SIAM J. Comput.
Vol. 30, No. 4, pp. 1253–1298

COMPUTATIONALLY SOUND PROOFS*

SILVIO MICALI†

Abstract. This paper puts forward a new notion of a proof based on computational complexity and explores its implications for 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 enable us to

1. prove that verifying is easier than deciding for all theorems;
2. provide a quite effective way to prove membership in computationally hard languages (such as $\mathcal{C}o$-$\mathcal{NP}$-complete ones); and
3. show that every computation possesses a short certificate vouching its correctness.

Finally, if a special type of computationally sound proof exists, we show that Blum's notion of program checking can be meaningfully broadened so as to prove that $\mathcal{NP}$-complete languages are checkable.

Key words. interactive proofs, probabilistically checkable proofs, random oracles, Merkle trees

PII. S0097539795284959

1. Introduction.

CS proofs. 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) 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, which (1) is as easy to find as possible, (2) is very easy to verify, and (3) offers a strong computational guarantee about the verity of 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 close to that needed to Turing-accept S. By "very easy to verify" we mean that the time necessary to inspect a CS proof of a statement S is substantially smaller than the time necessary to Turing-accept S. 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 is based on a random oracle and provably yields a CS proof system without any unproven assumption. The second relies on a new complexity conjecture: essentially, that it is possible to replace the random oracle of the first construction with a cryptographic function and obtain, mutatis mutandis, similar results.

Applications of CS proofs. In either implementation, 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 not only that the time necessary to verify is substantially smaller than the time necessary to accept, but, more importantly,

*Received by the editors April 21, 1995; accepted for publication (in revised form) October 28, 1999; published electronically October 11, 2000.
http://www.siam.org/journals/sicomp/30-4/28495.html

†Laboratory for Computer Science, MIT, Cambridge, MA 02139 (silvio@theory.lcs.mit.edu).

that this "speed-up" occurs for *all* theorems (rather than just *a few* theorems). In addition, they provide a quite effective way for proving membership in computationally hard languages (e.g., $\mathcal{C}o$-$\mathcal{NP}$ complete ones).

CS proofs also possess novel and important implications for computational correctness. In particular, in either implementation, they imply that every computation possesses a short certificate vouching for its correctness. In addition, if implementable in the second manner, CS proofs also imply that any heuristic or program for an $\mathcal{NP}$-complete problem is cryptographically checkable. This application at the same time extends and demonstrates the wide applicability of Blum's [11] original framework for checking program correctness.

Origins 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, 17] and that of a zero-knowledge argument [13] have been the closest sources of inspiration in conceiving the new notion itself. In exemplifying the new notion, most relevant has been a construction of Kilian's [23], and, to a lesser extent, the works of [18] and [9].

Naturalness of CS proofs. We wish to emphasize that, from the above starting point, the 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 follow quite naturally. This simplicity, in our opinion, lends support to our new perspective.

2. New goals for efficient proofs.

Proofs without demands for efficiency: Semirecursive languages. Truth and proofs have been traveling hand in hand. 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, the corresponding true statements form a *semirecursive* set.[2] In the expressive and elegant approach of Turing, such sets possess two equivalent characterizations particularly important for our enterprise, one in terms of accepting algorithms and one in terms of verifying algorithms.

1. A language (set of binary strings) L is semirecursive if and only if there exists an *(accepting)* Turing machine A such that
$$L = \{x \ : \ A(x) = YES\}.$$
2. A language L is semirecursive if and only if there exists a *(verifying)* Turing machine V, halting on all inputs, such that
$$L = \{x \ : \ \exists \sigma \in \{0,1\}^* \text{ such that } V(x,\sigma) = YES\}.$$

Establishing the verity of a statement is thus a purely algorithmic process, and (at least formally) classical proofs—the σs of the second definition—are just strings. Because in this paper a "true" theorem simply is one derivable in a given theory, for

[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 endeavors than may actually exist.

[2] Again, throughout this work, *true* is considered equivalent to *derivable*.

variety of discourse we may call a member of a semirecursive language a *theorem* or a *true statement*, and a classical proof a *derivation*.

Finally, we also wish to recall the definition of a recursive language. Namely,

3. a language L is recursive (decidable) if and only if there exists a *(deciding)* Turing machine D, halting on all inputs, such that

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

Most of the semirecursive languages considered later on are actually recursive, in which case we may refer to their accepting algorithms as deciding algorithms.

Prior demands for efficiency: $\mathcal{NP}$, $\mathcal{IP}$, and $\mathcal{PCP}$. The two classical definitions of semirecursiveness are syntactically different, but they are not formally distinct from the point of view of "computational efficiency." Indeed, though in many natural cases verifying a classical proof is computationally preferable to finding it, classical proofs are more a way of expressing what is *in principle true* rather than a way of capturing what is *efficiently provable.* Providing a derivation is certainly a way to convince someone that a given theorem is true but not necessarily an efficient one: a classical proof may be arbitrarily long, or its relative verifying algorithm may take arbitrarily many computational steps to verify it.

Therefore, the now familiar notions of $\mathcal{NP}$ (due to Cook [16] and, independently, to Levin [24]) and $\mathcal{IP}$ (due Goldwasser, Micali, and Rackoff [21] and, independently, to Babai and Moran [4]) have been put forward in an effort to capture the essence of an efficient proof. Despite the notable differences between $\mathcal{NP}$ and $\mathcal{IP}$, in both cases this effort consists of *demanding that verifying be easy.*

Our notion too demands ease of verification (and in a *stronger* sense), but also broadens the perceived essence of an efficient proof by demanding some novel properties.

Another type of "proof efficiency" is provided by the notion of $\mathcal{PCP}$ [3, 17] (which we shall discuss in some detail later on). Quite succinctly, this notion consists of an explicit algorithm 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 polylogarithmic time by random accessing (at unit cost) selected bits of τ. This immediately yields the following $\mathcal{NP}$-like proof system: the prover transforms an $\mathcal{NP}$-witness σ into a longer but samplable proof-string τ, and sends τ to the verifier, who then will verify τ by selectively sampling its bits.

In terms of overall verifying time, however, such a proof system is not more efficient than its $\mathcal{NP}$ counterpart (i.e., than just sending σ). Indeed, though few chosen bits of τ will be "truly checked," to ensure that he is truly dealing with, say, the ith bit of τ, the verifier must *read/receive* every bit of τ and keep precise track of the order in which it is read/received. And such read/receiving operations, according to any natural measure, have in themselves a cost proportional to τ's length, which is greater than the length of σ.[3]

(Notice that having the prover not send τ at all, but rather having him answer any question the verifier may ask about specific bits of τ, does not work: a dishonest prover may cheat successfully with probability 1.)

Our demands for efficiency. Our notion of an efficient proof system is expressed in relation to Turing acceptability. We believe this to be a necessary step.

[3]This continues to be true if the prover sends to the verifier a piece of randomly-accessible hardware containing τ, if such a transmission is deemed preferable.

Indeed, we perceive *accepting* as the solitary process of determining what is true and *proving* as the social process of conveying to others the results of this determination.

Let us begin with the desideratum common to prior notions of our proof: ease of verification. Both $\mathcal{NP}$ and $\mathcal{IP}$ interpret ease of verification in *absolute terms*, namely, by requiring that verifiers run in polynomial-time. This interpretation automatically and strictly narrows the class of the efficiently provable theorems. By contrast, we believe that efficient verifiability should be expressed in *relative* rather than absolute terms, namely, by comparing the complexities of (1) *verifying* the proof of a given statement S and (2) *accepting* S (i.e., establishing S's verity without any help).

In addition, we perceive two new desiderata. First, because a proof system specifies (implicitly or explicitly) two processes, that of verifying and that of *proving*, we believe that proving too should be efficient, and this latter efficiency should again be relative to the complexity of accepting. Second, while $\mathcal{NP}$ and $\mathcal{IP}$ narrow the provable theorems to a small subset of all true statements (e.g., $\mathcal{PSPACE}$), we believe that *all* true statements (i.e., all Turing-acceptable languages) should be efficiently provable.

Our main goals. In sum, at the highest and informal level, the objective of our new notion of a proof is *finding the right relationship between* accepting*,* efficiently proving*, and* efficiently verifying *a true statement.* We articulate this general objective in the following goals.

1. *(Relative) efficiency of verifying.* Construct proof systems so that, for *all* theorems, the complexity of verifying is substantially smaller than that of accepting.
2. *(Relative) efficiency of proving.* Construct proof systems so that the prover's complexity is close to that of accepting.
3. *(Recursive) universality.* Construct proof systems capable of efficiently proving membership in *every* semirecursive language.

As we shall point out in what follows, our notion of a CS proof system also achieves additional goals, but we do not consider them essential to the "right" notion of an efficient proof.

2.1. Efficiency of verifying.

The relative nature of efficient verifiability. As outlined above, we regard a proof system to be efficient if it makes verifying a given statement easier than Turing accepting it (i.e., easier than establishing its verity without the help from any prover). Ignoring a small, fixed polynomial, we demand that the complexity of verifying be polylogarithmic in that of accepting. Though somewhat arbitrary, the latter choice stems from two simple reasons: "logarithmic" because we wish the advantage of verifying over accepting to be *substantial* (whenever the accepting time is substantial!), and "poly" because we wish such an advantage to be reasonably *independent* from any specific computational model.

The ubiquitous nature of efficient verifiability. There is an additional and novel aspect to our goal of efficient verifiability, namely, that such efficiency should arise for *all* theorems and not for just some of them. Let us explain this point focusing on the $\mathcal{NP}$ proof system.

To begin with, within $\mathcal{P}$, the $\mathcal{NP}$ mechanism does not guarantee that verification is computationally easier than acceptance (which in this case coincides with decision). For instance, in principle, for infinitely many positive constants c there could be a language L_c decidable in time $O(n^c)$ but for which any type of $\mathcal{NP}$-witness needs $\Omega(n^c)$ steps to be verified. If this were the case, the $\mathcal{NP}$ proof system could not make verifying membership in these languages any easier than deciding them.

In addition, in principle again, assuming $\mathcal{P} \neq \mathcal{NP}$ only entails the existence of a superpolynomial gap between the complexities of Turing accepting and verifying for some, *but possibly rare*, inputs, such as certain instances of satisfiability. By contrast, according to our present point of view, a proof system does not make verification sufficiently efficient unless it makes it polylogarithmically easier than accepting for essentially *all* theorems.

2.2. Efficiency of proving.

The relative nature of efficient provability. As mentioned already, 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 Turing accepting the problem at hand.

Measuring prover efficiency relative to the complexity of accepting 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 accepting is the process of convincing ourselves), the complexity of proving cannot be lower than that of accepting, while it could be much greater. We thus demand that our proof systems satisfy the following two properties.

(i) The prover must succeed in convincing the verifier whenever the theorem at hand is true (the old completeness property of an interactive proof system).

(ii) The amount of computation needed by the prover to convince the verifier must be polynomially close to that needed to accept that the given theorem is true.

In property (ii), we demand that the two amounts of computation be polynomially close to ensure a reasonable robustness.

We refer to the simultaneous holding of these two properties as *feasible completeness.* Feasible completeness is a novel *requirement* for proof systems. But do any of the prior proof systems "happen" to enjoy it anyway? Quite possibly, the answer is no. Consider, for instance, an $\mathcal{NP}$ language L (preferably not $\mathcal{NP}$-complete[4]) decidable by an algorithm D in, say, $n^{\log n}$ time. Then, in the $\mathcal{NP}$ 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 $|x|^{\log |x|}$ steps! Indeed, finding such an insightful string w_x might in the worst case require $O(2^{|x|})$ steps. In other words, while a few months of hard work may suffice for proving to ourselves (i.e., for accepting) that a given mathematical statement is true, it is conceivable that a lifetime may not be enough for finding an explanation followable by a verifier with a limited attention span.

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 general $\mathcal{IP} = \mathcal{PSPACE}$ protocol [26, 33], which is extremely wasteful of prover resources.

Realizing the importance of feasible completeness in a proof system allows us

[4]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 an $\mathcal{NP}$-complete language *appears* feasible. In fact, because of self-reducibility, if L is $\mathcal{NP}$-complete and decidable in $n^{\log n}$ time, then an $\mathcal{NP}$-witness of $x \in L$ is findable in $poly(|x|) \cdot |x|^{\log |x|}$ time. However, as we shall see in subsection 5.8, $\mathcal{NP}$ may not enjoy feasible completeness even when one focuses solely on $\mathcal{NP}$-complete languages.

to raise a variety of intriguing questions about $\mathcal{NP}$ and $\mathcal{IP}$.[5] But our point is not determining which proof systems *happen* to enjoy feasible completeness. Our point is that *feasible completeness must be* required *from any notion of a proof system that aims at achieving an adequate level of generality and meaningfulness.*

Related notions of feasible provability. Related notions of "feasible" provability have been considered in the past. In particular, Bellare and Goldwasser [6] discuss demonstrating membership in certain languages L by provers working in polynomial time and having access to an oracle for membership in L. Their notion, however, is weaker than ours because, in order to demonstrate membership in L of some given input x, their prover can query the oracle about other inputs x' for which accepting membership in L might be "much harder" than for x (despite the fact that such x' have been polynomial-time computed from x). Thus, if accessing the oracle for L were to be substituted with running an algorithm deciding L, then their provers may work much harder than needed for accepting that a specific x belongs to L.

Less relevantly, the protocols of [20] and [13] show that, if a prover were given for free an $\mathcal{NP}$ witness that an input x belongs to an $\mathcal{NP}$-complete language L, then proving in zero-knowledge that $x \in L$ only requires polynomial (in $|x|$) work. (In a sense, therefore, theirs is an example of feasible provability, but relative to the "nondeterministic complexity of x." That is, the prover complexity of a zero-knowledge proof system for $\mathcal{NP}$ is shown to be feasible relative to the prover complexity of another proof system: the $\mathcal{NP}$ one.) Such notion is nonetheless adequate when the prover is not handed the statement of a theorem (e.g., $x \in L$) as an input, but rather generates it together with suitable auxiliary information (e.g., an $\mathcal{NP}$-witness of $x \in L$) that enables feasible proving.

By contrast, our notion of feasible completeness refers to (1) *individual* inputs and (2) the *deterministic complexity* of these inputs.

2.3. Recursive universality. The previous proof systems discussed above 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, different interactive proof systems, or none. Moreover, as mentioned above, even considering the classes of all languages having an interactive proof system, one obtains a set of languages, $\mathcal{IP}$, that is quite small with respect to the set of all semirecursive languages.

We instead consider *universality* (i.e., the capability of handling the entire range of semirecursive languages) to be a necessary property of a "sufficiently right" proof system. By this we do not just mean that every semirecursive 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* semirecursive 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 will be seen, we consider an accepting algorithm for L to be a suitable description of L

[5]For instance, in an intuitive language,

$Q1$: *What is the computational complexity required from any $\mathcal{IP}$-prover of unsatisfiability?*

$Q2$: (In light of better-than-exhaustive-search algorithms for graph isomorphism) *What is the complexity required from any $\mathcal{NP}$-prover of, say, graph isomorphism?*

$Q3$: *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,

$Q3'$: *What is the computational complexity required from any $\mathcal{IP}$-prover of satisfiability?*

and one that facilitates our establishing the efficient provability and verifiability of a proof system.)

3. CS proofs with a random oracle.

To approximate our new goals, we put forward the notion of a CS proof system. As we shall see, this actually is a family of closely related notions. The first of such notions, that of a CS proof with a random oracle, will be presented and implemented in detail in this section; the others will be more briefly discussed in section 4. All these proof systems aim at proving membership in the following special language.

3.1. The CS language.

Encodings. Throughout this paper, we assume usage of a standard binary encoding, and often identify an object with its encoding. (In particular, if A is an algorithm, we may—meaningfully, if informally—give A as an input to another algorithm.) The length of an (encoded) object x is denoted by $|x|$. If q is a quadruple of binary strings, $q = (a, b, c, d)$, then our quadruple encoding is such that, for some positive constant c,

$$1 + |a| + |b| + |c| + |d| < |q| < c(1 + |a| + |b| + |c| + |d|).$$

Steps. If M is a Turing machine and x an input, we denote by $\#M(x)$ the number of steps that M takes on input x.

Definition 3.1. *We define the* CS language*, denoted by $\mathcal{L}$, to be the set of all quadruples $q = (M, x, y, t)$, such that M is (the description of) a Turing machine, x and y are a binary strings, and t a binary integer such that*

1. $|x|, |y| \leq t$;
2. $M(x) = y$; and
3. $\#M(x) = t$.

Notice that, as long as M reads each bit of its inputs and writes each bit of its outputs, the above property 1 is not a real restriction. Notice too that, due to our encoding, if $q = (M, x, y, t) \in \mathcal{L}$, then $t < 2^{|q|}$.

3.2. The notion of a CS proof-system with a random oracle.

Oracles and oracle-calling algorithms. We denote the set of all binary strings having length i by Σ^i, and the set of all functions from a-bit strings to b-bit strings by $\Sigma^a \to \Sigma^b$. By an *oracle* we mean a function in $\Sigma^a \to \Sigma^b$, for some choice of a and b.

We consider algorithms making calls to one or two oracles. To emphasize that an algorithm A makes calls to a single oracle, we write $A^{(\cdot)}$. If A is such an algorithm and f an oracle, we write A^f to denote the algorithm obtained by answering A's queries according to function f, that is, by answering each query α with $f(\alpha)$. Similarly, to emphasize that an algorithm A makes calls to two oracles, we write $A^{(\cdot,\cdot)}$. If A is such an algorithm and (f_1, f_2) a pair of oracles, we write $A^{(f_1,f_2)}$ to denote the algorithm obtained by answering A's queries to the first oracle according to function f_1, and those to the second oracle according to function f_2.

For complexity purposes, in a computation of an oracle-calling algorithm, the process of writing down a query and receiving its answer from the proper oracle f is counted as a single step. No result of this paper would change in any essential way if this process costed $poly(a, b)$ steps whenever $f \in \Sigma^a \to \Sigma^b$.

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

Integer presentation. If x is an integer given as an input to an algorithm A, unless otherwise clarified it is assumed that x is presented in binary to A. We only make an exception for the security paramenter, denoted by k, that is always presented in unary to all our algorithms: accordingly, we shall denote by 1^k (i.e., the concatenation of k 1s) the unary representation of integer k.[6]

Definition 3.2. *Let $P^{(\cdot)}$ and $V^{(\cdot)}$ be two oracle-calling Turing machines, the second of which is running in polynomial time. We say that (P, V) is a* CS proof system with a random oracle *if there exists a sequence of six positive constants, $c_1, \ldots, c_6$ (referred to as the* fundamental constants *of the system), such that the following two properties are satisfied.*

1′. *Feasible completeness.* For all $q = (M, x, y, t) \in \mathcal{L}$, for all k, for all $f \in \Sigma^{k^{c_1}} \rightarrow \Sigma^{k^{c_1}}$, $(1'.i)$ $P^f(q, 1^k)$ halts within $(|q|kt)^{c_2}$ computational steps, outputting a binary string $\mathcal{C}$ whose length is $\leq (|q|k)^{c_3}$, and $(1'.ii)$ $V^f(q, 1^k, \mathcal{C}) = YES$.
2′. *Computational soundness.* For all $\widetilde{q} \notin \mathcal{L}$, for all k such that $2^k > |q|^{c_4}$, and for all (cheating) deterministic $2^{c_5 k}$-call algorithm $\widetilde{P}$, for a random oracle $\rho \in \Sigma^{k^{c_1}} \rightarrow \Sigma^{k^{c_1}}$,

$$Prob_\rho[V^\rho(\widetilde{q}, 1^k, \widetilde{P}^\rho(\widetilde{q}, 1^k)) = YES] \leq 2^{-c_6 k}.$$

Thus, an execution of (P, V) requires a common oracle f and two common inputs: a quadruple of binary strings q (allegedly a member of $\mathcal{L}$) and a unary-presented integer k. We refer to q as *the CS input*, and to k as *the security parameter*. Such an execution consists of first running P^f on inputs q and 1^k, so as to produce a binary output $\mathcal{C}$, and then running V^f on inputs q, 1^k and $\mathcal{C}$. If $q = (M, x, y, t)$ and $V^f(q, 1^k, \mathcal{C}) = YES$, we may call string $\mathcal{C}$ a *random-oracle CS proof of $M(x) = y$*, or, more precisely, a *random-oracle CS proof, of security k, of $M(x) = y$ in less than t steps.* For variation of discourse, we may sometimes refer to such a $\mathcal{C}$ as a CS *witness* or a CS *certificate.* If it is clear from the context that we are dealing with CS proof systems with a random oracle, we may simplify our language by dropping the qualification "random-oracle."

