CS PROOFS

(EXTENDED ABSTRACT)

Silvio Micali*

Abstract

This paper puts forward a computationally-based notion of proof and explores its implications to computation at large.

In particular, given a random oracle or a suitable cryptographic assumption, we show that every computation possesses a short certificate vouching its correctness, and that, under a cryptographic assumption, any program for a $\mathcal{NP}$-complete problem is checkable in polynomial time.

In addition, our work provides the beginnings of a theory of computational complexity that is based on "individual inputs" rather than languages.

1 Introduction

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

Informally, a CS proof of a statement S consists of a short string σ, both easy to find and to verify, offering a strong computational guarantee about the verity of S. By "easy to find" we mean that a CS proof of a true statement can be computed in a time essentially comparable to that needed to decide the statement. By "easy to verify" we mean that the time necessary to inspect a CS Proof of a statement S is poly-logarithmically shorter than that required to decide S. 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 their "proofs" are practically impossible to find. The value of this new notion, of course, crucially depends on whether it can be sufficiently exemplified, and we shall prove that CS proofs can be explicitly constructed given a random oracle or a suitable cryptographic assumption.

* Laboratory for Computer Science, MIT, Cambridge, MA 02139.

In conceiving and constructing CS proofs, we have benefitted from the research effort concerning interactive and zero-knowledge proofs. In particular, probabilistically-checkable proofs [8] [9] and zero-knowledge arguments [7] have been the closest sources of inspiration for the new notion itself; while the works of [14] and [15], in particular, but also those of [23] and [17] —though all addressing somewhat different scenarios— have been crucial for its construction.

CS proofs provide, in my opinion, the most effective way to date for proving membership in Co-NP-complete and other hard languages. They also provide, in a new and meaningful framework, very natural answers to some of our oldest questions in complexity theory (e.g., whether verifying is easier than deciding). Moreover, they yield novel and important implications for computational correctness. In particular, they imply that every computation possesses a short certificate vouching its correctness, and that any heuristic or program for a NP-complete problem is checkable in polynomial-time —an application that, at the same time, extends and demonstrates the wide applicability of Blum's [26] original framework for checking program correctness.

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

2 Our Goals for the Notion of a Proof

A fuller explanation of our goals, and a better comparison between our notion and preceeding ones can be found in Section 2 of reference [1].

Intuitively, our MAIN objective in introducing the notion of a CS proof is finding the right relationship between deciding and efficiently proving that a statement is true. This main objective is articulated in the following sub-goals.

$\mathcal{G}1$: *Convenience of verifying.* We should construct proof-systems so that verifying is poly-logarithmically easier than deciding for almost all theorems.

By contrast, $\mathcal{NP}$ and $\mathcal{IP}$ (1) express easyness of verification in "absolute terms," by requiring that Verifiers

0272-5428/94 $04.00 © 1994 IEEE

run in polynomial time, and (2) may, at best, guarantee that verification is poly-logarithmically easier than decision only for special members of special languages —e.g. (assuming that $\mathcal{NP} \neq \mathcal{P}$ in "the proper way"), for certain instances of satisfiability, since SAT is easy on most inputs.

$\mathcal{G}2$: *Feasibility of proving.* We should construct proof-systems so that the Prover's computational complexity is reasonably close to that of deciding.

By contrast, consider a language L, in $\mathcal{NP}$ but not $\mathcal{NP}$-complete, that is decidable in, say, $n^{\log n}$ time. Then, in the $\mathcal{NP}$ mechanism, to save the Verifier that many steps of computation, proving that a given x belongs to L entails finding a polynomially-long and polynomially-inspectable witness w_x. But the complexity necessary to find such a string may vastly exceed that of a running a decision algorithm D on x for $n^{\log n}$ steps. Indeed, it may be as high as $O(2^n)$, particularly if one reduces "$x \in L$" to a polynomially-longer instance of a $\mathcal{NP}$-complete problem. (Moreover, as we shall see below, "$\mathcal{NP}$ proving" may be infeasible even when the given language L happens to be $\mathcal{NP}$-complete.) A similar drawback is exhibited by the $\mathcal{IP}$ proof mechanism. Indeed, often the best way to prove membership in a IP-languange consisting of invoking the general $IP = PSPACE$ protocol [5] and [6], which is extremely wasteful of prover resources.

