in abundance. We see the problem, intuitively, if we try to embed two computationally sound proof systems. The result would be a (computationally sound) proof that "There exists a computationally sound proof π of x." The problem is that *of course* there exists a computationally sound proof of x, *even when x is false*. So a proof that there exists a computationally sound proof of x implies nothing about the truth or falsehood of x.

Significantly, however, proofs of knowledge *can* be combined in this way: the result is a (computationally sound) proof that "Prover A has seen a computationally sound proof that Prover B has seen a witness w of x." Intuitively, this is the difference between saying "A is convinced that B is convinced of x" and saying "A is convinced that B *could be* convinced of x" —the first statement is reasonable evidence of x when both A and B are reasonable, but the second statement holds no weight since even a reasonable person *could be* mislead. In essence, the proof of knowledge property lets "reasonableness" be transferred down a sequence of provers. The formal statement of this assertion is that by sufficient repeated application of the *knowledge extractor E* associated with the proof system one can extract a valid witness w from any procedure that returns embedded proofs.

Remark 1: This simple intuition unfortunately translates into neither simple definitions nor simple proofs. Because this work seeks to optimize both prover and verifier time and space as well as the overall soundness of the proofs, we need to keep track everywhere not only of who is proving who's knowledge of what to whom, but also the time and space bounds of all involved parties, along with the security parameters. Nevertheless, it is our hope that the simple intuition underlying the constructions here will make the technical details less opaque.

Remark 2: We note that embedding proof systems deprives us of another principal tool: the use of random oracles. Specifically, suppose we have an oracle-based prover-verifier system (P^O, V^O) that can prove statements about the results of computation like "Machine M accepts the following string within t time steps. . . ." When we try to recursively embed this system the recursion breaks down because, even at the first level of recursion, we are no longer trying to prove statements about classical computation but rather statements of the form "M with oracle access to O accepts the following string...." Thus standard applications of random oracles do not appear to help. It remains an interesting question whether the goals of this paper may be attained in some other way using random oracles.

The Noninteractive CS Knowledge Assumption. Random oracles are intricately tied to CS proofs, in that the only known constructions of noninteractive CS proofs make use of random oracles (see [13]). Nevertheless, as with most random oracle constructions, the hope is that in practice the random oracle may be replaced by a suitably strong hash function plus access to a common random string.

In Section 4 we extend Micali's construction of CS proofs to a construction of CS proofs of knowledge: there exists an efficient extractor E that, given a statement X, a CS proof π, and access to the CS prover that produced π, outputs in quasilinear time a (classical) proof Π of X. We highlight this construction as a motivation for our assumption that oracle-less CS proofs of knowledge exist.

In essence, our assumption states that, in a specific construction of non-interactive CS proofs (Constructions 4 and 5), it is possible to replace the random oracle with a random string and still preserve the strength of the proofs. (That is, we do not invoke the random-oracle hypothesis in its general form. As shown by Canetti, Goldreich, and Halevi [8] and others in different contexts, we expect that there may be other non-interactive CS proof constructions for which no way to replace the oracle exists.)

We note that, while the Fiat-Shamir heuristic of replacing random oracle calls with a deterministic hash function yields feasible proposals for how to remove the oracle calls from the prover and verifier, it says nothing about how to translate the knowledge extractor into this new setting. For this reason we cannot explicitly conjecture a noninteractive CS proof of knowledge. However, in the context of this paper, the knowledge extractor component of the CS proof system serves only as a technique to argue security and is not invoked in our construction of incrementally verifiable computation. Thus we may propose the following much more explicit conjecture: our construction of incrementally verifiable computation (Theorem 1) works when using the prover-verifier pair (P, U) from Construction 4, modified by replacing the random oracle with a suitably strong hash function plus access to a common random string.

Knowledge ⇒ Time/Space Efficiency. In this work we start with an unusual and very strong assumption about (proofs of) knowledge and conclude with a proof system of unprecedented time and space efficiency. In this paragraph we wish to draw the reader's attention not to the assumption or the conclusion, but to the nature of the relationship between them. On the left we make an assumption about *knowledge* in CS proofs: we take a restricted system that only deals with witnesses of length $3k$ and compresses them to proofs of length k, the security parameter, and assume that there is a linear-time *knowledge extractor* that can extract the witness given access to the prover. On the right we conclude with a proof system that compresses any proof to length $poly(k)$, uses space polynomial in the space needed to classically accept the language, and is time-efficient in the tightest possible sense, using only $poly(k)$ time to process each step of the classical acceptance algorithm. We note that current constructions of non-interactive CS proofs based on random oracles need time *polynomial* in the time to classically accept, and space of the same order as their *time*[13]. Our results constitute a new technique to leverage *knowledge* to gain time and space efficiency, and is in a sense a completeness result for CS proof systems.

2 Definitions

2.1 Noninteractive proofs and the Common Random String model

It is a well-known aphorism in cryptography that "security requires randomness". In many standard settings, a participant in a protocol injects randomness into his responses to protect him from some pre-prepared deviousness on the part of the other participant.