Discussion.

Controlled inconsistency. CS proofs (similarly to zero-knowledge arguments discussed later on) allow the existence of false proofs but ensure that these are computationally hard to find. That is, *false CS proofs may exist, but they will "never" be found.*

Equivalently, CS proof systems are deliberately inconsistent but practically indistinguishable from consistent systems. 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 who produced them, though more powerful than their corresponding verifiers, are themselves computationally bounded.[7] From a practical point of view, this is hardly a limitation. As long we restrict our attention to physically implementable processes, no prover in our universe can perform $2^{1{,}000}$ steps of computation, at least during the existence of the human race. Thus, "practically speaking" all provers are computationally bounded.

[6]This is to ensure that a polynomial-time algorithm is guaranteed to be able to make "*poly(k)*" steps when the security parameter is k.

[7]The transition from an interactive proof system to a CS proof system is analogous to the transition from a perfect zero-knowledge proof system to a computational zero-knowledge proof system [21], which has proved to be a more flexible and powerful notion [20].

Deterministic cheating. In the above definition of a CS proof system with a random oracle, we have considered cheating provers $\widetilde{P}$ to be deterministic because probabilism does not help in our context. Indeed, since we are not concerned about the size of the description of $\widetilde{P}$, nor about its running time (except for the number of oracle calls it may make), $\widetilde{P}$ may easily have built in any "lucky" sequence of coin tosses.

Security parameters. Informally, the security parameter k controls the probability of something going wrong, and CS proofs become more meaningful as k grows. But, at a minimum, we require that k be large enough so that $2^k > |q|^{c_4}$. This "mild" lower-bound is relied upon in our proof of Theorem 3.8.[8] At the same time, it is also reasonable in that, on input a quadruple q, the honest prover P is allowed at least $poly(|q|)$ steps of computation, and it would thus be strange not to assume that a cheating prover can make a similar number of steps.

Running time. A member of $\mathcal{L}$, $q = (M, x, y, t)$, includes the exact number of steps, t, in which M outputs y on input x. More simply, however, we could have demanded that t upperbounds $\#M(x)$. But since the CS proof system with a random oracle of section 3.4.2, $(\mathcal{P}, \mathcal{V})$, actually proves the exact value of $\#M(x)$, it would have been a pity to loose this exact information.[9]

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 $\mathcal{EXPTIME}$ language to a verifier whose running time is upperbounded by a fixed polynomial in the input length alone. By contrast, the unbounded prover of an interactive proof system can only prove membership in $\mathcal{PSPACE}$ languages to a polynomial-time verifier, and it is widely believed that $\mathcal{PSPACE}$ is a proper subset of $\mathcal{EXPTIME}$.

However, at a second thought, there is no paradox. Indeed, a prover, being someone more powerful than us, may be a potentially useful *ally* (willing to enlarge our state of knowledge by letting us verify that some very difficult theorems are indeed true), but may also be a potentially dangerous *enemy* (wishing to trick us into believing some false statements). It is thus not too surprising that, when a prover is powerful but not-too-powerful, we can "trust him to a larger extent" and can thus "critically receive" from him more theorems than before.

Achieving our goals. Let us now point out how CS proofs achieve our main goals for the notion of an efficient proof.

1. *Efficient verifiability.* Our first goal required that, for every theorem, verifying should be polylogarithmically easier than accepting. This goal is approximated by a CS proof system (P, V) in the following sense. Let L be a semirecursive language, x a member of L, and A an accepting algorithm for L. Then the theorem $x \in L$ can be verified by running A on input x and verifying that $A(x) = YES$. Assume now that the latter computation takes t steps and that (P, V) is a CS proof system with a random oracle. Then, by choosing a proper security parameter k and running P on inputs

[8]Roughly, for proving Theorem 3.5 it suffices that $2^k > poly(|t|)$, which it is implied by $2^k > poly(|q|)$, because $|t| < |q|$.

[9]In any case, he who considers more natural choosing t as an upperbound to $\#M(x)$ may notice that with minor changes all of the results of this paper, including those about CS checking, remain true. In particular, a CS proof that some $q = (M, x, y, T)$ belongs to the modified CS language is computable in $poly(|q|k\#M(x))$ even when $\#M(x) << T$.

$q = (A, x, YES, \#A(x))$ and 1^k, and access to a random oracle ρ, one obtains a CS certificate of $x \in L$, $\mathcal{C}$ that (a) is length-bounded by a fixed polynomial in $|q|$ and k, and (b) is accepted by V^ρ running on additional inputs q and 1^k. Therefore, because V is polynomial time (and because k is unary-presented), V accepts within a number of steps that are bounded by a fixed polynomial in $|q|$ and k. Thus, because $\#A(x)$ enters q in binary representation and because of our quadruple-encoding conventions, V accepts in time polynomial in $|x|$ and k (as well as in $|A|$) but polylogarithmic in $\#A(x)$.

2. *Efficient provability.* Our second goal called for the complexity of proving being polynomially close to that of accepting. This property is immediately guaranteed by the feasible completeness of a CS proof system. Feasible completeness in fact states that there exists a fixed constant c_2 such that, if an algorithm A accepts that a string x belongs to a semirecursive language L (in $\#A(x)$ steps), then a CS prover can, on inputs $q = (A, x, YES, \#A(x))$ and 1^k and with access to a random oracle, find a CS proof of $x \in L$ within $(|q|k\#A(x))^{c_2}$ steps. That is, a CS prover can find a CS proof of $x \in L$ in a time that is polynomial in $\#A(x)$ and k (as well as in $|A|$ and $|x|$).
3. *Recursive universality.* Our third goal called for proof systems capable of proving membership in all possible semirecursive languages. In apparent contrast with this requirement, a CS proof system is defined to prove membership only in the CS language $\mathcal{L}$. But $\mathcal{L}$ is designed so as to encode membership questions relative to any possible semirecursive language. In fact, to each semirecursive language L corresponds a 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 $(M_L, x, YES, t) \in \mathcal{L}$, thus achieving the third goal.

Efficient verifiability within $\mathcal{P}$. Note that a CS proof system with a random oracle makes verifying computationally preferable to accepting even for polynomial-time languages.

The process behind the curtains. In a classical proof system, what convinces us of the verity of a given statement is the existence of a string satisfying a proper syntactic property. By contrast, in an interactive proof system there are no strings that do the convincing: the "proof is in the process." Differently from both scenarios, in a noninteractive CS proof system, proofs are strings possessing a special property, but such strings may exist also for false statements. Therefore, what is convincing is not the existence of such strings but our belief that the *process* behind the generation/selection of such strings had computationally limited resources.[10]

Comparison with zero-knowledge arguments. Goldwasser, Micali, and Rackoff [21] introduced and first exemplified the notion of a zero-knowledge proof system, and Goldreich, Micali, and Wigderson[20] showed that all languages in $\mathcal{NP}$ possess

[10]Indeed, when debating whether a given statement is true, we do not have "serendipitous" access to some CS proof of it, if any. Thus, if we are given such a string, then there must have been an active process that generated/selected it. For instance, assume that, while walking on a beach pondering our favorite statement S, we encounter a sand pattern that looks like the sequence of bits of a CS proof, σ, of S. Then, we may consider that the grains of sand have been arranged in such a σ-shape by natural elements (such as wind, waves, and sun), and view the universe as a computer and its age as computing time, so that, in a final analysis, our 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.

a (computationally) zero-knowledge proof, under a general complexity assumption.[11] Brassard, Chaum, and Crépeau [13] then put forward a related notion, that of a *zero-knowledge argument for $\mathcal{NP}$*, and proved a related theorem: under a specific complexity assumption, there exist zero-knowledge arguments for all $\mathcal{NP}$ languages.[12]

We wish to disregard the zero-knowledge component of the latter protocols and focus instead on their proof-system component, which we call the *argument (proper)*. Such arguments in fact provide an earlier example of proof systems which (as our CS proofs) are "convincing if its provers are *computationally bounded*." Let us explain.

In a (zero-knowledge) argument system for an $\mathcal{NP}$ language L, all agents, the prover P, the verifier V, and any possible malicious prover $\widetilde{P}$, are assumed to be polynomial-time machines. Before proving that a given input x belongs to L, P is *assumed* to have available, on a special tape inaccessible by V, an $\mathcal{NP}$-witness of $x \in L$, w. (Without this assumption, such a w might be uncomputable by the polynomial-time P.) During the protocol, P is provided by V with a special encryption scheme and uses it so as to convince V of the existence of w (without revealing it) by means of an interactive process that is less efficient than merely sending w. Vice versa, if $x \notin L$, no such w exists, and it is hard for a malicious $\widetilde{P}$ to convince V of the opposite. In fact, succeeding in such a malicious convincing entails "breaking" the provided encryption scheme, and the chance that a polynomial-time $\widetilde{P}$ may do that is quite remote.

Arguments, therefore, do not enlarge (nor aim at enlarging) the class of theorems that are efficiently provable. Rather, they constitute an alternative way of proving membership in $\mathcal{NP}$, a way that is less efficient than simply providing the witness but satisfies an additional property, *zero-knowledgeness* (which is in fact an integral part of their very definition).

Note that, leaving aside zero-knowledgeness and interaction (there are, after all, interactive CS proof systems), our CS proofs with a random oracle differ from the arguments of Brassard, Chaum, and Crépeau in the following ways.

- *Their arguments for $\mathcal{NP}$ may not enjoy efficient verifiability.* As discussed above, such $\mathcal{NP}$ arguments are less efficient than classical $\mathcal{NP}$ proof systems, and, as pointed out in subsection 2.1, the latter systems may not satisfy (ubiquitous efficiency, and thus) efficient verifiability.
- *Their arguments for $\mathcal{NP}$ may not enjoy efficient provability.* As discussed above, on input a member x of an $\mathcal{NP}$ language L, the prover of an $\mathcal{NP}$ argument is assumed to have "for free" (as an additional input) an $\mathcal{NP}$-witness w of $x \in L$. But, as we have pointed out in subsection 2.2, the time necessary for a prover to find such a witness w may vastly exceed that necessary to accept that $x \in L$.
- *Their arguments for $\mathcal{NP}$ do not enjoy recursive universality.* $\mathcal{NP}$ languages are a proper subset of all recursive languages.

3.3. The intuition behind our CS proof-system with a random oracle. Our construction is based on an earlier one of Kilian's [23], which is itself based on

[11] Informally, they exhibit interactive protocols enabling a prover to convince a polynomial-time verifier that an input belongs to an $\mathcal{NP}$ language L, but without conveying any more knowledge than the mere fact that a given witness of such a membership exists. Their protocols privilege the "proof aspect" rather than the "zero-knowledge aspect". Indeed, even if endowed with unbounded computational power, their provers cannot convince their verifiers that inputs outside L are in L (but with a negligible probability). However, for their verifiers to gain no information about inputs of L (other than their belonging to L) it is crucial that they be time-bounded.

[12] Differently from Goldreich, Micali, and Rackoff, rather then the proof aspect, they privilege the zero-knowledge aspect (see previous footnote).

Merkle's trees [27] and probabilistically checkable proofs [3, 17]. Let us thus start by recalling the latter two notions.

3.3.1. Probabilistically checkable proofs. Babai, Fortnow, Levin, and Szegedy [3], and Feige, Goldwasser, Lovasz, Safra, and Szegedy [17] have put forward, independently and with different aims,[13] some related and important ideas sharing a common technique: *proof-samplability.*[14] In essence, they present two algorithms. The first transforms an $\mathcal{NP}$-witness, w, into a slightly longer "samplable proof," w'. The second algorithm can check the correctness of such a string w' by "random" accessing (i.e., accessing at unit cost) selected few of its bits. In our paper, we refer to these two algorithms as the (sampling-enabling) prover and the (sampling) verifier, which we, respectively and consistently, denote by SP and SV, so as to differentiate them from other types of provers and verifiers.

The following version of their result has proved useful in most applications so far.

THEOREM 3.3 (samplable proofs: Version 0). *For all $\mathcal{NP}$ languages L there exist two polynomial-time algorithms, a deterministic SP, a probabilistic SV, and a polynomial Q, such that*

(a) *For all n-bit strings $x \in L$ and for all $\mathcal{NP}$-witness w of $x \in L$, $SP(x,w) = w'$, wherein string w' is such that, on input x and access to any $Q(\log n)$ bits of w' of its choice, algorithm SV accepts; and*

(b) *for all $x \notin L$ and for all w', algorithm SV, on input x and access to any $Q(\log n)$ bits of w' of its choice, rejects with probability $\geq 1/2$.*

We shall, however, rely on a more precise and general version of their result, namely the following theorem.

THEOREM 3.4 (samplable proofs: Version 1). *For any polynomial-time relation R over $\Sigma^* \times \Sigma^*$, there exist a deterministic polynomial-time algorithm $SP(\cdot,\cdot)$, a probabilistic polynomial-time algorithm $SV(\cdot,\cdot)$, and two polynomials $L(\cdot)$ and $\Lambda(\cdot)$, such that the following hold.*

1. *For all strings x and y such that $R(x,y)$ holds, $SP(x,y)$ outputs a string y' such that*
   1.1. *$y' < L(|x|+|y|)$, and*
   1.2. *$SV(x,|y'|)$, having a random tape of length $\Lambda(\log|y'|)$ and random access to y', accepts.*
2. *For all strings x such that for all y $R(x,y)=0$, and for any string σ, the probability (computed over SV's coin tosses) that $SV(x,|\sigma|)$, having a random tape of length $\Lambda(\log|\sigma|)$, having random access to σ, and actually accessing $\Lambda(\log|\sigma|)$ bits of σ, accepts is $\leq 1/2$.*

In the statement of Theorem 3.4, as customary, inputs $|y'|$ and $|\sigma|$ are presented to SV in binary. (Thus the second input of SV is polylogarithmically shorter than $|x|+|y|$.) Note that, unlike in the case of $\mathcal{NP}$, $R(x,y)=1$ may not imply any a priori

[13]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 [17] 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 the fact that inputs appear in any special format.)

[14]Though improved in [2, 1, 35, 29], the original proof-samplability techniques of [3] and [17] suffice for our purposes.

bound for the length of y relative to that of x.

3.3.2. Merkle's trees. Recall that a binary tree is a tree in which every node has at most two children, hereafter called the 0-*child* and the 1-*child*. A *collision-free hash function* is, informally speaking, a polynomial-time computable function H mapping binary strings of arbitrary length into reasonably short strings, so that it is computationally infeasible to find any *collision (for H)*, that is, any two different strings x and y for which $H(x) = H(y)$. (Popular candidate collision-free hash function is the standardized *secure hash function* [32] and Rivest's MD4 [31].)

A *Merkle tree* [27] then is a binary tree whose nodes store (i.e., are associated to) values, some of which are computed by means of a collision-free 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.[15] Thus, if the collision-free hash function produces k-bit outputs, 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, and a 1-value otherwise.

The crucial property of a Merkle tree is that, unless one succeeds in finding a collision for H, *it is computationally hard to change any value in the tree (and, in particular, a value stored in a leaf node) without also changing 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 root value rv "implicitly defines" what the n original values were. Assume in fact that, as some point in time, A gives rv, but not the original values, to another party B. 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 then 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), $Y_1, \ldots, Y_d$. Party B verifies the received alleged leaf-value v_i and the received alleged authentication path $Y_1, \ldots, Y_d$ as follows. She sets $X_1 = v_i$ and, letting $i_1, \ldots, i_d$ be the binary expansion of i, computes the values $X_2, \ldots, X_d$ as follows: if $i_j = 0$, she sets $X_{j+1} = H(Y_j X_j)$; otherwise, she sets $X_{j+1} = H(X_j Y_j)$. Finally, B checks whether the computed k-bit value X_d equals rv.

3.3.3. Kilian's construction. In [23], Kilian presents a special zero-knowledge argument for $\mathcal{NP}$, (P, V), exhibiting a polylogarithmic amount of communication, where prover P uses a Merkle tree in order to provide to V "virtual access" to a samplable proof.[16]

In essence, disregarding zero-knowledge aspects, the polynomial-time prover P, as in any zero-knowledge argument, possesses a polynomially long witness, w, proving

[15] 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 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.

[16] Essentially the same construction (minus its zero-knowledge aspects) was independently discovered by the author and privately comunicated to Shafi Goldwasser prior to Kilian's publication that same year. (It was not, however, written up or circulated until after Kilian's publication, and then only in the context of a broader notion of an efficient proof.)

that a given input x belongs to a given $\mathcal{NP}$-language L. In virtue of Theorem 3.3, P then transforms w into a longer, but still polynomially long in the length of x, "samplable proof" w' by running on inputs x and w the algorithm SP of Theorem 3.3. In order to yield more efficient verifiability, 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-bit 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 set to be the 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 the algorithm SV of Theorem 3.3. 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 polylogarithmic (in n) number of bits of w', and because each such a virtual access is answered by $k\,poly(\log n)$ bits of authentication path, the overall amount of communication is polylogarithmic 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.

3.3.4. Our modifications. Like all prior argument systems, Kilian's is not a CS proof system (nor even an interactive one, as defined later on). To begin with, it only proves membership in $\mathcal{NP}$ languages and thus does not satisfy recursive universality. Further, even relative to the $\mathcal{NP}$ languages, it may not satisfy feasible completeness. Indeed, in his construction, in order to convince verifier V that x belongs to an $\mathcal{NP}$ language L, prover P needs an $\mathcal{NP}$-witness, w, of $x \in L$. But, again, the time necessary to compute w on input x may vastly exceed that necessary to accept (in this case, *decide*) that $x \in L$ (in a way that does not produce an $\mathcal{NP}$-witness).

We do, however, obtain a CS proof system with a random oracle, $(\mathcal{P}, \mathcal{V})$, by modifying his argument system. First, as a necessary step towards recursive universality, we assume that an input to $(\mathcal{P}, \mathcal{V})$ consists of a member of the CS language $\mathcal{L}$, (M, x, y, t).[17] On such a CS input, $(\mathcal{P}, \mathcal{V})$ 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 a computation of $M(x)$ in which string y is produced as an output. Such a history σ is then thought of as a proof that $M(x) = y$. This proof will not be insightful, and, because no restriction is put on M, can be arbitrarily long relative to x. (Notice that $\mathcal{P}$'s computation so far satisfies, by definition, feasible completeness.)

Next, $\mathcal{P}$ will put such a proof σ in samplable form. Consider in fact the following relation R.

[17] Again, in order to prove membership in a given semirecursive language L, M will then be a Turing machine accepting L, and y will be the special string YES.

$R(q, \sigma) = 1$ *if and only if* σ *is the* t*-step history of a computation of* M *outputting* y *on input* x.

Then, notice that R is $poly(|q|, |\sigma|)$-time computable. Thus, due to Theorem 3.4 proof σ can be put in a probabilistically checkable form τ by algorithm SP within $poly(|q|)$ —and thus $poly(t)$— steps. Given random access to the so obtained τ, algorithm SV, on inputs q and τ, then efficiently checks its correctness using polylog$(|\tau|)$ queries and a random tape, RT, whose length is $polylog(|\tau|)$. (Notice that also this second piece of $\mathcal{P}$'s computation satisfies feasible completeness.)

