Computationally-Sound Proofs

Silvio Micali

Department of Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139, USA
silvio@theory.lcs.mit.edu

Abstract. This paper puts forward a new notion of a proof based on computational complexity, and explores its implications to computation at large.

Computationally-sound proofs provide, in a novel and meaningful framework, answers to old and new questions in complexity theory. In particular, given a random oracle or a new complexity assumption, they allow us to prove that verifying is easier than deciding; to provide a quite effective way to prove membership in computationally hard languages (such as $\mathcal{C}o\text{-}\mathcal{NP}$-complete ones); and to show that every computation possesses a short certificate vouching its correctness.

1 Introduction

A new notion. Proofs are fundamental to our lives, and as for all things fundamental we should expect that answering the question of what a proof is will always be an on-going process. Indeed, we wish to put forward the new notion of a *computationally-sound proof* (*CS proof* for brevity) which achieves new and important goals, not attained or even addressed by previous notions.

Informally, a CS proof of a statement S consists of a short string σ, very easy to verify and as easy to find as possible, offering a strong computational guarantee about the verity of S. By "very easy to verify" we mean that the time necessary to inspect a CS Proof of a statement S is poly-logarithmically shorter than that required to decide S. By "as easy to find as possible" we mean that a CS proof of a *true* statement (i.e., for the purposes of this paper, *derivable* in a given axiomatic theory) can be computed in a time essentially comparable to that needed to decide the statement. Finally, by saying that the guarantee offered by a CS proof is "computational" we mean that false statements either do not have any CS proofs, or such "proofs" are practically impossible to find.

Implementations of CS proofs. The value of a new notion, of course, crucially depends on whether it can be sufficiently exemplified. We provide two

main implementations of our notion. The first consists of an explicit construction, based on a random oracle, that provably yields a CS proof system without any complexity assumption. The second implementation, which is the subject of a forthcoming paper, is cryptography-based, and yields a CS proof system under a new complexity assumption.

Applications of CS proofs. CS proofs provide, in a new and meaningful framework, very natural answers to some of our oldest questions in complexity theory. In particular, they imply that verifying is poly-logarithmically easier than deciding. More importantly, they actually imply that this is *almost always* the case, rather than a phenomenon possibly occurring for some special theorems. CS proofs also provide a quite effective way for proving membership in computationally hard languages (e.g., $\mathcal{C}o$–$\mathcal{NP}$ complete ones).

In addition, CS proofs have novel and important implications for computational correctness. In particular, they imply that every computation possesses a short certificate vouching its correctness. Cryptography-based CS proofs also imply that any heuristic or program for a $\mathcal{NP}$-complete problem is cryptographically checkable. This application at the same time extends and demonstrates the wide applicability of Blum's [8] original framework for checking program correctness.

We wish to emphasize that the above mentioned applications of CS proofs to computation at large have been obtained by means of surprisingly simple arguments. Indeed, after setting up the stage for the new notion, the results about computational correctness fall with the ripeness of a Newtonian apple. This simplicity, in our opinion, eloquently vouches for the power of the new perspective.

Roots of CS proofs. In conceiving and constructing CS proofs, we have benefited from the research effort in interactive and zero-knowledge proofs. In particular, the notion of a probabilistically-checkable proof [3] [13] and and that of a zero-knowledge argument [11] have been the closest sources of inspiration in conceiving the new notion itself. In exemplifying the new notion, most relevant is a construction of Kilian's [22]; also relevant have been the works of [14] and [6].

2 Prior Notions and New Goals

The problem of defining the intuitive notion of an efficient proof has definitively attracted a lot of attention in the last three decades; indeed, the notions of $\mathcal{NP}$, $\mathcal{IP}$, and $\mathcal{PCP}$ are notable fruits of this long research effort. Yet, our developing CS proofs has been motivated by our perceiving and wishing to express some new

aspects of an efficient proving process. These additional aspects are informally presented in this section, and our way to capture them formally in the next one.

Prior to sketching out our new goals, we recall some prior notions of an efficient proof, both because they have been inspiring for our venture, and because they will, by contrast, help us express what our new desiderata are. We actually find it useful going as far back as recalling the "classic notion of a proof," but in a way that instrumentally paves the road to our new notion.

2.1 Prior Notions of a Proof

Proceeding directly to give the definition of a CS Proof is attractive in that it would minimize the number of possible objections. (After all, one can define anything.) We feel, however, that CS Proofs are better understood "in context", that is, by comparing them to the notions that preceded them.

To be sure, no such comparison can be objective. Classical proofs, for instance, and CS Proofs are so *different* that any comparison presupposes reducing them first to a common "denominator". In so doing (apples and oranges as they are), one must introduce some distortion or oversimplification (e.g., overemphasizing some of their aspects at the expense of others), and rely heavily on the fact that "words" may remain the same, while their "meanings" do not. But the alternative would be not to compare at all.

Accordingly, below we provide a summary of prior notions for pure comparison purposes (no survey intended), with the understanding that its value is that of a *subjective* summary. In a sense, it is the author's account of how he came to think of CS Proofs.

The Classical Notion of a Proof

What is true? What does it mean to prove that something is true? These are questions that go hand in hand. Indeed, as formalized in the first half of this century by a brilliant series of works, the classical notion of a proof [1] is inseparable from that of a true statement. Given any finite set of axioms and inference rules,

[1] Thinking that the intuitive notion of a proof has remained unchanged from the times of classic Greece (i.e., thinking that people like Peano, Zermelo, Frankel, Church, Turing, and Gödel have only contributed its rigorous formalization and the discovery of its inherent limitations) is certainly appealing, but unrealistic. Personally, we believe that no notion so fundamental and so human can remain, not even intuitively, the same across so different spiritual experiences and historical contexts. No doubt, our yearning for permanence (dictated by our intrinsically transient nature) predisposes us to perceive more continuity in our endevours than may actually exist.

Having said this, the author firmly believes that the above mentioned formalizations of the notion of a classical proof are the only meaningful ones!)

the corresponding true statements form a *semi-recursive* set. (Again, throughout this work *true* is considered equivalent to *derivable*.) In the expressive and elegant approach of Turing, such sets possess two equivalent characterizations particularly important for our enterprise, one in terms of deciding algorithms and one in terms of (proof-)verifying algorithms:

1. A language (set of binary strings) L is semi-recursive if and only if there exists a *(deciding)* Turing machine D such that

$$L = \{x \ : \ D(x) = YES\}.$$

2. A language L is semi-recursive if and only if there exists a total *(verifying)* Turing machine V such that

$$L = \{x \ : \ \exists \sigma \in \{0,1\}^* \text{ such that } V(x,\sigma) = YES\}.$$

Deciding the verity of a statement is thus a purely algorithmic process, and classical proofs —the σ's of the second definition— are just strings.

Stating that classical proofs are "just string" might appear quite diminishing. In an ordinary sense, proofs possess additional characteristics (insightfulness, elegance, usefulness, etc.) and may even be emotionally charged, but, formally speaking, a classical proof is a string satisfying special syntactic constraints. Call it "Turing machine" or "procedure", any proof system that is formal enough must specify in sufficient detail a *verification algorithm*. Proofs then are those strings that (relative to a given statement) are "accepted" by such an algorithm.[2]

We should also clarify that, though we all experience abbreviations and "reusability" of proofs to be essential to the mathematical enterprise (we rarely, if ever, deal with proofs from "scratch"), in this paper proofs are thought of as self-contained and abbreviation-free.[3]

Finally, let us recall that, for the purposes of this paper a "true" theorem simply is one derivable in a given theory. Thus, for variety of discourse, we may call a member of a semirecursive language a *theorem* or a *true statement*, and a classical proof a *derivation*.

---

[2] The "non-logician" should realize that this is also the case for formalizations other than Turing machines. For instance, roughly, a classical proof of a statement can be viewed as a sequence of "lines", each being an axiom (out of a given set) or obtained from previous lines according to one inference rule out of a finite set (e.g., if one line is "A" and another is "A implies B", then it is fine to write down a new line consisting of "B") whose last line is the statement in question. Here too, therefore, proofs are strings satisfying some proper syntactic constraints.

[3] Modularity is however so important that we actually wonder whether it is possible to develop a meaningful and formal model to measure the "centrality" and "usefulness" of lemmas and theorems in a given theory (e.g., one where weights are algorithmically associated to each derivable statement).

Proofs Versus Efficient Proofs

The two classical definitions of semi-recursiveness recalled above are equivalent in the sense that they identify the same sets.[4] Of course, they are syntactically different, but they are indistinguishable from the point of view of computational convenience. Indeed, classical proofs are more a way of expressing what is *in principle true* rather than a way of capturing what is *efficiently provable*. In fact, providing a derivation is certainly a way to convince someone that a given theorem is true, but is not necessarily at all efficient: a classical proof may be arbitrarily long, or its relative verifying algorithm may take arbitrarily many computational steps to verify it.

The above criticism, of course, unfairly presupposes the more modern comple xity-theoretic point of view. Indeed, going back to a recent past, one might legitimately object that what is true has nothing to do with how long it takes to determine it. It is thus better to say that the advent of complexity theory has allowed us to perceive a *new concept* by distinguishing two notions, "*truth*" and "*proof*," in what used to be —at least formally if not intuitively— an undifferentiated one.

To be sure, proving and deciding may have always been psychologically different notions. (Personally, we perceive the first as the solitary process of determining what is true, and the second as the social process of conveying to others the results of this determination.) But from a computational point of view, *if one discards efficiency*, proving would just be a synonym of deciding, whose only *raison d'être* would be variation of discourse. Indeed, if the time needed for "critically receiving" a proof were not substantially less than that of deciding, one might as well run a deciding algorithm rather than receiving a proof from someone else.

But realizing that proving should be efficient (for being a meaningful and distinct notion), and finding the "right" notion of an efficient proof are two very different things; particularly because what we perceive as "right" changes over time.

2.2 Complexity, Polynomial Time and Classical Proofs

Intuitively, we expect that the complexity of —say— deciding primality grow as a function of integer size. Indeed, underlying any notion of complexity lies

[4] Indeed, it is immediately seen that if a language L possesses a verifying Turing machine V, then it also possesses a deciding algorithm D, namely, the one that, on input a string x, looks for a classical proof that $x \in L$ by examining, in lexicographic order, all strings, stopping only when one is found. Conversely, the history of the accepting computation of a deciding algorithm D on input x is itself a string that is "easy to verify."

the question of how inputs are represented. Complexity assumes an infinity of inputs and that these inputs are coded in binary.[5]

A bit more formally, let $f(n)$ be a non decreasing function from Z^+ to Z^+, and let L be an infinite language. We say that L's complexity is $O(f(n))$ if there is a (deciding) Turing machine M such that, for (any n and) any n-bit input belonging to L, halts outputting YES within $f(n)$ steps, and rejects otherwise. (By a step of a Turing machine we mean any of its elementary operations —such as a state transition, reading or writing a symbol, etc.) By saying that a language is polynomial-time (decidable) we mean that it has complexity $O(n^c)$ for some constant c.

Sometimes, a different definition of input length is used. For instance, it is traditional -as well as natural– to assume that a graph of n nodes has input length n. "Correctly", however, this length should be $O(nlogn)$: such a graph has at most n^2 edges, and $\log m$ bits suffice to give distinct names to m objects. It should be noticed, however, that this "slight" difference in input length does not affect which graph languages are recognizable in polynomial time. (Indeed, a large part of the success of the notion of polynomial time is due to robustness properties such this.)

Given the nature of the most familiar examples, one may wonder whether the complexity point of view applies to "all proofs" and "all theorems" or to just a subset of them of a more-or-less combinatorial flavor. For instance: What is the complexity of Fermat's last theorem? What is the complexity of establishing whether a given multivariate polynomial of degree 4, $P(x_1, ..., x_n)$, has real roots? What is the length of such inputs?

To address such questions, one should realize that complexity is a very *language-dependent* notion: it all depends on how one groups together *infinitely many* strings, rather than on their combinatorial nature. (In particular, the complexities of a language and a sub-language can be vastly different.)

In fact, though addressing a combinatorial problem, . speaking of the complexity of establishing whether a *single* specific graph is Hamiltonian is not meaningful in a complexity framework: a trillion steps performed on a given graph G is a "constant number" of operations. (It is like speaking of the asymptotic behaviour of a function f whose only known value is that at point 3.) By contrast, speaking about the complexity of deciding Hamiltonicity is meaningful: one considers the infinite language consisting of all Hamiltonian graphs and then see how the computational effort of deciding membership in it grows with input size.

Similarly, the question of the complexity of Fermat's last theorem (or that

[5] Decision algorithms would appear "much faster" if inputs were coded in unary; while adopting any base greater than 2 would result in a constant savings in input length (which, for instance, would not affect the notion of polynomial time).

of a given polynomial P having real roots) is not well defined. The problem is not that Fermat's last theorem is not of a combinatorial nature. Nor does the problem consist of how one can measure the input size of Fermat's last theorem. (Its statement, in fact, though making assertions about infinitely many integers, has a finite description. Thus, choosing a suitable encoding scheme yields a clear input size: the length of the string encoding the statement.) The problem is that, in order to discuss complexity, one needs to be dealing with an infinite language. But this does not mean that the question of complexity does not apply to a classical setting! It only means that we must *re-phrase* the question in a proper manner.

While in complexity theory one is usually conscious of selecting "arbitrarily" which languages to investigate, in a classical setting one "ordinarily" assumes to be dealing with a single set of axioms (and inference rules) —the "natural" ones— and thus with a single language: that of all statements derivable in such a *fixed* system.

One can thus both formally and meaningfully consider "all theorems" as a language. All that is needed is an encoding for statements and proofs, so that a (verifying) Turing machine can be specified. This language is obviously undecidable, but complexity applies quite naturally if one considers proper sublanguages of it by bounding proof length: for instance, the language of all statements whose proof is at most n^2 bit-long (where n is the statement length), which we may conjecture to be not polynomial-time decidable.

As we shall see, for any possible language L, CS Proofs succeed in proving membership in L by means of strings that are very short, very easy to verify, and as easy to produce as possible. In particular, therefore, CS Proofs apply to the language of "all theorems," by replacing any classical proof by a much shorter string.

NP, IP, PCP, and Zero-Knowledge Arguments

A survey paper discussing in greater detail these prior complexity-theoretic approaches to the notion of a proof would be very valuable. But, as already said, ours is not such a paper: all we want is to put forward and discuss a new notion of a proof. We thus recall these prior notions only to the extent necessary to illustrate and provide context for our new one.

Non-deterministic Polynomial time. The first attempt to capture what is efficiently provable is Cook's [12], and independently Levin's [25], beautiful notion of $\mathcal{NP}$. Informally, a language L belongs to the class $\mathcal{NP}$ if every string x in it possesses a *short* (i.e., polynomially-long in the length of x) derivation,

and its verifying algorithm is efficient (i.e., runs in polynomial time). Thus, $\mathcal{NP}$-proofs are a special type of classical proofs.

Interactive proofs. A non-classical notion of a proof, that of an *interactive proof*, was subsequently put forward by Goldwasser, Micali, and Rackoff [18] and independently by Babai and Moran [4]. According to their point of view, proofs are *processes* rather than syntactic objects. Because the notion of a CS proof "mirrors" to a large extent that of an interactive proof, it will be useful to recall the latter a bit. Informally, an interactive proof is a *game* between two algorithms: a Prover P, who can perform an arbitrary amount of computation, and a Verifier V, who is bound to computing in polynomial time. Given the same statement as an input, the two algorithms compute *interactively* (i.e., they send each other messages during the proving process), and *probabilistically* (i.e., they can toss coins as part of their strategies).

A Prover-Verifier pair computing as above, (P, V), is said to be an *interactive proof-system* for a language L if, on input a string x, the following two conditions hold:

1. (Completeness): If $x \in L$, then P always convinces V (i.e., V outputs YES at the end of any such computation); and,
2. (Soundness): If $x \notin L$, then, for any false prover P', the probability of P' convincing V (solely computed over the coin tosses made by the two algorithms on input x) is negligible.

It is easily seen that the class of languages having an interactive proof-system, $\mathcal{IP}$, contains $\mathcal{NP}$, but it may contain much more. (Indeed, thanks to the works of Fortnow, Karloff, Lund, and Nisan [26] and Shamir [37] we now know that $\mathcal{IP} = PSPACE$.)

Probabilistically checkable proofs. We shall talk about $\mathcal{PCP}$ [3] [13] in some detail when constructing CS proofs. For now, let us just recall the notion of a *probabilistically checkable proof* at a very high level. Quite succinctly, Babai Fortnow, Levin and Szegedy [3] and Feige, Goldwasser, Lovasz, Safra and Szegedy [13] have, independently, provided explicit algorithms transforming an $\mathcal{NP}$-witness, σ, into a new proof (i.e., string), τ, which is polynomially longer, but whose correctness can be detected in probabilistic poly-logarithmic time, by properly sampling it in a few locations rather than by reading it in its entirety. Probabilistically checkable proofs can thus be viewed as a special type of syntactic proofs.

Zero-knowledge arguments. As introduced by Brassard, Chaum, and Crépeau's [11], *zero-knowledge argument* are special proof systems. ' These systems do not make a larger class of theorems efficiently provable. Rather, in the spirit of Goldwasser, Micali, and Rackoff's prior notion of a *zero-knowledge proof* [18], they aim at minimizing the "amount of knowledge" conveyed in a "proof" of membership in an $\mathcal{NP}$ language.

Informally, a zero-knowledge argument is an interactive protocol enabling a Prover to convince a Verifier that an input belongs to an $\mathcal{NP}$-language, but without conveying any more knowledge than the mere fact that a given witness of such a membership exists. (In particular, no Verifier may learn such a witness in its entirety, nor whether w has more 0's than 1's, etc.)

In a zero-knowledge argument, both Prover and Verifier are polynomial-time machines. Further, when they interact on input a string x belonging to an $\mathcal{NP}$-language L, it is assumed that the Prover also has (on a special tape inaccessible to the Verifier) an $\mathcal{NP}$-witness, w, of $x \in L$.