Infeasibility of proving is one of the main limitations of prior proof-systems with respect to applications. Indeed, in any practical application, the role of the Prover must be taken by a real person or a real physical device. Now, assuming that either agent succeeds in feasibly deciding a given statement (i.e., in convincing him/itself that the statement holds), if too much extra computation is required from the Prover to convince someone else, no person or physical device may be up to such a role, and no proof may effectively occur.

$\mathcal{G}2'$: *Individual inputs vs. complexity classes.* Refining our second goal, we should construct proof-systems so that, if an individual theorem x can be decided to be true in T computational steps by a given algorithm D, then the number of computational steps required from the Prover to show that x is true is upperbounded by "essentially the same" value T.

By contrast, as elucidated in the following remark, $\mathcal{NP}$ and $\mathcal{IP}$ emphasize classes of inputs rather than individual inputs.

Remark: Assume, for instance, that the $\mathcal{NP}$-complete language of all Hamiltonian graphs, $\mathcal{H}$, can be decided in time $O(2^{\sqrt{n}})$ by means of some algorithm D. Then, finding a $\mathcal{NP}$-proof that $G \in \mathcal{H}$ (e.g., a Hamiltonian tour in G) can be accomplished in $O(n^2 \cdot 2^{\sqrt{n}})$ steps: by calling D on graphs obtained from G by removing some of its edges. In traditional terms, this complexity can be taken to be "reasonably close to that of deciding." This is so because "complexity of deciding" is traditionally interpreted as asymptotically, the worst case across all possible inputs —i.e., whole of $\mathcal{H}$.

However, let me argue that the two complexities above may be very distant from one another if one takes an "individual-input" point of view. Indeed, it may be that, on input G, D actually halts in $n^{\log n}$ steps; while finding a Hamiltonian circuit in G by running D while deleting edges from G may require $2^{\sqrt{n}}$ steps. Indeed, it is conceivable that it is the "density" of edges of the original graph that helps D deciding that G has a Hamiltonian tour in a relatively easy manner; however, after sufficiently many edges of G have been removed, the difficulty of D's job may grow dramatically (though, towards the end, may drop again).

$\mathcal{G}3$: *Universality.* We should construct universal proof-systems, capable, that is, of efficiently proving membership in any semi-recursive language (rather than, say, in $PSPACE$-languages).

$\mathcal{G}4$: *Transferability.* We should construct proof-systems so that he who has verified the proof of a theorem x, is also enabled to prove x to anyone else in an easy manner.

(By contrast, the convinced verifier of an interactive proof for a theorem x may not be able to prove x to anyone else. This feature, while useful in cryptography, is limiting in other contexts.)

$\mathcal{G}5$: *Confidence.* We should construct proof-systems whose soundness does not depend on conditions that are not checkable to hold.

(By contrast, multi-prover interactive proof-systems [4] require that, while interacting with the same verifier, the Provers do not talk to one-another; something that the verifier can never check.)

$\mathcal{G}6$: *Good accounting.* We should construct proof-systems that are efficient according to concrete and realistic complexity measures.

(By contrast, probabilistically-checkable proofs count the complexity of verification assuming that a probabilistically-checkable proof has already been transfered, some-how and free-of-charge, to the Verifier's random-accessible memory. For more details, see [1], subsection 2.5.)

3 The Notion of a CS Proof

For meaningfully approximating the above goals, we wish to introduce a new notion, that of a CS proof.

Up to now, proof-systems have been requiring that all (deterministically or probabilistically) *provable* statements be *true*. CS proofs break with this tradition: they allow the existence of false proofs, but they ensure that they are computationally impossible to find. That is,

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

In practice, this shift of paradigm is absolutely *adequate*, and, judging from some of the practical consequences that it entails, *highly desirable*. (Indeed, we genuinely believe that the right notion is the one that allows us to prove the right theorem —especially if in a natural way.)

Accordingly, the Provers of a CS proof-system, though more powerful than their Verifiers, are themselves computationally bounded.

In other words, the transition from probabilistic proofs to CS proofs is analogous to the transition from (statistical) zero-knowledge to computational zero-knowledge, which has proved to be a more flexible and powerful notion.