Though proving that "$M(x) = y$ in t steps," such τ is again too long. Thus, $\mathcal{P}$ "Merkle hashes" τ as in [23] using a collision-free hash function H and gives $\mathcal{V}$ only virtual access to it (something that still preserves feasible completeness). The verifier is thus guaranteed that he is properly accessing τ (i.e., that $\mathcal{P}$ is not choosing on-line the bits of τ based on the bit-locations that $\mathcal{V}$ wishes to access) *provided* that $\mathcal{P}$ is computationally incapable of finding a collision in H.[18] To provide such a "guarantee," for a *specific* input $q = (M, x, y, t) \in \mathcal{L}$, it is possible to choose the security parameter k big enough so that finding a collision in a k-bit-output H requires a number of steps enormously bigger than those required above from $\mathcal{P}$ on the input q at hand, but not too big so as to violate feasible completeness.[19] In sum, therefore, we propose to keep honest a prover working on a given *individual* problem by means of another much harder *individual* problem, that of finding a collision for H (though the latter problem may belong to a "much lower" complexity class than the first one[20]).

So far, our $(\mathcal{P}, \mathcal{V})$ satisfies both recursive universality and feasible completeness but still is interactive. Indeed, recall that, to give $\mathcal{V}$ virtual access to τ, $\mathcal{P}$ uses function H to Merkle-hash the samplable proof τ and sends $\mathcal{V}$ the resulting root value RV. In response, $\mathcal{V}$ runs the sampling verifier SV with a random tape RT. During this execution SV computes which bits of τ it wishes to see; CS verifier $\mathcal{V}$ then sends these requests to CS prover $\mathcal{P}$; and prover $\mathcal{P}$ replies with both the requested bits and their authentication paths relative to RV. Because RT is genuinely random, if the input $(M, x, y, t) \notin \mathcal{L}$ and if $\mathcal{V}$ interacts with a malicious prover $\widetilde{\mathcal{P}}$ that does not succeed in finding a collision for H, then SV (and thus $\mathcal{V}$) accepts with probability at most $1/2$.

Let us now introduce further modifications in order to dispense with any interaction between $\mathcal{P}$ and $\mathcal{V}$ during the proving process. We first decrease the probability of SV accepting a false statement to less than 2^{-k} by repeating the above process k times, each time using an independently-selected random tape RT. We then use a random oracle, as follows, to retain more or less this same probability, while eliminating any interaction between $\mathcal{P}$ and $\mathcal{V}$.

In some sense, we have the CS prover $\mathcal{P}$ "choose" the k random tapes of SV, so that $\mathcal{V}$ is no longer needed. In fact, given these tapes, $\mathcal{V}$ runs deterministically, and

[18] In Kilian's case such guarantee stemmed from the fact that the prover was polynomial-time, while collision finding is assumed not to be (and to enable him to prove membership in $\mathcal{NP}$-complete languages it was assumed that he had access to an $\mathcal{NP}$ witness for free). In our case, however, $\mathcal{P}$ cannot be assumed to be polynomial-time, because it ought to be able to run M on x for t steps for all possible $(M, x, y, t) \in \mathcal{L}$, and t may vastly exceed $|(M, x, y, t)|$.

[19] Assume, for instance, that the complexity of finding an H-collision for a k-bit output H is $\Omega(2^{k^d})$ for some constant d between 0 and 1. Then, because the honest prover works in time polynomial in t, setting $k = (\log t)^{2/d}$ seems a reasonable choice. This choice in fact increases only by a $poly((\log t)^{2/d})$ factor the amount of work of the honest prover but forces any malicious prover to work in time $2^{(\log t)^2}$.

[20] Accordingly, we view a prover working on a given input as an *individual* device, endowed with a fixed amount of computational resources, rather than a mechanism capable of handling all members of a given complexity class.

thus a prover $\mathcal{P}$ who knows them can simulate perfectly $\mathcal{V}$'s actions (and in particular compute the bit locations of τ that SV wishes to see). Of course, however, this a dangerous way of proceeding. In fact, it is already dangerous having a malicious CS prover $\widetilde{\mathcal{P}}$ simply *know* these k random tapes (let alone choose them), or even just predict the bit locations that SV wishes to see in its k runs. In fact:

> Letting λ denote the number of bit-locations of τ SV wishes to access in a single run, the total number of such bit-locations will be $k\lambda$. Now, if $k\lambda$ is sufficiently small with respect to the total number of bits in τ (which we would like to be our case in order to satisfy efficient verifiability), it is not hard to see that if $\widetilde{\mathcal{P}}$ knows in advance these $k\lambda$ bit-locations, then he could provide (1) a root value RV for the Merkle tree, (2) bit values for said locations, and (3) authetication paths for these values relative to RV, so as to cheat $\mathcal{V}$ with probability 1.

Notice too, however, that, if the k random tapes are selected *after* $\widetilde{\mathcal{P}}$ provides the root value RV, then, roughly said, unless he succeeds in finding at least one collision for H, his probability of cheating still is $< 2^{-k}$. (This continues to be true even if $\widetilde{\mathcal{P}}$ knows 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 $(M, x, y, t) \in \mathcal{L}$, prover $\mathcal{P}$, as before, (a) computes a classical proof of it by running M on input x and costructing a history σ of such computation, (b) puts σ in a samplable form τ, and (c) stores τ in the leaves of a suitable binary tree and constructs a corresponding Merkle tree, using a collision-free hash function H, so as to compute a root value RV. At this point, $\mathcal{P}$ uses the random oracle 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. Then, it outputs, as a CS proof with a random oracle for $(M, x, y, t) \in \mathcal{L}$, 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.

The intuition that this strategy works is quite strong. Consider 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. (Despite this simple and strong intuition, however, formally proving that this strategy works appears to be more difficult.)

Our strategy is reminiscent of a step used by Fiat and Shamir [18]. 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 collision-free hash function on the first party's message. (By now, similar strategies have been discussed in the literature in many a context.)

As a final modification, in lieu of k-bit-output collision-free hash function H, we construct our Merkle tree using a random oracle mapping $2k$-bit strings to k-bit ones. Indeed, finding collisions for random oracles is provably hard in a precisely quantifiable way, and adoption of such oracles also for this task dispenses us for relying on additional complexity assumptions (i.e., the existence of a collision-free H).

Note that the random oracle used for removing the interaction between prover and verifier and that used for building the Merkle tree had better be different. Alternatively, using standard techniques, one may use a single random oracle to "extract" two independent ones: one for each of these two tasks.

3.4. Description of $(\mathcal{P}, \mathcal{V})$: Our CS proof-system with a random oracle.

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

3.4.1. Preliminaries.

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 exhibit a CS proof system with a random oracle $(\mathcal{P}, \mathcal{V})$ by having $\mathcal{P}$ 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 holds 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, it suffices to 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 $f_j(y)$.[21]

From k runs to one run: Sampling proof systems and their length bounds. Rather than having the probability of successful cheating be less than 1/2, let us restate Theorem 3.4 so as to reduce this probability to 2^{-k} by means of a sampling verifier that (at least formally) still uses a single random tape but receives an additional, independent security parameter.

THEOREM 3.5 (samplable proofs: Version 2). *There exists a deterministic polynomial-time algorithm $SP(\cdot,\cdot)$, a probabilistic polynomial-time algorithm $SV(\cdot,\cdot,\cdot)$, and two polynomials $L(\cdot)$ and $\Lambda(\cdot)$ such that, for any polynomial-time relation R over $\Sigma^* \times \Sigma^*$, the following two properties hold.*

1. *For all strings x and y such that $R(x, y)$ holds, $SP(x, y)$ outputs a string y' such that*
   1.1. *$y' < L(|x| + |y|)$, and*
   1.2. *for every security parameter k, $SV(x, |y'|, 1^k)$, having a random tape of length $k \cdot \Lambda(\log |y'|)$ and random access to y', accepts.*
2. *For all strings x such that for all y $R(x, y) = 0$ and for any string σ, the probability (computed over SV's coin tosses) that $SV(x, |\sigma|, 1^k)$, having a random tape of length $k \cdot \Lambda(\log |\sigma|)$, having random access to σ, and actually accessing $k \cdot \Lambda(\log |\sigma|)$ bits of σ, accepts is $\leq 2^{-k}$.*

The CS-history relation. In what follows we shall use Theorem 3.5 only for a specific relation $\mathcal{H}$, the *CS-history relation*, defined as follows.

[21]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 $\{0,1\}^{1+max(a_1+a_2)}$ into $\{0,1\}^{max(b_1+b_2)}$ allows us to achieve our goal quite straightforwardly.

$\mathcal{H}(q,h) = 1$ if and only if string $q = (M, x, y, t) \in \mathcal{L}$ and string h is an encoding of the history of the execution of M on input x.

Notice that, assuming the use of a proper encoding for these histories, not only is $\mathcal{H}$ polynomial-time computable, but there also is a fixed polynomial Q such that $\mathcal{H}(q,h) = 1$ implies $|h| \leq Q(2^{|q|})$. (In fact, our quadruple conventions imply that $|t| < |q|$, and thus that $t < 2^{|q|}$.) Let us now use the CS-history relation to restate Theorem 3.5 in the following form more directly useful to us.

THEOREM 3.6 (samplable proofs: Version 3). *There exists a deterministic polynomial-time algorithm $SP(\cdot,\cdot)$, a probabilistic polynomial-time algorithm $SV(\cdot,\cdot)$, and two polynomials $\ell(\cdot)$ and $\lambda(\cdot)$ such that, letting $\mathcal{H}$ be the history relation, the following two properties hold.*

1. *For all strings q and h such that $\mathcal{H}(q,h) = 1$, $SP(q,h)$ halts within $\ell(|q|)$ steps outputting a string h' such that*
   1.1. $\log|h'| < \ell(|q|)$*, and*
   1.2. *for every security parameter k, $SV(q, |h'|, 1^k)$, having a random tape of length $k \cdot \lambda(|q|)$ and random access to h', accepts.*
2. *For all strings q such that for all h $\mathcal{H}(x,h) = 0$, for any security parameter k, and for any string σ, the probability (computed over SV's coin tosses) that $SV(q, |\sigma|, 1^k)$, having a random tape of length $k \cdot \lambda(|q|)$, having random access to σ, and actually accessing $k \cdot \lambda(|q|)$ bits of σ, accepts is $\leq 2^{-k}$.*

DEFINITION 3.7. *Let SP, SV, ℓ, and λ be as in Theorem 3.6. Then, we shall refer to (SP, SV) as a* sampling proof system (for the CS-history relation)*, and to ℓ and λ as its* length bounds *(respectively, for the samplable proof produced by SP and the number of queries and length of the random tape used by SV).*

Notation.

- *Basics.* 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 $\bar{b}$.
- *Strings.* If α is a binary string, then $|\alpha|$ denotes α's length; $\alpha_1 \cdots \alpha_i$ denotes α's i-bit prefix; and $\alpha_1 \cdots \bar{\alpha}_i$ denotes α's i-bit prefix with the last bit complemented.
- *Labeled trees.* If N is a power of two, we let $\mathcal{T}_N$ denote the complete binary tree with N leaves, whose vertices are labeled by binary strings whose lengths range from 0 to $\log N$ as follows. Vertex v_ε is the root, v_0 and v_1 are, respectively, its the left and right child, and, more generally, for all $i \in [0, \log N)$ and for all $\alpha \in \Sigma^i$, $v_{\alpha 0}$ and $v_{\alpha 1}$ are, respectively, the left and right child of node v_α. (Consequently, $v_{\alpha_1 \cdots \alpha_j}$ and $v_{\alpha_1 \cdots \bar{\alpha}_j}$ 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 $(\log_2 N)$-bit binary representation of integer j, with possible leading 0s. (Accordingly, the jth leaf of $\mathcal{T}_N$ is node $v_{[j]}$.)

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

Let us now describe two oracle-calling algorithms, $\mathcal{P}$ and $\mathcal{V}$, and then prove that they are, respectively, the prover and verifier of a CS proof system with a random oracle, $(\mathcal{P}, \mathcal{V})$.

Common inputs: $q = (M, x, y, t)$, an n-bit (alleged) member of $\mathcal{L}$, and 1^k, a security parameter.

{*Comment:* For the purpose of the code of (honest) prover $\mathcal{P}$, $q = (M, x, y, t) \in \mathcal{L}$ and thus $t = \#M(x)$.}

Common subroutines: (SP, SV), a sampling proof system—as per Definition 3.7—with length bounds ℓ and λ.

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

{*Comment:* The first oracle, when randomly selected, is used as a collision-free hash function of a Merkle tree, by which $\mathcal{P}$ commits to a samplable proof of $q \in \mathcal{L}$. The second oracle, when randomly selected, is used to generate the random tape of sampling verifier SV.}

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

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

```
ALGORITHM $\mathcal{P}$
  $P1$. (Commit to a samplable proof of $q \in \mathcal{L}$.)
    $P1.1$ (Find a proof, denoted by $\sigma$, of $x \in \mathcal{L}$.)
        Run machine $M$ on input $x$ so as to output $y$ in $\#M(x)$ steps and
        generate an encoding, $\sigma$, of $M$'s computational history.
        {Comment: $\sigma$ can be considered a proof that $x \in \mathcal{L}$.}
    $P1.2$ (Compute a samplable form, denoted by $\tau$, of proof $\sigma$.)
        $\tau \leftarrow SP(q, \sigma)$.
        {Comment: Theorem 3.6, our quadruple encoding, and $n = |q|$ imply
        $|\tau| \le \ell(n)$.}
    $P1.3$ (Commit to $\tau$ by means of a $k$-bit value $R_\varepsilon$.)
        Assume, for simplicity only, that $|\tau|/k = N$, where $N$ is an integral
        power of 2. Then, we shall associate to (figuratively speaking, "store
        in") each node $v_\alpha$ of a labeled tree $\mathcal{T}_N$ a value $R_\alpha$ computed as follows.
        Subdivide $\tau$ into the concatenation of $N$ substrings, each $k$-bit long,
        $\tau = \tau_1 \cdots \tau_N$, and for $0 \le j < N$, assign to the $j$th leaf, $v_{[j]}$, the $k$-bit
        value
(3.1)                         $R_{[j]} = \tau_j$.
        Then, in a bottom-up fashion, assign to each interior node $v_\alpha$ of $\mathcal{T}_N$ the
        $k$-bit value
(3.2)                         $R_\alpha = f_1(R_{\alpha 0} | R_{\alpha 1})$.
        {Comment: $R_\varepsilon$ thus is the $k$-bit value assigned to the root of $\mathcal{T}_N$. $R_\varepsilon$ is
        considered a commitment to all values stored in the vertices of $\mathcal{T}_N$ and
        thus a comitment to all of $\tau$.}
  $P2$. (Build a CS certificate, $\mathcal{C}$, of $q \in \mathcal{L}$.)
    $P2.1$ (Start building the CS certificate with the $k$-bit commitment $R_\varepsilon$ as prefix.)
        $\mathcal{C} \leftarrow R_\varepsilon$.
    $P2.2$ (Choose a random tape, $T$, for $SV$.)
        $T \leftarrow f_2(q\,|R_\varepsilon)$.
    $P2.3$ (Run $SV$ with random tape $T$ and virtual access to $\tau$.)
        Run $SV$ with random tape $T$, inputs $q$ and $|\tau|$, and virtual access to $\tau$.
        Whenever $SV$ wishes to access bit-location $i$ of $\tau$, perform the following
        instructions.
   $P2.3.1$ (Find the index, $I$, of the substring of $\tau$ containing $b_i$.)
        $I \leftarrow \lceil i/k \rceil$;
   $P2.3.2$ (Add leaf $I$ to the CS certificate.)
           $\mathcal{C} \leftarrow \mathcal{C}|R_{[I]}$; and
   $P2.3.3$ (Add to the certificate the authentication path of leaf $I$.)
           Set $\alpha = [I]$;
           {*Comment:* $|\alpha| = \log_2 N$.}
        Set $AUTHPATH_I = R_{\alpha_1\cdots\bar{\alpha}_{\log N}}|\cdots|R_{\alpha_1\bar{\alpha}_2}|R_{\bar{\alpha}_1}$;
        {*Recall:* The authentication path of leaf $I$ consists of the values stored
        in the siblings of the vertices of the path from leaf $I$ to the root.}
        $\mathcal{C} \leftarrow \mathcal{C}|AUTHPATH_I$.
        {*Example:* if $N = 8$ and $I = 3$, then $[I] = 011$ and $AUTHPATH_I =$
        $(R_{010}, R_{00}, R_1)$.}
  $P3$. (Output a certificate for $q \in \mathcal{L}$.)
   Output $\mathcal{C}$.
   {*Comment:* $\mathcal{C}$'s $k$-bit prefix is $R_\varepsilon$.}
  ALGORITHM $\mathcal{V}$
  $V1$. (Read and delete $R_\varepsilon$ from certificate $\mathcal{C}$, and compute $SV$'s random tape $T$.)
   $ALLEGEDROOT \leftarrow \mathcal{C}_1\cdots\mathcal{C}_k$;
   $\mathcal{C} \leftarrow \mathcal{C}_{k+1}\cdots$; and
   $T \leftarrow f_2(q|R_\varepsilon)$.
  $V2$. (Run $SV$ with random tape $T$, inputs $q$, $\ell(n)$ and $1^k$, and virtual access to
   samplable proof $\tau$.)
   Execute $SV(q, \ell(n), 1^k)$ with random tape $T$. Whenever $SV$ wishes to access
   bit-location $i$ of the samplable proof, do the following.
    $V2.1$ (Find the index, $I$, of the $k$-bit segment of $\tau$ containing $b_i$, and read the
        value of leaf $I$ from the certificate.)
        $I \leftarrow \lceil i/k \rceil$; $\alpha \leftarrow [I]$; and $R_\alpha \leftarrow \mathcal{C}_1\cdots\mathcal{C}_k$.
        Provide $SV$ with the $(i - kI)$th bit of $R_\alpha$.
    $V2.2$ (Delete the value of leaf $I$ from the certificate.) $\mathcal{C} \leftarrow \mathcal{C}_{k+1}\cdots$.
    $V2.3$ (Check and remove from the certificate the authentication path of leaf $I$.)
        For $m = 1$ to $\log N$,
            $R_{\alpha_1\cdots\bar{\alpha}_m} \leftarrow \mathcal{C}_1\cdots\mathcal{C}_k$ and
            $\mathcal{C} \leftarrow \mathcal{C}_{k+1}\ldots$.
        For $m = \log N, \ldots, 1$, compute $R_{\alpha_1\cdots\alpha_{m-1}}$ as follows:

            $R_{\alpha_1\cdots\alpha_{m-1}} \leftarrow \begin{cases} f_1(R_{\alpha_1\cdots\alpha_m}|R_{\alpha_1\cdots\bar{\alpha_m}}) & \text{if } \alpha_m = 1, \\ f_1(R_{\alpha_1\cdots\bar{\alpha_m}}|R_{\alpha_1\cdots\alpha_m}) & \text{if } \alpha_m = 0 \end{cases}$

        and check whether the computed value $R_\varepsilon$ equals the value
        $ALLEGEDROOT$.
        {*Example:* If $N = 8$ and $I = 3$, then $[I] = 011$ and the verifier computes

            $R_{01} = f_1(R_{010}|R_{011})$,
            $R_0 = f_1(R_{00}|R_{01})$, and
            $R_\varepsilon = f_1(R_0|R_1)$,

        where values $R_{011}$, $R_{010}$, $R_{00}$, and $R_1$ are retrieved from $\mathcal{C}$.}
  $V3$. (Accept if and only if $SV$ accepts and the authentication path of each leaf is
    correct.)
    If $SV$ accepts and each $V2.3$ check is passed, output $YES$. Otherwise, output
    $NO$.
```

3.5. $(\mathcal{P},\mathcal{V})$ works.

THEOREM 3.8. *$(\mathcal{P},\mathcal{V})$ is a CS proof system with a random oracle.*

As per our Definition 3.2, to prove Theorem 3.8 we need to prove that $(\mathcal{P},\mathcal{V})$ satisfies both feasible completeness and computational soundness.

3.5.1. Proof of feasible completeness. Adopting the two-oracle formulation of Definition 3.2 and recalling the domain and range of each of the two oracles in our construction of $(\mathcal{P},\mathcal{V})$, what we need to prove is the following.