In the noninteractive proof setting such an approach is inadequate: the verifier is unable to protect himself with randomized messages to the prover, since he cannot even *communicate* with the prover. To address these issues, the *common random string* (CRS) model was introduced [7, 6].

The CRS model —sometimes called the common *reference* string model— assumes that all parties have access to the same random string, and further that each can be confident that this string is truly random and not under the influence of the other parties. Potential examples of such a string are measurements of cosmic background radiation or, for a string that will appear in the future, tomorrow's weather.

In the analysis of the security of a CRS protocol leeway must be given for "unlucky" choices of strings, since if every choice of string worked in the protocol we would not need a random one. Thus even if a CRS protocol has a chance of failing, we still consider it secure if this chance is negligible as a function of the size of the random string.

2.2 Incremental computation

Basic notation We denote a Turing machine M with no inputs by $M()$, a Turing machine with one input by $M(\cdot)$, a Turing machine with two inputs by $M(\cdot, \cdot)$, etc. We assume a standard encoding, and denote by $|M|$ the length of the description of M. For a Turing machine M running on input s, we denote by $time_M(s)$ the time M takes on input s, and by $space_M(s)$ the space M takes on input s; we denote the empty input by ϵ, so that $space_M(\epsilon)$ is the space of Turing machine M when run on no input.

Incremental outputs Commonly, Turing machines make an output only once, and making this output ends the computation. Instead, we interpret Turing machines as being able to output their current memory state at certain times in their operation: explicitly, consider a Turing machine with a special state "Output" where whenever the machine is in state "Output" the entire contents of its tape are outputted. [4] This captures our

[4] We note that this is a slightly unusual model of output, as the machine would be unable to output a string such as "Hello World" without first deleting all other memory locations on the tape. In the context of this paper, we expect machines to not delete this other information: since we consider only $poly(k)$-space machines, it imposes no undue burden on the prover to output this information, and no undue burden on the verifier to ignore it.

intuitive notion of an "incremental computation," namely one divided into "generations" where at the end of each generation the entire memory configuration is output so that the next generation may resume the computation from the current configuration.

2.3 Incrementally verifiable computation

We formally define incrementally verifiable computation here. We consider a Turing machine $M()$ that we wish to simulate for t time steps using k memory, where $k \geq \log t$. We consider a fixed *compiler* $C(\cdot,\cdot)$ that produces from (M,k) an *incrementally verifiable* version of M, namely a machine $C(M,k) = T(\cdot)$ that takes as input the common random string, runs in time $t \cdot k^{O(1)}$, uses memory $k^{O(1)}$, and every $k^{O(1)}$ time steps outputs its memory configuration. The jth memory configuration output should be interpreted as a pair consisting of a claim about the memory configuration of M at time j, and a CS proof of its correctness. There is a fixed machine V, the verifier, that will accept all pairs of configurations and proofs generated in this way, and will reject other pairs, subject to the usual condition of the CRS model that the verifier may be fooled with negligible probability, and the computational soundness caveat that an adversary with enormous computational resources may also fool the verifier.

Definition 1. *An increasing sequence of integers $\{t_j\}$ is an α-*incremental timeline *if for any j, $t_j - t_{j-1} \leq \alpha$.*

Definition 2. *A Turing machine that makes outputs at every time on an α–incremental timeline is called an α–*incremental output Turing machine*.*

Definition 3 (Feasible Compiler). *Let $C(\cdot,\cdot)$ be a polynomial time Turing machine. We say that C is a* feasible compiler *if there exists a constant c such that for all $k > 0$ and all $M()$ such that $|M| \leq k$, $C(M,k)$ is a Turing machine $T(\cdot)$ taking as input the common random string, such that*

1. *T is a k^c-incremental output Turing machine.*
2. *$space_T(r) = k^c$ for all inputs r.*

In other words, properties 1 and 2 guarantee that each compiled machine T outputs its internal configuration "efficiently often" while working in "efficient space."

Definition 4 (Incrementally Verifiable Computation). *The pair (C,V) is an* incrementally verifiable computation scheme (in the CRS model) *with security K if C is a feasible compiler, V is a polynomial-time Turing machine ("the verifier") and $K(k) : \mathbb{Z}^+ \to \mathbb{Z}^+$, such that the following properties hold: For any Turing machine M with $|M| \leq k$ let the jth output of the compiled machine $C(M,k)$ be parsed as an ordered pair (m_j, π_j^r), representing a claim about the jth memory configuration of M, and its proof; and let r denote the common random string of length k^2. We require:*

1. *(Correctness) The compiled machine accurately simulates M, in that m_j is indeed the jth memory configuration of $M(\epsilon)$ for all j, independent of r.*
2. *(Completeness) The verifier V accepts the proofs π_j^r: $\forall r, V(M,j,m_j,\pi_j^r,r) = 1$.*
3. *(Computational soundness) For any constant c and for any machine P' that for any length k^2 input r outputs a triple $(j, m_j'^r, \pi_j'^r)$ in time K, we have for large enough k that*

$$Prob_r[m_j'^r \neq m_j \wedge V(M,j,m_j'^r,\pi_j'^r,r) = 1] < k^{-c}.$$

We note that for the incrementally verifiable computation scheme to be secure against polynomial-time adversaries we must have K super-polynomial.