CS PROOFS VS. ZERO-KNOWLEDGE ARGUMENTS. A related but different idea occurs in Brassard. Chaum, and Crépeau's notion of a zero-knowledge argument [7]. Following [2], a ZK argument provides an alternative way of proving membership in $\mathcal{NP}$-languages so as to hide the "unnecessary knowledge" contained in a $\mathcal{NP}$-proof. In their setting, both Prover and Verifier are polynomial-time machines. However, when they are given a member x of an $\mathcal{NP}$ language L as a common input, it is assumed that the Prover is also provided (on a special tape that is inaccessible to the Verifier) with a $\mathcal{NP}$-witness of $x \in L$. (In essence, the proof-system of [7] is an interactive protocol. The Prover sends the Verifier special encryption of the $\mathcal{NP}$-witness in his possession, and then handles queries of the Verifier by decrypting selective pieces of the witness. In order to guarantee perfect (rather than computational) zero-knowledgeness, the encryption used is not uniquely decodable, but the Prover must first, say, factor of find discrete logs before he can "choose" how to decrypt.) These proof-systems differ from CS proofs in many ways; in particular, they are interactive, do not produce short strings certifying long computations, are interested only in $\mathcal{NP}$, and, even within this class, do not guarantee goal $\mathcal{G}2$: feasibility of proving. (As we have pointed out, $\mathcal{NP}$-witnesses cannot be given for free: coming up with a $\mathcal{NP}$-witness of $x \in L$ (where $L \in \mathcal{NP}$) may be much harder than simply deciding that $x \in L$, since not all deciding algorithm produce a short and easily inspectable witness.) Indeed the aim of these proof-systems was zero-knowledgeness, not improving the domain of the provable.

3.1 Guaranteed CS proofs

We now wish to define a variant of the notion of a CS proof that can be exemplified *without* any complexity assumptions. In this variant, all Provers and Verifiers have access to a common random oracle. Actually, it suffices that they have common access to something less demanding: a random function from $\{0,1\}^k$ to $\{0,1\}^k$, where k is sufficiently large, but finite, security parameter.

Though somewhat impractical, this variant presents a main advantage:

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

If you are interested in truth and have no concern for time, you do not need proofs and provers: you may be equally happy to run a decision algorithm for establishing whether a given statement holds. Proofs are in fact mechanisms that aim at quickly and "critically" transfer truths that have been hard to obtain. Thus,

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

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 to the notion of a proof. But if it turned out that $\mathcal{P} = \mathcal{NP}$, this would only mean that $\mathcal{NP}$ did not provide an adequate approximation. Our intuition that proofs are an independently meaningful notion could not be shaken by any such a result. It is thus important to establish meaningful models in which we can show that proofs do exist as an independent notion.

Guaranteed CS proof-systems provide us with such a model. This model is further attractive because, as we shall see, it also achieves all the goals we have set forward in the last section.

WHERE DOES THE POWER COME FROM? Guaranteed CS proofs are based on a random oracle-functions. Now it should be noted that a random function $f : \Sigma^k \to \Sigma^k$ essentially consists of an exponentially-long (in k) string. Thus, having f available as an *oracle* rather than as a *string* allows one to have *polynomial-time* rather than *exponential-time* access to its bits. And it is precisely this exponential speed-up that will be converted, thanks to our specific construction, to the much more interesting exponential gap between decision and verification.

But, first, let us see what the exact definition of a guaranteed CS proof-system is.

3.1.1 Basic Definitions

This definition is presented below in a dry manner, without explanations or justifications. A "step-by-step" discussion of the definition of a guaranteed CS proof-system can be found in Section 3.2 (Micro Discussion) of reference [1], and a discussion of its high-level implications and significance can be found in Section 3.3 (Macro Discussion) of the same reference.

ORACLES AND ORACLE-CALLING ALGORITHMS. By an *oracle*, we mean a function $f : \Sigma^i \to \Sigma^i$, for some positive integer i. When the set Σ^i is given, by a *random oracle* I mean a randomly selected function from Σ^i into Σ^i.

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

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

this call would "cost" $poly(k)$ steps whenever $f : \{0,1\}^k \rightarrow \{0,1\}^k$.)

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

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

Definition: Let (P, V) be a pair of Turing machines capable of oracle calls, the second of which runs in polynomial-time. Pair (P, V) is executed in a special way. An execution of (P, V) starts with both machines having a common input $\vec{q}$ (presented in binary), a common security parameter k (presented in unary), and access to a common oracle f. The execution then continues by running P on $\vec{q}$ and k and access to f, so as to produce a binary output $\mathcal{C}$, and then running V on $\vec{q}, k$, and $\mathcal{C}$, and access to f.