> *There exists $c_1, c_2, c_3 > 0$ such that for all $q = (M, x, y, t) \in \mathcal{L}$, for all k, for all $f_1 \in \Sigma^{2k} \to \Sigma^k$, and for all $f_2 \in \Sigma^{k+|q|} \to \Sigma^{k\cdot\lambda(|q|)}$:*
> (i) *$\mathcal{P}^{f_1,f_2}(q, 1^k)$ halts within $(|q|kt)^{c_2}$ computational steps, outputting a binary string $\mathcal{C}$ whose length is $\leq (|q|k)^{c_3}$, and*
> (ii) *$\mathcal{V}^{f_1,f_2}(q, 1^k, \mathcal{C}) = YES$.*

It is immediately seen that subproperty (ii) of feasible completeness holds. That is, for all $q = (M, x, y, t) \in \mathcal{L}$, 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) steps for producing a certificate, follows as easily. Indeed, prover $\mathcal{P}$ performs the following operations: (1) initially invests t steps of computation for running M on input x; (2) takes a number of steps polynomial in q's length (i.e., n) and t for computing the samplable proof τ; (3) makes less than $t \log t$ queries (each at unit cost) to the second random oracle for generating the Merkle tree; and, finally, (4) makes additional polynomially many steps for running $\mathcal{V}$ "in his head" and answering its queries so as to build the desired CS certificate.

Finally, let us argue that the length of $(\mathcal{P},\mathcal{V})$'s certificates are in accordance to Definition 3.2. Namely, letting $q = (M, x, y, t)$ be a member of $\mathcal{L}$ and $\mathcal{C} = \mathcal{P}^{f_1,f_2}(q, 1^k)$, then $\mathcal{C}$'s length is polynomial in $|q|$ and k. To this end, notice that $\mathcal{C}$ contains a k-bit root value, plus one authentication path (in the constructed Merkle tree) for each bit that the samplable verifier SV wishes to access when run on inputs q and $|\tau|$ and (virtual) access to the samplable proof τ of "$M(x) = y$ in t steps." Now, because τ can be computed in a number of steps upperbounded by a fixed polynomial in $|q|$ and t, and because according to our conventions $t < 2^{|q|}$, it follows that the length of the binary representation of $|\tau|$ is upperbounded by some other fixed polynomial in $|q|$ alone. Therefore, because SV runs in polynomial time, the number of bits of τ it accesses (i.e., the number of authentication paths included in $\mathcal{C}$) is polynomial in $|q|$ alone. The claim about the length of $\mathcal{C}$ then follows from the fact that each authentication path contains a k-bit value for each level of the constructed Merkle tree and thus $k \log \tau < k|q|$ bits overall. □

3.5.2. Proof of computational soundness. Adopting the two-oracle formulation of Definition 3.2 and recalling the domain and range of each of the two oracles in our construction of $(\mathcal{P},\mathcal{V})$, what we need to prove is the following.

> *There exist positive constants c_4, c_5, and c_6 such that for all $\widetilde{q} \notin \mathcal{L}$, for all k such that $2^k > |\widetilde{q}|^{c_4}$, and for all (cheating) deterministic $2^{c_5 k}$-call algorithm $\widetilde{P}$, for random oracles $\rho_1 \in \Sigma^{2k} \to \Sigma^k$ and $\rho_2 \in \Sigma^{k+|\widetilde{q}|} \to \Sigma^{k\cdot\lambda(|\widetilde{q}|)}$,*
>
> $$Prob_{\rho_1;\rho_2}[V^{\rho_1,\rho_2}(\widetilde{q}, 1^k, \widetilde{P}^{\rho_1,\rho_2}(\widetilde{q}, 1^k)) = YES] \leq 2^{-c_6 k}.$$

We shall actually prove the following theorem.

THEOREM 3.9. *$(\mathcal{P},\mathcal{V})$ satisfies the above condition for the following choice of c_4, c_5, and c_6:*

$$c_4 \overset{\text{def}}{=} 64c, \qquad c_5 \overset{\text{def}}{=} 1/8, \quad \text{and} \qquad c_6 \overset{\text{def}}{=} 1/16,$$

where $c \overset{\text{def}}{=}$ the smallest positive integer C such that $n^C > \ell(n)$ for all integers $n > 1$.

Recall that ℓ is the first length-bound of our underlying sampling proof system (SP, SV).

Proof idea. In essence, the proof is by contradiction. Assume that $q \notin \mathcal{L}$, and that, nonetheless, we are given a cheating prover $\widetilde{\mathcal{P}}$ that has a nonnegligible probability of outputting a CS proof for $q \in \mathcal{L}$. Then, we derive a contradiction by using such $\widetilde{\mathcal{P}}$ to build a *samplable* proof for $q \in \mathcal{L}$. Let us explain.

By hypothesis, for many oracles ρ_1 and ρ_2, $\widetilde{\mathcal{P}}$ should be able to produce a CS proof $\mathcal{C}_{\rho_1,\rho_2}$ for $q \in \mathcal{L}$. Such string $\mathcal{C}_{\rho_1,\rho_2}$ allegedly includes the contents of a few bit-locations of an underlying samplable proof, τ_{ρ_1,ρ_2}. By varying ρ_1 and ρ_2, and looking at their corresponding $\mathcal{C}_{\rho_1,\rho_2}$, we obtain the contents of more and more bit-locations, until we discover the bits in all the locations having a nonnegligible probability of being queried by the sampling verifier on input q. Despite the fact that such contents are pieced together from different CS proofs (and thus potentially from different underlying samplable proofs), we shall prove that, with high probability, the discovered contents are consistent with a *single* samplable proof τ.

Local definition 1.

- *Probabilities.* Let $S_1, S_2, \ldots,$ be finite sets, and let E be an event. Then, by $PROB_{x_1 \in S_1; x_2 \in S_2; \ldots}[E]$ we denote the probability of E in the experiment consisting of selecting elements $x_1 \in S_1, x_2 \in S_2, \ldots$ randomly and independently. If, for some x_i, it is already clear that x_i ranges in S_i, we may omit specifying S_i and more simply denote the same probability by $PROB_{\ldots; x_i; \ldots}[E]$.
- *Pseudoexecutions.* We shall consider executing a cheating, N-call, prover $\widetilde{\mathcal{P}}^{(\cdot,\cdot)}$ by answering its queries to the first oracle by means of a function f, and its queries to the second oracles by means of a predetermined, N-long, sequence S (i.e., the ith query to the second oracle will be answered with the ith element of S). We shall call such a process a *pseudoexecution (of $\widetilde{\mathcal{P}}$)*, or an execution of *algorithm* $\widetilde{\mathcal{P}}^{f,S}$. When $\widetilde{\mathcal{P}}^{f,S}$ is run on an n-bit input ($q \notin \mathcal{L}$) and security parameter k, then each element of S will consist of a $k\lambda(n)$-bit string (i.e., a possible random tape for $\mathcal{V}$). If σ is a string and m an integer between 1 and N, by the expression $\widetilde{\mathcal{P}}^{f,S_m=\sigma}$ we denote the algorithm identical to $\widetilde{\mathcal{P}}^{f,S}$, except that the mth query to the second oracle is answered by σ (i.e., the mth element of S—no matter what it originally was—is "forced" to be σ). By the expression $\widetilde{\mathcal{P}}^{f,S_m=\sigma_1,\sigma_2}$ we denote the algorithm that first executes $\mathcal{P}^{f,S_m=\sigma_1}$ and then $\mathcal{P}^{f,S_m=\sigma_2}$.
- *Collisions.* Let f be an oracle, $A(\cdot)$ an oracle-calling algorithm, and z an input. Then, by the expression *an f-collision in $A^f(z)$*, we mean that executing A on z with oracle f, A queries f about two distinct strings a and b and obtains the same string, c $(= f(a) = f(b))$, in response.
- *Pseudocertificates and pseudoroots.* Without loss of generality, we assume that a cheating prover $\widetilde{\mathcal{P}}$ never asks the same query twice to the same oracle. Again, without loss of generality, we assume that each cheating prover $\widetilde{\mathcal{P}}$ verifies all its nonempty outputs. That is, if $\widetilde{\mathcal{C}}$ is a nonempty string and $\widetilde{\mathcal{P}}^{f_1,f_2}(\widetilde{q}, \widetilde{1^k}) = \widetilde{\mathcal{C}}$, then, prior to outputting $\widetilde{\mathcal{C}}$, $\widetilde{\mathcal{P}}$ runs $\mathcal{V}$ making all required calls to f_1 and f_2 so as to verify that $\mathcal{V}^{f_1,f_2}(\widetilde{q}, \widetilde{1^k}, \widetilde{\mathcal{C}}) = YES$, and thus that $\widetilde{\mathcal{C}}$ is a CS certificate for $\widetilde{q}$. If $\widetilde{q}$ does not belong to the CS language $\mathcal{L}$, to emphasize this fact we refer to $\widetilde{\mathcal{C}}$ itself as a *pseudocertificate (for $\widetilde{q}$)* and to

its $\widetilde{k}$-bit prefix as a *pseudoroot*.

In an execution of a cheating prover $\widetilde{\mathcal{P}}$ producing a pseudocertificate $\widetilde{\mathcal{C}}$ for $\widetilde{q}$, we say that $\widetilde{\mathcal{C}}$ is *relative to tape-number* m if (1) the pseudoroot of $\widetilde{\mathcal{C}}$ is a string $\widetilde{R}_\varepsilon$ such that the mth query of $\widetilde{\mathcal{P}}$ to its second oracle consists of the pair $(\widetilde{q}, \widetilde{R}_\varepsilon)$.

To indicate a generic pseudocertificate having pseudoroot $\widetilde{R}_\varepsilon$, we write $\widetilde{R}_\varepsilon \ldots$.

A proof by contradiction. We proceed by contradiction. Assume that Theorem 3.9 is incorrect; then, because $\widetilde{\mathcal{P}}$ verifies all its nonempty outputs, the following proposition holds.

PROPOSITION 3.10. *There exist an integer $\widetilde{n} > 1$, a $\widetilde{n}$-bit string $\widetilde{q} \notin \mathcal{L}$, an integer $\widetilde{k}$ such that $2^{\widetilde{k}} > \widetilde{n}^{64c}$, and a deterministic, $2^{\widetilde{k}/8}$-call, cheating prover $\widetilde{\mathcal{P}}$ such that, for random oracles $\rho_1 \in \Sigma^{2\widetilde{k}} \to \Sigma^{\widetilde{k}}$ and $\rho_2 \in \Sigma^{\widetilde{k}+\widetilde{n}} \to \Sigma^{\widetilde{k}\cdot\lambda(\widetilde{n})}$,*

(P1) $PROB_{\rho_1;\rho_2}[\widetilde{\mathcal{P}}^{\rho_1,\rho_2}(\widetilde{q}, 1^{\widetilde{k}}) \neq \varepsilon] > 2^{-\widetilde{k}/16}$.

We now show that Proposition 3.10 contradicts the fact that $(\mathcal{P}, \mathcal{V})$'s subroutine (SP, SV) is a sampling proof system. We start by stating without proof some easy probabilistic facts.

Basic lemmas. Let A and B be finite sets, $A \times B$ their Cartesian product, and E a subset of $A \times B$. Then, in the following two lemmas $PROB_{a;b}[(a,b) \in E]$ denotes the probability that (a,b) belongs to E by selecting uniformly and independently a in A and b in B, and $PROB_b[(a,b) \in E]$ denotes the probability that (a,b) belongs to E by selecting uniformly b in B.

LEMMA 3.11. *Assume $PROB_{a;b}[(a,b) \in E] > x$, and let $G = \{a : PROB_b[(a,b) \in E] > 2^{-1} \cdot x\}$. Then,*

$$PROB_a[a \in G] > x/2.$$

LEMMA 3.12. *Assume $PROB_{a;b}[(a,b) \in E] < x$, and let $L = \{a : PROB_b[(a,b) \in E] < nx\}$. Then,*

$$PROB_a[a \in L] > 1 - n^{-1}.$$

LEMMA 3.13. *For all positive integers k and N, for all N-call algorithms $\mathcal{A}(\cdot)$, and for all inputs z,*

$$PROB_{f \in \Sigma^{2k} \to \Sigma^k}[\text{ an } f\text{-collision in } \mathcal{A}^f(z)] < N^2 2^{-k}.$$

Note that Lemma 3.13 continues to hold if A has additional inputs and oracles, provided that f is randomly selected independently of them.

An averaging argument.

LEMMA 3.14. *Let $\widetilde{n}$, $\widetilde{q}$, and $\widetilde{k}$ be as in Proposition 3.10. Then, there exist an oracle $f_1 \in \Sigma^{2\widetilde{k}} \to \Sigma^{\widetilde{k}}$, a $2^{\widetilde{k}/8}$-long sequence S of $\widetilde{k} \cdot \lambda(\widetilde{n})$-bit strings, an integer $m \in [1, 2^{\widetilde{k}/8}]$, and a $\widetilde{k}$-bit (pseudoroot) $\widetilde{R}_\varepsilon$ such that*

(L1.1) $PROB_{\sigma \in \Sigma^{\widetilde{k}\cdot\lambda(\widetilde{n})}}[\widetilde{\mathcal{P}}^{f_1, S_m = \sigma}(\widetilde{q}, 1^{\widetilde{k}}) = \widetilde{R}_\varepsilon \cdots] > 2^{-1} \cdot 2^{-3\widetilde{k}/16}$, and

(L1.2) $PROB_{\sigma_1,\sigma_2 \in \Sigma^{\widetilde{k}\cdot\lambda(\widetilde{n})}}[\widetilde{\mathcal{P}}^{f_1, S_m = \sigma_1}(\widetilde{q}, 1^{\widetilde{k}}) = \widetilde{R}_\varepsilon \cdots = \widetilde{\mathcal{P}}^{f_1, S_m = \sigma_2}(\widetilde{q}, 1^{\widetilde{k}}) \wedge f_1$-collision in $\mathcal{P}^{f_1, S_m = \sigma_1, \sigma_2}(\widetilde{q}, \widetilde{k})] < 32 \cdot 2^{-9\widetilde{k}/16}$.

Proof. Let ρ_1 and ρ_2 be oracles as in Proposition 3.10. Then, because $\widetilde{\mathcal{P}}^{\rho_1,\rho_2}$ verifies its nonempty outputs (and because it is $2^{\widetilde{k}/8}$-call), each of its pseudocertificates is relative to some tape-number between 1 and $2^{\widetilde{k}/8}$. Thus, inequality P1 implies that there exists a positive integer $m \in [1, 2^{\widetilde{k}/8}]$ such that

(L1.3) $PROB_{\rho_1;\rho_2}[\widetilde{\mathcal{P}}^{\rho_1,\rho_2}(\widetilde{q}, 1^{\widetilde{k}}) \neq \varepsilon \wedge \widetilde{\mathcal{P}}^{\rho_1,\rho_2}(\widetilde{q}, 1^{\widetilde{k}})$ is relative to tape-number $m] \geq 2^{-\widetilde{k}/16}/2^{\widetilde{k}/8} = 2^{-3\widetilde{k}/16}$.

Therefore, by focusing our attention on tape-number m (i.e., on the mth answer of our random oracle $\rho_2 \in \Sigma^{\widetilde{k}+\widetilde{n}} \to \Sigma^{\widetilde{k}\cdot\lambda(\widetilde{n})}$) and by averaging, inequality L1.3 implies that there exists a $2^{\widetilde{k}/8}$-long sequence, S, of $\widetilde{k}\cdot\lambda(\widetilde{n})$-bit strings (i.e., of possible second-oracle answers), such that

(L1.4) $PROB_{\rho_1;\sigma\in\Sigma^{\widetilde{k}\cdot\lambda(\widetilde{n})}}[\widetilde{\mathcal{P}}^{\rho_1,S_m=\sigma}(\widetilde{q}, 1^{\widetilde{k}}) \neq \varepsilon \wedge \widetilde{\mathcal{P}}^{\rho_1,S_m=\sigma}(\widetilde{q}, 1^{\widetilde{k}})$ is relative to tape-number $m] > 2^{-3\widetilde{k}/16}$.

Define now an oracle $\rho_1 : \Sigma^{2\widetilde{k}} \to \Sigma^{\widetilde{k}}$ to be *good* if

$$PROB_{\sigma\in\Sigma^{\widetilde{k}\cdot\lambda(\widetilde{n})}}[\widetilde{\mathcal{P}}^{\rho_1,S_m=\sigma}(\widetilde{q}, 1^{\widetilde{k}}) \neq \varepsilon \wedge \widetilde{\mathcal{P}}^{\rho_1,S_m=\sigma}(\widetilde{q}, 1^{\widetilde{k}}) \text{ is relative to tape-number } m] > 2^{-1}\cdot 2^{-3\widetilde{k}/16}.$$

Then, inequality L1.4 and Lemma 3.11 imply that

(L1.5) $PROB_{\rho_1}[\rho_1 \text{ good }] > 2^{-1}\cdot 2^{-3\widetilde{k}/16}$.

Note now that because both $\widetilde{\mathcal{P}}^{\rho_1,S_m=\sigma_1}$ and $\widetilde{\mathcal{P}}^{\rho_1,S_m=\sigma_2}$ make at most $2^{\widetilde{k}/8}$ oracle calls, algorithm $\widetilde{\mathcal{P}}^{\rho_1,S_m=\sigma_1,\sigma_2}$ makes at most twice as many calls to ρ_1. Thus, Lemma 3.13 implies that

(L1.6) $PROB_{\rho_1\in\Sigma^{2\widetilde{k}}\to\Sigma^{\widetilde{k}};\sigma_1,\sigma_2\in\Sigma^{\widetilde{k}\cdot\lambda(\widetilde{n})}}[\rho_1\text{-collision in } \widetilde{\mathcal{P}}^{\rho_1,S_m=\sigma_1,\sigma_2}(\widetilde{q}, 1^{\widetilde{k}})] < 4\cdot 2^{\widetilde{k}/4}\cdot 2^{-\widetilde{k}} = 4\cdot 2^{-3\widetilde{k}/4}$.

Define now an oracle $\rho_1 : \Sigma^{2\widetilde{k}} \to \Sigma^{\widetilde{k}}$ to be *lucky* if

$$PROB_{\sigma_1,\sigma_2\in\Sigma^{\widetilde{k}\cdot\lambda(\widetilde{n})}}[\rho_1\text{-collision in } \widetilde{\mathcal{P}}^{\rho_1,S_m=\sigma_1,\sigma_2}(\widetilde{q}, 1^{\widetilde{k}})] < (8\cdot 2^{3\widetilde{k}/16})\cdot(4\cdot 2^{-3\widetilde{k}/4}) = 32\cdot 2^{-9\widetilde{k}/16}.$$

Then, inequality L1.6 and Lemma 3.12 imply that

(L1.7) $PROB_{\rho_1}[\rho_1 \text{ lucky }] > 1 - 8^{-1}\cdot 2^{-3\widetilde{k}/16} > 1 - 2^{-1}\cdot 2^{-3\widetilde{k}/16} \geq 1 - PROB_{\rho_1}[\rho_1 \text{ good}] = PROB_{\rho_1}[\rho_1 \text{ not good}]$.