At an intuitive level, the Prover sends the Verifier what may be conceptualized as "special encodings" of w,[6] and then answers some special questions of the Verifier about such encodings. The properties enjoyed by such an interaction are the following: (1) the Verifier, no matter how long he might compute, cannot learn anything about the specific witness w possessed by the Prover, and (2) if the Prover properly answers the Verifier's questions, the Verifier is guaranteed that, unless the Prover succeeds in breaking the special encryption scheme, the Prover must possess a witness of $x \in L$.

In sum, therefore, zero-knowledge arguments have two distinct aspects, correctness and zero-knowledgeness, and privilege the second rather than the first. In fact, a zero-knowledge argument offers an *unconditional* guarantee of zero-knowledgeness but only a *conditional* guarantee of correctness. (I.e., if the Prover could break the special scheme, he could cheat the Verifier into believing false statements about $\mathcal{NP}$-language membership. But, knowing that the Prover is bound to polynomial-time computation, the Verifier also knows that he will not succeed in such a breakage, but with negligible probability.)

It should be noticed that these guarantees are the exact reverse of those offered by the type of zero-knowledge proofs used earlier by Goldreich, Micali, and Wigderson to prove that any $\mathcal{NP}$-language has a zero-knowledge proof-system

[6] Each special encoding scheme, E, is actually sent by the Verifier to the Prover, is probabilistic, and enjoys the following two properties: (a) each ciphertext is equally likely to encode any cleartext (i.e., for any ciphertext C and any cleartext x, there is the same number of random strings r —of a given length— such that $C = E(x, r)$); however, (b) without succeeding in breaking the scheme, he who has generated a ciphertext C by encoding a cleartext x with a random string r, cannot decode C in any other way (i.e., he cannot find a different cleartext x' and a random string r' such that $E(x', r') = C$).

[20]. In fact, their zero-knowledge proofs offer an *unconditional* guarantee of correctness (i.e., every cheating prover, no matter how much computational power it may have, has but a negligible probability of convincing its verifier that a false statement is true), but only a *conditional* guarantee of zero-knowledgeness, that is, one holding for any Verifier that is bound to polynomial-time computation.[7]

2.3 New Goals For Efficient Proofs

A General Objective. A common objective of all prior notions of an efficient proof is guaranteeing efficient verification. We too share this objective, but provide a different interpretation of what "efficient verification" should mean. In addition, believing that a proof system should specify both the process of *proving* and that of *verifying*, we demand that proving too be efficient.

At the highest and informal level, our objective is

finding the right relationship between deciding, efficiently proving, *and* efficiently verifying *that a statement is true.*

As it is often the case, realizing and stating the objective carries in itself the means to reach it. It implicitly states that the right notion of proving should arise from that of deciding. As we have already said, we view deciding as the (solitary) process of *convincing ourselves* of what is true, and proving as the (social) process of *convincing others* of what is true, and we believe that one should be able to convince himself for being able to convince others.

Accordingly, the overall usefulness (i.e., the efficiency) of the proof of a theorem arises only when contrasted with the effort necessary for deciding it. In other

[7] That is, given much longer time to compute, the Verifier could reconstruct any piece of hidden knowledge, and thus a proof of the given statement in its entirety. Zero-knowledge proofs of this type, as introduced by [18], are called *computational zero-knowledge proofs*. In the same paper [18], also other types of zero-knowledge proofs (called *perfect* and *statistical* zero-knowledge proofs) have been introduced and explicitly constructed. These latter zero-knowledge proofs offer an unconditional guarantee of *both* correctness and zero-knowledgeness. Though this would appear to be a preferable notion, only a few notable languages are known to possess zero-knowledge proofs of this type (e.g., graph isomorphism and graph non-isomorphism [20]). Fortnow [15] has actually shown that if a zero-knowledge proof of this type were found for a single $\mathcal{NP}$-complete problem, then the polynomial-time hierarchy would collapse at the second level. Some researchers have interpreted this result as saying that this type of zero-knowledge proof is unlikely to be powerful enough so as to encompass all $\mathcal{NP}$ languages (i.e., that, most likely, one has to live with some form of conditional guarantee if one wishes to have a general enough notion of zero-knowledge proofs). By contrast, based on a very weak cryptographic assumption, [20] have shown that any language in $\mathcal{NP}$ possesses a computational zero-knowledge proof system.

words, our main and informal goal consists of arguing that the right notions of *efficiently proving* and *efficiently verifying* are not definable in "absolute terms" (as in some sense it has been done so far to express "efficient verifiability"), but must be expressed "relative to" that of *deciding.*

Three Main Goals. We articulate the above vast and vague objective into three slightly smaller and more concrete goals for an efficient proof-system.

1. **Efficient Verifiability.** We should construct proof-systems so that, for *all* theorems, the complexity of verification is poly-logarithmic in the complexity of deciding.
2. **Efficient Provability.** We should construct proof-systems so that the prover's computational complexity is polynomially close to that of deciding.
3. **Universality:** We should construct proof-systems capable of efficiently proving membership in *any* semi-recursive language.

As we shall point out in the sequel, our CS proofs also achieve additional goals, but we do not consider them essential to the notion of an efficient proof. Let us now explain the novelty of our goals by contrasting them with what was achievable by some prior proof systems.

Efficient Verifiability

Relative Efficiency. As in previous proof-systems, we too require that verification be "efficient." Differently from those previous notions, however, we do not express this efficiency requirement in absolute terms. Rather, in accordance to our premise, we regard a proof-system as making verification of a given statement efficient if it enables the computational effort of the verifier to be much easier than that he would have to invest, without any help from a prover, in order to establish the verity of the same statement. (I.e., rather than requiring that verifying be *easy* —e.g., polynomial-time, like in $\mathcal{NP}$ or $\mathcal{IP}$,— we require that it be *easier* than deciding.)

Having settled on a relative measure, we now face the problem of quantifying what computational savings are deemed to make verification efficient. We choose to consider verification efficient if its complexity is polylogarithmic in the complexity of deciding. Though somewhat arbitrary, our choice stems from two simple reasons: "logarithmic" because we wish that the advantage of verification over decision be *substantial* (whenever the decision time is substantial!), and "poly" because we wish such an advantage to be reasonably *independent* from any specific computational model.

Ubiquitous Efficiency. There is an additional, novel, and crucial aspect to our goal of efficient verifiability; namely, that such convenience should arise for *all* theorems, and not for just some rare ones. Let us explain.

According to prior notions, one might consider expressing that verification is easier than decision by saying that $\mathcal{P} \neq \mathcal{NP}$. The latter statement, however, might only entail the existence of a super-polynomial gap between decision and verification for some *rare* inputs, such as certain instances of satisfiability. (Indeed, satisfiability appears to be "easy on most inputs.")

According to our present point of view, instead, a proof-system does not make verification sufficiently efficient unless it makes it polylogarithmically easier than deciding for *all* theorems. (Of course, however, for certain trivial statements, "polylogarithmically easier" may be no savings at all in absolute terms! Indeed, for some polynomials Q and some sufficiently small values v, $Q(\log v)$ may be greater than v.)

Efficient Provability

Implicitly or explicitly, proofs involve *two* agents, a Prover and a Verifier. We thus believe that the right notion of a proof should require efficiency for *both* agents and that the efficiency of a prover should not be measured in absolute terms, but relatively to the complexity of the decision problem at hand.

To us, measuring prover efficiency relative to decision complexity is a quite natural choice. Indeed, based on our intuition that the complexity of convincing someone else cannot be lesser than that of convincing ourselves (and based on our view that deciding is the process of convincing ourselves), we believe that the complexity of proving cannot be (at least much) lower than that of deciding; while it could be much greater. We thus regard as necessary properties of an efficient proof-system not only that (1) the Prover succeeds in convincing the Verifier whenever the theorem at hand is true (the older *completeness* condition of an interactive proof-system), but also that (2) the amount of computation needed by the Prover to convince the Verifier is reasonably close to that needed to decide that the given theorem is true. We refer to the simultaneous holding of these two properties as *feasible completeness*.

Feasible completeness a novel *requirement* for proof-systems, but we may wonder whether some prior proof-systems "happened" to enjoy it anyway. The answer is possibly no. Consider, for instance a $\mathcal{NP}$-language L (preferably not $\mathcal{NP}$-complete[8]) decidable by an algorithm D in, say, $n^{\log n}$ time. Then, in the

[8] Above, we assume that L is not $\mathcal{NP}$-complete to avoid raising two issues at once. Indeed, due to our current complexity measures, $\mathcal{NP}$-proving membership in a $\mathcal{NP}$-complete language *appears* feasible. In fact, because of the self-reducibility, if L is $\mathcal{NP}$-complete and decidable in $n^{\log n}$ time, then a $\mathcal{NP}$-witness of $x \in L$ is findable

$\mathcal{NP}$ mechanism, proving that a given string x belongs to L entails finding a polynomially-long and polynomial-time inspectable witness w_x. But the complexity necessary to find such an *insightful* string may vastly exceed that of running algorithm D on input x for $n^{\log n}$ steps! Indeed, it may be as high as $O(2^n)$. In other words, while a few months of hard work may suffice for proving to ourselves (i.e., for deciding) that a given mathematical statement is true, it is conceivable that a life time may not be enough for finding an explanation followable by a verifier with a limited attention span (e.g., a child).

Efficient provability might also not hold for the $\mathcal{IP}$ proof mechanism. Indeed, often the best way to prove membership in an $\mathcal{IP}$-language consists of invoking the mentioned and general $\mathcal{IP} = PSPACE$ protocol, which is extremely wasteful of prover resources.

Realizing the importance of feasible completeness in a proof-system allows us to raise a variety of intriguing questions about $\mathcal{NP}$ and $\mathcal{IP}$.[9] But our point is not determining which proof-systems *happen* to enjoy feasible completeness. Our point is that feasible completeness is a *must* for every "sufficiently right" proof-system.

Universality

The proof-systems discussed in Subsection 2.1.3 have only a limited "range of action." For instance, an interactive proof-system (P, V) is defined only with respect to proving membership in a *specific* language L. Different languages have therefore had different interactive proof-systems, or none. Indeed, even considering the classes of all languages having a proof-system of a *given type*, among those discussed in that subsection, one obtains sets of languages (e.g., $\mathcal{NP}$, $\mathcal{IP}$, etc.) quite small with respect to the set of all recursive languages.

In contrast with those proof-systems, we consider *universality* (i.e., the capability of acting on the entire range of recursive languages) to be a necessary property of every "sufficiently right" proof-system. By this we do not just mean

---

in $poly(n) \cdot n^{\log n}$ time. However, as we shall see in subsection 6.5, $\mathcal{NP}$ may not enjoy feasible completeness even when one focuses solely on $\mathcal{NP}$-complete languages.

[9] For instance, in an intuitive language,

$\mathcal{Q}1$: *What is the computational complexity required from any $\mathcal{IP}$-prover of Unsatisfiability?*

$\mathcal{Q}2$: *What is the complexity required from any $\mathcal{NP}$-prover for, say, Graph Isomorphism?*

$\mathcal{Q}3$: *Are there $\mathcal{NP}$-languages L, such that proving membership in L may require much less computation from an $\mathcal{IP}$-prover than from an $\mathcal{NP}$-prover?*

(I.e., can giving a Prover "more freedom" save him much work?) In particular,

$\mathcal{Q}3'$: *What is the computational complexity required from any $\mathcal{IP}$-prover of Satisfiability?*

that every recursive language should admit a proof-system of the "right" type. We actually mean that a "right" proof-system should be able to prove membership in *any* recursive language. That is, for any language L and any member x of L, on input x and a suitable description of L, a right proof-system should be able to prove, *efficiently*, that x belongs to L.

As we shall see, we consider a deciding algorithm for L to be a suitable description of L (and one that immediately allows us to assess the efficient provability and efficient verifiability of a proof system).

3 Computationally-Sound Proofs

In this section we put forward the notion of a CS proof to approximate in a reasonable manner the goals set forth in the previous section. We actually present three main types of CS proof-systems, according to the type of randomness source they use. All types of CS proofs, however, share the same paradigm.

A new paradigm. Similarly to $\mathcal{IP}$, a CS proof-system consists of a pair of algorithms: a Prover and a Verifier. Differently from $\mathcal{IP}$, however, in a CS proof-system Prover and Verifier do not interact with one another. Rather, on input the statement of a theorem, the Prover outputs a string, called a CS proof of the theorem. The Verifier computes on inputs the statement of the theorem and an alleged CS proof of it, and either accepts or rejects.

So far, therefore, CS proofs are *syntactic objects*, much in the spirit of classical proofs. There is, however, a major difference between CS proofs and classical ones. Classical proofs ensure that all *provable* statements are *true*. CS proofs break with this tradition: they allow the existence of false proofs, but they ensure that these are computationally impossible to find. That is,

False CS proofs may exist, but they will "never" be found.

Indeed, each CS proof specifies a security parameter, controlling the amount of computing resources necessary to "cheat" in the proof, so that these resources can be made arbitrarily high. Accordingly, CS proofs are meaningful only if we believe that the Provers that have produced them, though more powerful than their corresponding Verifiers, are themselves computationally bounded.[10]

From a practical point of view, this shift of paradigm is *adequate*. Indeed, as long we restrict our attention to physically implementable Provers, no physical

[10] Those familiar with the relevant literature may appreciate that the transition from an interactive proof-system to a CS proof-system is analogous to the transition from perfect zero-knowledge proof-system to a computational zero-knowledge proof-system[18], which has proved to be a more flexible and powerful notion [20].

process in our Universe can perform $2^{1,000}$ steps of computation, at least as long the human race will exist. Thus, "practically speaking," all Provers are computationally bounded.

From a theoretical point of view, the new paradigm (though contrary to a long established tradition) is *natural*, in the sense that the new notion of a proof allows us to answer in an appealingly simple way, many of the oldest questions in complexity theory, and some new and fundamental ones as well. After all, "the right notion is the one that allows us to prove the right theorem in the right way!"

We also wish to clarify that the possibility of "proving a false statement" does not arise during the proving process (as in the case of a zero-knowledge argument[11]), Indeed, there is no interaction in a CS proof-system between Provers and Verifiers. Thus, if one wished to look for a false CS proof, he could do so "off-line", alone, and having all the elements that might be needed, or deemed useful, for such a search. And given that false CS proofs are strings, if such strings exist (and they do!), they can be surely found, *but only if more than a prescribed (and arbitrarily high) amount of computation is performed.*

If this may provide adequate protection against a cheating prover that tries to compute a good-looking CS proof, σ, of a false statement, what about such a string σ that "already exists out there"? For instance, what if, while walking on a beach we find a sand pattern that looks like the sequence of bits of such a σ?

Our answer is that though CS proofs are strings, and strings can be written in a variety of ways (on the sand, in a DNA sequence, or a plain piece of paper), we must consider the *process* that has generated them. For instance, in the above hypothetical example, the grains of sand have been arranged in such a σ-shape by natural elements such as wind, waves, sun, etc. We can thus view planet Earth as a computer, and Earth's age as its computing time, so that, in a final analysis, such a σ has been found in a few billion years: an unlikely event if we have chosen our parameters so that the age of the Universe is negligible with respect to the time necessary to find a good-looking CS proof of a false statement.

In sum,

CS proof-systems are deliberately inconsistent,
but indistinguishable, from a practical point of view, from consistent systems.

Three types of CS proof-systems. A CS proof-system may be probabilistic, because its Prover and Verifier share a *common source of randomness.* We actually distinguish three types of CS proof-systems, according to the type of

[11] Where the Prover may break an encoding scheme provided to him by the Verifier only at the beginning of an individual proving process.

randomness source Prover P and Verifier V may share (roughly said, a random oracle, a random string, and no randomness at all):

1. *CS proof-systems with a random oracle.* In this type of CS proofs, when having a given statement as an input, both P and V have access to the same random oracle. A bit more precisely (given that a random oracle can be viewed as an infinite string of random bits), P and V have oracle access to a random function $f : \Sigma^k \to \Sigma^k$, where k is a security parameter. Equivalently said, therefore, P and V can access in a single step any bit of a long random string, that is, one having length exponential in the security parameter k.
2. *CS proof-systems with a random string.* Here P and V have access to a short random string, that is, one having length polynomial in a security parameter k.
3. *Deterministic CS proof-systems.* In such systems P and V do not share any source of randomness.

The order of presentation of these three types of CS proof-systems is determined by the strength of the assumptions needed for their explicit construction. Indeed, we show how to construct explicitly CS proof-systems of the first type without relying on any additional assumption, CS proof-systems of the second type based on a new (but to us reasonable) complexity assumption, and CS proof-systems of the third type based a *very strong* complexity assumption.

All three types of CS proof-systems are *non-interactive*, in the sense that, to convince the verifier or the verity of a given statement, the Prover just sends him a single message (the CS proof), without any need to exchange messages back and forth (not even once).

As the reader may notice, however, it is possible to define *interactive CS proof-systems* which can actually be constructed more simply (in particular, their construction can be immediately obtained from that of subsection 3.1.5.) and under more traditional complexity assumptions (in particular, the existence of collision-free hash functions). We refrain from defining such proof-systems because we wish to "contain" this already long paper and because non-interactive CS proof-systems are more interesting and powerful. For instance, CS proofs with a random string and deterministic CS proofs yield some results on *checking* that are not possible to obtain with interactive CS proofs (or CS proofs with a random oracle for that matter).

3.1 CS Proofs With A Random Oracle

Significance of the model. As announced above, in a CS proof-system with a random oracle Prover and Verifier are given oracle access to a random function from $\{0,1\}^k$ to $\{0,1\}^k$, where k is an integer representing a security parameter (controlling the "trustworthiness" of the system).