2.4 Noninteractive CS proofs of knowledge

We now specify the assumption we make: the existence of noninteractive CS proofs of knowledge.

We note that proofs of knowledge are typically studied in the form of *zero knowledge proofs of knowledge*. In this setting, one party wants to convince another party that he possesses certain knowledge without revealing this knowledge. The reason why he does not simply transmit all his evidence to the other party is that he wishes to maintain his privacy.

In our setting the reason one generation does not just transmit all its evidence to the next generation is not a privacy concern, but rather the concern that the following generation will not have the time to listen to all this evidence.

In both settings, the "knowledge" that must be proven may be considered to be a witness for a member of an NP-complete language: one party proves to the other that he knows, for example, a three-coloring of a certain graph.

In the zero-knowledge setting, our prover does not wish for the verifier to learn a three-coloring of the graph. In the incremental computation setting, our prover is worried that the verifier may not want to spare the resources to learn a three-coloring of the graph.

Related issues were considered in a paper of Barak and Goldreich where they investigated efficient (interactive) ways of providing proofs and proofs of knowledge [4]. Our definition of a noninteractive CS proof of knowledge contains elements from their definition of a *universal argument*.

For the sake of concreteness, we work with a specific NP-complete language, which has the property that for any k the strings in the language of length $4k$ have witnesses of length $3k$. We will require of our CS proof system that instead of returning proofs of length $3k$ (for example, the witnesses) the proofs are shortened to be of length k.

Definition 5. *Let c be a constant. The language $\mathcal{L}_c$ consists of the ordered pairs (M, x) where M is a Turing machine and x is a string such that, letting $k = |M|$ we have:*

1. *$|x| = 3k$.*
2. *There exists a string w of length $3k$ such that M when run on the concatenation (x, w) accepts within time k^c.*

We note that the string w may be thought of as the *NP witness* for (M, x)'s membership in the language. Further, since M may express any polynomial-time computation (for large enough k), the language $\mathcal{L}_c$ is NP complete.[5]

Definition 6 (Noninteractive CS proof of knowledge). *The pair (P, U) is a* noninteractive CS proof of knowledge (in the CRS model) *with parameters $K'(k) : \mathbb{Z}^+ \to \mathbb{Z}^+, c, c_1, c_2$ if P and U are Turing machines such that for all machines M, defining $k = |M|$, and all strings x of length $3k$ the following properties hold:*

1. *(Efficient prover) For any (CRS) string r of length k, $time_P(M, x, w, r) = k^{O(1)}$*
2. *(Length shrinking) For any (CRS) string r of length k, $|P(M, x, w, r)| = k$.*
3. *(Efficient verification) For any (CRS) string r of length k, $time_U(P(M, x, w, r), M, x, r) \leq k^{c-1}$*
4. *(Completeness) For any (CRS) string r of length k, $U(P(M, x, w, r), M, x, r) = 1$*
5. *(Knowledge extraction) There exists a constant c_2 such that for any Turing machine P' there exists a randomized Turing machine $E_{P'}$, the* extractor*, such that for any input (M, x) of length $4k$ such that for all r of length k, $time_{P'}(M, x, r) \leq K'(k)$ and $Pr_r[U(P'(M, x, r), M, x, r) = 1] = \alpha > 1/K'$ we have*

$$Prob[w \leftarrow E_{P'}(M, x) : M(x, w) = 1] > 1/2$$

and the running time of $E_{P'}(M, x)$ is at most k^{c_2}/α times the expected running time (over choices of r) of $P'(M, x, r)$.

[5] One can easily manipulate any NP language into one whose members and witnesses have lengths in the 4:3 ratio by appropriate padding.

3 Constructing incrementally verifiable computation

3.1 Merging proofs

We aim here to reexpress claims of the form $M : s_1 \xrightarrow{t} s_2$ as claims of membership in the language $\mathcal{L}_c$. The equivalence will not be exact but instead, in light of the goals of this paper, computationally sound. We define this relation inductively, for t that are powers of 2. The base case, when $t = 1$, is an exact relation.

Construction 1 (Base Case) *Let T_0 be the machine that interprets its input as a pair of length $3k$ strings (x, w) where x is interpreted as a triple of length k strings $x = (M, s_1, s_2)$, and checks that M when simulated for one step on configuration s_1 ends up in configuration s_2, ignoring the auxiliary input w.*

We note that for strings M, s_1, s_2 of length k, the pair $(T_0, (M, s_1, s_2))$ is in $\mathcal{L}_c$ if and only if $M : s_1 \xrightarrow{1} s_2$. The language $\mathcal{L}_c$ is crucial here, because this is the language which (by assumption) we may find CS proofs for.