Because inequality L1.7 implies that there exist oracles in $\Sigma^{2\widetilde{k}} \to \Sigma^{\widetilde{k}}$ that are both good and lucky, let f_1 be one such oracle. Then, because F_1 is good, letting F_1 be oracle f_1 of Lemma 3.14 satisfies inequality L1.1. In fact, L1.1 simply states that f_1 is a good oracle. Let us now show that there exists a k-bit value $\widetilde{R}_\varepsilon$ such that letting F_1 be oracle f_1 of Lemma 3.14 satisfies inequality L1.2. In fact, notice that, for any possible choice of σ, the computation of $\widetilde{\mathcal{P}}^{F_1,S_m=\sigma}(\widetilde{q}, 1^{\widetilde{k}})$ is always identical up to its mth query to the second oracle, and thus that there exists a $\widetilde{k}$-bit value $\widetilde{R}_\varepsilon$ such that all mth queries consist of the same pair $(\widetilde{q}, \widetilde{R}_\varepsilon)$. Therefore, whenever an execution of $\widetilde{\mathcal{P}}^{f_1,S_m=\sigma}(\widetilde{q}, 1^{\widetilde{k}})$ produces a nonempty pseudocertificate with respect to tape-number m, $\widetilde{R}_\varepsilon$ will be the pseudoroot of this certificate. Now, because F_1 is lucky, we have

$PROB_{\sigma_1,\sigma_2\in\Sigma^{\widetilde{k}\cdot\lambda(\widetilde{n})}}[\widetilde{\mathcal{P}}^{F_1,S_m=\sigma_1}(\widetilde{q}, 1^{\widetilde{k}}) = \widetilde{R}_\varepsilon\cdots = \widetilde{\mathcal{P}}^{F_1,S_m=\sigma_2}(\widetilde{q}, 1^{\widetilde{k}}) \wedge F_1$-collision in $\widetilde{\mathcal{P}}^{F_1,S_m=\sigma_1,\sigma_2}(\widetilde{q}, 1^{\widetilde{k}})]$

$< PROB_{\sigma_1,\sigma_2\in\Sigma^{\widetilde{k}\cdot\lambda(\widetilde{n})}}[F_1\text{-collision in } \widetilde{\mathcal{P}}^{F_1,S_m=\sigma_1,\sigma_2}(\widetilde{q}, 1^{\widetilde{k}})] <$ (because F_1 is lucky) $32\cdot 2^{-9\widetilde{k}/16}$. □

A collision argument.

Local definition 2. If $\widetilde{\mathcal{C}}$ is a pseudocertificate, we write $\widetilde{\mathcal{C}} \ni (i, b)$ if $\widetilde{\mathcal{C}}$ indicates that location i (in the samplable proof that is allegedly Merkle-hashed to the pseudoroot of $\widetilde{\mathcal{C}}$) contains bit b.

LEMMA 3.15. *Let $\widetilde{n}$, $\widetilde{q}$, $\widetilde{k}$, $\widetilde{\mathcal{P}}$, f_1, S, m, and $\widetilde{R}_\varepsilon$ be as in Lemma 3.14, let σ_0 and σ_1 be two distinct $k \cdot \lambda(\widetilde{n})$-bit strings, and let E_0 be the execution of $\widetilde{\mathcal{P}}^{f_1, S_m=\sigma_0}(\widetilde{q}, 1^{\widetilde{k}})$ and E_1 the execution of $\widetilde{\mathcal{P}}^{f_1, S_m=\sigma_1}(\widetilde{q}, 1^{\widetilde{k}})$. Now, if E_0 and E_1 produce two pseudocertificates, respectively, $\mathcal{C}_0$ and $\mathcal{C}_1$, such that (1) both $\mathcal{C}_0$ and $\mathcal{C}_1$ have pseudoroot $\widetilde{R}_\varepsilon$ and (2) for some common location i, $\mathcal{C}_0 \ni (i, 0)$ and $\mathcal{C}_1 \ni (i, 1)$, then an f_1-collision occurs in $\widetilde{\mathcal{P}}^{f_1, S_m=\sigma_0,\sigma_1}(\widetilde{q}, 1^{\widetilde{k}})$.*

Proof. Recall that, supposedly, the underlying samplable-proof of $\widetilde{q} \in \mathcal{L}$ has been divided into N substrings, each $\widetilde{k}$-bit long and stored in a separate leaf of a Merkle tree of depth $\log N$. Thus bit-location i should be stored in the Ith leftmost leaf of the Merkle tree, where $I = \lceil i/\widetilde{k} \rceil$, or, equivalently, in the node whose $\log N$-bit name is $[I] = \alpha_1 \cdots \alpha_{\log N}$. According to our notation, the value stored in this node is denoted by $R_{[I]}$. Now, because they indicate different bit-values for location i, pseudocertificates $\mathcal{C}_0$ and $\mathcal{C}_1$ also indicate different $\widetilde{k}$-bit values for the content of node $[I]$, respectively, $R^0_{[I]}$ and $R^1_{[I]}$.

Pseudocertificates $\mathcal{C}_0$ and $\mathcal{C}_1$ also include two authentication paths for these values. Let them be, respectively,

$$R^0_{\alpha_1 \cdots \alpha_{\log N-1} \bar{\alpha}_{\log N}} | R^0_{\alpha_1 \cdots \bar{\alpha}_{\log N-1}} | \cdots | R^0_{\bar{\alpha}_1} | \widetilde{R}_\varepsilon$$
and
$$R^1_{\alpha_1 \cdots \alpha_{\log N-1} \bar{\alpha}_{\log N}} | R^1_{\alpha_1 \cdots \bar{\alpha}_{\log N-1}} | \ldots, R^1_{\bar{\alpha}_1} | \widetilde{R}_\varepsilon.$$

Because cheating prover $\widetilde{\mathcal{P}}$ verifies all its outputs, for each $j \in [1, \log N]$ it queries oracle f_1 about string S^0_j in the first execution, and about string S^1_j in the second execution, where $S^0_j = R^0_{\alpha_1 \cdots \alpha_j} | R^0_{\alpha_1 \cdots \bar{\alpha}_j}$ and $S^1_j = R^1_{\alpha_1 \cdots \alpha_j} | R^1_{\alpha_1 \cdots \bar{\alpha}_j}$.

This implies that there exists a value $j \in [1, \log N]$ such that S^0_j and S^1_j are different queries but oracle f_1 returns the same answer on them. In fact, for $j = 1$, the two queries $S^0_1 (= R^0_{\alpha_1} | R^0_{\bar{\alpha}_1})$ and $S^1_1 (= R^2_{\alpha_1} 1 | R^1_{\bar{\alpha}_1})$ are both answered by $\widetilde{R}_\varepsilon$, and for $j = \log N$ the two queries $S^0_{\log N} (= R^0_{[I]})$ and $S^1_{\log N} (= R^1_{[I]})$ are different, because they coincide with the two different values for leaf I. □

Reaching the desired contradiction.

Local definition 3. Letting $\widetilde{n}$, $\widetilde{q}$, $\widetilde{k}$, S, m, and $\widetilde{R}_\varepsilon$ be as in Lemma 3.14, define the following.

- *Conditional probabilities $P_{i,0}$ and $P_{i,1}$.* For each bit-location $i \in [1, \ell(\widetilde{n})]$, define
  $$P_{i,0} = PROB_{\sigma \in \Sigma^{\widetilde{k} \cdot \lambda(\widetilde{n})}} [\widetilde{\mathcal{P}}^{f_1, S_m=\sigma}(\widetilde{q}, 1^{\widetilde{k}}) = \widetilde{R}_\varepsilon \cdots \wedge \widetilde{R}_\varepsilon \cdots \ni (i, 0)]$$
  and
  $$P_{i,1} = PROB_{\sigma \in \Sigma^{\widetilde{k} \cdot \lambda(\widetilde{n})}} [\widetilde{\mathcal{P}}^{f_1, S_m=\sigma}(\widetilde{q}, 1^{\widetilde{k}}) = \widetilde{R}_\varepsilon \cdots \wedge \widetilde{R}_\varepsilon \cdots \ni (i, 1)].$$
- *String τ.* For each location $i \in [1, \ell(\widetilde{n})]$, define
  $\tau_i = 0$ if $P_{i,0} \geq P_{i,1}$ and $\tau_i = 1$ otherwise.
- *Probabilities P_i and $P_{\bar{i}}$.* For each bit-location $i \in [1, \ell(\widetilde{n})]$, define
  $P_i = P_{i,\tau_i}$ and $P_{\bar{i}} = P_{i,\bar{\tau}_i}$.
- *Event "all queries answered by τ."*
  If $\sigma \in \Sigma^{\widetilde{k} \cdot \lambda(\widetilde{n})}$, in a pseudoexecution of $\widetilde{\mathcal{P}}^{f_1, S_m=\sigma}(\widetilde{q}, 1^{\widetilde{k}})$ with no f_1-collision and producing a pseudocertificate $\widetilde{\mathcal{C}}$ with pseudoroot $\widetilde{R}_\varepsilon$, "all queries answered

by τ" denotes the event that, for all bit-locations $i \in [1, \ell(\widetilde{n})]$ and for all bit b, $\widetilde{\mathcal{C}} \ni (i,b)$ implies $b = \tau_i$.

LEMMA 3.16. *Let $\widetilde{n}$, $\widetilde{q}$, $\widetilde{k}$, $\widetilde{\mathcal{P}}$, f_1, S, m, and $\widetilde{R}_\varepsilon$ be as in Lemma 3.14 Then,* $PROB_{\sigma\in\Sigma^{\widetilde{k}\cdot\lambda(\widetilde{n})}}[\widetilde{\mathcal{P}}^{f_1,S_m=\sigma}(\widetilde{q},1^{\widetilde{k}}) = \widetilde{R}_\varepsilon \cdots \wedge \text{ all queries answered by } \tau] > 2^{-\widetilde{k}}$.

Proof. We start by claiming that

(L3.1) $P_{\bar{i}} < 4\cdot 2^{-9\widetilde{k}/32}$.

To prove our claim, consider selecting two $\widetilde{k}\cdot\lambda(\widetilde{n})$-bit strings, σ_0 and σ_1, and then executing $\widetilde{\mathcal{P}}^{f_1,S_m=\sigma_0,\sigma_1}(\widetilde{q},1^{\widetilde{k}})$. That is, consider executing first $\widetilde{\mathcal{P}}^{f_1,S_m=\sigma_0}(\widetilde{q},1^{\widetilde{k}})$ and then $\widetilde{\mathcal{P}}^{f_1,S_m=\sigma_1}(\widetilde{q},1^{\widetilde{k}})$. Then, by definition of P_i and $P_{\bar{i}}$, with probability $\geq 2P_iP_{\bar{i}}$, one of the latter two executions outputs a pseudocertificate $\mathcal{C}_0$ and the other a pseudocertificate $\mathcal{C}_1$ such that (1) $\mathcal{C}_0$ and $\mathcal{C}_1$ are nonempty pseudocertificates having pseudoroot $\widetilde{R}_\varepsilon$, and (2) $\mathcal{C}_0 \ni (i,0)$ and $\mathcal{C}_1 \ni (i,1)$. By Lemma 3.15, (1) and (2) imply that, with probability $\geq 2P_iP_{\bar{i}}$ (taken over the choices of σ_1 and σ_2), (3) an f_1-collision occurs in $\widetilde{\mathcal{P}}^{f_1,S_m=\sigma_0,\sigma_1}(\widetilde{q},1^{\widetilde{k}})$. Thus, by Lemma 3.14 (inequality L1.2), we have $2P_iP_{\bar{i}} < 32\cdot 2^{-9\widetilde{k}/16}$. Now, because $P_{\bar{i}} \leq P_i$ by definition, we have $2P_{\bar{i}}^2 \leq 2P_iP_{\bar{i}} < 32\cdot 2^{-9\widetilde{k}/16}$, and thus $P_{\bar{i}} < 4\cdot 2^{-9\widetilde{k}/32}$ as initially claimed.

Define now $P_{\bar{\tau}}$ as the probability that "$\widetilde{\mathcal{P}}^{f_1,S_m=\sigma}(\widetilde{q},1^{\widetilde{k}}) = \widetilde{R}_\varepsilon\cdots$, but *not* all queries answered by τ", that is,

$$P_{\bar{\tau}} \stackrel{\text{def}}{=} PROB_{\sigma\in\Sigma^{\widetilde{k}\cdot\lambda(\widetilde{n})}}[\widetilde{\mathcal{P}}^{f_1,S_m=\sigma}(\widetilde{q},1^{\widetilde{k}}) = \widetilde{R}_\varepsilon\cdots \wedge\ \exists i \widetilde{R}_\varepsilon\cdots \ni (i,\bar{\tau}_i)].$$

Then, because there are at most $\ell(\widetilde{n})$ bit-locations i, and because (due to Proposition 3.10 and our definition of c) $\ell(\widetilde{n}) < (\widetilde{n})^c < 2^{\widetilde{k}/64}$, we have

(L3.2) $P_{\bar{\tau}} \leq \sum_{i=1}^{\ell(\widetilde{n})} P_{\bar{i}} < 4\cdot 2^{-9\widetilde{k}/32}\cdot\ell(\widetilde{n}) < 4\cdot 2^{-9\widetilde{k}/32}\cdot 2^{\widetilde{k}/64} = 4\cdot 2^{-17\widetilde{k}/64}$.

Thus

(L3.3) $PROB_{\sigma\in\Sigma^{\widetilde{k}\cdot\lambda(\widetilde{n})}}[\widetilde{\mathcal{P}}^{f_1,S_m=\sigma}(\widetilde{q},1^{\widetilde{k}}) = \widetilde{R}_\varepsilon\cdots \ \wedge\ \text{ all queries answered by } \tau]$
$\geq PROB_{\sigma\in\Sigma^{\widetilde{k}\cdot\lambda(\widetilde{n})}}[\widetilde{\mathcal{P}}^{f_1,S_m=\sigma}(\widetilde{q},1^{\widetilde{k}}) = \widetilde{R}_\varepsilon\cdots] - P_{\bar{\tau}} \geq$ (by inequalities L1.1 and L3.2) $2^{-1}\cdot 2^{-3\widetilde{k}/16} - 4\cdot 2^{-17\widetilde{k}/64} >$ (because $2^{\widetilde{k}} > n^{64c}$, $n\geq 2$ and $c \geq 1$ imply $\widetilde{k} > 64$) $2^{-\widetilde{k}}$. □

Notice now that Lemma 3.16 contradicts the fact that the underlying (SP, SV) is a sampling proof system according to Definition 3.7. In fact, because $\widetilde{\mathcal{P}}$ verifies all its nonempty outputs, we have

$2^{-\widetilde{k}} <$ (because of inequality L3.3) $PROB_{\sigma\in\Sigma^{\widetilde{k}\cdot\lambda(\widetilde{n})}}[\widetilde{\mathcal{P}}^{f_1,S_m=\sigma}(\widetilde{q},1^{\widetilde{k}}) = \widetilde{R}_\varepsilon\cdots$ $\wedge$ all queries answered by $\tau] = PROB_{\sigma\in\Sigma^{\widetilde{k}\cdot\lambda(\widetilde{n})}}[\mathcal{V}^{f_1,S_m=\sigma}(\widetilde{q},1^{\widetilde{k}},\widetilde{R}_\varepsilon\cdots) =$ $YES\ \wedge$ all queries answered by τ].

But, by our construction of $\mathcal{V}$, the last inequality implies that, by running sampling verifier SV on inputs $\widetilde{q}$ ($\notin\mathcal{L}$), $|\tau|$ (i.e., $\widetilde{n}$), and $\widetilde{k}$, having a random tape of length $\widetilde{k}\cdot\lambda(|\tau|)$, and having random access to τ, SV accepts with probability $> 2^{-\widetilde{k}}$.

Because $\widetilde{q}\notin\mathcal{L}$, the existence of τ contradicts property 2 of Theorem 3.6. The contradiction establishes Theorem 3.9, and thus that our $(\mathcal{P},\mathcal{V})$ enjoys computational soundness, completing our proof of Theorem 3.9. □

3.6. The significance of CS proofs with a random oracle. Random oracles may be quite theoretical, and, as discussed later on, one might consider implementing CS proofs cryptographically (with or without interaction). But the latter implementations would be meaningless if it turned out that $\mathcal{P} = \mathcal{NP}$. This would not be too bad, one might say, because, if $\mathcal{P} = \mathcal{NP}$, the very notion of an "efficient" proof system would be meaningless, at least in any broad sense.[22]

Personally, we disagree: fundamental intuitions such as proofs being a notion that is both meaningful and separate (from that of accepting[23]) could not be shaken by a formal result such as $\mathcal{P} = \mathcal{NP}$. Indeed, 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 of the notion of a proof. But if it turned out that $\mathcal{P} = \mathcal{NP}$, to the author this would only mean that $\mathcal{NP}$ did not provide such a reasonable approximation after all.

It is thus important to establish meaningful models for which we can *show* that *proofs do exist as an independent notion.* CS proofs with a random oracle provide us with such a model: indeed, they guarantee that

> *even if $\mathcal{NP} = \mathcal{P}$, given a sufficient amount of randomness in the proper form, fundamental intuitions like verification being polylogarithmically easier than decision are indeed true.*

4. Other types of CS proofs. Many variants of the basic notion of a CS proof exist. Below, we confine ourselves to briefly presenting just two additional ones: that of an interactive CS proof (because it can be implemented based on standard cryptographic assumptions) and that of a noninteractive CS proof (because it implies the existence of CS checkers for $\mathcal{NP}$-complete problems).

4.1. Interactive CS proofs.

The notion of an interactive CS proof system. Let us first quickly recall interactive Turing machines (ITMs) as defined by Goldwasser, Micali, and Rackoff [21]. Informally, an ITM is a probabilistic Turing machine capable of "sending and receiving messages." An ITM is meant to be run a number of times, each time starting with the internal configuration reached at the end of the previous run. Each run of an ITM starts by reading one incoming message—i.e., reading a string on a special tape—and ends by sending one outgoing message—i.e., writing a string on another special tape. (In an initial run we assume that the incoming message is the input, and that the internal configuration consists of a blank work tape and a distinguished start state.) An ITM halts when, in a given run, it enters a special halting state, from which it takes no further action. ITMs are meant to be executed in pairs. If A and B are ITMs, an execution of (A, B) on input x is obtained by having x be both A's and B's input and by running alternatively A and B, so that each outgoing message of A is B's incoming message in the next run of B, and vice versa. The number of rounds in an execution of (A, B) is the number of times in which either of the two ITMs sends a message. By convention, an execution of (A, B) starts and ends with a run of B. In its last run, B may *accept* by outputting the special symbol YES, or *reject* by outputting the special symbol NO. Except for their exchanged messages, in an execution of (A, B) neither ITM has access to the internal computation (in particular the coin tosses) of the other. The probability that, after a random execution of (A, B) on a given input x, B accepts is taken over all coin tosses of A and B.

[22]Though concurring with us that properly capturing efficient verification may require more than $\mathcal{P} \neq \mathcal{NP}$, one might also believe that if $\mathcal{P} = \mathcal{NP}$, there would be little or no notion of efficient verification to be captured.

[23]Again, 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 whenever he wishes to establish whether a given statement holds. Proofs cannot properly exist as a separate notion unless they succeed in making verification of truth much easier than accepting truth.

Slightly more generally, we shall also consider ITM pairs (A, B), where A actually is an *interactive circuit.* This allows us to use the size of A to bound more effectively the number of "steps" A may make in an execution with B.[24] Recall that 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).

Notice that an interactive circuit A may be taken to be deterministic, because it might have wired in any finite lucky sequence of coin tosses. (In this case the probability that B is convinced in a random execution with A on input x solely depends on B's coin tosses.)

DEFINITION 4.1. *Let (P, V) be a pair of ITMs, the second of which running in polynomial time, and let $\mathcal{L}$ be the CS language. We say that (P, V) is an* interactive CS proof system *if there are four positive constants a, b, c, and d such that the following two properties are satisfied.*

$1''$. Feasible completeness. *For all $q = (M, x, y, t) \in \mathcal{L}$, and for all integers k, in every execution of (P, V) on inputs q and 1^k,*
($1''$.i) *P halts within $(|q|kt)^a$ computational steps, and*
($1''$.ii) *V outputs YES.*

$2''$. Computational soundness. *For all $\widetilde{q} \notin \mathcal{L}$, for all k such that $2^k > |q|^b$, and for all (cheating) interactive circuit $\widetilde{P}$ of size $\leq 2^{ck}$, in a random execution of $\widetilde{P}$ with V on inputs $\widetilde{q}$ and 1^k,*

$$Prob[V \text{ outputs } YES] < 2^{-dk}.$$

The constructability of interactive CS proof systems. Let us make the mentioned notion of a collision-free hash function a bit more formal.