We say that a pair (P, V) like above is a *guaranteed computationally-sound proof-system* (*guaranteed CS proof-system* for short) if there exist a sequence of 6 positive constants, $c_1, \ldots, c_6$ (refered to as the *fundamental constants* of the system), such that the following two properties are satisfied:

1'. *Feasible Completeness.* $\forall$ integers $n > 1$, $\forall$ n-bit input $\vec{q} = (\bar{M}, x, t) \in \mathcal{L}$, $\forall$ unary integers k, and $\forall$ oracle $f : \Sigma^{k^{c_1}} \rightarrow \Sigma^{k^{c_1}}$,

(*i*) P_f halts within $(nkt)^{c_2}$ computational steps, and outputs a binary string $\mathcal{C} = P_f(\vec{q}, k)$ whose length is $\leq (nk \log t)^{c_3}$, and

(*ii*) $V_f(\vec{q}, k, \mathcal{C}) = YES$.

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

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

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

3.1.2 Constructing Guaranteed CS Proofs

Our explicit construction of a specific CS proof-system, $(\mathcal{P}, \mathcal{V})$, utilizes probabilistically-checkable proofs and several ideas from cryptography, and zero-knowledge theory in particular. We start with a brief recollection of the original works on probabilistically-checkable proofs.

PROBABILISTICALLY-CHECKABLE AND SAMPLABLE PROOFS.

Very recently, Babai Fortnow, Levin and Szegedy [8] and Feige, Goldwasser, Lovasz, Safra and Szegedy [9] have put forward, independently and with different aims,[1] some related and important ideas sharing a common technique: *proof-samplability*, that is, an explicit algorithm transforming a $\mathcal{NP}$-witness, σ, into a new proof (i.e., string), τ, which is polynomially longer, but whose correctness can be detected by properly sampling it in a few locations rather than by reading it in its entirety.

The simpler technique of [9], called *probabilisticaly-checkable proofs* ever since the work of Arora and Safra [10], suffices for the goals of this paper. Actually, for simplicity purposes, we shall rely on the following "stripped-down" version of this technique, which we call *samplable proofs.*[2]

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

The reader familiar with the recent advancements on probabilisticall checkable proofs —in particular the beautiful papers of Arora and Safra [10], Arora, Lund, Motwani, Sudan, and Szegedy [11], Sudan [12], Polishuck and Spielman [13],— will realize that, for the sake of simplicity, we are sacrificing quite a bit of efficiency. Indeed, Polishchuk and Spielman show that a $\mathcal{NP}$-witness with length n possesses a probabilistically-checkable version that is only $n^{1+\epsilon}$-bit long for any constant $\epsilon > 0$. Nonetheless, we shall ignore polynomial improvements in the running time of sampling provers and poly-log improvements in the running time of sampling verifiers. Unlike for approximation theory, in fact, improvements in the

[1] The authors of [8] focus on proofs of membership in $\mathcal{NP}$-langguages, 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 [9] use proof-samplability to establish the difficulty of finding approximate solutions to important $\mathcal{NP}$-complete problems. (With this goal in mind, these other authors do not mind Verifiers working in time polynomial in the length of the input, and do not use or need that inputs appear in any special format.)

[2] Indeed, a probabilistically-checkable proof is a richer and more flexible notion, allowing one to specify an arbitrary number for the queries the Verifier may make, and for the coins it may toss. Moreover, the careful reader may realize that, even restricting one's attention to $PCP(\log(n)^{O(1)}, log(n)^{O(1)})$, the PCP techniques yield a reacher structure than samplable proofs; for instance, samplable proofs do not guarantee that any particular bit of the samplable proof is accessible by the Verifier with positive probability —only the final (and proper) ACCEPT and REJECT probabilities are guaranteed.

efficiency of proof-samplability will only affect the efficiency of CS proofs.

THE MAIN IDEAS IN THE CONSTRUCTION. We wish to construct a specific guaranteed CS proof-system, $(P_{\mathcal{L}}, V_{\mathcal{L}})$. We are thus assuming that both $P_{\mathcal{L}}$ and $V_{\mathcal{L}}$ access the same random oracle $\mathcal{R}$.