We extend this construction, defining machines T_i such that $(T_i, (M, s_1, s_2)) \in \mathcal{L}_c$ is equivalent in a computationally sound sense to $M : s_1 \xrightarrow{2^i} s_2$. In particular, T_i is such that, given CS proofs of the claims $(T_i, (M, s_1, s_2)) \in \mathcal{L}_c$ and $(T_i, (M, s_2, s_3)) \in \mathcal{L}_c$ we can construct a CS proof of the claim $(T_{i+1}, (M, s_1, s_3)) \in \mathcal{L}_c$. Reexpressing these three statements, we see that given a CS proof that "$(M : s_1 \xrightarrow{2^i} s_2)$" and a CS proof that "$(M : s_2 \xrightarrow{2^i} s_3)$" we may construct a CS proof that "$(M : s_1 \xrightarrow{2^{i+1}} s_3)$." Since the lengths of each of these CS proofs is (by definition) k, this is our desired notion of *merging proofs*.

Construction 2 *Define T_{i+1} as a machine that interprets its input as the pair (x, w) where x is interpreted as (M, s_1, s_3) and w is interpreted as (p_1, p_2, s_2), and does the following:*

Check if p_1, p_2 are CS proofs of knowledge respectively that $(T_i, (M, s_1, s_2)) \in \mathcal{L}_c$ and $(T_i, (M, s_2, s_3)) \in \mathcal{L}_c$.

Given x, w, i such that w witnesses the fact that $(T_{i+1}, x) \in \mathcal{L}_c$, we can efficiently construct a CS proof of this fact as $P(T_{i+1}, x, w, r_{i+1})$ by assumption. (We note that we take the common random string r_{i+1} to be dependent on i.) We prove that this construction is computationally sound. In the following, we call a pair $(x = (M, s_1, s_2), p)$ *deceptive* if p proves to the verifier that $(T_i, x) \in \mathcal{L}_c$ but it is not the case that running M for 2^i steps from memory state s_1 reaches memory state s_2. The proof is by induction; the base case of T_0, as observed above, is trivial.

Lemma 1. *For $\alpha \in (\frac{1}{K'}, 1)$ and $b \in (2(2^i + k), K')$, if T^i has the property that no machine running in time b, outputs a deceptive pair $((M, s_1, s_2), p)$ with probability $\frac{1}{2}$ over the random strings $r_0, \ldots, r_i$, then no machine running in time $\frac{\alpha}{2} b / k^{c_2}$ outputs a deceptive pair for the machine T_{i+1} with probability α, over the random strings $r_0, \ldots, r_{i+1}$.*

Proof. This result is a straightforward consequence of the *knowledge extraction* property of the proofs in Definition 6. Assume we have a machine P' that outputs deceptive pairs $(x = (M, s_1, s_3), p')$ for T_{i+1} with probability α (over r) in time $\frac{\alpha}{2} b / k^{c_2}$. We apply the *extractor* $E_{P'}$, and have by definition that $E_{P'}(T_{i+1}, x)$ returns a classical witness w (relative to r_{i+1}) with probability at least $1/2$ in time at most $b/2$. The witness w is a classical witness for (T_{i+1}, x) in the language L, and thus (by the definition of T_{i+1}) w may be interpreted as $w = (p_1, p_2, s_2)$. Further, since w is a classical witness, both the proofs p_1 and p_2 are accepted by the verifier. However, since p' is deceptive, at least one of p_1, p_2 must be deceptive (with respect to T, r_i). In time $2^i + k \le b/2$ we can classically check which one of p_1, p_2 is deceptive, by simply simulating M for 2^i steps on s_1 comparing the current state against s_2, and reporting "p_1" if they agree, "p_2" if they do not. Thus using $b/2 + b/2 = b$ time we have recovered a deceptive pair for T_i with probability at least $1/2$, contradicting our assumption. □

Applying Lemma 1 inductively starting from $b = K'$, letting $\alpha = \frac{1}{2}$ for the first $i - 1$ iterations and $\alpha = \epsilon$ for the last yields:

Lemma 2. *No machine running in time $2\epsilon K' / (4k^{c_2})^i$ outputs a deceptive pair for the machine T_i with probability ϵ, over the random strings $r_0, \ldots, r_i$.*

3.2 The main result

Theorem 1. *Given a noninteractive CS proof of knowledge (P, U, K', c, c_1, c_2), there exists an incrementally verifiable computation scheme (C, V, K) provided $Kk^{2\log k + c_2 \log K} \leq K'$.*

Proof. Making use of the CS proof of knowledge, Construction 2 describes a recursive procedure for generating a proof for 2^i steps of the computation using i levels of a binary recursion. Consider the tree that such a recursion would induce. The leaves of the recursive tree are the memory configurations of M, and the internal nodes j levels above the leaves are proofs of knowledge of recursive depth j (by way of machine T_j) asserting the results of simulating M for 2^j steps. Each node is computable in time polynomial in k from its two children, as this requires just one application of the polynomial-time prover P.

Let $C(M)$ be a machine that performs a depth-first traversal of the binary tree, starting at the leaf corresponding to time 0, visiting each leaf in order, and computing the value of every node it visits. At any moment in such a traversal the "stack" consists of the values of nodes on a path from a leaf to the root. Every time a leaf is visited, let $C(M)$ output the values of all the nodes along this path as a *proof of incremental correctness*. We note that processing any node takes time polynomial in k, and the depth of the recursion is less than k, and so a leaf is visited every $k^{O(1)}$ time. Thus this procedure uses the desired time and space.