Though somewhat "impractical", CS proofs of this type present a main advantage:

> *Even if $\mathcal{NP} = \mathcal{P}$, they guarantee that, given a sufficient amount of randomness in the proper form, fundamental intuitions like verification being poly-logarithmically easier than decision are indeed true.*

Let us explain. He who is concerned about truth, but not about time, does not need proofs and provers: he may be equally happy to run a decision algorithm for establishing whether a given statement holds. Proofs are in fact mechanisms that aim at quickly and "critically" transfer truths that have been hard to obtain. Thus,

> *Proofs cannot properly exist as a separate notion (i.e., separate from that of decision) unless they succeed in making verification of truth MUCH easier than deciding truth.*

The author's inclination to believe that $\mathcal{P} \neq \mathcal{NP}$ is only based on (1) his *a priori certainty* that proofs are a meaningful and separate notion, and (2) his *inclination* to believe that $\mathcal{NP}$ is a reasonable approximation to the notion of a proof. But if it turned out that $\mathcal{P} = \mathcal{NP}$, to us this would only mean that $\mathcal{NP}$ did not provide an adequate approximation. Indeed, fundamental intuitions such as proofs being an independently meaningful notion could not be shaken by a formal result such as $\mathcal{P} = \mathcal{NP}$. It is thus important to establish meaningful models in which we can *show* that *proofs do exist as an independent notion.*

CS proofs with a random oracle provide us with such a model. Better yet, they actually achieve, in their setting, *all the goals* set forward in subsection 2.2. It is thus legitimate to ask:

Where does their power come from?

That is, how can a random function help in making proofs an independently meaningful notion? The intuitive answer is that a random function

$$f : \Sigma^k \to \Sigma^k$$

essentially consists of an exponentially-long (in k) string. Thus, having f available as an *oracle* rather than as a *string* allows one to have *polynomial-time* (actually, constant-time) rather than *exponential-time* access to its bits. And it is precisely this exponential speed-up that will be converted, thanks to our specific construction, to the much more interesting exponential gap between decision and verification.

Preliminary Definitions

Oracles and oracle-calling algorithms. By an *oracle*, we mean a function $f : \Sigma^a \to \Sigma^b$, for some positive integers a and b. When the domain Σ^a and the codomain Σ^b are understood, by a *random oracle* we mean a function randomly selected among those mapping Σ^a into Σ^b.

Let A be an algorithm capable of making oracle calls. If A makes calls to a single oracle, we emphasize this fact by writing $A_{(\cdot)}$; similarly, if A makes calls to a pair of oracles, we may emphasize this fact by writing $A_{(\cdot,\cdot)}$; etc. If f is an oracle, we write A_f to denote the algorithm obtained by having algorithm $A_{(\cdot)}$ make its calls to oracle f; similarly, we write A_{f_1,f_2}, if (f_1, f_2) is the pair of oracles to which $A_{(\cdot,\cdot)}$ makes its calls; and so on.

For complexity purposes, in a computation of $A_{(\cdot,\ldots,\cdot)}$, the process of writing down a query σ to one of its oracles, f, and receiving $f(\sigma)$ in response is counted as a single step. (No result of this paper would change in an essential way if this call would "cost" $poly(a, b)$ steps whenever $f : \{0,1\}^a \to \{0,1\}^b$.)

An algorithm that, in any possible execution, makes exactly N calls to each of its oracles will be refereed to as a *N-call algorithm.*

A special language. Let $\mathcal{L}$ denote the language consisting of the triplets $\mathbf{q} = (\bar{M}, x, t)$, where $\bar{M}$ is the description (in some standard encoding) of a Turing machine M, x is a binary input to M such that $M(x) = YES$, and t is the *binary* representation of an upperbound to the number of steps M makes on input x.

Definition of a CS proof-system with a random oracle. Let (P, V) be a pair of Turing machines capable of oracle calls, the second of which runs in polynomial-time. At the start of an execution of (P, V), both machines have two common inputs, a binary *input* $\mathbf{q}$ (an alleged member of $\mathcal{L}$) and a unary *security parameter* k, and access to a common oracle f. The execution consists of running P on $\mathbf{q}$ and k with access to f, so as to produce a binary output $\mathcal{C}$, and then running V on $\mathbf{q}, k$, and $\mathcal{C}$ with access to f.

We say that (P, V) is a *CS proof-system with a random oracle* if there exist a sequence of 6 positive constants, $c_1, \ldots, c_6$ (refereed to as the *fundamental constants* of the system), such that the following two properties are satisfied:

1′. *Feasible Completeness.* $\forall$ integers $n > 1$, $\forall$ n-bit input $\mathbf{q} = (\bar{M}, x, t) \in \mathcal{L}$,
$\forall$ unary integers k, and $\forall$ oracle $f : \Sigma^{k^{c_1}} \to \Sigma^{k^{c_1}}$,
(*i*) P_f halts within $(nkt)^{c_2}$ computational steps, and outputs a binary string $\mathcal{C} = P_f(\mathbf{q}, k)$ whose length is $\leq (nk \log t)^{c_3}$, and
(*ii*) $V_f(\mathbf{q}, k, \mathcal{C}) = YES$.

2′. *Computational Soundness.* $\forall\, k > n^{c_4}$, $\forall \mathbf{q} \notin \mathcal{L}$, and $\forall$ deterministic (cheating) algorithm P' making $\leq 2^{c_5 k}$ oracle calls, for a random oracle $\mathcal{R} : \Sigma^{k^{c_1}} \to \Sigma^{k^{c_1}}$,

$$Prob_{\mathcal{R}}(V_{\mathcal{R}}(\mathbf{q}, k, P'_{\mathcal{R}}(\mathbf{q}, k)) = YES) < 2^{-c_6 k}.$$

If (P, V) is a CS proof-system with a random oracle, the process of running P on an input $(\bar{M}, x, t)$ with a security parameter k and a random oracle $\mathcal{R}$ will be refereed to as a *random-oracle CS proof*, and the output of P will be refereed to as a *random-oracle CS witness* or a *random-oracle CS certificate* (of security k) of $M(x) = YES$. If it is clear from the context that we are dealing with CS proofs with a random oracle, we may actually simplify our language by dropping the qualifications "with a random oracle" and "random-oracle."

Discussion

Deterministic Cheating. In the above definition of a CS proof-system with a random oracle, we have considered cheating provers P' to be deterministic. This choice simplifies the notational burden of later sections a bit, but does not cause any loss of generality. Indeed, since we are not concerned about the size of the description of P', nor about its running time (except for the number of oracle calls it may make), we could easily "wire-in" any "lucky" sequence of coin tosses if probabilism might help cheating.

Verified Cheating. Without loss of generality, the cheating provers of computational soundness always halt: either outputting nothing, or outputting *verified* certificates. That is, for all oracles f, if $\mathcal{P}'_f(q', k) = \mathcal{C}' \neq \varepsilon$, then, prior to outputting $\mathcal{C}'$, $\mathcal{P}'$ has appropriately called oracle f so as to verify that $\mathcal{V}_f(q', k, \mathcal{C}') = YES$.

Choosing the security parameter. Informally, the security parameter k controls the probability of something going wrong, and CS proofs become meaningful only when k is chosen large enough. Now, what should our considerations be when choosing k?

In the physical world as we know it, no one can perform 2^{300} computational steps, and any probability less than 2^{-300} is undistinguishable from 0. This suggests choosing our security parameter so that, by making less that 2^{300} oracle calls, one cannot cheat with probability greater than 2^{-300}.

The above choice of k may be particularly adequate when dealing with a single theorem statement, but if one considers all the false statements which are 300-bit long, then the probability that a cheating prover, in a relatively few oracle calls, may construct a good-looking CS certificate of correctness for at

least one of them may be constant, because there may be up to 2^{300} of such false statements. It may thus be preferable to ensure that, with probability $> 1-2^{-300}$, no false statement, no matter how long, has a feasibly-found CS certificate of correctness. This is easily accomplished by choosing k as a (fixed) moderately-growing function of n, the input length. Indeed, we recommend in our definition that k should grow polynomially in the input length: $k > n^{c_4}$.

Finally, letting k grow with n maximizes the meaningfulness of a CS proof-system (P, V) in a theoretical sense. Loosely speaking, when "CS-proving" that $(\bar{M}, x, t) \in \mathcal{L}$, we should choose k so that the effort necessary for cheating, even with a miniscule probability of success, is much larger than the decision time, t. In fact, a Verifier who is willing to check a CS witness for "$M(x) = YES$ within t steps," is implicitly acknowledging that in the world there is already someone —namely, the honest P himself— who can perform $poly(k, t)$ steps of computation.

In other words, after having argued that "feasible proving" and "efficient verifying" should not be expressed in absolute terms, but relative to the complexity of deciding, we now argue that the "hardness of cheating" should not be expressed in absolute terms either. Rather, we should require that cheating is "much harder than deciding." But, what values may be reasonably considered to be much greater than t (and actually achieved by an explicit construction)?

Our answer is $t^{O(\log t)}$. In fact, such difficulty of cheating is easily enforceable by choosing, as we do, k to grow polynomially in n. For instance, assume that one chooses $c_4 \geq 2$, thereby lowerbunding k by n^2. Such a lowerbound guarantees that the number of oracle calls necessary to a cheating prover to cheat with probability of success $2^{c_6 k}$ must exceed $2^{c_5 n^2}$. Now, the fact that $n > \log t$ (because the binary value t is part of the n-bit input $\mathbf{q}$) implies $t^{c_5 \log t}$.

In addition to the above reasons, choosing k to grow polynomially in n is also essential to our proof of Theorem 1, where we show that an explicit pair $(\mathcal{P}, \mathcal{V})$ is a CS proof-system with a random oracle.

Achieving our goals. Let us now see how our goals for the "right" notion of a proof have been "hard-wired" in the definition of CS proofs.

1. *Efficient Verifiability.* Our first goal required that, for every theorem, verifying should be polylogarithmically easier than deciding. This goal is approximated by a CS proof-system (P, V) in the following sense.
   Let L be any recursive language, x be any member of L, and D any deciding algorithm for L. Then, the theorem $x \in L$ can be verified by running D on input x and verifying that $D(x) = YES$. Assume now that the latter computation takes t steps. Then, by choosing a proper security parameter k and running P on inputs $(\bar{D}, x, t, k)$ and access to a random oracle $\mathcal{R}$, one obtains a CS proof, σ, of $x \in L$ such that (a) σ's length is bounded by a fixed

polynomial in $\bar{D}$, k, and $\log t$, and (b) σ is accepted by $V_{\mathcal{R}}$ running on input $(\bar{D}, x, k)$. Therefore, being V polynomial-time, there is a fixed polynomial Q such that V verifies σ within $Q(|\bar{M}|, |x|, \log t, k)$ steps.
In sum, no matter what algorithm one specifies for deciding whether $x \in L$, there is a corresponding CS proof of $x \in L$ which is verifiable in a time that is polylogarithmic in the number of steps taken by that algorithm on input x (and polynomial in all other parameters).

2. *Efficient Provability.* The second goal called for the complexity of proving being polynomially closer to that of deciding. This property is immediately guaranteed by the feasible-completeness of a CS proof-system. Feasible completeness in fact states that there exists a fixed polynomial $Q(\cdot, \cdot, \cdot)$ such that, if an algorithm D decides in t steps that a string x belongs to a recursive language L, then a CS prover can, on input $\mathbf{q} = (\bar{D}, x, t)$ and security parameter k, demonstrate that $x \in L$ within $Q(|\mathbf{q}|, k, t)$ steps.
3. *Universality.* Our third goal called for proof-systems capable of proving membership in all possible semi-recursive languages. In apparent contrast with this requirement, a CS proof-system is defined to prove membership only in the very special language $\mathcal{L}$. But $\mathcal{L}$ is designed so as to encode membership questions relative to any possible semi-recursive language. In fact, to each semi-recursive language L corresponds a (deciding) Turing machine M_L so that $x \in L$ if and only if M_L, on input x, outputs YES in some number of steps t. Thus, $x \in L$ if $(M_L, x, t) \in \mathcal{L}$, thus achieving the third goal.

The explicitness of t. There are two reasons for explicitly including in $\mathcal{L}$ the number of steps, t, within which M_L accepts x. First, our explicitly constructed CS proof-systems while certifying that $M_L(x) = YES$ also certify within how many steps M_L outputs YES. Second, as we have already argued in this subsection, the value t —or, at least, an upperbound for it— must be known in order to choose meaningfully the security parameter k.

An additional property. CS proofs also satisfy additional properties. In particular, like in a classical proof but unlike in an interactive proof, a Verifier who has inspected a CS proof can easily convince someone else of the verity of its corresponding statement. Indeed, being CS proofs strings checkable with a publically available oracle, all the Verifier has to do is to send someone else the same string he has successfully verified. (We do not wish, however, to make this property a requirement for any "sufficiently right" notion of a proof. Indeed, there are important contexts —in particular, cryptographic ones— in which it may be preferable or advantageous that a convinced verifier cannot convince anyone else. Indeed, this is one of the main advantages of a zero-knowledge proof.)

Back to zero-knowledge arguments. We can now better contrast CS proofs with Brassard, Chaum and Crépeau's zero-knowledge arguments. Taking aside zero-knowledge considerations, there are the following differences:

- Zero-knowledge arguments are interactive.
  (Incidentally, interactive versions of CS proofs can be also defined and constructed in an easy manner, though we omit doing so not to burden further this paper.)
- Zero-knowledge arguments may not enjoy Efficient Verifiability.
  (In fact, $\mathcal{NP}$ may not satisfy Ubiquitous Efficiency, an important sub-requirement of Efficient Verifiability.)
- Zero-knowledge arguments may not enjoy Efficient Provability.
  (Indeed, on input a member x of a $\mathcal{NP}$-language L, it is assumed that their Provers possess, "for free", a $\mathcal{NP}$-witness of $x \in L$. But, as we have already pointed out in Subsection 2.2.2, the time necessary for a Prover to find such a witness may vastly exceed that necessary to decide that $x \in L$.)
- Zero-knowledge arguments do not enjoy Universality.
  (Indeed, they are only designed to work for $\mathcal{NP}$-languages.)

(We wish to point out, however, that it might be possible to change the definition and the design of zero-knowledge arguments so as to enjoy most, if not all, of the properties they miss so far. Presumably, this could be accomplished by using Kilian's construction, the ideas of our paper, and the non-interactive zero-knowledge proofs of [7] and [6]. Investigating further this possibility is beyond the scope of the present paper.)

A paradox. CS proofs are paradoxical in that a computationally-bounded prover appears able to "prove more theorems" than an unbounded one. Indeed, if we choose k's value as a suitable function of the input length, then a properly-bounded CS prover can demonstrate membership in any $EXPTIME$ language to a Verifier whose running time is upperbounded by a fixed polynomial of the input length alone. By contrast, the unbounded prover of an interactive proof-system can only prove membership in $PSPACE$ languages to a polynomial-time verifier, and it is widely believed that $PSPACE$ is a proper subset of $EXPTIME$.

A few years ago, this seemed puzzling to us.

Deconstructing our construction

Warning. The reader hoping to see a catchy, simple example of a CS Proof system is about to be disappointed. Our construction of a CS proof-system is hardly practical, and we do not know of any simpler ones. In my "defense" I can

only say that not everything needs to be practical (I trust that most abstract mathematicians would agree), and that things important often become simpler with time.

Because of the construction's complexity, we shall use the present subsection to provide an informal introduction to it, and to highlight its ideas. This subsection, to be sure, is itself quite long (though hopefully less tedious than the construction itself and its proof). This is purposely so. We in fact wish to give the reader a chance to bypass subsections 3.1.5 and 3.1.7 altogether. We encourage him, however, to glance at 3.1.6 (because it is possible to believe that the construction works for the wrong reasons!).

Our construction is based on an earlier one of Kilian's [22], which is itself based on Merkle's trees and probabilistically-checkable proofs. Let us thus start by recalling these other notions.

Probabilistically-checkable and samplable proofs. Very recently, Babai Fortnow, Levin and Szegedy [3] and Feige, Goldwasser, Lovasz, Safra and Szegedy [13] have put forward, independently and with different aims,[12] some related and important ideas sharing a common technique: *proof-samplability*. A bit more precisely, they present an explicit algorithm transforming a $\mathcal{NP}$-witness, σ, into a new proof (i.e., string), τ, which is polynomially longer, but whose correctness can be detected by properly sampling it in a few locations rather than by reading it in its entirety.

Though the original constructions have been greatly improved (in particular, by Arora and Safra [2]), the simpler technique of [13] suffices for the goals of this paper. Actually, for simplicity purposes, we shall rely on the following "stripped-down" version of their technique, which we call *samplable proofs*.[13]

[12] The authors of [3] focus on proofs of membership in $\mathcal{NP}$-languages, and show that it is possible to construct Verifiers that work in time poly-logarithmic in the length of the input. (Since in such a short time the Verifier could not even read the whole input —and thus check that the proof he is going to sample actually relates to the "right" theorem,— these authors have devised a special error-correcting format for the input, and assume that it is presented in that format. An input that does not come in that format can be put into it in polynomial-time.)

The authors of [13] use proof-samplability to establish the difficulty of finding approximate solutions to important $\mathcal{NP}$-complete problems. (With this goal in mind, these other authors do not mind Verifiers working in time polynomial in the length of the input, and do not use or need that inputs appear in any special format.)

[13] Indeed, a probabilistically-checkable proof is a richer and more flexible notion, allowing one to specify a wide range for the number of queries the Verifier may make, and for the number of coins it may toss. Moreover, the careful reader may realize that, even restricting one's attention to $PCP(\log(n)^{O(1)}, log(n)^{O(1)})$, the PCP techniques yield a reacher structure than samplable proofs; for instance, samplable proofs do

Definition. A *sampling proof-system* consists of an elementary interactive proof-system, (SP, SV), where both the sampling prover SP and the sampling verifier SV run in probabilistic polynomial time. On input x (a n-bit string belonging to a given $\mathcal{NP}$-language L) and w_x (a $\mathcal{NP}$-witness that indeed $x \in L$), SP computes a (slightly longer) string w'_x, a samplable proof that $x \in L$. On input strings x (candidate member of L) and random access to σ (candidate samplable proof for $x \in L$), Verifier SV, after accessing only poly-log(n) bit-locations of σ, outputs either ACCEPT or REJECT with the following constraints. If $x \in L$ and $\sigma = SP(x, w_x)$ for some correct $\mathcal{NP}$-witness w_x, then SV always outputs ACCEPT. But if $x \notin L$, then, $\forall \sigma$, SV outputs REJECT with probability $\geq 1/2$.