DEFINITION 4.2. *Let KG (for key generator) be a probabilistic polynomial-time algorithm, $KG : 1^* \to \Sigma^*$, and let E (for evaluator) be a polynomial-time algorithm, $E : \Sigma^* \times 1^* \to \Sigma^*$ (more precisely, $E : \Sigma^* \times 1^k \to \Sigma^k$ for all positive integers k). We say that the pair (KG, E) is a* collision-free hash function *if there exist positive constants r, s, and t such that for all $k > r$ and for all (collision-finding) circuits CF of size $< 2^{sk}$, letting h be a random output of KG on input k,*

$$Prob_h[CF(h, 1^k) = (x, y) \text{ such that } x \neq y \ \wedge\ E(h, x) = E(h, y)] < 2^{-tk}.$$

From the proof of Theorem 3.9 (indeed, even from the informal arguments of subsection 3.4) the reader can easily derive a proof of the following corollary.

COROLLARY 4.3. *Interactive CS proof systems exist if collision-free hash functions exist.*

(Notice that such CS proof systems would actually be four-round ones: in essence, the verifier runs KG to generate a random h and sends it to the prover as the first message; the prover uses h to construct the Merkle tree and sends its root to the

[24]An ITM making, say, at most s steps, may de facto make "many more steps" if it has a large description—e.g., by encoding a large finite function in its state control. 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 system, a cheating prover is envisaged to be an algorithm whose running time and description, in some standard encoding, are both bounded. Having a cheating prover be a Boolean combinatorial circuit with a bounded number of gates is actually just a specific but simple way to accomplish this.

verifier as the second message; the verifier runs the sampling algorithm SV to compute a sequence of bit-positions and sends them to the prover as the third message; and the prover returns to the verifier the bit-values of those positions—together with their authenticating paths—as the fourth message.)

Two-round CS proof systems. Informally, a *two-round CS proof system* is an interactive CS proof system (P, V) in every execution of which only two messages are sent: the first by V and the second by P. Such proof systems appear significantly more difficult to implement than general interactive ones. Nonetheless they could be constructed based on Cachin, Micali, and Staedler's new computationally private information-retrieval system [14]. Their construction relies on the difficulty of deciding, given a prime p and an integer n (whose factorization is unknown), whether p divides $\phi(n)$.

4.2. CS proof-systems sharing a random string.

The notion of a CS proof system sharing a random string. In a CS proof system sharing 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. If string r is universally known, it can be shared by all provers and verifiers. (CS proof systems sharing a random string are a special case of one-round CS proof systems because r could be the message sent by the verifier to the prover.)

DEFINITION 4.4. *Let (P, V) be a pair of Turing machines, the second of which runs in polynomial time. We say that (P, V) is a* CS proof system sharing a random string *if there exists a sequence of five positive constants, $c_2, \ldots, c_6$ (referred to as the* fundamental constants *of the system*[25]*), such that the following two properties are satisfied.*

$1'''$. *Feasible completeness.* For all $q = (M, x, y, t) \in \mathcal{L}$ and for all binary string r, $(1'''.i)$ on inputs q and r, P halts within $(|q| \cdot |r| \cdot t)^{c_2}$ computational steps outputing a binary string $\mathcal{C}$, whose length is $\leq (|q| \cdot |r|)^{c_3}$, such that $(1'''.ii)$ $V(q, r, \mathcal{C}) = YES$.

$2'''$. *Computational soundness.* For all $\widetilde{q} \notin \mathcal{L}$, for all k such that $2^k > |q|^{c_4}$, and for all (cheating) circuits $\widetilde{P}$ whose size is $\leq 2^{c_5 k}$, for a random k^{c_1}-bit string r

$$Prob_r[\widetilde{P}(\widetilde{q}, r) = \widetilde{\mathcal{C}} \wedge V(\widetilde{q}, r, \widetilde{\mathcal{C}}) = YES] \leq 2^{-c_6 k}.$$

We refer to the above strings r and $\mathcal{C}$ as, respectively, a *reference string* and a *CS certificate (of $q \in \mathcal{L}$, relative to r and (P, V)).*

The constructability of CS proof systems sharing a random string. We conjecture that CS proof systems with a random string exist. In particular, their existence is guaranteed by an ad hoc (and stronger) assumption: informally the "replaceability," in Theorem 3.6's $(\mathcal{P}, \mathcal{V})$, of random oracles with (hopefully) adequate functions (e.g., collision-free hashing ones). Such replacements have been advocated,

[25]The "numbering" of these constants has been chosen to facilitate comparison with CS proof systems with a random oracle.

in more general contexts, by Bellare and Rogaway [7].[26]

5. Computationally sound checkers. CS proofs have important implications for validating one-sided heuristics for $\mathcal{NP}$. Generalizing a prior notion of Blum's, we put forward the notion of a *CS checker* and show that CS proofs with a random string imply CS checkers for $\mathcal{NP}$.

To begin with, we state the general problem of heuristic validation we want to solve, explain why prior notions of checkers may be inadequate for solving it, discuss the novel properties we want from a checker, and convey in a simple but *wishful* manner what our checkers are. Then we shall approximate this wishful version of a CS checker by a more formal definition and construction.

Warning. Approaching meaningfully the problem of validating efficient heuristics for $\mathcal{NP}$-complete problems is nontrivial (as we shall see, it entails reconciling "two opposites"), and our particular approach to it may prove quite subjective (if not outright controversial). Nonetheless, the problem at hand is so crucial that we might be excused for putting forward some quite preliminary contributions and ideas.

5.1. The problem of validating one-sided heuristics for $\mathcal{NP}$.

A general problem. $\mathcal{NP}$-complete languages contain very important and useful problems that we would love to solve. Unfortunately, it is extensively believed that $\mathcal{P} \neq \mathcal{NP}$ and $\mathcal{NP} \neq \mathcal{C}o\text{-}\mathcal{NP}$, and thus that our ability of successfully handling $\mathcal{NP}$-complete problems is severely limited. Indeed, if $\mathcal{P} \neq \mathcal{NP}$, then no efficient (i.e., polynomial-time) algorithm may decide membership in an $\mathcal{NP}$-complete language without making any errors. Moreover, if $\mathcal{NP} \neq \mathcal{C}o\text{-}\mathcal{NP}$, then no efficient algorithm may, in general, prove nonmembership in an $\mathcal{NP}$-complete language by means of "short and easy-to-verify" strings.

In light of the above belief, the "best natural alternative" to deciding $\mathcal{NP}$-complete languages efficiently and conveying efficiently to others the results of our determinations consists of tackling $\mathcal{NP}$-complete languages by means of efficient *heuristics* that are *one-sided.* Here by "heuristic" we mean a program (emphasizing that no claim is made about its correctness) and by "one-sided" we mean that such a program, on input a string x, outputs either (1) a proper $\mathcal{NP}$-witness, thereby *proving* that x is in the language, or (2) the symbol NO, thereby *claiming* (without proof) that x is not in the language.

But for an efficient one-sided heuristic to be really useful for tackling $\mathcal{NP}$-complete problems we should *know when it is right.* Of course, when such an heuristic outputs an $\mathcal{NP}$-witness, we can be confident of its correctness on the given input. However, when it outputs NO, skepticism is mandatory: even if the heuristic came with an a priori guarantee of returning the correct answer on most inputs, we might not know whether the input at hand is among those. Thus, in light of the importance of $\mathcal{NP}$-

[26] A word of caution is now due about such replacements. For certain tasks, it is now known how to replace random oracles successfully with ordinary algorithms based on traditional assumptions. For instance, a random oracle provably is "collision-free," but collision-free hash functions can be built, say, under the assumption that the integer factorization or the discrete logarithm problems are computationally intractable. On the other hand, "random-oracle replacement" does not always work: Canetti, Goldreich, and Halevi [15] show that it is possible to construct special algorithms that behave very differently when given access to a random oracle than they do when given access to *any* pseudorandom function. (In light of their result, it should be possible to construct, somewhat artificially, some CS proof systems sharing a random oracle that can never be transformed into CS proof systems sharing a random string by replacing their oracle with a pseudorandom function. But this does not imply that the same holds for every CS proof system with a random oracle, in particular for the $(\mathcal{P}, \mathcal{V})$ of Theorem 3.9).

complete languages and in light of the many efficient one-sided heuristics suggested for these languages, a fundamental problem naturally arises.

> *Given an efficient one-sided heuristic H for an $\mathcal{NP}$-complete language, is there a meaningful and efficient way of using H so as to validate some of its NO outputs?*

Interpreting the problem. The solvability of the above general problem critically depends on its specific interpretation.

One such interpretation was proposed by Manuel Blum when, a few years ago, he introduced the beautiful notion of a checker [11][27], and asked whether $\mathcal{NP}$-complete languages are checkable. Under this interpretation, the general problem still is totally open. Moreover, as we shall argue below, unless this interpretation is suitably broadened, even a positive solution might have a limited usefulness.

In this paper we thus propose a new interpretation of the general problem and, assuming the existence of CS proofs with a random string, provide its first (and positive) solution.

5.2. Blum checkers and their limitations.

The notion of a Blum checker. Intuitively, a Blum checker for a given function f is an algorithm that either (a) determines with arbitrarily high probability that a given program, run on a given input, correctly returns the value of f at that input, or (b) determines that the program does not compute f correctly (possibly, at some other input). Let us quickly recall Blum's definition.

DEFINITION 5.1. *Let f be a function and C a probabilistic oracle-calling algorithm running in expected polynomial time. Then, we say that C is a Blum checker for f if, on input an element x in f's domain and oracle access to any program P (allegedly computing f), the following two properties hold.*

1. *If $P(y) = f(y)$ for all y (i.e., if P correctly computes f for every input), then $C^P(x)$ outputs YES with probability* 1.
2. *If $P(x) \neq f(x)$ (i.e., if P does not compute f correctly on the given input x), then $C^P(x)$ outputs YES with probability $\leq 1/2$.*

The probabilities above are taken solely over the coin tosses of C whenever P is deterministic, and over the coin tosses of both algorithms otherwise.

The above notion of a Blum checker slightly differs from the original one.[28] In particular, according to our reformulation any correct program for computing f immediately yields a checker for f, though not necessarily a useful one (because such a checker may be too slow, or because its correctness may be too hard to establish).[29]

Despite their stringent requirements, Blum checkers have been constructed for a variety of specific functions (see, in particular, the works of Blum, Luby, and Rubinfeld [12] and Lipton [25]).

Note that the notion of a checker is immediately extended to languages: an algorithm C is a Blum checker for a language L if it is a Blum checker for L's characteristic function. Indeed, the interactive proof systems of [26] and [33] yield Blum checkers

[27] We shall call his notion a *Blum checker* to highlight its difference from ours.

[28] Disregarding minor issues, Blum's original formulation imposes an additional condition, roughly, that C run asymptotically faster than the fastest known algorithm for computing f—or asymptotically faster than P when checking P. This additional constraint aims at rebuffing a natural objection: who checks the checker? The condition is in fact an attempt to guarantee, in practical terms, that C is sufficiently different from (and thus "independent" of) P, so that the probability that both C and P make an error in a given execution is smaller than the probability that just P makes an error.

[29] Thus, running a checker C (as defined by us) with a program P may be useful only if C is much faster than P, or if C's correctness is much easier to prove—or believe—than that of P.

for, respectively, any $\#\mathcal{P}$- or $\mathcal{PSPACE}$-complete language.[30]

Blum checkers vs. efficient heuristics for $\mathcal{NP}$-complete problems. We believe that the question of whether Blum checkers for $\mathcal{NP}$-complete languages exist should be interpreted more broadly than originally intended. We in fact argue that, even if they existed, Blum checkers for $\mathcal{NP}$-complete languages might be less relevant than desirable.

Definition 5.2 (informal). *We say that a Blum checker A for a function f is* irrelevant *if, for all efficient heuristics H for f, and for all x in f's domain, with high probability $A^H(x) = NO$ without ever calling H on input x.*

Note that, if $\mathcal{P} \neq \mathcal{NP}$, then no efficient heuristic for an $\mathcal{NP}$-complete language is correct on all inputs. Thus it is quite legitimate for a Blum checker for an $\mathcal{NP}$-complete language to output NO whenever its oracle is an efficient heuristic, without ever calling it on the specific input at hand: a NO-output simply indicates that the efficient heuristic is incorrect on some inputs (possibly different from the one at hand). However, constructing an irrelevant Blum checker for the characteristic function of SAT (the language consisting of all Boolean formulae in conjunctive normal form) under the assumption that $\mathcal{P} \neq \mathcal{NP}$ is not trivial. The difficulty lies in the fact that a checker does not know whether it is accessing a polynomial-time program (in which case, if $\mathcal{P} \neq \mathcal{NP}$, it could always output NO), or an exponential-time program that is correct on all inputs (in which case it should always output YES). We can, however, construct such an irrelevant Blum checker under the assumption that one-way functions exist. This assumption appears to be stronger than $\mathcal{P} \neq \mathcal{NP}$ but is widely believed and provides the basis of all modern cryptography.

Definition 5.3 (informal). *We say that a function f mapping binary strings to binary strings is* one-way *if it is length-preserving, polynomial-time computable, but not polynomial-time invertible in the following sense: for any polynomial-time algorithm A, if one generates at random a sufficiently long input z and computes $y = f(z)$, then the probability that $A(y)$ is a counter-image of f is miniscule.*

Theorem 5.4 (informal). *If one-way functions and Blum checkers for $\mathcal{NP}$-complete languages exist, then there exist irrelevant Blum checkers for $\mathcal{NP}$-complete languages.*

Proof (informal). Let SAT be the $\mathcal{NP}$-complete language of all satisfiable formulae in conjunctive normal form, let P be a program allegedly deciding SAT, let C be a Blum checker for SAT, let f be a one-way function, and let $\mathcal{C}$ be the following oracle-calling algorithm.

On input an n-variable formula F in conjunctive normal form, and oracle access to P, $\mathcal{C}$ works in two stages. In the first stage, $\mathcal{C}$ randomly selects a (sufficiently long) string z and computes (in polynomial time) $y = f(z)$. After that, $\mathcal{C}$ utilizes the completeness of SAT to construct, and feed to P, n formulae in conjunctive normal form, $F_1, \ldots, F_n$, whose satisfiability "encodes a counter-image of y under f."

(For instance, F_1 is constructed so as to be satisfiable if and only if there exists a counter-image of y whose first bit is 0. The checker feeds such an F_1 to P. If P outputs "F_1 is satisfiable," then $\mathcal{C}$ constructs F_2 to be a formula that is satisfiable if and only if there exists a counter-image of y whose 2-bit prefix is 00. If, instead, P responds

[30]In fact, the definition of a Blum checker for a language L is analogous to a restricted kind of interactive proof for L: one whose prover is a probabilistic polynomial-time algorithm with access to an oracle for membership in L. Indeed, whenever a language L possesses such a kind of interactive proof system, a checker C for L is constructed as follows. On inputs P (a program allegedly deciding membership in L) and x, the checker C simply runs both prover and verifier on input x, giving the prover oracle access to program P. C outputs YES if the verifier accepts, and rejects otherwise.

"F_1 is not satisfiable," then $\mathcal{C}$ constructs F_2 to be a formula that is satisfiable if and only if there exists a counter-image of y whose 2-bit prefix is 10. And so on, until all formulae $F_1, \ldots, F_n$ are constructed and all outputs $P(F_1), \ldots, P(F_n)$ are obtained.)

Because string y is, by construction, guaranteed to be in the range of f, at the end of this process one either finds (a) a counter-image of y under f, or (b) a proof that P is wrong (because if no f-inverse of y has been found, then P must have provided a wrong answer for at least one of the formulae F_i). If event (b) occurs, $\mathcal{C}$ halts outputting NO. Otherwise, in a second phase, $\mathcal{C}$ runs Blum checker C on input the original formula F and with oracle access to P. When C halts so does $\mathcal{C}$, outputting the same YES/NO value that C does.

Let us now argue that $\mathcal{C}$ is a Blum checker for SAT. First, it is quite clear that $\mathcal{C}$ runs in probabilistic polynomial time. Then there are two cases to consider.

1. *P correctly computes SAT's characteristic function.* In this case, a counter-image of y is found, and thus $\mathcal{C}^P$ does not halt in the first phase. Moreover, in the second phase, $\mathcal{C}$ runs Blum checker C with the same correct program P. Therefore, by property 1 of a Blum checker, C^P will output YES no matter what the original input formula F might be, and, by construction, so will $\mathcal{C}^P$. This shows that $\mathcal{C}$ enjoys property 1 of a Blum checker for SAT.
2. *$P(F)$ provides the wrong answer about the satisfiability of F.* In this case, either $\mathcal{C}^P$ halts in phase 1 outputting NO, or it executes phase 2 by running $C^P(F)$, that is, the original Blum checker for SAT, C, on the same input F and the same oracle P. Therefore, by property 2 of a Blum checker, the probability that $C^P(F)$ will halt outputting YES is no greater than 1/2. By construction, the same holds for $\mathcal{C}^P(F)$. This shows that $\mathcal{C}$ enjoys property 2 of a Blum checker for SAT.

Finally, let us argue that, for any input F and any efficient P (no matter how well it may approximate SAT's characteristic function), almost always $\mathcal{C}^P(F) = NO$, without even calling P on F. In fact, because $\mathcal{C}$ runs in polynomial time, whenever P is polynomial-time, so is algorithm $\mathcal{C}^P$. Therefore, $\mathcal{C}^P$ has essentially no chance of inverting a one-way function evaluated on a random input. Therefore, $\mathcal{C}$ will output NO in phase 1, where it does not call P on F. $\square$

In sum, differently from many other contexts, the notion of a Blum checker may not be too useful for handling efficient heuristics for $\mathcal{NP}$-complete languages, either because no such checkers exist[31] or because they may exist but not be too useful.

Blum checkers are not complexity-preserving. The lesson we derive from the above sketched proof of Theorem 5.4 is that Blum's notion of a checker lacks a new property that we name *complexity preservation*. Intuitively, a Blum checker for satisfiability, when given a "not-so-difficult" formula F, may ignore it altogether and instead call the to-be-tested efficient heuristic on *very special* and *possibly much harder* inputs, thus forcing the heuristic to make a mistake and justifying its own outputting NO (i.e., "the heuristic is wrong").

The possibility of calling a given heuristic H on inputs that are harder than the given one essentially erodes the chances of meaningfully validating H's answer whenever it happens to be correct.[32] Such a possibility may not matter much if "the

[31] Notice that this possibility does not contradict the fact that $\mathcal{NP}$ is contained in both $\#\mathcal{P}$ and $\mathcal{PSPACE}$ and that $\#\mathcal{P}$- and $\mathcal{PSPACE}$-complete languages are Blum checkable!

[32] Note that such possibility not only is present in the *definition* of a Blum checker but also in all known *examples* of a Blum checker. Typically, in fact, a Blum checker works by calling its given heuristic on random inputs, and these may be more difficult than the specific, original one.

difference in computational complexity between any two inputs of similar length" is somewhat bounded. But it may matter a lot if such a difference is enormous—which may be the case for $\mathcal{NP}$-complete languages, as they encode membership in both easy and hard languages. We thus wish to develop a notion of a "complexity-preserving" checker.

5.3. New checkers for new goals.

The old goal. Blum checkers are very useful to catch the occasional mistake of programs believed to be correct on all inputs. That is, they are ideally suited to check *fast programs for easy functions* (or slow program for hard functions). In fact, if f is an efficiently computable function, then we know a priori that there are efficient and correct programs for f. Therefore, if a reputable software company produces a program P for f, it might be reasonable to expect that P is correct. In this framework, by running a Blum checker for f, with oracle P, on a given input x we have nothing to lose[33] and something to gain. Indeed, if the checker answers YES, we have "verified our expectations" about the correctness of P at least on input x (a small knowledge gain), and if the checker answers NO, we have proved our expectations about P to be wrong (a big knowledge gain).