We now show that these "stack dumps" in fact constitute computationally-sound proofs.

Consider a subtree whose leaves consist of a range $[t_1, t_2]$. (If the subtree has depth j then t_1 and t_2 will be consecutive multiples of 2^j.) When the recursion finishes processing this subtree, it will store in the parent node parameters $x = (M, s_1, s_2)$ and a proof of knowledge that M when starting in configuration s_1 reaches configuration s_2 in time $t_2 - t_1$.

We note that when the recursion processes leaf t' it must have finished processing all the leaves before t', and thus the leaves spanned by those subtrees in the "stack" must constitute all the leaves before t'. Thus these proofs of knowledge, when considered together, assert the complete result of simulating M from time 0 to time t'.

To check such a sequence of proofs, V verifies their individual correctness, and checks that the start and end memory states for each of the corresponding "theorems" match up.

We note, as above, that if such a sequence of proofs is *deceptive*, then we can (classically) isolate the deceptive proof using $O(t)$ additional time by simulating M. From Lemma 2 with $\epsilon = k^{-\log k}$, the probability that this incrementally verifiable computation scheme fools the verifier is negligible in k provided the time to execute of $C(M)$ plus the additional $O(t)$ classical verification time is at most $2k^{-\log k}K'/(4k^{c_2})^{\log t}$. We note that $C(M)$ consists essentially of constructing t CS proofs, each of which takes time $k^{O(1)} < k^{\log k}$. Thus (C, V, K) is an incrementally verifiable computation scheme for computations of length $t \leq K$ provided $Kk^{\log k} \leq k^{-(\log k) - c_2 \log K} K'$. Rearranging terms yields the desired result. □

4 CS proofs of knowledge in the random oracle model

To explicitly introduce CS proofs of knowledge, and support our hypothesis that there exist noninteractive CS proofs of knowledge in the common reference string model we provide details of such proofs in the random oracle model. Specifically, our construction will satisfy Definition 6 modified by replacing the string r everywhere with access to an oracle $\mathcal{R}$.

The construction of the proofs is based closely on the constructions of Kilian and Micali[12, 13]. The construction of the witness extractor is inspired by a construction of Pass[14].

4.1 Witness-extractable PCPs

One of the principal tools in the construction of CS proofs is the probabilistically checkable proof (PCP)[1, 2]. The PCP theorem states that any witness w for a string x in a language in NP can be encoded into a probabilistically checkable witness, specifically, a witness of length n can be encoded into a PCP of length $n \cdot (\log n)^{O(1)}$ with an induced probabilistic scheme (based on x) for testing $O(1)$ bits of the encoding such that:

– For any proof generated from a valid witness the test succeeds.
– For any x for which no witness exists the test fails with probability at least $\frac{2}{3}$.

In practice, the test is run repeatedly to reduce the error probability from $\frac{1}{3}$ to something negligible in n. In addition to the above properties of PCPs, we require one additional property that is part of the folklore of PCPs but rarely appears explicitly:

Definition 7 (Witness Extracting PCP). *A PCP is* witness-extracting with radius γ *if there exists a polynomial time algorithm W that, given any string s on which the PCP test succeeds with probability at least $1-\gamma$, extracts an NP witness w for x.*

We sketch briefly how this additional property can be attained. Consider the related notion of a *PCP of proximity* (PCPP)[5]:

Definition 8 (Probabilistically checkable proof of proximity). *A pair of machines (P, V) are a PCPP for the NP relation $L = \{(x, w)\}$ with* proximity parameter ϵ *if*

– *When $(x, w) \in L$ the verifier accepts the proof output by the prover:*

$$Prob[V(P(x, w), (x, w)) = 1] = 1.$$

– *If for some x, w is ϵ-far from any w' such that $(x, w') \in L$, then the verifier will reject any proof π with high probability:*

$$Prob[V(\pi, (x, w)) = 1] < \frac{1}{3}.$$

We note that this property is stronger than the standard PCP property since in addition to rejecting if no witness exists, the verifier also rejects if the prover tries to significantly deceive him about the witness. Ben-Sasson et al. showed the existence of PCPPs with $O(1)$ queries and length $n \cdot (\log n)^{O(1)}$ [5]. We use these PCPPs to construct witness-extractable PCPs:

Construction 3 *Let R be an error-correcting code of constant rate that can correct ϵ fraction of errors, with ϵ the PCPP parameter as above. Let $L = \{(x, w)\}$ be the NP relation for which we wish to find a witness-extractable PCP. Modify L using the code R to obtain a relation*

$$L' = \{(x, R(w)) : (x, w) \in L\}.$$

Let P be a PCPP prover for this relation. The verifier for this proof system is just the PCPP verifier for L', which expects inputs of the form $(P(x, R(w)), (x, R(w)))$. Let the witness extractor W for the proof system run the decoding algorithm on the portion of its input corresponding to $R(w)$ and report the result.