The reader familiar with the recent advancements on probabilistically-checkable proofs —in particular the beautiful papers of Arora and Safra [2], Arora, Lund, Motwani, Sudan, and Szegedy [1], Sudan [39], Polishuck and Spielman [30],— will realize that, for the sake of simplicity, we are sacrificing quite a bit of efficiency.[14]

Merkle's trees. Merkle trees [27] are based on binary trees and collision-free hash functions.

- A *binary tree* is a tree in which every node has at most two children (in which case, each one of them is said to be the sibling of the other).
- A *collision-free hash function* is, informally speaking, a polynomial-time computable function H mapping binary strings of arbitrary length into reasonably short ones, so that it is computationally infeasible to find two different strings x and y for which $H(x) = H(y)$. Such pair of strings x and y is called a *collision (for H)*. (Popular candidate collision-free hash function is the standardized *secure hash function* [36] and Rivest's MD4 [35].)
- A *Merkle tree*, informally speaking, is a binary tree whose nodes store (i.e., are associated to) values, some of which computed by means of a one-way hash function H in a special manner. A leaf node can store any value, but each internal node should store a value that is the one-way hash of the concatenation of the values in its children. (I.e., if an internal node has a 0-child storing the value U and a 1-child storing a value V, then it stores

not guarantee that any particular bit of the samplable proof is accessible by the Verifier with positive probability —only the final (and proper) ACCEPT and REJECT probabilities are guaranteed.

[14] Indeed, Polishchuk and Spielman show that a $\mathcal{NP}$-witness with length n possesses a probabilistically-checkable version that is only $n^{1+\varepsilon}$-bit long for any constant $\varepsilon > 0$. Nonetheless, we shall ignore polynomial improvements in the running time of sampling provers and poly-log improvements in the running time of sampling verifiers. Unlike for approximation theory, in fact, improvements in the efficiency of proof-samplability will only affect the efficiency of CS proofs.

the value $H(UV)$. If a child of an internal node does not exist, we assume by convention that it stores a special value, denoted by EMPTY.)
If the one-way hash function produces k-bit outputs, then each internal node of a Merkle tree, including the root, stores a k-bit value. Except for the root value, each value stored in a node of a Merkle tree is said to be a 0-value, if it is stored in a node that is the 0-child of its parent, a 1-value otherwise.

The crucial property of a Merkle tree is that, unless one succeeds in finding a collision for H, *it is impossible to change any value in the tree without also changing the root value.* In particular, one cannot change the original values stored in the leaves without changing also the root value.

This property allows a party A to "commit" to n values, $V_1, \ldots, V_n$ (for simplicity assume $n = 2^a$ for some integer a), by means of a single k-bit value. That is, A stores value V_i in the ith leaf of a full binary tree of depth d, and uses a collision-free hash function H to build a Merkle tree, thereby obtaining a k-bit value RV stored in the root. This value RV "implicitly defines" what the n original values were. Assume that A gives RV, but not the original values, to another party, B, at some point in time. Then, whenever, at a later point in time, A wants to "prove" to B what the value of, say, V_i was, he may just reveal all n original values to B, so that B can recompute the Merkle tree and the verify that the newly computed root-value indeed equals RV.

More interestingly, A may "prove" what V_i was by revealing just $d+1$ (i.e., $\log n + 1$) values: V_i together with its *authentication path*, that is, the values stored in the siblings of the nodes along the path from leaf i (included) to the root (excluded), $Y_1, \ldots, Y_d$. B verifies the received alleged leaf-value V_i and the received alleged authentication path $Y_1, \ldots, Y_d$ as follows. Letting $i_1, \ldots, i_d$ be the binary expansion of i, B sets $V = X_1$ and computes the values $X_2, \ldots, X_d$ as follows: if $i_j = 0$, she sets $X_{j+1} = H(Y_j X_j)$; else, she sets $X_{j+1} = H(X_j Y_j)$. Finally, B checks whether the computed k-bit value X_d equals RV.

Kilian's construction. In [22], Kilian presents a zero-knowledge argument for $\mathcal{NP}$, (P, V), exhibiting a polylogarithmic amount of communication, where prover P uses Merkle trees in order to provide to V "virtual access" to a samplable proof.

In essence, disregarding zero-knowledge considerations, the polynomial-time P, as in any zero-knowledge argument, possesses a polynomially-long witness, w, proving that a given input x belongs to a given $\mathcal{NP}$-language L. P thus transforms w in a longer, but still polynomially-long in the length of x, samplable proof w'. In order to obtain a logarithmic amount of computation, P cannot send V witness w, nor can he send him the longer samplable proof w'. Rather, P uses a Merkle tree with a collision-free hash function H, producing k-bits outputs, as above to compute a k-bit string, RV, that commits him to w', and then sends RV

to the verifier V. (For instance, disregarding further efficiency considerations, if w' is n-bit long and , for simplicity, n is a power of 2, the ith bit of w' is made to be the original value V_i in the above described construction, the Merkle tree is a full binary tree of depth $\log n$, and RV is the k-bit value stored in its root.)

Verifier V runs as a subroutine a sampling verifier SV. When SV wishes to consult the jth bit of w', V asks P for it, and P responds by providing the original value b_j together with its authentication path. V then checks whether b_j's authentication path is correct relative to RV, and, if so, he is assured that b_j is the original value because he trusts that P, being polynomial-time, cannot find a collision for H. V then feeds b_j to SV. The computation proceeds this way until V finds that an authentication path is incorrect, in which case it halts and REJECTS, or until SV halts, in which case V REJECTS if SV does, and ACCEPTS otherwise. Because SV "virtually" accesses a logarithmic (in n) number of bits of w', and because each such a virtual access is answered by $k \log n$ bits of authentication path, the overall amount of communication is logarithmic in n and thus in the length of x.

Notice that the above construction only shows how a Verifier can be given virtual access to w'. Let us reiterate that, in order to obtain a communication efficient zero-knowledge argument, Kilian's construction is actually more complicated, but the additional zero-knowledge constraint is irrelevant for our goals.

Our first variant. Essentially this same construction (minus its zero-knowledge addition) was independently found by the author [29], but our version included a small, essentially conceptual, variant which is crucial to the present enterprise. Indeed, while Kilian aimed at providing more efficient zero-knowledge arguments for $\mathcal{NP}$, we aimed (as we do now) at providing a more general and powerful notion of a proof.

Our variant follows from the following two, simple but important, observations: (1) realizing that probabilistically-checkable proofs are more general than traditionally claimed, and (2) considering proofs of membership in a given language as individual problems. Let us explain.

1. The salient features of samplable proof-systems, although traditionally claimed for $\mathcal{NP}$ languages, actually hold in a more general context. Indeed, the results of [13] and [3] (sometimes referred to as "poly-PCP") are mostly seen as a "reduction of PCP to NP." To us this is a *big understatement* of their results. (For instance, it would not allow us to prove Theorem 1 at all.) In fact, a more careful reading of those papers yields the following

   Theorem 0 ([13], [3]): There exists a deterministic algorithm $SP(\cdot,\cdot)$, a probabilistic algorithm $SV(\cdot,\cdot)$ running in time polynomial in the first input and polylogarithmic in the length of the second input, and two polynomials

$\ell(\cdot,\cdot)$ and $\lambda(\cdot,\cdot)$ such that, for any polynomial-time relation R over $\Sigma^* \times \Sigma^*$ (i.e., $R(x,y) = 1$ may not imply any a priori bound for the length of y relative to that of x),

(a) for any strings x and y such that $R(x,y)$ holds, algorithm $SP(x,y)$ halts within $\ell(|x|,|y|)$ steps outputting a string y'; and algorithm SV, on inputs x and $|y|$ and random access to y', flips at most $\lambda(|x|,|y|)$ coins and accepts; and

(b) For any string x such that $\forall y\ R(x,y) = 0$, and for any string σ, the probability (computed over SV's coin tosses) that algorithm SV, on inputs x and $|\sigma|$ and random access to σ, flips at most $\lambda(|x|,|y|)$ coins and accepts is at most $1/2$.

2. If the first observation consists of realizing that samplable proofs apply "above $\mathcal{NP}$, the second observation consists of realizing that one can meaningfully use cryptography, in particular the infeasibility of finding collisions in a collision-free hash function, in order to keep honest a Prover capable of proving membership in languages "above $\mathcal{NP}$".

   Erroneously adopting a complexity-classes perspective, using cryptography "against" such a Prover was not considered meaningful. Though re-evoking in a convincing way past conceptual barriers is impossible, one could have raised the following "objection":

   > because a collision-free hash function H is polynomial-time computable, finding two strings x and y such that $H(x) = H(y)$ is "within $\mathcal{NP}$". Thus, how can a Prover with more than $\mathcal{NP}$-power be kept honest by the difficulty of finding collisions?[15]

   This objection vanishes if one views a Prover, working on an input string x and a given language L, not as an universal mechanism for proving membership in L or in a given complexity class, but as an *individual process/device*, endowed with an *individual amount of computational resources*, sufficient for the *individual input* at hand.

   Often, this amount of computational resources can be upperbounded in a natural and absolute manner, no matter what the complexity of the language L may be. For example, if x is n-bit long and L is $PSPACE$-complete, then there is a positive constant c such that 2^{n^c} steps certainly suffice to determine whether $x \in L$.

   Thus, no matter how high n may be, it is meaningful, for a Verifier that has interacted with a Prover so as to become convinced that a specific n-bit input x belongs to L, to believe that he has interacted with some agent capable 2^{n^c} steps of computation: indeed, $\Omega(2^{n^c})$ could be L's complexity, and the agent has convinced the Verifier to be capable of proving membership in L!

[15] Notice that cryptography was used against the Prover of a zero-knowledge argument, but that such a Prover was defined to be polynomial-time!

But it is also meaningful, for the *same* Verifier, after the *same* interaction, to believe the *same* agent incapable of —say— $2^{(n^c)^2}$ steps of computation. Thus, if, for some constant d, the complexity of collision-finding for a k-bit-output hash function H is $\Omega(2^{k^d})$, and if k is greater than $(n^c)^{2/d}$, it is totally meaningful to believe that the same agent cannot find a k-bit collision for H.

The above two remarks yield what may be called an *interactive* CS proof-system, $(\mathcal{P}, \mathcal{V})$ (a notion that, as already said, we refrain from formalizing for more quickly getting to the notion we really care about). Namely, by adopting a proper representation of "the input" and by using collision-free hash functions whose output is suitably larger than the length of the input at hand, we can use Kilian's construction to prove membership into any language, of any complexity class, while satisfying both Efficient Verifiability and Efficient Provability.

In essence, on input $\mathbf{q} = (\bar{M}, x, t) \in \mathcal{L}$, $(\mathcal{P}, \mathcal{V})$ works as follows. First, $\mathcal{P}$ runs machine M on input x so as to generate, in t steps, the history (i.e., sequence of instantaneous configurations), σ, of an accepting computation of M. Such an history σ is then thought of as a proof that $M(x) = YES$. This proof will not be insightful, and, because no restriction is put on M, can be arbitrarily long relative to x.

Consider now the following relation R: $R(x, \sigma) = 1$ if and only if σ is the (t-step) history of an accepting computation of M on input x. Then, R is $poly(|x|, |\sigma|)$ computable. Thus, due to Theorem 0, σ can be put in a probabilistically-checkable form τ by algorithm SP (the prover of a sampling proof-system (SP, SV)) within $poly(|x|, t)$ steps. The so obtained τ is efficiently checkable by algorithm SV with $poly(|x|, \log t)$ coin tosses.

But, while proving that "$M(x) = YES$ in t steps," τ is again too long. Thus, assuming that finding a collision in an k-bit-output collision-free hash function requires 2^{k^d} for some constant d, a collision-free hash function H whose output has size $(\log t)^{2/d}$ is chosen and, instead of handing τ to verifier $\mathcal{V}$, $\mathcal{P}$ gives him virtual access to τ using, as above, a Merkle tree with a hash function H. This increases moderately (i.e., by a factor polynomial in $\log t$) the amount of computation of $\mathcal{P}$, but makes cheating practically impossible for a malicious Prover (conceived as an individual process/device endowed with an individual amount of computational resources).

In sum, without any contradiction, we may keep honest a Prover working on a given individual problem by rescaling and choosing a much harder individual problem.

Our second variant. The CS proof-system with a random oracle presented later on differs from the above outlined $(\mathcal{P}, \mathcal{V})$ in another way.

On the minor side, rather than an k-bit-output collision-free hash function, the new construction uses a random oracle-function mapping $2k$-bit strings to k-bit strings. Indeed, finding collisions for such a "function" is provably hard (and can be quite precisely computed).

More importantly, we use our random oracle for obtaining a non-interactive CS proof system. Notice, in fact, that the proof system $(\mathcal{P}, \mathcal{V})$ informally described so far is interactive. Indeed, during the proving process, the sampling verifier SV keeps on computing which bits in the samplable proof it wishes to see; verifier $\mathcal{V}$ sends then these requests to prover $\mathcal{P}$; and prover $\mathcal{P}$ responds with the requested bits and their authentication paths.

Throughout the entire process, subroutine SV flips coins; more precisely, it uses the same random tape RT fed to it by verifier $\mathcal{V}$ at the start of SV's computation. Because RT is genuinely random, when the input statement "$M(x)$ =YES in t steps" is false and verifier $\mathcal{V}$ interacts with a malicious prover $\mathcal{P}'$ that does not succeed in finding a collision for the random oracle, SV and thus $\mathcal{V}$ accept with probability at most $1/2$.

This probability can be made at most 2^{-k} by repeating the whole process k times, each time using an independently-selected random tape RT.

Now it can be seen that if k (and thus the number of bit positions virtually accessed by the sampling verifier SV throughout its k runs) is sufficiently small with respect to the length of the samplable proof (which we would like to be our case in order to satisfy Efficient Verifiability), and if a malicious Prover knew in advance which k random tapes the Verifier is going to feed SV, then he could cheat with a much higher probability, even with probability 1. However, roughly said, if the k tapes are selected *after* the malicious Prover provides the root value RV, then, roughly said, unless he succeeds in finding at least one collision for the random oracle, his probability of cheating is still 2^{-k}. This continues to be true even if he receives the so selected k tapes in their entirety, rather than just the bit-requests that the sampling verifier computes from them.

This suggests replacing interaction in the above proof-system as follows. On input the statement "$M(x)$ =YES in t steps", Prover $\mathcal{P}$, as before, (a) computes a classical proof of it by running M on input x and listing the sequence of instantaneous configurations of the computation, (b) puts this classical proof in samplable form, and (c) stores this samplable proof in the leaves of a suitable binary tree and constructs a corresponding Merkle tree, using the random oracle in lieu of a collision-free hash function, so as to compute a root value RV. At this point, $\mathcal{P}$ uses the random oracle (actually, an independent random oracle "extracted" from the given one) on input RV so as to compute k suitably-long random tapes, $RT_1, \ldots, RT_k$. He then runs ("in his head") verifier $\mathcal{V}$ and its subroutine SV as in the whole process described above, for k times, using RT_i as SV's random tape in the ith iteration. Therefore, he computes (in his head)

all the bit-locations of the samplable proof that SV requests to access, and outputs, as a CS proof with a random oracle for "$M(x)$ =YES in t steps", the value RV and the requested bits, each with its own authentication path relative to RV. Such proof can be verified, in the obvious way, by using verifier $\mathcal{V}$ (with subroutine SV) and the same random oracle.

Consider now a malicious Prover trying to "CS-prove with a random oracle" a false statement. Of course, he can choose a root value RV' of his liking and consult the oracle so as to see whether he can produce a good-looking CS proof relative to RV'. However, roughly said, because for each RV' (as long as he does not succeed in finding a collision for the random oracle), his chance of finding a good-looking proof is at most 2^{-k}, we expect that he tries 2^k times before he succeeds. Thus, if k is large enough, and the running time of the malicious prover is properly and meaningfully upperbounded, his chance of finding a CS proof of a false statement is negligible.

Our variant is reminiscent of a step used by Fiat and Shamir [14]. Indeed, they construct their digital signature scheme by starting with an interactive two-party protocol, in which the first party sends a first message to the second party and the second party responds with a random string, and then replacing the random message of the second party by evaluating a one-way function on the first party's message. (Prior to that, Manuel Blum actually suggested to the author using a random oracle in order to replace interaction in a zero-knowledge proof-system, though this suggestion never appeared in print.[16])

A conceptual contribution. We believe the main contribution of this paper to be a *conceptual* one. Namely (in addition to new but natural requirements such as feasible completeness) our paper provides a complexity-based notion of a proof which is *universal* (i.e., applicable to all recursive languages) and deliberately *inconsistent*, but in a *controllable* (and meaningfully advantageous) way.

It has been surprising to many (and to us in particular) why such a notion had not been put forward much longer ago. Indeed, the main algorithmic ideas of our construction (1) samplable proofs [13] [3], (2) Merkle's trees [27] and (3) the Fiat-Shamir signature scheme [14], have been individually known for quite a while. In addition, (1) and (2) had already been used *together* in [22]. (To these ideas we just added the two simple remarks mentioned above[17] and the use of

[16] His suggestion was in fact the starting point of the non-interactive zero-knowledge proofs of [7] and [6]. The suggestion did not enter the final version of these papers because, for their specific task of zero-knowledge proofs of membership in $\mathcal{NP}$ languages, we succeeded in replacing random oracles by short random strings, under quite-standard complexity assumptions.