The new goal. We instead want to develop checkers for a related but different goal: validating efficient heuristics that are known to be incorrect on some inputs. That is, we wish to develop checkers suitable for handling *fast programs for hard functions.* Now, if f is a function hard to compute, then we know a priori that no efficient program correctly computes it. Therefore, obtaining from a checker a proof that such an efficient program does not compute f correctly would be quite redundant. We instead want checkers that, at least occasionally, if an efficient heuristic for f happens to be correct on some input x, are capable of convincing us that this is the case.

Interpreting the new goal. Several possible valid interpretations of this general constraint are possible. In this paper we focus on a single one: namely, we want checkers that are *complexity-preserving.* Let f be a function that is hard to compute (at least in the worst case). Then, intuitively, a complexity-preserving checker for f will, on input x, call a candidate program for f only on inputs for which evaluating f is essentially as difficult as for x.

Our point is that, while a given heuristic for satisfiability, $\mathcal{H}$, may make mistakes on some formulae, it may return remarkably accurate answers on some class of formulae (e.g., those decidable in $O(2^{cn})$ time, for some constant $c < 1$, by a given deciding algorithm D). Intuitively, therefore, checkers should be defined (and built!) so that, if the input formula belongs to that class and $\mathcal{H}$ happens to be correct on the input formula, they call $\mathcal{H}$ only on additional formulae in that class.

5.4. The wishful version of a CS checker.

The spirit of a CS checker is best conveyed wishfully assuming (for a second) that $\mathcal{NP}$ equaled $Co\text{-}\mathcal{NP}$. In that case, our CS checkers would take the following simple and appealing form.

Wishful checkers. Define a *wishful checker* to be a polynomial-time algorithm C that, on input a Boolean formula F, outputs a Boolean formula $\mathcal{F}$ satisfying the following two properties.

1. *Membership reversion.* $\mathcal{F} \in$ SAT if and only if $F \notin$ SAT.

[33]Blum checkers are often so fast (e.g., running in time sublinear in that of the algorithm they check) that not even this is much of a concern.

2. *Complexity preservation.* "The satisfiability of $\mathcal{F}$ is as hard to decide as that of F."

How to use a wishful checker. We interpret the above algorithm C as a kind of checker because it immediately yields the following algorithm C' (more closely matching our intuition of a checker).

C' : Given an efficient one-sided heuristic for SAT, H, and an input formula, F, compute $\mathcal{F} = C(F)$. Then, call H so as to obtain the two values $H(F)$ and $H(\mathcal{F})$. If either value is different than NO, then (H being one-sided) a satisfying assignment has been computed either proving that $F \in$ SAT or that $F \notin$ SAT. Otherwise, $H(F) = H(\mathcal{F}) = NO$ proves that H is incorrect.

Note that, by the very definition of a wishful checker, the above proof that H is incorrect has been obtained without querying H on formulae harder than the original input F.

5.5. The notion of a CS checker.

Let us now informally explain how, without assuming $\mathcal{NP} = Co\text{-}\mathcal{NP}$, CS checkers may approximate wishful ones to a sufficiently close extent. Renouncing to achieving greater generality, we limit our discussion to CS checkers for SAT.

CS checkers. Informally speaking, a *CS checker* is a polynomial-time algorithm $\mathcal{C}$ that, on input a formula F, outputs a Boolean formula $\mathcal{F}$, called the *coinput*, satisfying the following properties.

1. *Membership semireversion.*
   1.1. At least one of F and $\mathcal{F}$ is satisfiable.
   1.2. If F is satisfiable, then no efficient algorithm has a nonnegligible chance of finding a satisfying assignment for $\mathcal{F}$.
2. *Complexity semipreservation.* If $F \notin$ SAT, then the satisfiability of $\mathcal{F}$ is as hard to decide as that of F.
   (Note: We explain why complexity preservation is restricted to the "$F \notin$ SAT" case a few lines below.)

How to use CS checkers. We interpret the above $\mathcal{C}$ as a checker because it immediately yields the following algorithm $\mathcal{C}'$ (that better matches what we may intuitively expect from a checker).

$\mathcal{C}'$: Given an efficient one-sided heuristic for SAT, H, and an input formula, F, do the following.
   – Compute the coinput $\mathcal{F} = \mathcal{C}(F)$.
   – Call H so as to obtain $H(F)$.
   – If $H(F) \neq NO$, HALT.
   – If $H(F) = NO$, call H so as to obtain $H(\mathcal{F})$ and HALT.

The usefulness of CS checkers. The usefulness of the above $\mathcal{C}'$ stems from the following two properties.

(a) $\mathcal{C}'$ is informative about the satisfiability of F or the correctness of H.

(b) If H is correct on F, then $\mathcal{C}'$ never calls H on a formula harder than F.

Indeed, a computation of $\mathcal{C}'$ results in (1) showing a satisfying assignment of F, (2) showing a satisfying assignment of $\mathcal{F}$, or (3) showing that $H(F) = H(\mathcal{F}) = NO$.

A type-1 result clearly proves that $F \in$ SAT.

A type-2 result is interpretable as saying that F is unsatisfiable. This is so because if F belonged to SAT, then either a satisfying assignment of coinput $\mathcal{F}$ does not exist, or (by the very definition of a CS checker) the probability that it can be obtained in polynomial time is negligible. (Notice, in fact, that $\mathcal{C}'$ is efficient because both $\mathcal{C}$ and H are.)

A type-3 result proves that H is wrong. In fact, if $H(F) = NO$ is correct, then (by property 1.1 of a CS checker) $\mathcal{F} \notin \text{SAT}$, and thus $H(\mathcal{F}) = NO$ is incorrect. Let us now argue that *if H is correct on F*, our proof of H's incorrectness has been obtained in a complexity-preserving manner. We distinguish two cases.

1. If H is correct on F and $H(F) \neq NO$, then $H(F)$ is an (easy-to-verify) satisfying assignment of F, and thus $\mathcal{C}'$ does not call H on any coinput. Therefore, $\mathcal{C}$ vacuously does not call H on any $\mathcal{F}$ harder than F.
2. If H is correct on F and $H(F) = NO$, then $F \notin \text{SAT}$, in which case (by property 2 of a CS checker) $\mathcal{F}$ is guaranteed to have the same complexity as F.

If instead H is *not* correct about our original input F, then $H(F) = H(\mathcal{F}) = NO$ still is a proof of H's incorrectness, but not necessarily one obtained in a complexity-preserving manner. Notice, however, that lacking complexity preservation in this case is of no concern: *if H happens wrong about our own original input, we are happy to prove that H is wrong in any manner.* Recall that in checking we care about our own original input x more than about H. Thus if $H(x)$ is correct, we aim at "proving" this fact, and we do not want to throw H away by calling it on much harder inputs. But if $H(x)$ is wrong, we do not mind dismissing H in any way. Least of all, we want to be convinced that $H(x)$ is right!

The complexity preservation of a CS checker. To complete our informal discussion of CS checkers we must explain in what sense, whenever $F \notin \text{SAT}$, the complexity of F is close to that of $\mathcal{F}$. That is, we must explain (1) how we measure the complexity of the original input, and (2) how the coinput preserves this complexity.

1. *Complexity meters.* The complexity of the original input F is defined to be the number of steps made by a chosen deciding algorithm for SAT, D, on input F. That is, when a CS checker for SAT is given an input formula F, it is also given as an additional input the description of this chosen D. We refer to D as *the complexity meter.* In fact, by specifying D, we (implicitly) pin down the complexity of the original formula F. By insisting that D be a decider for SAT (i.e., that D be correct) we insist that the complexity of the original input be a "genuine" one.[34]

   By properly choosing the complexity meter, one may be able to force the complexity of the original input to be small (and thus force the checker to query its given heuristic on a coinput of similarly small complexity). Choosing D to be the algorithm that tries all possible satisfying assignments for F is certainly legitimate but not too meaningful. (Because any formula would have "maximum complexity" relative to such a complexity meter, the checker would essentially be free to call its given heuristic on any possible coinput.) Quite differently, if the original input F is known to belong to a class of formulae for which a given SAT algorithm performs very well (e.g., runs in subexponential time), by specifying that algorithm as our complexity meter, we force the checker to call its given heuristic only on a coinput of similarly low complexity.

   Let us stress that we do not require that the checker, or someone choosing a complexity meter D, know how many steps D takes on the original input F. Nor do we require that one distinguish (somehow) for which inputs, if any, algorithm D (slow in the worst case) may be reasonably fast. Rather, we require that, if F happens to belong to those inputs on which D is fast, then

[34]In particular, if D were allowed to make errors, all formulae F could have constant complexity.

it is this lower (and possibly unknown) complexity that should be preserved by a CS checker.

By specifying D one specifies implicitly *the complexity of F, whatever it happens to be.*

For technical reasons, however, we require that D's running time be upper-bounded by 2^{2n}, a bound that essentially poses no real restrictions. Within these bounds, any algorithm for satisfiability could always be "timed-out" and then converted to an exhaustive search for a satisfying assignment).

2. *Complexity cometers.* The complexity of a coinput $\mathcal{F}$ is defined to be the number of steps taken on input $\mathcal{F}$ by a decider for SAT, $\mathcal{D}$, specified before hand. We refer to such a $\mathcal{D}$ as *the complexity cometer.*
   Thus a complexity cometer is independent of the chosen complexity meter: the first is fixed once and for all (in fact, it could be made part of the very definition of a CS checker), while the second is chosen afresh each time a CS checker is run. Under these circumstances, at first glance, it may appear surprising that a CS checker may succeed in keeping the complexity of the coinput close to that of the original input. But the fixed cometer $\mathcal{D}$ includes the code of the universal algorithm, so that, in a sense, the complexity of a coinput is measured relative to a "decider for SAT that is easily constructed on input D."
   Notice that one could conceive stating complexity preservation by simply saying that the number of steps taken by a chosen D on the original input is polynomially close to the number of steps taken by the same D on the coinput. This would be a simpler way of having the cometer easily depend on the meter. However, we needed to endow CS checkers with a bit more room to maneuver than that. In any case, we believe it preferable to have the meter that is a fixed component of the CS checker to be a *universal meter.*

5.6. The actual definition of a CS checker.

Preliminaries.

- We let CNF denote the language of all formulae in conjunctive normal form, and SAT the set of all satisfiable formulae in CNF. If $F \in \text{SAT}$, then we denote by $\text{SAT}(F)$ the set of all satisfiable assignments of F. For any positive integer n, CNF_n and SAT_n will denote, respectively, all formulae in CNF and SAT whose binary length is n.
- By an SAT *decider* we mean an algorithm that (correctly) decides the language SAT. (Deciders need not output a satisfying assignment in case the input formula is satisfiable.) We say that an SAT decider D is *reasonable* if, for all $F \in \text{CNF}$, $\#D(F) \le 2^{|F|}$.
- If A is a probabilistic algorithm and E an event (involving executions of A on specified inputs), by $Prob_A[E]$ we denote the probability of E, taken over all possible coin tosses of A.

DEFINITION 5.5. *Let Φ be a probabilistic polynomial-time algorithm, $\mathcal{D}$ an SAT decider, and $\mathcal{Q}(.,.,.,.)$ a positive polynomial. We say that $(\Phi, \mathcal{D}, \mathcal{Q})$ is a* CS checker *if, for all positive constant c, for all CNF formulae F, for all reasonable SAT deciders D, and for all sufficiently large k, on input $(F, D, 1^k)$ Φ outputs a formula $\mathcal{F}$ such that:*

1. $F \vee \mathcal{F} \in \text{SAT}$;
2. $F \in \text{SAT} \Rightarrow$ *for all k^c-size circuits A, $Prob_\Phi[A(\mathcal{F}) \in \text{SAT}(\mathcal{F})] < 2^{-k}$; and*
3. $F \notin \text{SAT} \Rightarrow \#\mathcal{D}(\mathcal{F}) < \mathcal{Q}(|F|, |D|, 1^k, \#D(F))$.

If $\mathcal{C} = (\Phi, \mathcal{D}, \mathcal{Q})$ is a CS checker, we refer to Φ as the *reducer*, to $\mathcal{D}$ as the *complexity cometer*, and to $\mathcal{Q}$ as the *complexity slackness.*

Remark. Note that even the existence of triplets $(\Phi, \mathcal{D}, \mathcal{Q})$ satisfying just properties 1 and 2 alone constitutes a surprising statement about SAT. Informally, they say that any formula F can be efficiently transformed to a formula $\mathcal{F}$ such that (1) at least one of them is satisfiable, while (2) every efficient algorithm can find a satisfying assignments for at most one them. That is,

properties 1 *and* 2 *"almost" say that* $\mathcal{NP} = Co\text{-}\mathcal{NP}$,

in the sense that, to convince a verifier V that a formula F is not satisfiable, a prover P may first run Φ on input $(F, D, 1^k)$ so as to compute $\mathcal{F}$, and then (because if $F \notin$ SAT, then, by property 1, $\mathcal{F}$ is satisfiable) produce a satisfying assignment for $\mathcal{F}$. The verifier is convinced because, if also F were satisfiable, then a satisfying assignment for F could be found in less than 2^k steps, which would violate property 2 whenever P is poly(k) size, and k is large enough!

5.7. Implementing CS checkers for SAT. Let us recall some known properties of Cook's [16] and Levin's [24] $\mathcal{NP}$-completeness constructions.

Key properties of Cook's and Levin's constructions. Given a polynomial-time predicate $A(\cdot,\cdot)$ and a positive constant b, these constructions consist of a polynomial-time algorithm that, on input a binary string x, outputs a CNF formula ϕ that is satisfiable if and only if there is a binary string σ such that $|\sigma| \le |x|^b$ and $A(x, \sigma) = YES$. We refer to such a string σ as a *witness (for x).* The construction further enjoys the following extra properties (which are actually required by Levin's definition of $\mathcal{NP}$-completeness):

(i) x is polynomial-time retrievable from ϕ;

(ii) a proper witness for x is polynomial-time computable from any satisfying assignment for ϕ (if one exists); and

(iii) a satisfying assignment for ϕ is polynomial-time computable from any proper witness for x (if one exists).

THEOREM 5.6. *If CS proof systems sharing a random string exist, then CS checkers for SAT exist.*

Proof. Let (P, V) be a CS proof system sharing a random string with fundamental constants $c_2, \ldots, c_6$, and consider the following algorithm.

ALGORITHM Φ

Inputs: F, a CNF formula, D, a reasonable SAT solver, and 1^k, a security parameter.

Subroutines: P and V.

Output: a CNF formula $\mathcal{F}$.

Code: Randomly select a k-bit (reference) string r for (P, V), and use Cook's (or Levin's) construction to compute a CNF formula $\mathcal{F}$ that is satisfiable if and only if there exist two binary strings t and σ such that, setting $q = (F, D, NO, t)$, the following three properties hold: (1) $|t| \le 2|F|$,[35] (2) $|\sigma| \le (|q| \cdot k)^{c_3}$, and (3) $V(q, r, \sigma) = YES$.
{*Comment:* If it exists, σ is a CS certificate of $(D, F, NO, t) \in \mathcal{L}$, relative to (P, V) and reference string r. The existence of such a σ, however, does not guarantee that $D(F) = NO$.[36]}

[35] i.e., because D is reasonable, considering t as an integer, $t = \#D(F) \le 2^{|F|}$.

[36] In fact, we "expect" that σ exists (and thus that $\mathcal{F}$ is satisfiable) with "overwhelming probability," even when F is satisfiable. But σ is hard to find.

Let us now show that there exist an SAT decider $\mathcal{D}$ and a positive polynomial $\mathcal{Q}$ such that $\mathcal{C} = (\Phi, \mathcal{D}, \mathcal{Q})$ is a CS checker.

To begin with, notice that, because of the polynomiality of V and of Cook's construction, Φ is polynomial-time.[37]

Further, because properties 1 and 2 of a CS checker (as per Definition 5.5) only depend on its reducer, let us show that they hold for our Φ prior to defining $\mathcal{D}$ and $\mathcal{Q}$.

Property 1 holds trivially if $F \in$ SAT. Assume therefore that $F \notin$ SAT. Then, because of the correctness and running time of the complexity meter D, we have $D(F) = NO$ within $t \le 2^{2n}$ steps. Thus, by the (feasible) completeness of (P, V) for any possible reference string r there is a CS certificate σ of $q = (D, F, NO, t) \in \mathcal{L}$. Thus, $\mathcal{F} \in$ SAT, proving that property 1 holds in all cases.

Property 2 is established by contradiction. Assume that there exists an input formula $F \in$ SAT and a poly(k)-size circuit A that, with nonnegligible probability, computes a satisfying assignment of a so-constructed coinput $\mathcal{F}$. Then, by property (ii) of Cook's construction, from such a satisfying assignment (if it exists and is found) one computes in polynomial time both t and a CS certificate σ of $q = (D, F, NO, t) \in \mathcal{L}$. But if $F \in$ SAT, then for no t is $q = (D, F, NO, t) \in \mathcal{L}$. Therefore, this contradicts the computational soundness of (P, V).

Let us finally show that there exist an SAT decider $\mathcal{D}$ and a positive polynomial $\mathcal{Q}$ such that, for all formulae $F \notin CNF$, for all complexity meters D, and for all security parameters k, if D, on input F, takes t ($\le 2^{2|F|}$) steps to decide that no satisfying assignment for F exists, then, given any coinput $\mathcal{F}$ of F, $\mathcal{D}$ finds a satisfying assignment for $\mathcal{F}$ in at most $Q(|F|, |D|, k, t)$ steps.

Algorithm $\mathcal{D}$ works in four phases as follows.

$\mathcal{D}1$. Computes F, D, and r from $\mathcal{F}$.
(Due to property (i) of Cook's construction, $\mathcal{D}$ can execute this phase in time polynomial in $|\mathcal{F}|$. Thus, because $\mathcal{F}$ has been computed by $\mathcal{C}$ in time polynomial in $|F|$, $|D|$, and k, this phase is executable in time polynomial in $|F|$, $|D|$, and k.)

$\mathcal{D}2$. Runs D on input F to find the exact number of steps, t, taken by D to output NO on input F.
(Because D can be simulated with a slow-down polynomial in $|D|$, this phase takes time polynomial in $|D|$ and t.)

$\mathcal{D}3$. Run prover P on input $q = (D, F, NO, t)$ and reference string r to produce a CS certificate, σ, of $q \in \mathcal{L}$.
(Due to the feasible completeness of (P, V), this phase is executable in time polynomial in $|q|$, k, and t; and thus in time polynomial in $|F|$, $|D|$, k, and t.)

$\mathcal{D}4$. Use σ to compute a satisfying assignment for $\mathcal{F}$.
(Due to property (iii) of Cook's construction, this phase also can be implemented in time polynomial in $|F|$, $|D|$, and k.)

Because each phase is implementable in time polynomial in $|F|$, $|D|$, k, and t, there exists a polynomial $\mathcal{Q}$ such that $\mathcal{D}(\mathcal{F})$ outputs a satisfying assignment of $\mathcal{F}$ in $Q(|F|, |D|, k, t)$ steps.

[37]Indeed, define $A(\cdot,\cdot)$ as follows: $A((F, D, r), (t, \sigma)) \overset{\text{def}}{=} V((D, F, NO, t), r, \sigma)$. Notice now that A is polynomial-time: in fact, V is the verifier of a CS proof system with a random string. Notice also that $|\sigma|$ is polynomially bounded in $|F|$, $|D|$, and $|r|$: in fact $q = (D, F, NO, t)$, $|t| \le 2|F|$, and $|\sigma| \le (|q|k)^{c_3}$.

Finally, notice that the above four-phase procedure can be converted to an SAT decider by interleaving two different computations. In the first, an exhaustive search is conducted for deciding whether $\mathcal{F}$ is satisfiable. In the second, $\mathcal{F}$ is interpreted as a coinput of F, and the above four-phase procedure is run. The so-modified $\mathcal{D}$ halts when either computation halts and outputs what the halting computation does. $\square$

5.8. Remarks.

An alternative formulation. As we said, any CS checker $\mathcal{C}$ (as per Definition 5.5) immediately yields an oracle-calling algorithm that, on input a formula F (a complexity meter D, and a security parameter k) and access to a one-sided efficient heuristic H, computes a coinput $\mathcal{F}$ and obtains $H(F)$ and $H(\mathcal{F})$.