Claim. Construction 3 is a witness-extractable PCP with quasilinear expansion, where the verifier reads only a constant number of bits from the proof.

Proof. We note that since R is a constant-rate code and P expands input lengths quasilinearly, this scheme also has quasilinear expansion. Since the PCPP system reads only $O(1)$ bits of the proof, this new system does too.

For any pair $(x, w) \in L$ the proof generated will be accepted by the verifier, so this scheme satisfies the first property of PCPs. If x is such that no valid w exists for the L relation, then no valid $R(w)$ exists under the L' relation and the verifier will fail with probability at least $\frac{2}{3}$, as required by the second property of PCPs.

Finally, to show the witness extractability property we note that by definition of a PCPP, if the verifier succeeds with probability greater than $\frac{1}{3}$ on $(\pi, (x, s))$ then s is within relative distance ϵ from the encoding of a valid witness $R(w)$. Since the code R can correct ϵ fraction errors, we apply the decoding algorithm to s to recover a fully correct witness w. We have thus constructed a witness-extractable PCP for $\gamma = \frac{2}{3}$. □

4.2 CS proof construction

We now outline the construction of noninteractive CS proofs of knowledge, which is essentially the CS proof construction of Kilian and Micali[12, 13]. We present the knowledge extraction construction in the next section.

The main idea of this CS proof construction is for the prover to construct a (witness-extractable) PCP, choose random queries, simulate the verifier on this PCP and queries, and send only the results of these queries to the real verifier, along with convincing evidence that the queries were chosen randomly and independent of the chosen PCP. For security parameter k' (we differentiate from the parameter k used in the non-oracle-based definitions.) the prover sends only data related to k' runs of the PCP verifier, and thus the length of the proof essentially depends only on the security parameter k'.

The technical challenge in the construction is to convince the verifier that the queries to the PCP are independent of the PCP. To accomplish this we use a *random oracle*. Let $\Re$ denote the set of functions

$$\mathcal{R} : \{0,1\}^{2k'} \to \{0,1\}^{k'}.$$

By a *random oracle* we mean a function $\mathcal{R}$ drawn uniformly at random from the set $\Re$. The machines in our construction will have oracle access to such an $\mathcal{R}$.

We start by defining a *Merkle hash*:

Definition 9 (Merkle hash). *Given a string s and a function $\mathcal{R} : \{0,1\}^{2k'} \to \{0,1\}^{k'}$, do the following:*

- *Partition s into chunks of length k', padding out the last chunk with zeros.*
- *Let each chunk be a leaf of a full binary tree of minimum depth.*
- *Filling up from the leaves, for each pair of siblings s_0, s_1, assign to their parent the string $\mathcal{R}(s_0, s_1)$.*

To aid in the notation we define a *verification path* in a tree:

Definition 10 (Verification path). *For any leaf in a full binary tree, its* verification path *consists of all the nodes on the path from this node to the root, along with each such node's sibling.*

The construction of CS proofs is as follows:

Construction 4 *Given a security parameter k', a polynomial-time relation $L = \{(x, w)\}$ with $|w| < 2^{k'}$ and a corresponding witness-extractable PCP with prover and verifier PP, PV respectively, we construct a CS prover P and verifier U.*

P on input (x, w) and a function $\mathcal{R} : \{0,1\}^{2k'} \to \{0,1\}^{k'}$ does the following:

1. *Run the PCP prover to produce $s = PP(x, w)$.*
2. *Compute the Merkle hash tree of s, letting s_r denote the root.*
3. *Using $\mathcal{R}$ and s_r as a seed, compute enough random bits to run the PCP verifier PV k' times.*
4. *Run PV k' times with these random strings; let the CS proof $P^{\mathcal{R}}(x, w)$ consist of the $k' \cdot O(1)$ leaves accessed here, along with their complete verification pathways.*

U on input x, a purported proof π and a function $\mathcal{R}$ does the following:

1. *Check for consistency of the verification pathways, i.e. for each pair of claimed children (s_0, s_1) verify that $\mathcal{R}(s_0, s_1)$ equals the claimed parent.*
2. *From the claimed root s_r run the procedure in steps 3 and 4 of the construction of P, failing if the procedure asks for a leaf from the tree that does not have a verification pathway.*
3. *Accept if both steps succeed, otherwise reject.*

These are essentially the CS proofs of Killian and Micali. In the next section we exhibit the knowledge extraction property of these proofs, and thereby infer their soundness; further properties and applications may be found in the original papers.

4.3 Knowledge extraction

We now turn to new part of this construction, the *knowledge extractor* from part 5 of Definition 6. We construct a *black-box extractor*, that is, a fixed E that takes a description of the machine P' as an input argument, instead of depending arbitrarily on P'.

Recall that we want to construct a machine E that when given a (possibly deceptive) prover P' will efficiently extract a witness w for any x on which

$$Pr[U^{\mathcal{R}}(P'^{\mathcal{R}}(x), x) = 1] > 1/K'.$$

In other words, if P' reliably constructs a proof for a given x, then there is a witness "hidden" inside P', and E can extract one. The general idea of our construction is to simulate $P'^{\mathcal{R}}(x)$ while noting each oracle call and response, construct all possible Merkle trees that P' could have "in mind", figure out based on the output of P' which Merkle tree it finally chose, read off the PCP at the leaves of the tree, and use the PCP's witness extraction property to reveal a witness.