[17] I.e., realizing that probabilistically-checkable proofs are more general than tradition-

random oracles.) Yet, their techniques have been successfully applied to many contexts (approximability of $\mathcal{NP}$-complete problems, zero-knowledge arguments for $\mathcal{NP}$, etc.), but not to distilling a new notion of a proof.

Perhaps, we have been focusing on consistency and efficient verifiability for so long that we became unable (the author first) to see any other issues and desiderata in the notion of a proof.

Having presented all the ideas entering in our construction at an intuitive level, let us now proceed more formally.

Our construction

The readers finding this (semi-) algorithmic rendition of the above ideas somewhat superfluous has my full solidarity: I hate "programming" in all its approximations. However, failure to provide an explicit algorithmic form would have upset plenty of other readers. (One simply cannot please everyone!)

From one oracle to two oracles. According to our definition, in a CS proof-system with a random oracle Prover and Verifier have oracle-access to a single function f, where feasible completeness holds for any f, and computational soundness for a random f.

It will be easier, however, to exemplify a CS proof-system with a random oracle $(\mathcal{P}, \mathcal{V})$, where $\mathcal{P}$ and $\mathcal{V}$ have oracle-access to two distinct functions: f_1 and f_2, where feasible completeness holds for any possible choice of f_1 and f_2, while computational soundness when f_1 and f_2 are random and independent.

Oracle-access to these two functions can be simulated by accessing a single, properly selected, function f: to ensure that f_1 and f_2 are randomly and independently selected when f is random, we arrange that whenever $(i, x) \neq (j, y)$ no query made to f in order to compute $f_i(x)$ coincides with a query made to f in order to compute $f_j(x)$. For instance, if, for $i = 1, 2$, $f_i : \{0,1\}^{a_i} \to \{0,1\}^{b_i}$ (for some positive integer values a_i and b_i, $i = 1, 2$), letting f map $\{0,1\}^{a_1+a_2}$ into $\{0,1\}^{b_1+b_2}$, allows us to achieve our goal quite straightforwardly.

Length bounds. From Theorem 0 we distill the following

Definition: Let SP, SV, ℓ and λ be as in Theorem 0. Then, we shall refer to (SP, SV) as a *sampling proof system*, and to ℓ and λ as its *length bounds* (respectively, for the samplable proof produced by SP and the random tape used by SV).

---

ally claimed, and considering proofs of membership in a given language as individual problems.

Note that in our setting we deal with membership in the special language $\mathcal{L}$, consisting of triplets $\mathbf{q} = (\bar{M}, x, t)$ such that $M(x)$ accepts within t steps, and we are interested in putting in samplable form a proof y consisting of the history of the computation of $M(x)$, so that y's length is bounded by a fixed polynomial in t. In fact, y is describable by a Turing-machine tableau of side t. Thus, if we denote by n the binary length of $\mathbf{q}$, then $|x| + |\bar{M}| \leq n$, $t < 2^n$, and y's length is bounded by a fixed polynomial in n. Accordingly, in our setting, we can view length bounds ℓ and λ as polynomials in the single variable n.

Notation. We denote the empty word by ε, the set $\{0,1\}$ by Σ, the set of all natural numbers by $\mathcal{N}$, the set of all positive integers by Z^+, the concatenation of two strings x and y by $x|y$ (or more simply by xy), and the complement of a bit b by $\bar{b}$.

We denote the length of a binary string α by $|\alpha|$. If α is a n-bit string whose ith bit is α_i, then for any integer j between 1 and n we let $\alpha[1 \cdot \cdot j]$ denote the j-bit string $\alpha_1 \cdots \alpha_{j-1}\alpha_j$. We let $\alpha[1 \cdot \cdot \bar{j}]$ the string obtained by complementing the last bit of $\alpha[1 \cdot \cdot j]$, that is, $\alpha[1 \cdot \cdot \bar{j}] = \alpha_1 \cdots \alpha_{j-1}\bar{\alpha}_j$. Finally, if α is any binary string and i a natural number, we let $\alpha[i+1 \cdot \cdot]$ denote α after deleting its first i bits (i.e., if α is a n-bit string, $\alpha[i+1 \cdot \cdot] = \alpha[i+1 \cdot \cdot n]$.)

If N is a power of two, we let $\mathcal{T}_N$ denote the complete binary tree with N leaves, labeled so that v_ε is the root, v_0 and v_1 are, respectively, the left and right children of v_ε, and, $\forall \alpha \in \{0,1\}^i$ and $\forall i < \log N$, $v_{\alpha 0}$ and $v_{\alpha 1}$ are, respectively, the left and right children of node v_α. Consequently, $v_{\alpha[1 \cdot \cdot j]}$ and $v_{\alpha[1 \cdot \cdot \bar{j}]}$ are siblings whenever $0 < |\alpha| \leq \log N$ and $0 < j \leq |\alpha|$. The leaves of $\mathcal{T}_N$ are thought to be ordered "from left to right." Within the context of a tree $\mathcal{T}_N$, we denote by $[j]$ the $\log N$-bit binary representation of integer j (thus, $[j]$ consists of the binary representation of j with the right number of leading zeros). Accordingly, the jth leaf of $\mathcal{T}_N$ is node $v_{[j]}$.

In the following protocol we associate values to the nodes of $\mathcal{T}_N$ so as to obtain a Merkle tree (but constructed with a random-oracle function rather than a collision-free hash function). In so doing, we shall consistently denote the value of v_α as R_α. Thinking of each node of $\mathcal{T}_N$ as having its own memory location, we may also say that "R_α has been stored in node v_α."

Algorithms $\mathcal{P}$ and $\mathcal{V}$.

Common inputs: $\mathbf{q} = (\bar{M}, x, t)$, and k

respectively, a n-bit triplet in $\mathcal{L}$ and a unary security parameter.

Common subroutines: SP and SV,

respectively prover and verifier of a given samplable proof-system (SP, SV) with length bounds ℓ and λ.

Common oracles: $f_1 : \Sigma^{2k} \to \Sigma^k$ and $f_2 : \Sigma^{k+n} \to \Sigma^{k\lambda(n)}$.

(*Comment:* When randomly selected, oracle f_1 is used as a collision-free hash function in Merkle tree $\mathcal{T}_n$, and thus to enable $\mathcal{P}$ to commit to a samplable proof of $\mathbf{q} \in \mathcal{L}$. Oracle f_2 is used to generate k random tapes for SV.)

$\mathcal{P}$'s output: $\mathcal{C}$, a CS certificate that $\mathbf{q} \in \mathcal{L}$.

$\mathcal{V}$'s additional input: $\mathcal{C}$.

Algorithm $\mathcal{P}$

$P1$. (Commit to a samplable proof of $\mathbf{q} \in \mathcal{L}$.)

$P1.1$ (Find a proof σ of $x \in \mathcal{L}$.)
Run machine M on input x so as to generate the history, σ, of the $\leq t$-step accepting computation of $\mathcal{M}$.
(*Comment:* σ can be considered a proof that $x \in \mathcal{L}$)

$P1.2$ (Put σ in a samplable form τ.)
Run algorithm SP on input $\mathbf{q}$ and σ so as to obtain a samplable proof, τ.
(*Comment:* Because $|\mathbf{q}| = |(\bar{M}, x, t)| = n$, $|\tau| \leq \ell(n)$.)

$P1.3$ (Commit to τ by means of a k-bit value R_ε.)
Assume, for simplicity only, that $|\tau|/k = N$, where N, an integral power of 2. Then, sub-divide τ into the concatenation of N k-bit strings, $\tau = \tau_1 \cdots \tau_N$, and compute a value R_ε by associating to the vertices of tree $\mathcal{T}_N$ the following values. For $0 \leq j < N$, assign to the jth leaf, $v_{[j]}$, the k-bit value

$$R_{[j]} = \tau_j. \tag{1}$$

Then, in a bottom-up fashion, assign to each interior node v_α of $\mathcal{T}_N$ the k-bit value

$$R_\alpha = f_1(R_{\alpha 0} | R_{\alpha 1}). \tag{2}$$

(*Comment:* R_ε is the k-bit value assigned to the root of $\mathcal{T}_N$.)

$P2$. (Build a CS certificate $\mathcal{C}$ of $\mathbf{q} \in \mathcal{L}$.)

$P2.1$ (Start certificate.)
$\mathcal{C} \leftarrow R_\varepsilon$.

$P2.2$ (Choose k random tapes for SV.)
Call $TAPE$ the random $k\lambda(n)$-bit string obtained by computing $f_2(\mathbf{q}\,|R_\epsilon)$; divide this string in k disjoint segments, each $\lambda(n)$-bit long, and call $TAPE_j$ the jth such segment.

$P2.3$ (Run SV k times with virtual access to τ.)
For $j = 1, \ldots, k$, run SV with $TAPE_j$ as the random tape, inputs $\mathbf{q}$ and $|\tau|$, and virtual access to τ. Whenever SV wishes to access bit-location i of τ, perform the following instructions:

$P2.3.1$ (Find the index, I, of the substring of τ containing b_i.)
Let I be the smallest positive integer p such that $i \leq pk$;

$P2.3.2$ (Add leaf I to the CS certificate.)
$\mathcal{C} \leftarrow \mathcal{C}|R_{[I]}$; and

$P2.3.3$ (Add to the certificate the siblings of the path between leaf I and the root of $\mathcal{T}_N$.)
Set $\alpha = [I]$; set $v_j = R_{\alpha[1..\bar{j}]}$ for $j = 1, \ldots, \log N$;
set $SIBLINGPATH_I = (v_1, \ldots, v_{\log N})$; $\mathcal{C} \leftarrow \mathcal{C}|SIBLINGPATH_I$.
(*Example:*
if $I = 3$ and $N = 8$, then $SIBLINGPATH_I = (R_{010}, R_{00}, R_1)$.)

$P3$. (Output the certificate.)
Output $\mathcal{C}$, a CS certificate of $\mathbf{q} \in \mathcal{L}$ *relative to root-value* R_ϵ.

Algorithm $\mathcal{V}$

$V1$. (Read and delete the root of $\mathcal{T}_N$ from the certificate, and compute k random tapes for SV.)
Set $ALLEGED - ROOT = \mathcal{C}[1 \cdot\cdot k]$ and reset $\mathcal{C} \leftarrow \mathcal{C}[k+1 \cdot\cdot]$. Then compute the $k\lambda(n)$-bit string $TAPE = f_2(\mathbf{q}|R_\epsilon)$; divide $TAPE$ into k non-overlapping segments, each $\lambda(n)$-bit long, and call $TAPE_j$ the jth such segment.

$V2$. (Run SV k times with virtual access to samplable proof τ.)
For $j = 1, \ldots, k$ run SV with $TAPE_j$ as the random tape, input $\mathbf{q}$, and virtual access to samplable proof τ. Whenever SV wishes to access bit-location i of τ, do

$V2.1$ (Find the index, I, of the segment of τ containing b_i, and read the value of leaf I from the certificate.)
Set $I = \lceil i/k \rceil$, $\alpha = [I]$, and $R_\alpha = \mathcal{C}[1 \cdot\cdot k]$.

$V2.2$ (Delete the value of leaf I from the certificate.)
$\mathcal{C} \leftarrow \mathcal{C}[k+1 \cdot\cdot]$.

$V2.3$ (Check whether the certificate contains the siblings of the path between leaf I and the root of $\mathcal{T}_N$, and remove them.)

For $m = 1$ to $\log N$, set $R_{\alpha[1\cdots\bar{m}]} = \mathcal{C}[1 \cdots k]$ and reset $\mathcal{C} \leftarrow \mathcal{C}[k+1 \cdots]$. Then, for $m = \log N, \ldots, 1$, compute $R_{\alpha[1\cdots m-1]}$ as follows:

$$R_{\alpha[1\cdots m-1]} = \begin{cases} f_1(R_{\alpha[1\cdots j]}|R_{\alpha[1\cdots\bar{j}]}) & \text{if } \alpha_m = 1 \\ f_1(R_{\alpha[1\cdots\bar{m}]}|R_{\alpha[1\cdots m]}) & \text{if } \alpha_m = 0 \end{cases}$$

and check whether the computed value R_ε equals the read value $ALLEGED-ROOT$.
(*Example:* If $N = 8$ and $I = 3$, then the verifier computes

$$\begin{aligned} R_{011} &= \tau_3, \\ R_{01} &= f_1(R_{010}|R_{011}), \\ R_0 &= f_1(R_{00}|R_{01}), \text{ and} \\ R_\varepsilon &= f_1(R_0|R_1). \end{aligned}$$

In this example, the values of τ_3, R_{010}, R_{00}, and R_1 are part of the certificate provided by the prover, and the values of R_{011}, R_{01}, R_0, and R_ε are computed by the verifier.

*V*3. (Accept if and only if the sibling path have always been correct and if SV has always accepted.)
If each check performed in Verification Step 2.3 has been passed, and if SV has output YES in each of its k runs, then output YES. Else, output NO.

A flawed proof that the construction works. We prove that $(\mathcal{P}, \mathcal{V})$ is a CS proof-system with random oracle in the next subsection. The proof of feasible completeness is quite straightforward, but our proof of computational soundness is not particularly compact nor intuitive, due to our insistence in proving even intuitive properties. Though we might have exaggerated in details, our desire to be meticulous derives from the fact that more intuitive and simple "proofs" of computational correctness are actually flawed in *fundamental* ways. Indeed, their being intuitive derives from the traditional use of Merkle trees, which is quite different from ours.[18] Based on previous use of Merkle trees, we may be tempted of quickly "deriving" computational soundness from the following

Irrelevant (and informal) Fact: the value R_ε, computed in Proving Step 1.3 and included in the certificate, in practice commits the Prover to at most one possible string τ because, without making $2^{O(k)}$ oracle calls, the chances of finding two different strings τ and τ' that "Merkle-hash" to R_ε is negligible.

[18] Traditionally, one uses a Merkle tree to prevent someone else from cheating. (E.g., in the original application [27], a signer uses a Merkle tree to authenticate n values of his choice, so as to make it infeasible for an impostor to authenticate different values.) In our case, instead, the Prover uses a Merkle tree to prevent ***himself*** from cheating, which complicates the proof a great deal.

The above statement is certainly true, and, once properly formalized, not hard to prove. It is not, however, directly relevant to the scenario at hand. It applies, instead, to settings where one party wishes to secretly commit to a string τ that will be later revealed in its entirety.[19]

Unfortunately, the setting of Theorem 1 is quite different: prover $\mathcal{P}$ never de-commits R_ϵ by revealing the entire samplable proof τ. Indeed, because τ may be too long, and because feasible completeness must be satisfied, $\mathcal{P}$ only reveals relatively few bits of τ: those requested by the samplable verifier in its k simulated and virtual runs. It is thus irrelevant that a cheating prover cannot feasibly find two strings τ and τ' that tree-hash to the same k-bit value.

Equally irrelevant is the fact that successful cheating is highly improbable if a malicious prover *first* selects a suitable value R_ϵ, and *then* tries to answer satisfactorily all bit-requests that are induced by R_ϵ (via the k random tapes it specifies).

Rather, we must prove that it is infeasible for a cheating prover, on input $q' \notin \mathcal{L}$, to compute a k-bit value R'_ϵ (possibly obtained without Merkle-hashing any string) *together with*, somehow, (a) bit-values (corresponding to requests made by SV based on the tapes specified by R_ϵ) that lead SV to accept at every run, and (b) authentication paths corresponding to said bit-values that correctly merge with R_ϵ. This is what we are going to do next.

The construction works

Theorem 1: $(\mathcal{P}, \mathcal{V})$ is a CS proof-system with a random oracle.

Proof of Feasible Completeness.
It is immediately seen that subproperty (ii) (completeness) holds. That is, for all $\mathbf{q} = (\bar{M}, x, t) \in \mathcal{L}$, for all security parameter k, and for all oracles f_1 and f_2, the certificate output by $\mathcal{P}$ convinces $\mathcal{V}$. Subproperty (i), that is the fact that $\mathcal{P}$ performs only polynomially many (in n, k, and t) for producing a certificate, follows as easily. Indeed, $\mathcal{P}$ invests t steps of computation for running M on input x, a number of steps polynomial in $\mathbf{q}$'s length and t for obtaining the samplable proof τ, less than $t \log t$ queries to the random oracle for generating the Merkle tree, and much fewer steps for running $\mathcal{V}$ "in his head" and answering its queries.

(Also recall that each call to our f_1 and f_2 can be simulated very efficiently even if one were given access only to a single oracle $f : \Sigma^{3k+n} \to \Sigma^{k(\lambda(n)+1)}$). □

[19] For instance, a party A secretly chooses a string τ, Merkle-hashes it into a k-bit string R_ϵ, gives R_ϵ to party B, and permits him to look at a few bits of τ proving their values via authentication paths relative to R_ϵ. After that, B is asked to bet on whether or not A's secret string possesses a given property. After B's bet, A reveals τ, so that B can verify whether it Merkle-hashes to R_ϵ.

Proof of Computational Soundness.

Initial notation. To simplify the proof of computational soundness, we assume, without loss of generality, any of our oracle-calling algorithms does not query any of its oracles twice about the same string σ.

In our proof we shall also consider oracle-calling algorithms that are probabilistic. The sequence of coin tosses of a probabilistic oracle-calling algorithm A shall be always denoted by $C(A)$. Notice that the notion of a N-call algorithm include probabilistic ones.[20]

Probabilities of events occuring in the executions of an oracle-calling algorithm A are computed over the random choice of a specified subset of its oracles and, if A is probabilistic, over the choice of $C(A)$. To indicate the probability of an event E taken over "the possible choices of $X, Y, \ldots$" we write

$$Prob_{X,Y,\ldots}(E).$$

Let $\mathcal{A}_{(\cdot)}$ be a N-call algorithm querying its oracle about strings in a finite set D, let R be another finite set, and let $f : D \to R$ be a function. Then, an execution of $\mathcal{A}_f$ identifies an element of the Cartesian product R^N; that is, the sequence $T = r_1, \ldots, r_N$, where r_i is f's answer to $\mathcal{A}$'s ith query.