With this in mind, we can rephrase Theorem 5.6 as follows (and obtain—implicitly—a definition of a CS checker that is more closely tailored to our implemetation).

COROLLARY 5.7. *If CS proof systems sharing a random string exist, then there exist* (1) *a polynomial-time oracle-calling algorithm* $\mathcal{C}^{(\cdot)}(\cdot,\cdot,\cdot)$ *that, whenever its first input is a CNF formula F, queries its oracle at most twice: once about F, and possibly a second time about a second CNF formula* $\mathcal{F}$*;* (2) *an SAT decider* $\mathcal{D}$ *and a polynomial* $Q(.,.,.,.)$ *such that*

> *for all one-sided heuristics H for SAT, for all* $F \in$ *CNF, for all reasonable SAT deciders D solving F in* $\leq 2^{2|F|}$ *steps, for all sufficiently long random binary strings r, the following two properties hold.*

rm 1. Individual-complexity preservation. *If H is correct on F and* $\mathcal{C}^H(F,D,r)$ *queries H about a second CNF formula* $\mathcal{F}$, *then*

$$\#\mathcal{D}(\mathcal{F}) \leq Q(|F|,|D|,|r|,\#D(F)).$$

2. Computational meaningfulness. $\mathcal{C}^H(F,D,r)$ *produces one of the following three outputs:*
   (a) *a satisfying assignment for F*
   (i.e., a proof that F is satisfiable),
   (b) *a CS proof, relative to* (P,V) *and reference string r, of* $D(F) = NO$
   (i.e., evidence that F is not satisfiable),
   (c) *a formula* $\mathcal{F}$ *such that, by construction, either F or* $\mathcal{F}$ *is satisfiable, and yet* $H(F) = H(\mathcal{F}) = NO$
   (i.e., a proof that H is not correct).

Unlike Blum checkers, the above oracle-calling algorithm $\mathcal{C}$ does not provide answers that are correct with arbitrarily high probability (computed over its possible coin tosses). The type-(a) and type-(c) outputs of $\mathcal{C}$ are errorless, at least in the sense that any error here can be efficiently detected. But a type-(b) output of $\mathcal{C}$, interpreted as a (computationally) meaningful explanation that F is nonsatisfiable, may be wrong in a noneasily detectable manner: if F is satisfiable, $\mathcal{C}$ could output a "false" CS proof of $D(F) = NO$ with positive probability. However, this probability is reasonably high only if an enormous amount of computation is performed; whereas, in our application, all computation is performed by $\mathcal{C}$ which is polynomial-time and by oracle H which is also polynomial-time. Therefore, the probability of a false type-(b) output is absolutely negligible.

Another advantage of one-sided heuristics. Our CS checkers only deal with one-sided heuristics for SAT. As already discussed, given the one-sided nature of $\mathcal{NP}$, this is a natural choice. On the other hand, could we have dealt with heuristics just outputting YES (i.e., "satisfiable, but with no proof") or NO?

So far, because of the self-reducibility property of $\mathcal{NP}$-complete problems, choosing between either type of heuristics has often been a matter of individual taste. Indeed, it is well known that a decision oracle for SAT can, in polynomial time, be converted to a search oracle for SAT. As we explain below, however, this "equivalence" between decision and search relative to $\mathcal{NP}$-complete languages may cease to hold when one demands, as we do, that our reductions preserve the complexity of individual inputs, rather than just that of complexity classes.

When dealing with one-sided efficient heuristics H for SAT, assuming that H is correct on F, we need only to take care of complexity preservation when $H(F) \neq NO$. In fact, if $H(F) \in SAT(F)$, then there is no need to call H on any coinput $\mathcal{F}$, and thus there is no complexity to be preserved. Presumably, however, if H outputs just YES and NO, we would care about preserving F's complexity also when $H(F)$ is correct and $H(F) = YES$. Now, to convince ourselves that $H(F) = YES$ is correct, we could run the self-reducibility algorithm, calling H on a sequence of formulae $F_1, \ldots, F_n$ (obtained by "fixing" a new variable each time), so as to find a satisfying assignment of F, or prove that H is wrong (on either F or some F_i). The problem is, however, that this self-reducibility process may not be complexity-preserving: It may be the case that our F is relatively easy, while some of the F_i's are very hard. Indeed, it is conceivable that it is the "degree of freedom" of the variables of the original formula F that make it easy to decide (without finding any $\mathcal{NP}$-proof of it) that F is satisfiable. However, after sufficiently many variables of F have been fixed, the difficulty of deciding satisfiability may grow dramatically high (though later on, when sufficiently many variables have been fixed, it will dramatically drop).

Extra complexity preservation. Notice that, in the implementation of the proof of Theorem 5.6, the CS checker preserves the complexity of the original input F in a much closer manner than demanded by our definition. Indeed, the coinput $\mathcal{F}$ consists of the very encoding of the computation of the complexity meter D on input F (and we wonder whether this may yield a preferable formulation of complexity preservation).

Additional applications. We believe that complexity preservation, in different formulations, will be useful to other contexts as well. In particular, it will enhance the meaningfulness of many reductions in a complexity setting. For instance, using complexity preservation, [22] presents a more refined notion of a proof of knowledge [21, 36, 18, 6].

An open problem. Is it possible to (define and) construct CS checkers that, when given an heuristic H and an input x, also receive a concise algorithmic representation of a ("nontrivial") set S and call H only on x and elements of S? Such checkers could still be allowed to output a proof that the given heuristic H is wrong. But, if H happens to be correct on S, and the given input happens to belong to S, they should output a "validation" for $H(x)$ (rather than a proof that H is wrong).

6. Certified computation.

In this section we reinterpret the results of sections 3 and 4 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 whenever we are confident in the design of a given algorithm A but less so in the physical computer that runs it. For instance, the computer hardware may be defective. Alternatively, the hardware may function properly, but the operating system may be flawed. Alternatively yet, the hardware and software may be fine, but some α rays may succeed in flipping a bit together with its controls, so that the original bit value is not restored.

Moreover, even assuming that the physical computer is correct, after running A on input x so as to obtain a result y, can we convince someone else that y is indeed equal to $A(x)$ without having him redo the computation himself?

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

DEFINITION 6.1 (informal). *A* certified-computation system *is a compiler-verifier 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.*
3. *For each input x, algorithm A' produces the same output y as A, but also a short and easily inspectable string C vouching that indeed $y = A(x)$ in the following sense.*

   If $A(x)$ outputs y in t steps, then $\mathcal{V}(A, x, y, t, C) = YES$. Otherwise, it is very hard to find a string σ such that $\mathcal{V}(A, x, y, t, \sigma) = YES$.

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 that $\mathcal{V}$ is quite efficient (and runs 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 "$y = A(x)$ in t steps." Let us explain. On input x, algorithm $A' = \mathcal{C}(A)$ first runs A on x so that, after some number t of steps, an output y is obtained. Then A' outputs y and a CS proof of $(A, x, y, t) \in \mathcal{L}$. Thus the length of such a CS proof will be polynomial in $\log t$ (and, of course, $|A|$, $|x|$, $|y|$, and some suitable security parameter k). The additional time required to produce this CS proof is comparable to the running time of A. (This holds if one uses the construction of Theorem 3.8 rather than a generic CS proof system.) Indeed, while the definition of CS proofs allows a polynomial relation between these running times, the sketched construction actually yields a linear-time relation!

An alternative notion of certified computation. Another type of certified computation was previously proposed in [3] as an application of $\mathcal{PCP}$. In essence, in their notion, $A'(x)$ outputs both $A(x)$ and a string τ that vouches the correctness of $A(x)$. Though τ may be extraordinarily long (τ's length exceeds the number of steps taken by A on input x), it could be inspected in $\mathcal{PCP}$-style, by reading and verifying only selectively few of its bits. As we have argued in section 2, however, ensuring that one is really working with precisely these few bits of τ requires an overall verification time that is linear in τ's length (and thus greater than the time necessary to run A on input x).

Pros and cons of certified computation. One may view some older algorithms as specialized forms of certified computation, (i.e., applicable to certain functions only). For instance, the classical extended GCD algorithm not only outputs the greatest common divisor c of two input integers a and b, but also two integers x and y such that $ax + by = c$. Blum views such extended output as a relatively simple way of checking the validity of c. That is, by checking that indeed (1) $ax + by = c$ and that (2) c divides both a and b, one verifies that there is no divisor of a and b greater than c, and so that c actually is the greatest common divisor. Thus x and y are a sort of certificate proving the correctness of c. This certificate, however, is not sufficiently short (with respect to a, b, and c) and not sufficiently easy to verify (with respect to ordinary GCD computation).

A certified computation system can, instead, be considered as an "extended" *universal* algorithm, in the sense that it shows that certified computability is not a property enjoyed only by some special functions (such as the GCD function) but is an *intrinsic property of computation.* It should be noticed that, in addition to such "universality," a certified computation system produces relatively shorter and more easily inspectable certificates (than, say, the special ones produced by the extended GCD algorithm), but has a weaker guarantee of correctness (i.e., false certificates exist but are hard to find).

Finally, it should be appreciated that our suggested certified computation system may be impractical (as it refers to the execution history of Turing machines). One way to improve its efficiency may consist of finding a more convenient, and yet sufficient for our purposes, version or representation of the execution history of an algorithm.

Assumptions and implementations. The constructability of certified computation systems is implied by CS proof systems with a random oracle or CS proof systems with a random string. As explained below, if the latter proof systems exist, then their use in the current context does not require trusting the randomness of the extra string or the computational limitation of the prover.

Indeed, in this application, the random string need not be agreed upon by both prover and verifier (by means of some possibly difficult negotiation) nor chosen by means of a trusted third party or process. When seeking reassurance that indeed $y = A(x)$, the user U 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."

In addition, the physical device D implementing algorithm $\mathcal{P}$ "sits on top of user U's desk" and produces its output within a "reasonable time" monitored by U. Therefore, user D knows for a fact that D does not comprise more than 2^k gates and that it takes less than 2^k steps of computation for a security parameter k set by U.

In sum, if CS proofs with a random string exist, they yield certified computation systems that de facto offer 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 hire a supercomputing 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, that is, a CS certificate of "$A(x) = y$ in t steps." Note that, because such a certificate also vouches for the number of steps taken by A's execution, it is easy for the supercomputing company to charge according to the amount of computation actually invested.

Certified computation may also facilitate the verification of certain mathematical theorems proven with the help of a computer search (as in the case of the four-color theorem). For instance, the proof may depend on a lemma stating that there is no sparse graph with less than fifty nodes possessing a given property, and the lemma could be proved by means of an exhaustive search taking a few years of computing. Often, algorithms performing an exhaustive search are sufficiently simple so that we can be confident of the correctness of their design. 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 and easily verifiable certificate of correct execution relative to a random oracle. If the security parameter were 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 $2^{-1,000}$.

7. Concluding remarks.

Are CS proofs really proofs? In our minds this question really goes together with an older one: do probabilistic algorithms [34, 30] 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 "harmless." 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 from the large historical contexts in which they have arisen, and we expect that they will change with the changing of these contexts. And we believe and hope that, with time, CS proofs will be regarded to be as natural as probabilistic computations.

Truths versus proofs. According to our highest-level goal, CS proofs propose a new relationship between proving and deciding. In the thirties, Turing suggested that establishing (to yourself) the truthfulness of a mathematical statement consists of running a proper (accepting) algorithm. Today, we suggest that proving (to others) a mathematical statement consists of *feasibly speeding up* the verification of the result of any given accepting algorithm. That is, (1) proofs should make verifying the result of any accepting computation exponentially faster than the same accepting computation, but (2) proofs should not be more time-consuming to find than the accepting computations whose result verification they wish to facilitate. This, in our opinion, is an appealing relationship between proving and accepting, and one that guarantees that proving is both a useful and a distinct notion.

Living with error. In order to guarantee this "feasible speed up," CS proofs replace the traditional notion of a proof with a *computational* one. While CS proofs of true statements always exist, are suitably short, and are feasibly found, proofs of false statements either do not exist or are extraordinarily hard to find. Indeed, a CS proof is a short string that can be thought of as a "compressed version" of a long accepting computation. But the same conciseness that gives CS proofs their distinctive advantage also causes them to "lose quality" with respect to the accepting computations they compress: it makes them vulnerable to the possibility of error (though in a controllable way).[38]

To be sure, the possibility of inconsistency should not be taken lightly. But after realizing that the coherence of a sufficiently rich mathematical system cannot be decided within it, perhaps we should switch to *managing error* rather than trying desperately to ban it!

Acknowledgments. This paper is dedicated to Oded Goldreich, friend, scientist, and vigilant guardian angel. Most sincere thanks also go to Allan Borodin, Steve Cook, 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, Leo Reyzin, Janos Makowski, and Ray Sidney for their useful comments.

REFERENCES

[1] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy, *Proof verification and hardness of approximation problems*, in Proceedings of the 33rd IEEE Conference on Foundations of Computer Science, IEEE Computer Society, Los Alamitos, CA, 1992, pp. 14–23.

[2] S. Arora and M. Safra, *Probabilistic checking of proofs*, in Proceedings of the 33rd IEEE Conference on Foundations of Computer Science, IEEE Computer Society, Los Alamitos, CA, 1992, pp. 2–13.

[3] L. Babai, L. Fortnow, L. Levin, and M. Szegedy, *Checking computations in polylogarithmic time*, in Proceedings of the 23rd ACM Symposium on Theory of Computing, 1991, pp. 21–31.

[4] L. Babai and S. Moran, *Arthur-Merlin games: A randomized proof system and a hierarchy of complexity classes*, J. Comput. System Sci., 36 (1988), pp. 254–276.

[5] M. Bellare and O. Goldreich, *On defining proofs of knowledge*, in Advances in Cryptology—CRYPTO 92, Lecture Notes in Comput. Sci. 740, Springer-Verlag, Berlin, 1993, pp. 390–420.

[6] M. Bellare and S. Goldwasser, *The complexity of decision versus search*, SIAM J. Comput., 23 (1994), pp. 97–119.

[7] M. Bellare and P. Rogaway, *Random oracles are practical: A paradigm for designing efficient protocols*, in Proceedings of the 1st ACM Conference on Computer and Communications Security, Fairfax, VA, 1993, pp. 62–73.

[8] M. Ben-or, S. Goldwasser, J. Kilian, and A. Wigderson, *Multi prover interactive proofs: How to remove intractability*, in Proceedings of the 20th ACM Symposium on Theory of Computing, Chicago, IL, 1988, pp. 113–131.

[9] M. Blum, A. De Santis, S. Micali, and G. Persiano, *Noninteractive zero-knowledge*, SIAM J. Comput., 20 (1991), pp. 1084–1118.

[10] M. Blum, P. Feldman, and S. Micali, *Noninteractive zero-knowledge proof systems and applications*, in Proceedings of the 20th ACM Symposium on Theory of Computing, Chicago, IL, 1988, pp. 103–112.

[11] M. Blum and S. Kannan, *Designing programs that check their work*, in Proceedings of the 21st ACM Symposium on Theory of Computing, Seattle, WA, 1989, pp. 86–97.

[12] M. Blum, M. Luby, and R. Rubinfeld, *Self-testing/correcting with applications to numerical problems*, J. Comput. System Sci., 47 (1993), pp. 549–595.

[13] G. Brassard, D. Chaum, and C. Crepeau, *Minimum disclosure proofs of knowledge*, J. Comput. System Sci., 37 (1988), pp. 156–189.

[14] C. Cachin, S. Micali, and M. Stadler, *Computational Private-Information Retrieval With Polylogarithmic Communication*, in preparation, 1998.

[38]Let me emphasize that, though very natural and useful, the computational boundedness of a cheating prover is not, per se, a goal of our 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 types of CS proofs exist when a cheating prover can compute for an unbounded amount of time.)

In sum, the notion of a true theorem has not changed. What we advocate changing is the notion of what it means to *prove that a theorem is true*.

[15] R. Canetti, O. Goldreich, and S. Halevi, *The random oracle methodology, revisited*, in Proceedings of the 13th Annual ACM Symposium on Theory of Computing, ACM, New York, 1998, pp. 209–218.

[16] S. Cook, *The complexity of theorem proving procedures*, in Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, ACM, New York, 1971, pp. 151–158.

[17] U. Feige, S. Goldwasser, L. Lovasz, S. Safra, and M. Szegedy, *Approximating clique is almost NP-complete*, in Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Los Alamitos, CA, 1991, pp. 2–12.

[18] A. Fiat and A. Shamir, *How to prove yourselves: Practical solutions of identification and signature problems*, in Advances in Cryptology—CRYPTO 86, Lecture Notes in Comput. Sci. 263, Springer-Verlag, Berlin, 1987, pp.186–194.

[19] L. Fortnow, J. Rompel, and M. Sipser, *On the power of multi-prover interactive protocols*, Theoret. Comput. Sci., 134 (1994), pp. 545–557.

[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. ACM, 38 (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 Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science, New York, NY, 1986, pp. 174–187.)

[21] S. Goldwasser, 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 Proceedings of the 17th Annual IEEE Symposium on Theory of Computing, IEEE Computer Society, Los Alamitos, CA, 1985, pp. 291–304. Earlier yet versions include *Knowledge Complexity*, submitted to the 25th IEEE Annual Symposium on the Foundations of Computer Science, 1984.)

[22] S. Halevi and S. Micali, *A Stronger Notion of Proofs of Knowledge*, manuscript, 1997.

[23] J. Kilian, *A note on efficient zero-knowledge proofs and arguments*, in Proceedings of the 24th Annual ACM Symposium on Theory of Computing, Victoria, Canada, 1992, pp. 723–732.

[24] L. Levin, *Universal sequential search problems*, Problems Inform. Transmission, 9 (1973), pp. 265–266.

[25] R. Lipton, *New directions in testing*, in Distributed Computing and Cryptography, J. Feigembaum and M. Merritt, eds., DIMACS Ser. Discrete Math. Theory Comput. Sci. 2, AMS, Providence, RI, 1991, pp. 191–202.

[26] C. Lund, L. Fortnow, H. Karloff, and N. Nisan, *Algebraic methods for interactive proof systems*, J. ACM, 39 (1992), pp. 859–868.

[27] R. Merkle, *A certified digital signature*, in Advances in Cryptology—CRYPTO 89, Lecture Notes in Comput. Sci., 435, Springer-Verlag, New York, 1990, pp. 218–238.

[28] S. Micali, *CS proofs*, in Proceedings ofthe 35th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Los Alamitos, CA, 1994, pp. 436–453. (An earlier version of this paper appeared as Technical Memo MIT/LCS/TM-510. Earlier yet versions were submitted to the 25th Annual ACM Symposium on Theory of Computing, 1993, and the 34th Annual IEEE Symposium on Foundations of Computer Science, 1993.)

[29] A. Polishchuk and D. Spielman, *Nearly-linear size holographic proofs*, in Proceedings of the 26th Annual ACM Symposium on Theory of Computing, Montreal, Canada, 1994, pp. 194–203.

[30] M. Rabin, *Probabilistic algorithms for testing primality*, J. Number Theory, 12 (1980), pp. 128–138.

[31] R. Rivest, *The MD5 Message-Digest Algorithm*, Internet Activities Board, Request for Comments 1321, 1992.

[32] *Secure Hash Standard*, Federal Information Processing Standards, Publication 180 (1993).

[33] A. Shamir, *IP = PSPACE*, J. ACM, 39 (1992), pp. 869–877.

[34] R. Solovay and V. Strassen, *A fast Monte-Carlo test for primality*, SIAM J. Comput., 6 (1977), pp. 84–85.

[35] M. Sudan, *Efficient Checking of Polynomials and Proofs and the Hardness of Approximation Problems*, Ph.D. Thesis, University of California at Berkeley, Berkeley, CA, 1992.

[36] M. Tompa and H. Woll, *Random self-reducibility and zero-knowledge interactive proofs of possession of information*, in Proceedings of the 28th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Los Alamitos, CA, 1987, pp. 472–482.