We note that this extractor is slightly unusual in that it does not "rewind" the computation at any stage, but merely examines the oracle calls P' makes; such extractors have been recently brought to light in other contexts under the names *straight-line extractors*[14] or *online extractors*[9]. The principal reason we need such an extractor is that we require the extractor to run in time *linear* in the time of P', up to multiplicative constant k^{c_2}, and we cannot afford the time needed to match up data from multiple runs.

We show that the following extractor fails with negligible probability on the set of $\mathcal{R}$ where $P'^{\mathcal{R}}(x)$ is accepted by the verifier; to obtain an extractor that never fails, we re-run the extractor until it succeeds.

Construction 5 (CS extractor) *Simulate $P'^{\mathcal{R}}(x)$, and let $q_1, ..., q_t$ be the queries P' makes to $\mathcal{R}$, in the order in which they are made, duplicates omitted. Assemble $\{q_i\}$ and separately $\{\mathcal{R}(q_i)\}$ into data structures that can be queried in time logarithmic in their sizes,* $\log t$ *in this case. If for some $i \neq j$ $\mathcal{R}(q_i) = \mathcal{R}(q_j)$, or if for some $i \leq j$ $q_i = \mathcal{R}(q_j)$, then abort.*

Consider $\{q_i\}$ as the nodes of a graph, initially with no edges. For any q_i whose first k' bits equal some $\mathcal{R}(q_j)$ and whose second k' bits equal some $\mathcal{R}(q_l)$, draw the directed edges from q_i to both q_j and q_l.

In the proof output by $P'^{\mathcal{R}}(x)$ find the string at the root, s_r. If s_r does not equal $\mathcal{R}(q_r)$ for some r, then abort. If the verification paths from the proof are not embedded in the tree rooted at q_r, abort.

Compute from x the depth of the Merkle tree one would obtain from a PCP derived from a witness for x. (Recall that for the language $\mathcal{L}_c$ in Definition 5, witnesses have length identical to that of x; in general we could pad witnesses to a prescribed length.) Read off from the tree rooted at q_r all strings of this depth from the root; where strings are missing fill in $0^{2k'}$ instead. Denote this string by pcp.

Apply the PCP witness extractor to pcp, and output the result.

Lemma 3. *Construction 5 when given (P', x) such that $P'^{\mathcal{R}}(x)$ always runs in time at most $2^{k'/4}$ and that convinces the verifier with probability $Pr_{\mathcal{R}}[U^{\mathcal{R}}(P'^{\mathcal{R}}(x), x) = 1] = \alpha > 2^{-k'/8}$, will return a witness w for x on all but a negligible fraction of those $\mathcal{R}$ on which P' convinces the verifier in time $O(k/\alpha)$ times the expected running time of P'.*

Proof. We show that this construction fails with negligible probability. We begin by showing that the probability of aborting is negligible.

Suppose P' has already made $i-1$ queries to the oracle, and is just about to query $\mathcal{R}(q_i)$. This value is uniformly random and independent of the view of P' at this point, so thus the probability that $\mathcal{R}(q_i)$ equals any of q_j or $\mathcal{R}(q_j)$ for $j < i$ is at most $2i \cdot 2^{-k'}$. The probability that this occurs for any $i \leq t$ is thus at most $t^2 2^{-k'}$, which bounds the probability that the extractor aborts in the first half of the extractor.

We note that since no two q_i's hash to the same value, the trees will be constructed without collisions, and since $q_i \neq \mathcal{R}(q_j)$ for $i \leq j$, the graph will be acyclic and thus a valid binary tree. We may now bound the probability that some node on a verification path (including possibly the root) does not lie in the graph we have constructed. Let s_0, s_1 be a pair of siblings on a verification pathway for which the concatenation $(\mathcal{R}(s_0), \mathcal{R}(s_1))$ is not in the graph. Thus P' does not ever query $\mathcal{R}(\mathcal{R}(s_0), \mathcal{R}(s_1))$. Since the proof P' generates is accepted by the verifier, the value of $\mathcal{R}(\mathcal{R}(s_0), \mathcal{R}(s_1))$ must be on the verification path output by P'. Thus P' must have *guessed* this value without evaluating it, and further, the guess must have been right. This occurs with probability at most $2^{-k'}$. Thus the total probability of aborting is at most $(t^2+1)2^{-k'}$.

We now show that if the extractor does not abort, it extracts a valid witness on all but a negligible fraction of $\mathcal{R}$'s. Recall that the CS verifier makes k' calls to the PCP verifier, each of which, if seeded randomly, fails with probability $\frac{2}{3}$ whenever the string *pcp* does not encode a valid witness w.