Viceversa, because $\mathcal{A}$ does not, within the same execution, query twice its oracle about the same string, given an R-valued sequence $T = r_1, \ldots, r_N$, we can envisage executing $\mathcal{A}$ so that the ith query to the oracle is answered by r_i. The ensemble of executions so generated will be denoted by $\mathcal{A}_T$.

It should be noticed that the set of ensembles $\mathcal{A}_T$ and the set of ensembles $\mathcal{A}_f$ may be different. (If $\mathcal{A}$ is deterministic, then $\mathcal{A}_T$ consists of a single execution, in which case it is immediate to see that there exist functions f such that $\mathcal{A}_T = \mathcal{A}_f$. However, if $\mathcal{A}$ is probabilistic, it may happen that in one execution of $\mathcal{A}_T$, the ith query, q, is answered by a value v, while in a different execution the same string q, asked as the jth query, where $j \neq i$, is answered by a value other than v. In this case for no function f is $\mathcal{A}_f$ equal to $\mathcal{A}_T$.) Nonetheless, it is simple to notice that, whether or not $\mathcal{A}$ is probabilistic, for any event E, the following identity holds:

$$(*) : Prob_{C(\mathcal{A}),f:D\to R}\ (E \text{ occurs in } \mathcal{A}_f) = Prob_{C(\mathcal{A}),T\in R^N}(\ E \text{ occurs in } \mathcal{A}_T).$$

The above concept, notation, and identity naturally extend to more "complex" cases.[21]

---

[20] Indeed, a probabilistic oracle-calling algorithm $\mathcal{A}_{(\cdot,\ldots,\cdot)}(\cdot,\ldots,\cdot)$ is N-call if in each of its executions (i.e., for any possible choice of its inputs, oracles, and coin tosses) A makes N calls to each of its oracles.

[21] For instance, Let T be a N-long sequence whose values belong to a finite set R, and let $\mathcal{A}_{(\cdot,\cdot)}(\cdot)$ be a N-call algorithm that always queries its first oracle about a string

Let m and M be positive integers such that $m \leq M$, let σ be a string, and let $T = \{\sigma_i : i \in [1, M] - \{m\}\}$ be a sequence of strings. Then, by the notation $T_m = \sigma$ we shall denote the M-long sequence $\sigma_1, \ldots, \sigma_{m-1}, \sigma, \sigma_{m+1}, \ldots, \sigma_M$.

Let $T = (\sigma_1, \ldots, \sigma_m)$ be a sequence of strings (in Σ^r); then we say that the (Σ^r-valued sequence) $\bar{T} = (\bar{\sigma}_1, \ldots, \bar{\sigma}_M)$ is an *extension of T (over Σ^r)* if $m \leq M$ and $\bar{\sigma}_i = \sigma_i$ for $i = 1, \ldots, m$.

Let m be a non-negative integer and S a sequence of pairs of binary strings, $S = (x_1, y_1), \ldots, (x_m, y_m)$. We say that S is *extendible from Σ^{ℓ_1} to Σ^{ℓ_2}* if ℓ_1 is the length of all the x_i's, ℓ_2 is the length of all the y_i's, and, whenever $i \neq j$, we have $x_i \neq x_j$ and $y_i \neq y_j$. (The reason for insisting that $y_i \neq y_j$ whenever $x_i \neq x_j$ will become clear in Corollary 1.) If $S = (x_1, y_1), \ldots, (x_m, y_m)$ is extendible from Σ^{ℓ_1} to Σ^{ℓ_2} and the function $f : \{0,1\}^{\ell_1} \to \{0,1\}^{\ell_2}$ is such that $f(x_i) = y_i$ for $i = 1, \ldots, m$, then we say that f is an *extension of S;* in symbols, $f \in ext(S)$.

Definition: Let $\mathcal{A}$ be an algorithm calling one or more oracles. Then, in an execution of $\mathcal{A}$ where the function f was one of its oracles, we say that $\mathcal{A}$ *finds an f-collision* if $\mathcal{A}$ queries f about two different strings x and y such that $f(x) = f(y)$.

To proceed, we need to state (without proof) a "converse" of the well-known birthday paradox.

Lemma 1: $\forall$ positive integers k, $\forall$ possible inputs $z_1, z_2, \ldots$, $\forall$ $2^{k/4}$-call algorithms $\mathcal{A}_{(\cdot,\ldots,\cdot)}(\cdot, \ldots, \cdot)$, and $\forall$ possible oracles $f_1, f_2, \ldots, f'_1, f'_2, \ldots$,

$$Prob_{C(\mathcal{A}), f:\Sigma^{2k}\to\Sigma^k}(\mathcal{A}^{f_1,\ldots,f,f'_1,\ldots}(z_1, z_2, \ldots) \textit{ finds an } f\textit{-collision}) < 2^{-k/2}.$$

Corollary 1: $\forall$ positive integers k, $\forall$ positive integers $m \leq 2^{k/4}$, $\forall$ sequences $S = (x_1, y_1), \ldots, (x_m, y_m)$ extendible from Σ^{2k} to Σ^k, $\forall$ $2^{k/4}$-call algorithms $\mathcal{A}_{(\cdot,\ldots,\cdot)}(\cdot, \ldots, \cdot)$, $\forall$ possible inputs $z_1, z_2, \ldots$, and $\forall$ possible oracles $f_1, f_2, \ldots, f'_1, f'_2, \ldots$,

$$Prob_{f\in ext(S)}(\mathcal{A}^{f_1,\ldots,f,f'_1,\ldots}(z_1, z_2, \ldots) \textit{ finds an } f\textit{-collision}) < 2^{-k/2}.$$

Proof of Corollary 1: Algorithm $\mathcal{A}$ has no greater a chance of finding a f-collision in the present setting than in that of Lemma 1. In fact, if it makes two queries α and β both in the set $X = \{x_1, \ldots, x_m\}$, then we are guaranteed that $f(\alpha) \neq$

---

in a finite set D. Then, $\mathcal{A}_{T,f_2}(x)$ denotes an execution of $\mathcal{A}$, where x is the input, f_2 the second oracle, and where the ith query of $\mathcal{A}$ to the first oracle is answered by the ith string of a sequence T. It is immediately seen that, for all possible choices of f_2, x, D, and R,

$$Prob_{f_1:D\to R, f_2}(E \textit{ occurs in } \mathcal{A}_{f_1,f_2}(x)) = Prob_{T\in R^N, f_2}(E \textit{ occurs in } \mathcal{A}_{T,f_2}(x)).$$

$f(\beta)$; on the other side, if α and β are not both in X, then the probability that $f(\alpha) = f(\beta)$ still is 2^{-k}. □

Let us now prove that $(\mathcal{P}, \mathcal{V})$ satisfies computational soundness if we choose the 4th, 5th, and 6th fundamental constants as follows:

- c_4 = the smallest integer i such that $n^i > 8\ell(n) + 32$,
- $c_5 = 1/16$, and
- $c_6 = 1/16$.

(To facilitate the comprehension of what follows, we prefer to make use of the different "labels" c_5 and c_6 rather than their common numerical value 1/16.)

The proof is by contradiction. Assume that computational soundness does not hold for our choice of 4th, 5th, and 6th fundamental constants, then the following proposition holds:

℘1 : *There exist an integer $n' > 1$, a n'-bit string $q' \notin \mathcal{L}$, an integer $k' > (n')^{c_4}$, and a deterministic, $2^{c_5 k'}$-call, cheating prover P' such that*

$$Prob_{f_1:\Sigma^{2k'}\to\Sigma^{k'},f_2:\Sigma^{k'}\to\Sigma^{k'L(n')}}(\mathcal{P}'_{f_1,f_2}(q',k') = \mathcal{C}' \wedge \mathcal{V}_{f_1,f_2}(q',k',\mathcal{C}') = YES) \geq 2^{-c_6 k'}$$

Additional notation: A function mapping $\Sigma^{2k'}$ into $\Sigma^{k'}$ will always be indicated by f_1 (with possible superscripts). Similarly, f_2 (with possible superscripts) will always denote a function mapping $\Sigma^{k'}$ into $\Sigma^{k'L(n')}$.

We shall solely focus on the non-empty outputs of a cheating prover $\mathcal{P}'$; thus, when writing $\mathcal{P}'_{f_1,f_2}(q',k') = \mathcal{C}'$, we mean that $\mathcal{P}'$ has output a non-empty string, and that this string is $\mathcal{C}'$. If this occurs, given that a cheating prover always verifies its own non-empty outputs, we must have $\mathcal{V}_{f_1,f_2}(q',k',\mathcal{C}') = YES$. Assume now that, during the verification of $\mathcal{C}'$, samplable verifier V virtually asks about a bit-location i of the virtual samplable proof; then, we write $\mathcal{C}' \ni i$.

If $\mathcal{P}'_{f_1,f_2}(q',k') = \mathcal{C}'$, to emphasize that $q' \notin \mathcal{L}$, we refer to $\mathcal{C}'$ as a *pseudo-certificate* (of $q' \in \mathcal{L}$). We further say that $\mathcal{C}'$ is *relative to* $\sigma \in \Sigma^{k'L(n')}$ if $\sigma = f_2(q'|R'_\epsilon)$, where R'_ϵ is the *pseudo-root* (i.e., the k'-bit prefix) of $\mathcal{C}'$. Thus, whenever $\mathcal{P}'_{f_1,f_2}(q',k') = \mathcal{C}'$, then $\mathcal{C}'$ is relative to some string $\sigma \in \Sigma^{k'L(n')}$ sent by the second oracle to $\mathcal{P}'$ in reply to one of its queries in that execution.

Let $S = (s_1, \ldots, s_i)$ be a sequence. Then ("abusing" our concatenation operator), for any value s, we let $S|s$ denote the sequence whose first i values coincide with those of S, and whose i+1st value is s.

Lemma 2: Proposition ℘1 implies the following proposition

℘2 : *There exist a k'-bit value R'_ϵ; a $\leq 2^{c_5 k'}$-long sequence S extendible from $\Sigma^{2k'}$ to $\Sigma^{k'}$; an integer $m \in [1, 2^{c_5 k'}]$; and a sequence*

$$T = \{\sigma_i \in \Sigma^{k'L(n')} : i \in [1, 2^{c_5 k'}] - \{m\}\},$$

such that

$$Prob_{f_1 \in ext(S), \sigma \in \Sigma^{k'L(n')}}(\mathcal{P}'_{f_1, T_m = \sigma}(q', k') = R'_{\varepsilon} \cdots) > 2^{-(c_5+c_6)k'-1}.$$

Proof of Lemma 2: According to our just established notation, proposition $\wp 1$ implies

$$Prob_{f_1,f_2}(\mathcal{P}'_{f_1,f_2}(q', k') \neq \varepsilon) \geq 2^{-c_6 k'}.$$

Thus, because $\mathcal{P}'$ is a $2^{c_5 k'}$-oracle, and because it "verifies" all of its pseudo-certificates (i.e., each of them is relative to a string $\sigma \in \Sigma^{k'L(n')}$ obtained by $\mathcal{P}'$ in response to a query made to its second oracle), by averaging there must exist a positive integer $m \leq 2^{c_5 k'}$ such that $\mathcal{P}'$ outputs a pseudo-certificate relative to the mth reply of its second oracle with probability at least $2^{-(c_5+c_6)k'}$. Thus, denoting by $\mathcal{C}[m]$ a pseudo-certificate relative to the mth query to the second oracle, we have

$$p = Prob_{f_1,f_2}(\mathcal{P}'_{f_1,f_2}(q', k') = \mathcal{C}[m]) \geq 2^{-(c_5+c_6)k'}.$$

Let us now write $p = p_1 + p_2$ where

$$p_1 = Prob_{f_1,f_2}(\mathcal{P}'_{f_1,f_2}(q', k') = \mathcal{C}[m] \wedge \text{at least one } f_1\text{-collision})$$

and

$$p_2 = Prob_{f_1,f_2}(\mathcal{P}'_{f_1,f_2}(q', k') = \mathcal{C}[m] \wedge \text{no } f_1\text{-collisions}).$$

Now, in view of Corollary 1 and property $(*)$ and our constraints on c_5, c_6, and k', we have $p_1 \leq 2^{-k'/4} \leq 2^{-(c_5+c_6)k'-1}$. Thus,

(3): $Prob_{f_1,f_2}(\bar{\mathcal{P}}'_{f_1,f_2}(q', k') = \mathcal{C}[m] \wedge \text{no } f_1\text{-collisions}) > 2^{-(c_5+c_6)k'-1}$.

Now let us again make use of averaging for "growing" a sequence of pairs of strings, S, and for choosing the first $m-1$ entries of a sequence T as follows. We initially set S and T to be empty, and start executing algorithm $\mathcal{P}'$ on inputs q' and k' from the initial configuration, and handling oracle calls in the following manner. If $\mathcal{P}'$ queries its first oracle about a new $2k'$-bit string, x, we choose a reply y in $\Sigma^{k'}$ which (a) is different from all the second entries of the pairs currently belonging to S, and (b) "preserves Equation 4," that is, denoting by $\bar{T}$ a $2^{c_5 k'}$-long sequence with values in $\Sigma^{k'L(n')}$, such that

$$Prob_{f_1 \in ext(S|(x,y)), \bar{T} \in ext(T)}(\mathcal{P}'_{f_1,\bar{T}}(q', k') = \mathcal{C}[m] \wedge \text{no } f_1\text{-collisions}) > 2^{-(c_5+c_6)k'-1}.$$

reset $S = S|(x, y)$; feed $\mathcal{P}'$ with y, and resume the execution. If $\mathcal{P}'$ makes its jth query to the second oracle, and $j < m$, we choose a reply $\sigma_j \in \Sigma^{k'L(n')}$ so as to preserve Equation 4, that is, such that

$$Prob_{f_1 \in ext(S'), \bar{T} \in ext(T|\sigma_j)}(\mathcal{P}'_{f_1,\bar{T}}(q',k') = \mathcal{C}[m] \wedge \text{no } f_1\text{-collisions}) > 2^{-(c_5+c_6)k'-1};$$

reset $T = T|\sigma_j$; feed $\mathcal{P}'$ with y, and resume the execution. When $\mathcal{P}'$ makes its mth query to its second oracle, we stop this process, thereby having already constructed the entire desired sequence S (which is $\leq 2^{c_5 k'}$-long because $\mathcal{P}'$ is $2^{c_5 k'}$-call) and the first $m-1$ strings of T. As for the other the elements of T (i.e., from the $m+1$st on), we simply choose them so as to preserve Equation 4, that is, so that

$$Prob_{f_1 \in ext(S), \sigma \in \Sigma^{k'L(n')}}(\mathcal{P}'_{f_1,T_m=\sigma}(q',k') = \mathcal{C}[m]) \geq 2^{-(c_5+c_6)k'-1}.$$

Now notice that, given that $\mathcal{P}'$ is deterministic, $\forall\, f_1 \in ext(S)$ and $\forall\, \sigma \in \Sigma^{k'L(n')}$, the computation of $\mathcal{P}'_{f_1,T_m=\sigma}(q',k')$ is totally determined up to the mth query to the second oracle. Thus, there exists a unique k-bit string, R'_ε, such that, in any execution of $\mathcal{P}'_{f_1,T_m=\sigma}(q',k')$, the mth query to the second oracle consists of R'_ε. Thus, whenever such an execution ends in outputting a certificate relative to the mth query, R'_ε must be its k-bit prefix. □

Let now S, T and R'_ε be like in proposition ℘2, and consider the following algorithm $\mathcal{A}$ calling an oracle $f_1 \in ext(S)$:

Algorithm $\mathcal{A}_{f_1}$

*A*1. For $j = 1$ to $4 \cdot 2^{\ell(n')} \cdot 2^{(c_5+c_6)k'+2}$, randomly select $\sigma_j \in \Sigma^{k'L(n')}$ and execute $\mathcal{P}'_{f_1,T_m=\sigma_j}(q',k')$.

*A*2. If in Step *A*1 algorithm $\mathcal{P}'$ has output $< 2 \cdot 2^{\ell(n')}$ pseudo-certificates (of $q' \in \mathcal{L}$) whose prefix is R'_ε, HALT without any output. Else,

*A*3. Compute BL, the set of bit-locations i such that, for some execution j,

$$\mathcal{P}'_{f_1,T_m=\sigma_j}(q',k') = (R'_\varepsilon \cdots) \ni i$$

*A*4. If, for some $i \in BL$, there is no unique bit b_i such that all questions of V about bit-location i have been consistently answered with b_i, HALT without any output. Else,

*A*5. HALT outputting the $2^{\ell(n')}$-long string τ whose ith character is b_i, if $i \in BL$, and $*$ otherwise.

Lemma 3: There exists $f'_1 \in ext(S)$ such that

$$Prob_{C(\mathcal{A})}(\mathcal{A}_{f'_1} \text{halts in Step } A5) > 1/2$$
and
$$Prob_{\sigma \in \Sigma^{k'L(n')}}(\mathcal{P}'_{f'_1,T_m=\sigma}(q',k') = R'_\varepsilon \cdots) \geq 2^{-(c_5+c_6)k'-2}.$$

Proof of Lemma 3. Say that a function $f_1 \in ext(S)$ is *lucky* if

$$Prob_{\sigma \in \Sigma^{k' L(n')}}(\mathcal{P}'_{f_1, T_m = \sigma}(q', k') = R'_\varepsilon \cdots) \geq 2^{-(c_5+c_6)k'-2}.$$

Then, a simple counting argument shows that

$$Prob_{f_1 \in ext(S)}(f_1 \text{ is lucky}) \geq 2^{-(c_5+c_6)k'-2}.$$

Now, the probability that $\mathcal{A}$ (run with an oracle $f_1 \in ext(S)$) does not halt in Step $A5$ is bounded above by the sum of (1) the probability of halting at Step $A2$ and (2) the probability of halting at Step $A4$. Now a simple application of Chernoff's bounds shows that the first probability is clearly upperbounded by $2^{-(c_5+c_6)k'-3}$. Let us now show that also the second probability is upperbounded by $2^{-(c_5+c_6)k'-3}$. To this end, notice that, whenever $\mathcal{A}_{f_1}$ halts in Step $A5$, then a f_1-collision has been found. Consider in fact, the following two conceptual steps for "locating" such a collision.