Consider for some non-aborting $\mathcal{R}$ and some $i \leq t$ the distribution ρ on $\mathcal{R}$ obtained by fixing those values of $\mathcal{R}$ that $P'^{\mathcal{R}}(x)$ learns in its first i oracle calls, and letting the values of $\mathcal{R}$ on the remaining inputs be distributed independently at random. Consider an $\mathcal{R}$ drawn from the distribution ρ. Construct a Merkle tree from the values $\{(q_j, \mathcal{R}(q_j)) : j \leq i\}$ rooted at q_i, i.e., pretending that P', when it finishes, will output $\mathcal{R}(q_i)$ as the root, and let *pcp* be the string read off from the leaves, as in the construction of the extractor. Compute from $\mathcal{R}$ and $\mathcal{R}(q_i)$ as in step 3 of the construction of the CS prover P the k' sets of queries to the PCP verifier. Unless the oracle calls generated here collide with the i previous calls, the PCP queries will be independent and uniformly generated; if witness extraction fails on *pcp* then by definition, these PCP tests will succeed with probability at most $\frac{1}{3}^{k'}$. Adding in the at most $t^2 2^{-k'}$ chance that, under this distribution, one of the new oracle calls will collide with one of the old calls, the total probability that *pcp* is not witness-extractable, yet the tests succeed, is at most $(t^2+1)2^{-k'}$.

Consider all distributions ρ with i fixed values as above. We note that the distributions have disjoint support, since no fixed $\mathcal{R}$ could give rise to two different initial sequences of oracle calls. We note also that any $\mathcal{R}$ either aborts or induces such a distribution ρ with i fixed values. We now vary i from 1 to t. Consider the set of non-aborting $\mathcal{R}$ for which there is some i such that the string $pcp_i^{\mathcal{R}}$ is not witness-extractable yet the PCP tests generated by $\mathcal{R}$ all succeed. By the above arguments and the union bound this set has density at most

$$t(t^2+1)2^{-k'}.$$

By assumption the set of $\mathcal{R}$ for which the verifier accepts $P'^{\mathcal{R}}(x)$ has density at least $2^{-k'/8}$. Thus for all but a negligible fraction of these $\mathcal{R}$, the string *pcp* is witness-extractable, and we may recover a witness w as desired. □

We note that our extractor runs logarithmic factor slower than P'. Since the running time of P' is subexponential in k, the extractor takes time $o(k)$ more than P'. As noted above, if P' returns an acceptable proof with probability α we may have to run the extractor $1/\alpha$ times (in expectation) before it returns a witness. Since by the above construction $\alpha \sim 1$, our extractor runs k times slower than P' and always returns acceptable proofs, as desired.

5 Acknowledgements

I am indebted to Silvio Micali for introducing me to the area of CS proofs and for many invaluable discussions along the way to writing this paper. I am grateful to Ran Canetti, Brendan Juba, and Rafael Pass for their many helpful comments on earlier versions of this paper. I would also like to thank the anonymous referees for their suggestions; those mistakes that remain are my own, and likely due to lack of understanding of what the referees were recommending.

References

1. Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and the hardness of approximation problems. *Journal of the ACM*, 45(3):501–555, May 1998.
2. S. Arora and S. Safra. Probabilistic checking of proofs: A new characterization of NP. *Journal of the ACM*, 45(1):70–122, January 1998.
3. L. Babai, L. Fortnow, and C. Lund. Non-deterministic exponential time has two-prover interactive protocols. *Computational Complexity*, 1, p.3-40. 1991.
4. B. Barak and O. Goldreich. Universal Arguments. *Proc. Complexity* (CCC) 2002.
5. E. Ben-Sasson, O. Goldreich, P. Harsha, M. Sudan, and S. Vadhan. Robust PCPs of proximity, shorter PCPs and applications to coding. *STOC* 2004, pp. 1-10.
6. M. Blum, A. De Santis, S. Micali, G. Persiano. Noninteractive Zero-Knowledge. *SIAM J. Comput.* 20(6): 1084-1118 1991.
7. M. Blum, P. Feldman, S. Micali. Non-Interactive Zero-Knowledge and Its Applications (Extended Abstract). *STOC* 1988, pp. 103-112.
8. R. Canetti, O. Goldreich, and S. Halevi. The Random Oracle Methodology, Revisited, *STOC 1998*, pp. 209-218.
9. M. Fischlin. Communication-efficient non-interactive proofs of knowledge with online extractors. *Advances in Cryptology* 2005.
10. O. Goldreich and M. Sudan. Locally testable codes and PCPs of almost-linear length. *FOCS* 2002.
11. S. Goldwasser, S. Micali, and C. Rackoff. The Knowledge Complexity of Interactive Proof Systems. *SIAM J. on Computing*, 18(1), 1989, pp. 186-208.
12. J. Kilian. A note on efficient zero-knowledge proofs and arguments. *STOC*, 1992, pp. 723-732.
13. S. Micali. Computationally Sound Proofs.*SIAM J. Computing* 30(4), 2000, pp. 1253–1298.
14. R. Pass. On deniability in the common reference string and random oracle model. *Advances in Cryptology*, 2003, pp. 316-337.
15. Adi Shamir. IP = PSPACE. *Journal of the ACM*, 39(4), p.869-877. October 1992.