$C1$. Find a bit-location i, the leaf I containing it ($I = \lceil i/k' \rceil$), and two sibling-paths (possibly belonging to the same certificate) P_1 and P_2 between leaf I and the (alleged) root of the (alleged) $\mathcal{T}_N$, according to which the bit stored in location i is different. Then, denoting $[I] = \alpha_1 \cdots \alpha_{\log N}$, we can write P_1 and P_2 as follows:

$$P_1 = R^1_{\alpha_1 \cdots \alpha_{\log N - 1} \bar{\alpha}_{\log N}}, R^1_{\alpha_1 \cdots \bar{\alpha}_{\log N - 1}}, \ldots, R^1_{\bar{\alpha}_1}, R^1_\varepsilon$$

and

$$P_2 = R^2_{\alpha_1 \cdots \alpha_{\log N - 1} \bar{\alpha}_{\log N}}, R^2_{\alpha_1 \cdots \bar{\alpha}_{\log N - 1}}, \ldots, R^2_{\bar{\alpha}_1}, R^2_\varepsilon$$

where $R^1_{[I]} = R^1_{\alpha_1 \cdots \alpha_{\log N}}$ is the value stored in leaf I according to P_1, and $R^2_{[I]} = R^2_{\alpha_1 \cdots \alpha_{\log N}}$ is the value stored in leaf I according to P_2.

(*Comments:* First, $R^1_{[I]} \neq R^2_{[I]}$, because their $(i - ([I]-1)k')$th bits are different, since they represent their respective values stored in bit-location i. Second, $R^1_\varepsilon = R'_\varepsilon = R^2_\varepsilon$ because both sibling-path occur within a certificate or two certificates whose pseudo-root is R'_ε.)

$C2$. Find $m \in [1, \log N]$ such that

$$X = R^1_{\alpha_1 \cdots \alpha_m} | R^1_{\alpha_1 \cdots \bar{\alpha}_m} \neq R^2_{\alpha_1 \cdots \alpha_m} | R^2_{\alpha_1 \cdots \bar{\alpha}_m} = Y,$$

but $f_1(X) = f_1(Y)$.

(*Comment:* Such m must exist because of three reasons.

First, $f(R^1_{\alpha_1} | R^1_{\bar{\alpha}_1}) = R'_\varepsilon = f(R^2_{\alpha_1} | R^2_{\bar{\alpha}_1})$.

Second, $R^1_{\alpha_1 \cdots \alpha_{\log N}} | R^1_{\alpha_1 \cdots \bar{\alpha}_{\log N}} \neq R^2_{\alpha_1 \cdots \alpha_{\log N}} | R^2_{\alpha_1 \cdots \bar{\alpha}_{\log N}}$. In fact, their k'-bit prefixes

Third, $\mathcal{P}'$ verifies all its non-empty outputs, and thus has made all relevant queries to oracle f_1, including X and Y.)

Now, notice that A calls its oracle $4 \cdot 2^{\ell(n')} \cdot 2^{(c_5+c_6)k'+2}$ times. Therefore, because $c_5 = c_6 = 1/16$ and $k' > (n')^{c_4} > 8\ell(n') + 32$, $\mathcal{A}$ is a $2^{k'/4}$-call algorithm. Moreover, because sequence S is extendible from $\Sigma^{2k'}$ to $\Sigma^{k'}$, Corollary 1 implies

$$Prob_{C(\mathcal{A}),f_1 \in ext(S)}(\mathcal{A}_{f_1} \text{finds a } f_1\text{-collision}) < 2^{-k'/2} < 2^{-(c_5+c_6)k'-3}.$$

Consequently, the fraction of functions $f_1 \in ext(S)$ for which $\mathcal{A}_{f_1}$'s probability of halting in Step $A5$ exceeds $1/2$ is less than $2 \cdot 2^{-(c_5+c_6)k'-3} = 2^{-(c_5+c_6)k'-2}$. Thus there must exist a function f_1' as desired in our hypothesis. □

Definitions: Whenever $\sigma \in \Sigma^{k'L(n')}$, consider it as the concatenation of k' strings, each $L(n')$-bit long, and denote by $\sigma[i]$ the ith such segment.

For any string τ over the alphabet $\{0, 1, *\}$, define

$$P'_\tau = Prob_{\sigma \in \Sigma^{k'L(n')}}(\textstyle\bigwedge_{i=1}^{k'} SV_{\sigma[i]}(q', k', \tau) = accept \,|\, \mathcal{P}'_{f_1', T_m=\sigma}(q', k') = R'_\varepsilon \cdots),$$

where $SV_{\sigma_j}(q', k', \tau)$ denotes the execution of samplable verifier SV on inputs q' and k', coin tosses σ_j, and access to string τ; with the provision that SV *rejects* if it accesses a bit-location of τ storing the value $*$.

Lemma 4: There exists a string $\tilde{\tau}$, over $\{0, 1, *\}$, such that $P'_{\tilde{\tau}} > 1/2$.

Proof of Lemma 4. We shall prove our lemma by showing that algorithm $\mathcal{A}_{f_1'}$ has a positive probability of outputting a string $\tilde{\tau}$ as desired.

Let $\bar{\tau}$ be a string such that $P'_{\bar{\tau}} \leq 1/2$. Then,

$$Prob_{C(\mathcal{A})}(A_{f_1'} \text{ outputs } \bar{\tau}) < 2^{-2^{\ell(n')}}.$$

In fact, for outputting a non-empty string in Step $A5$, $\mathcal{A}_{f_1}$ should successfully compute at least $2 \cdot 2^{\ell(n')}$ pseudo-certificates with prefix R'_ε in Step $A1$. Thus, consider the first $> 2 \cdot 2^{\ell(n')}$ randomly- and independently-selected strings $\sigma \in \Sigma^{k'L(n')}$ such that

$$\mathcal{P}_{f_1', T_m=\sigma}(q', k') = (R'_\varepsilon \cdots).$$

Then, in order for $\mathcal{A}_{f_1'}$ to output $\bar{\tau}$, for each of these σ, the event

$$\bigwedge_{i=1}^{k'} SV_{\sigma[i]}(q', \bar{\tau}) = accept$$

should occur. Hence, by the definition of $P'_{\bar{\tau}}$ and the fact that its value is less than $1/2$, the probability that all these events occur is at most $2^{-2 \cdot 2^{\ell(n')}}$.

Now, because any string outputable by $\mathcal{A}_{f_1'}$ is $2^{\ell(n')}$-long, there may be at most $3^{2^{\ell(n')}}$ strings τ such that $P'_\tau \leq 1/2$. Thus,

$$Prob_{C(\mathcal{A})}(\mathcal{A}_{f_1'} = \tau \wedge P'_\tau \leq 1/2) < \sum_{j=1}^{3^{2^{\ell(n')}}} 2^{-2\cdot 2^{\ell(n')}} < 1/2.$$

But because, as shown in Lemma 3, $\mathcal{A}_{f_1'}$ halts in Step $A5$ (and thus outputs a non-empty string τ) with probability $> 1/2$, $\mathcal{A}_{f_1'}$ must output a string $\tilde{\tau}$ such that $P'_{\tilde{\tau}} > 1/2$. □

We are now ready to finish the proof of computational soundness. Define

$$P_{\tilde{\tau}} = Prob_{\sigma \in \Sigma^{k' L(n')}}(\bigwedge_{i=1}^{k'} SV_{\sigma[i]}(q', \tilde{\tau}) = accept)$$

and

$$\tilde{P}_{\tilde{\tau}} = Prob_{s \in \Sigma^{L(n')}}(SV_s(q', \tilde{\tau}) = accept).$$

Then,

$$(\tilde{P}_{\tilde{\tau}})^{k'} = P_{\tilde{\tau}} \geq P'_{\tilde{\tau}} \cdot Prob_{\sigma \in \Sigma^{k' L(n')}}(\mathcal{P}'_{f_1', T_m = \sigma}(q', k') = R'_\epsilon \cdots) \geq 2^{-(c_5 + c_6)k' - 3}.$$

Now, because $c_5 = c_6 = 1/16$ and k' is, in particular, ≥ 8, the above inequality implies

$$\tilde{P}_{\tilde{\tau}} \geq 1/2,$$

a contradiction to the fact that $q' \notin \mathcal{L}$ and SV is a samplable verifier. □

Approximating the Random Oracles

We wish to point out, in a quick and informal way, that the just constructed CS proof-system $(\mathcal{P}, \mathcal{V})$ would maintain some key properties if one substituted its random oracles with some suitable approximations. These approximations are important not only because we are not sure that a random oracle is realizable by a physical process, but also because all physical processes that behave as random oracles may be too inconvenient to use (e.g., because they are too slow in answering their queries). Of course, deciding which approximations are satisfactory depends from the particular context at hand. Let us mention some of them.

- *Quasi-random oracles.* To begin with, one can see that for $(\mathcal{P}, \mathcal{V})$ to be a guaranteed proof-system it is not necessary that f_1 and f_2 be truly random oracles, but sufficiently random ones.
  Assume, for instance, that one may easily select oracles $f_1' : \Sigma^{2k} \to \Sigma^k$ according to a distribution F_1^k such that, for some positive constants e and d,

$$\forall k \text{ and } \forall\ 2^{dk}\text{-call algorithm } A,\ |{p_A}^{k'} - {p_A}^{k}| \leq 2^{-ek},$$

where ${p_A}^{k'}$ denotes the probability (computed selecting f_1' according to ${F_1}^{k}$) that $A_{f_1'}$ outputs 1, and ${p_A}^{k}$ the probability (computed selecting $f_1 : \Sigma^{2k} \to \Sigma^{k}$ at random) that A_{f_1} outputs 1. Then, such f_1' may be used instead of truly random oracles within the construction of $(\mathcal{P}, \mathcal{V})$.

- *Men in the cave.* Lipton and Rabin have, independently and in different occasions, suggested that a random oracle can be implemented by a trusted party with enough memory.
  In Rabin's suggestive language, the oracle (like respectable ones of times past) is a man living in a cave, with a coin and lots of pen-and-paper. Whenever someone queries him about a string σ, the wise and old man first consults his notes to see whether such a request was ever made before. If so, he retrieves the answer given in the past, and gives that same answer now. If not, he flips his coin the right number of times so as to construct a random string τ; gives τ as his answer; and records that, from now on, query σ should always be answered with τ.
  It would appear that such an implementation of a random oracle is useless in our application, because if our man is corruptable, it would be easy for a cheating Prover to find CS witnesses for false statements. This is not true, because there are contexts in which we can be sure that a man-in-the-cave implementation of a random oracle is trustworthy. For instance, in the context of Certified Computation (see later sections), we may wish to construct a man-in-the-cave by means of a simple probabilistic algorithm, run as a subroutine by other programs for our own use. (Jumping ahead, our oracle only "talks" to an algorithm of *ours* whose results *we* wish to test.) In this application, being the only beneficiary of a correct man-in-the-cave implementation, we have no incentive to cheat (ourselves)!
- Finally, a random oracle can be implemented by having the cryptographic random-function construction of Goldreich, Goldwasser, and Micali [17] (with suitably large security parameters) run within in a tamper-proof chip; one, that is, whose content (program and data) cannot be altered or read from the outside (without destroying it altogether). If the seeds are secret, their pseudo-random functions are provably indistinguishable from truly random ones by any observer, who access them as an oracle, that does not have sufficient resources for solving a given problem (e.g., factoring a 10,000-bit number).
  To ensure its secrecy, the seed could be selected within the chip by some "physical" method (e.g., noise diodes, Geiger counter, etc.). Alternatively, different people can each provide to the chip his own random and secret number, and the chip obtains its own seed by exclusive-oring all these inputs. This way, unless all the parties providing inputs to the chip are corrupt and

reveal what their individual input was, the resulting seed will be both random and secret. (Incidentally, if the seed were known, certain unpredictability properties of the corresponding pseudo-random function would disappear, but, for all we know, the difficulty of finding CS proofs for false statements may remain sufficiently intact.)

3.2 CS Proofs with a Random String

Though in this paper we do not address the problem of explicitly constructing CS proofs with a random string, we wish to present their definition and mention their applications.

The Notion of a CS Proof-System with a Random String

In a CS proof-system with a random oracle a cheating Prover was allowed to be an algorithm which could make only a bounded number of oracle calls, but have an arbitrarily long description. Informally, this was so because finding guaranteed CS certificates for a false statement implied finding special structures in a random oracle —e.g., two different strings x and y such that $f_1(x) = f_1(y)$— discoverable only by making extraordinarily many queries. Having lots of information "built-in" in the finite-state control of an oracle-calling algorithm could not help at all in finding these structures.

In a CS proof-system with a random string, Prover and Verifier are ordinary (as opposed to oracle-calling) algorithms, sharing a short random string r. That is, whenever the security parameter is k, they share a string r that both believe to have been randomly selected among those having length k^c, where c is a positive constant.

Because string r is assumed to be universally known (at lest by all those who wish to check CS proofs of security-level k), and because a CS certificate of $\mathbf{q}$ is verifiable in our new setting given $\mathbf{q}$ and r, a good-looking certificate for a false statement is certainly discoverable by exhaustive search. Thus, the only hope to stop a malicious Prover from cheating consists of bounding the number of computational steps he can perform. But bounding a cheating Prover's running time is meaningless unless its description is also bounded. (For instance, factoring a randomly chosen k-bit integer appears to be computationally intractable when k is large, but not for an algorithm whose description is about 2^k-bit long! Indeed, the finite state control of such a Turing machine could easily encode the factorization of all k-bit integers.) For this reason, in a cryptographic CS proof-systems, a cheating prover is envisaged to be an algorithm whose running time and description, in some standard encoding, are both bounded. We actually accomplish both tasks at once, by letting cheating provers be Boolean combinatorial circuits with a bounded number of gates.

Definition: A *circuit of size* $\leq s$ is a finite function computable by at most s Boolean gates, where each gate is either a NOT-gate (with one binary input and one binary output) or an AND-gate (with two binary inputs and one binary output).

Definition: Let (P, V) be a pair of Turing machines, the second of which runs in polynomial-time. We say that such a pair (P, V) is a *CS proof-system with a random string* if there exists a sequence of 6 positive constants, $c_1, \ldots, c_6$ (refereed to as the *fundamental constants* of the system), such that the following two properties are satisfied:

1''. *Feasible Completeness.* $\forall$ integers $n > 1$, $\forall$ n-bit input $\mathbf{q} = (\bar{M}, x, t) \in \mathcal{L}$, $\forall$ unary integers k, and $\forall$ $r \in \Sigma^{k^{c_1}}$,
(i) on inputs $\mathbf{q}, k$, and r, P halts within $(nkt)^{c_2}$ computational steps, and outputs a binary string whose length is $\leq (nk \log t)^{c_3}$, and
(ii) Whenever $\mathcal{C} = P(k, r, \mathbf{q})$, $V(k, r, \mathbf{q}, \mathcal{C}) = YES$.

2''. *Computational Soundness.* $\forall$ integers $n > 1$, $\forall$ $n-$bit $\mathbf{q}' \notin \mathcal{L}$, $\forall$ $k > n^{c_4}$, and $\forall$ (cheating) circuits P' whose size is $\leq 2^{c_5 k}$,

$$Prob_{r \in \Sigma^{k^{c_1}}}(P'(\mathbf{q}', k, r) = C' \wedge V(\mathbf{q}', k, r, C') = YES) \leq 2^{c_6 k}.$$

The Importance of CS Proofs with a Random String

CS proof-systems with a random string are important because of at least three reasons.

First, because (as we shall explain in a forthcoming paper) they are explicitly constructable under a new, but in our opinion plausible, complexity assumption.

Second, because it is much easier for two players (i.e., Prover and Verifier) or society at large to establish a common random string than a common random oracle.

Third, because the certificates for $\mathbf{q} \in \mathcal{L}$ they produce are polynomial-time recognizable.

The latter technical reason is full of notable consequences. In particular, it allows us to enlarge Blum's notion of *program checking* in new and powerful ways, and to prove that heuristics for $\mathcal{NP}$-complete problems are checkable in this broader sense. For instance, given an efficient heuristic H for graph Hamiltonicity and a specific graph G, we show how it is possible to run H twice (once on G and once of an efficiently computed graph G' for which deciding Hamiltonicity

is roughly "as hard as for G") and either (1) prove that H is wrong on at least one of these two graphs, or (2) establish with a "great degree of confidence" that H's answer about graph G is correct.

These applications of CS proofs with a random string will also appear in the mentioned forthcoming paper.

3.3 Deterministic CS Proofs

Definition: Let (P, V) be a pair of Turing machines, the second of which runs in polynomial-time. We say that such a pair (P, V) is a *deterministic CS proof-system* if there exists a sequence of 5 positive constants, $c_1, \ldots, c_5$ (refereed to as the *fundamental constants* of the system), such that the following two properties are satisfied:

1'''. *Feasible Completeness.* $\forall$ integers $n > 1$, $\forall$ n-bit input $\mathbf{q} = (\bar{M}, x, t) \in \mathcal{L}$, and $\forall$ unary integers k,
(i) on inputs $\mathbf{q}$ and k, P halts within $(nkt)^{c_1}$ computational steps, and outputs a binary string whose length is $\leq (nk \log t)^{c_2}$, and
(ii) Whenever $\mathcal{C} = P(\mathbf{q}, k)$, $V(\mathbf{q}, k, \mathcal{C}) = YES$.

2'''. *Computational Soundness.* $\forall$ integers $n > 1$, $\forall n-$bit $\mathbf{q}' \notin \mathcal{L}$, $\forall$ $k > n^{c_3}$, and $\forall$ (cheating) probabilistic algorithms $P'(\cdot, \cdot)$ halting in less than $\leq 2^{c_4 k}$ steps whenever their second input is k (in unary),

$$Prob(P'(\mathbf{q}', k) = C' \wedge V(\mathbf{q}', k, C') = YES) \leq 2^{c_5 k}.$$

As we said, we believe that the complexity assumption needed to build deterministic CS proof-systems is much stronger than that necessary for CS proof-systems with a random string.

4 Certified Computation

In this short section we reinterpret the results obtained so far in terms of *computation* rather than proofs. More precisely, we aim at obtaining certificates ensuring that no error has occurred in a *given* execution of a *given* algorithm on a *given* input. That is, certified computation does not deal with semantic questions such as "is algorithm A correct?" Rather it addresses the following syntactic question:

Is string y what algorithm A should output on input x (no matter what A is supposed to do)?

This question is quite crucial. Consider running executing A on an input x so as to obtain a result y. How can we be sure that $y = A(x)$? In real life, such an execution occurs by means a physical computer, and all sort of errors may occur, yielding a wrong value for $A(x)$. For instance, the computer hardware may be defective. Alternatively, the hardware may function properly, but the operating system may be flawed. Alternatively yet, hardware and software may be fine, but some α rays succeed in flipping a bit together with its controls, so that the original bit value is not restored.

Also, having run A on input x so as to obtain a result y, can we convince someone else that y is the right result without having him re-do the computation himself?

Certified computation provides a way to answer these basic questions, for any algorithm and any input.

"Defining" Certified Computation. In view of our work so far, formalizing and exemplifying (at least given a random oracle) the notion of certified computation is rather straightforward, but tedious indeed. We thus think that is best to proceed at a very intuitive level.

> **Informal Definition:** A *certified-computation system* is a pair of efficient algorithms, $(\mathcal{C}, \mathcal{V})$. Given any algorithm A as input, $\mathcal{C}$ outputs an equivalent algorithm A' enjoying the following properties.
>
> 1. A' runs in essentially the same time as A does;
> 2. A' receives the same inputs applicable to A and produces the same outputs; and
> 3. for each input x, A' produces, besides the right output y, also a short and easily inspectable string, C_{Axy}, vouching that indeed $y = A(x)$ in the following sense:
>
>   If $y = A(x)$, then $\mathcal{V}(A, x, y, C_{Axy}) = YES$. Else, it is very hard to find a string σ such that $\mathcal{V}(A, x, y, \sigma) = YES$.

Of course, one may ask who verifies the correctness of the verifier (i.e., either of algorithm $\mathcal{V}$ itself or of its executions). Note, however, that such a $\mathcal{V}$ is a *unique* program, capable of verifying certificates for the correct execution of *all* other programs. It is thus meaningful to invest sufficient time in proving the correctness of this particular algorithm (e.g., by verification methods). Also, being $\mathcal{V}$ quite efficient (and running on short inputs) we may afford to execute it on very "conservative" hardware (i.e., with particular redundancy, resiliency, and so on), or even on a multiplicity of hardwares.

Constructing certified-computation systems. One possible way of constructing program certification systems essentially consists of giving a CS proof

of the statement "$y = A(x)$." That is, on input x, $A' = \mathcal{C}(A)$ first runs A on input x so that, after some number T of steps, an output y is obtained. Then, A' outputs y and a CS proof that $y = A(x)$. Because the latter statement can be decided in time T, the length of its corresponding CS proof will be polynomial in $\log T$ (and, of course, $|x|$, $|y|$, and some suitable security parameter k), and so will be the time to inspect it, and the time to construct it.

Pros and cons. A certified-computation system can be considered as a *universal and guaranteed casting-out-nines procedure.* (Such a procedure is executed after a multiplication, but does does not guarantee an "arbitrarily high guarantee of correctness" for every single input.) In other words, certified computation is not a special property enjoyed by a few computational problems (such as multiplication or GCD computation.[22]), but is an *intrinsic* property of computation.

However, unless one can find a version or representation of the execution history of an algorithm that is both convenient and sufficient for our purposes,[23] our result will not be practical.

Assumptions and implementations. Certified-computation systems can be built based on a random oracle, but are particularly meaningful when constructed, under our assumption, based on a CS proofs with a random string for at least two reasons.

First, in this application, the random string needs not be agreed upon by both Prover and Verifier by virtue of some possibly difficult negotiation. When seeking reassurance that indeed $y = A(x)$, the user of a certified-computation system $(\mathcal{P}, \mathcal{V})$ controls both $\mathcal{P}$ and $\mathcal{V}$, and thus can choose their common string in a way that he believes to be genuinely random, without "asking for their consent."

---

[22] Indeed, the extended GCD algorithm may be a bit more cumbersome than the ordinary GCD onalgorithm, but more "reliable". In fact, according to Blum's point of view, by outputting not only the greatest common divisor, c, of two integers a and b, but also two integers x and y such that $ax + by = c$, the extended CGD algorithm allows one to check the validity of c quite easily. That is, by checking that indeed $ax + by = c$ and that c evenly divides both a and b, all operations much simpler and computationally less intensive than GCD computation. Note, however, that x and y consist of a quite long certificate for c's correctness (in relative terms), and that three multiplications and one addition are quite long (relative to the time required by GCD computation). By comparison, our certified-computation system may produce relatively shorter and more easily inspectable certificates, but has a probability of error.

[23] Certainly, one that does not consist of the sequence of all instantaneous Turing machine configurations.

In addition, in this application, algorithm $\mathcal{P}$ "sits on top of the user desk" and thus there is no question of it being implemented by a circuit comprising 2^k gates (nor of its making more than 2^k steps of computation), even for moderately small k. That is, if the underlying complexity assumption is true, a certified-computation system *de facto* offers the same guarantees of a probabilistic algorithm.

Conceivable applications of certified computation. Certified computation can in principle be quite useful when "contracting out" computer time. Indeed, consider an algorithm A that we believe to be correct, but is very time-consuming. Then, we can temporarily hire a super-computing company for executing A on a given input x on their computers, and agree that we will pay for their efforts if they give us back the value $y = A(x)$ together with a certificate of correct execution.

Still in principle, certified computation may facilitate the verification of certain mathematical theorems proven with the help of a computer search (as in the case of the 4-color theorem). For instance, the proof may depend on a Lemma stating that there is no sparse graph with less than 50 nodes possessing a given property, and the Lemma could be proved by means of an exhaustive search taking a few years of computing. Usually, algorithms performing an exhaustive search appear to be correct "by eye inspection". Thus, rather than (1) asking the reader of trusting that such a search has been done and has returned a negative result, or (2) asking the reader to perform such an extensive computation himself, one might publish together with the rest of the proof a compact, correct-execution certificate. If the security parameter where chosen to be —say— 1,000, then even the most skeptic reader might believe that no one has invested $2^{1,000}$ steps of computation in order to find a false certificate, nor that he has succeeded in finding one by relying on a probability of success less than $2^{-1,000}$.

5 Final Remarks

Are CS proofs really proofs? In our minds this question really goes together with an older one: do probabilistic algorithms [38] [31] really compute? There is a sense in which both answers should be NO. These negative answers, in our opinion, may stem from two different reasons: (1) a specific interpretation of the words "proof" and "computation," and (2) our mathematical tradition. The first reason is certainly true, but also "innocuous." The second is more "dangerous" and less acceptable: not because it is false that these notions break with a long past, but because the unchallenged length of a tradition should not be taken as implying that specific formalizations of fundamental intuitions are "final."

Indeed, we believe that even fundamental intuitions cannot be divorced by the large historical contexts in which they have arisen, and we expect that they will change with the changing of these contexts. Up to now, the ideas of this paper have not met much favor, exception made for the invaluable support of a trusted group of scientific friends. We hope, however, that, in the future, they will be regarded to be as natural as is, *by now*, the notion of probabilistic computation.

Truths versus proofs. According to our highest-level goal, CS proofs propose a *very appealing* relationship between proving and deciding. In the thirties, Turing, Church, and others suggested that deciding a mathematical statement consists of running a specific algorithm. Today, we are suggesting that proving a mathematical statement consists of feasibly speeding up the verification of the output of any decision algorithm. That is, proofs should guarantee to any third party an exponential speed-up in verifying that the output of any deciding process is YES; but they should not be more time-consuming than the computations whose verification they wish to facilitate. This, in our opinion, is the right relationship between proving and deciding, and one that guarantees that proving is both a useful and a distinct notion.

In order to offer this guarantee, CS proofs replace the traditional notion of truth with a *computational* one. Namely, the difference between proving true a correct statement and proving true a false one now is the difference between an easy computation and an extraordinarily hard one. Indeed, wishing to convince *any* third party (and not just a Verifier interacting with them) of the verity of a given theorem, CS Provers issue a CS certificate, a string that can be thought of as a "compressed version" of a much longer deciding computation. But the same conciseness that gives CS certificates their distinctive advantage, also causes them to "lose information" with respect to the computations they compress. It is thus possible that correct-looking certificates of false theorems exist, and whenever they do, of course, they can also be found by means of an exponential search. CS proofs guarantee, however, that essentially only an exponential search could be successful in finding misleading certificates.

In sum, the notion of a true theorem has not changed. What has changed, and in a computational-complexity direction, is the notion of what it means to *prove that a theorem is true.* Besides being very adequate in practice and very advantageous, this change also is, in our opinion, very natural. Indeed, proofs have no meaning outside efficiency: with or without modern terminology, they have always been, at least implicitly, a complexity-theoretic notion.

Let me emphasize, however, that, though very natural, the computational boundedness of a cheating prover is not, *per se*, a goal of the notion of a proof. Rather, it is the Trojan horse that lets us sneak in and achieve our desiderata. It is actually very important to establish whether some type of CS proofs exist

when a cheating Prover can compute for an unbounded amount of time. [24]

We believe and hope that CS proofs maybe useful in many more applications than the ones envisaged here.

6 Acknowledgments

My most sincere thanks go to Allan Borodin, Steve Cook, Oded Goldreich, Shafi Goldwasser, Leonid Levin, and Michael Rabin for encouraging and criticizing this research, in different ways and at various stages of its development.

Thanks also to Shai Halevi, Ray Sidney and Janos Makowski (and two referees) for their comments.

References

1. S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. *Proof verification and hardness of approximation problems.* Proc. 33rd. IEEE Conference on Foundation of Computer Science, 1992, pp. 14-23.
2. S. Arora and M. Safra. *Probabilistic checking of proofs.* Proc. 33rd. IEEE Conference on Foundation of Computer Science, 1992, pp. 2-13.
3. L. Babai and L. Fortnow and L. Levin and M. Szegedy. *Checking Computation in Polylogarithmic Time.* Proc. of STOC91.
4. L. Babai and S. Moran. JCSS 1988. A preliminary version due to the first author, "Trading Group Theory for Randomness," appeared in Proc. 17th Annual Symposium on Theory of Computing, 1985, pp. 421-429.
5. M. Ben-or and S. Goldwasser and J. Kilian and A. Wigderson. *Multi Prover Interactive Proofs: How to Remove Intractability.* Proc. 20th ACM Symp. on Theory of Computing, 1988, pp. 113-131.
6. M. Blum, A. De Santis, S. Micali, and G. Persiano. *Non-Interactive Zero-Knowledge.* SIAM J. on Comp. 1991.
7. M. Blum, P. Feldman, and S. Micali. *Non-Interactive Zero-Knowledge Proof Systems and Applications.* STOC 1988.
8. M. Blum and S. Kannan. *Designing Programs that check their work.* Proc. 21st Symposium on Theory of Computing, 1989, pp. 86-97.
9. M. Blum, M. Luby, and R. Rubinfeld. *Self-Testing and Self-Correcting Programs, With Applications to Numerical Problems.* Proc. 22nd ACM Symp. on Theory of Computing, 1990, pp. 73-83.

[24] By means of a weaker construction (in particular, one not enjoying feasible completeness), we had previously observed that, via Merkle trees and the self reducibility of the permanent, a bounded prover could demonstrate membership in any $\#P$-language. Our observation in fact followed the preliminary announcement that $\#\mathcal{P}$ is efficiently accepted by a 2-Prover system, but lost any interest a few days later, with the final announcement that $\#\mathcal{P} \subset \mathcal{IP}$ [26]. We wonder if also CS proofs (at least their interactive counter-part) may be subject to the same fate.

10. M. Blum and S. Micali. *How to Generate Cryptographically-Strong Sequences of Pseudo-Random Bits.* SIAM J. on Comp. vol 13, 1984

11. G. Brassard and D. Chaum and C. Crepeau. *Minimum Disclosure Proofs of Knowledge.* J. Comput. System Sci., 37, 1988, pp. 156-189.

12. S. Cook. *The Complexity of Theorem Proving Procedures.* Proc. 3rd Annual ACM Symposium on Theory of Computing, 1971, pp. 151-158.

13. U. Feige and S. Goldwasser and L. Lovasz and S. Safra and M. Szegedi. *Approximating Clique is Almost NP-complete.* 32nd FOCS, 1991, pp. 2-12.

14. A. Fiat and A. Shamir. *How to Prove Yourselves: Practical Solutions of Identification and Signature Problems.* Proc. Crypto 86, Springer- Verlag, 263, 1987, pp.186-194.

15. L. Fortnow. *The Complexity of Perfect Zero Knowledge.* Randomness and Computation, Advances in Computer Research, ed. S. Micali, JAI Press, 1989, pp. 327-344.

16. L. Fortnow, J. Rompel, and M. Sipser. *On the Power of Multi-Prover Interactive Protocols.* Proc. 3rd Structure in Complexity Theory Conf., 1988, pp. 151-158.

17. O. Goldreich, S. Goldwasser, and S. Micali. *How To Construct Random Functions.* J. of ACM 1986

18. S. Goldwasser and S. Micali and C. Rackoff. *The Knowledge Complexity of Interactive Proof Systems.* SIAM J. Comput., 18, 1989, pp. 186-208. An earlier version of this result informally introducing the notion of a proof of knowledge appeared in Proc. 17th Annual Symposium on Theory of Computing, 1985, pp. 291-304. (Earlier yet versions include "Knowledge Complexity," submitted to the 25th Annual Symposium on the Foundations of Computer Science, 1984.)

19. S. Goldwasser, S. Micali, and R. Rivest, *A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks*, SIAM J. Comput., Vol 17, No. 2, April 1988, pp. 281-308.

    (A preliminary version of this article appeared with the title "A paradoxical solution to the signature problem" in Proc. of 25th Annual IEEE Symposium on the Foundations of Computer Science, FL, November 1984, pp. 464-479.)

20. O. Goldreich, S. Micali, and A. Wigderson. *Proofs that Yield Nothing But Their Validity or All Languages in $\mathcal{NP}$ have Zero-Knowledge Proof Systems.* J. of the ACM, Vol 38, No. 1, July 1991, pp. 691-729.

    (A preliminary version of this paper, under the title "Proofs that yield nothing but their validity and a methodology for cryptographic protocol design," appeared in Proc. 27th Annual Symposium on Foundations of Computer Science, IEEE, New York, 1986, pp. 174-187.)

21. R. Karp. *Reducibility among combinatorial problems.* Complexity of Computer Computations, R. Miller and J. Thatcher eds., Plenum, New York, 1972, pp. 85-103.

22. J. Kilian. *A Note on Efficient Zero-Knowledge Proofs and Arguments.* Proc. 24th Ann. Symp. on Theory of Computing, Victoria, Canada, 1992.

23. R. Impagliazzo, J. Hastad, L. Levin, and M. Luby. *Pseudo-Random Generation under uniform Assumptions.* STOC 1990.

24. R. Impagliazzo, L. Levin, and M. Luby. *Pseudo-Random Generation From one-way functions.* STOC 1989
25. L. Levin. *Universal Sequential Search Problems.* Problems Inform. Transmission, Vol. 9, No. 3, 1973, pp. 265-266.
26. C. Lund and L. Fortnow and H. Karloff and N. Nisan. *Algebraic Methods for Interactive Proof Systems.* Proc. 22nd STOC, 1990.
27. R. Merkle. *A Certified Digital Signature.* Proc. Crypto 1989. Springer Verlag, 1990.
28. S. Micali. *CS Proofs.* Proc. 35th Annual Symposium on Foundations of Computer Science, 1994, pp.
    (An earlier version of this paper appeared as Technical Memo MIT/LCS/TM-510. Earlier yet versions were submitted to the 25th Annual Symposium on Theory of Computing, 1993, and the 34th Annual Symposium on Foundations of Computer Science, 1993.)
29. S. Micali, Private Communication to Shafi Goldwasser, 1992.
30. A. Polishchuk and D. Spielman. *Nearly-linear Size Holographic Proofs.* Proc. STOC 1994.
31. M. Rabin. *Digitalized Signatures.* in Foundations of Secure Computation, Academic Press, 1978, pp. 155-168.
32. M. Rabin. *Digitalized Signatures as Intractable as Factorization.* MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-212, Massachusetts Institute of Technology, Cambridge, MA, January 1979.
33. M. Rabin. *Probabilistic algorithms for testing primality.* J. Number Theory, Vol. 12, 1980, pp. 128-138.
34. R. Rivest, A. Shamir, and L. Adleman. *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems.* Comm. ACM, Vol. 21, 1978, pp. 120-126.
35. R. Rivest. *The MD5 Message-Digest Algorithm.* Internet Activities Board, Request for Comments 1321, April 1992.
36. *Secure Hash Standard.* Federal Information Processing Standards, Publication 180, 1993.
37. A. Shamir. $IP = PSPACE$. Proc. 31st IEEE Foundation of Computer Science Conference, 1990, pp. 11-15.
38. R. Solovay and V. Strassen. *A fast Monte-Carlo test for primality.* SIAM J. Comp., Vol. 6, 1977, pp. 84-85.
39. M. Sudan. *Efficient checking of polynomials and proofs and the hardness of approximation problems.* Ph.D. Thesis, University of California at Berkeley, 1992.
40. A. Yao. *Theory and Applications of Trap-Door Functions.* Proc. 23rd IEEE on Foundations of Computer Science, 1982.