The universality of $(P_{\mathcal{L}}, V_{\mathcal{L}})$ (i.e., its capability of proving membership in any semi-recursive language) is based on the realization that the salient features of probabilistically-checkable proof-systems, although claimed for $\mathcal{NP}$ languages, actually hold in a more general context. Namely, a more careful reading of [9] and [8] shows the existence of two algorithms, pcP and pcV, satisfying the following properties. Let $R(x, y)$ be a polynomial-time relation, where y may be arbitrarily long with respect to x. Then, on inputs x and y, such that $R(x, y)$ holds, algorithm pcP, running in $poly(|x|, |y|)$ time, computes a string y' such that, on input x and random access to y', algorithm pcV, running in $poly(|x|, \log |y'|)$ time and flipping $polylog(|y|)$ coins, can verify (in their original sense) that the statement " $\exists y R(x, y)$" is true.

Keeping in mind this observation, the CS proof-system $(\mathcal{P}, \mathcal{V})$, on input $\vec{q} = (M, x, t) \in \mathcal{L}$, works as follows. First, $\mathcal{P}$ runs machine M on input x so as to generate, in t steps, the sequence of instantaneous configurations, σ, of an accepting computation of M. Such an history σ can be thought of as a proof that $M(x) = YES$. This proof will not be insightful, and, because no restriction is put on M, can be arbitrarily long relative to x. Consider now the following relation R: $R(x, \sigma) = 1$ if and only if σ is the history of an accepting computation of M on input x. Then R is $poly(|x|, |\sigma|)$ computable. Thus, because of our initial observation, σ can be put in a probabilistically-checkable form τ by algorithm pcP within $poly(|x|, t)$ steps. In fact, the length of σ is $O(t^2)$, and thus the length of τ is upperbounded by a fixed polynomial in t. The so obtained τ can then be efficiently checked by pcV with the help of a fixed polynomial (in $\log t$) of coin tosses. Since these two fixed polynomials are of special interest to our analysis, let us highlight them in the following

Unfortunately, a string τ produced as above may be too long to constitute a CS certificate for "$M(x) = YES$ in t steps." Thus, Prover $\mathcal{P}$, instead of outputting τ, outputs a much shorter string $\mathcal{C}$ vouching that *if pcV were run on input x and random access to τ, then it would accept x.* Prover $\mathcal{P}$ produces $\mathcal{C}$ in three conceptual stages.

(*i*) He commits in a special way to every single bit of τ by outputting a short string R_ϵ.

(The special property of this commitment is that each bit of τ can be decommitted *individually*, that is, without revealing all the bits of τ.)

(*ii*) He produces the histories of several random executions of pcV on inputs x and random access to τ; and

(*iii*) (3) whenever pcV wishes to access bit location i, not only does $\mathcal{P}$ feed pcV the ith bit of τ, but he also provides evidence that b_i is indeed the ith bit of the string committed by R_ϵ.

The special property of the commitment of conceptual step (*i*) is that, while the whole string τ is commited to by R_ϵ, it is possible for prover $\mathcal{P}$ to decommit each single bit of τ *individually*, by releasing an amount of information that is poly-logarithmic on τ. Prover $\mathcal{P}$ is committed by R_ϵ to each of the bits of τ in the sense that, for each bit position i, $\mathcal{P}$ can feasibly prove either that "$b_i = 0$" or "$b_i = 1$," but not both).

CREDITS. The special commitment algorithm used by $\mathcal{P}$ to commit to each individual bit of τ is the tree-hashing scheme of Merkle [14]. (He used this scheme in the context of digital signatures: for authenticating, by a single short value, many one-time public keys.)

Merkle's scheme and interaction have been used by Kilian [15] for giving a probabilistically-checkable verifier "virtual random access" to the bits of a probabilistically-checkable proof. He used this idea for providing zero-knowledge arguments for $\mathcal{NP}$ where Provers and Verifiers exchanged poly-logarithmically less bits Disregarding zero-knowledge considerations, this same idea was independently conceived by the author (the aim was to allow theorems even outside $\mathcal{NP}$ to be proved with poly-logarithmically lesser bits of communication). The present construction of guaranteed CS proof-systems can be obtained by (1) stripping Kilian's construction of its zero-knowlegde complications, (2) replacing one-way hash functions with oracles, and (3) replacing interaction with the use of random oracles (so as to provide short certificate); and, in addition, by (4) observing that the resulting construction still works "above $\mathcal{NP}$," and (5) insisting that Provers decide themselves the theorems they wish to prove, rather than receiving from a *deus ex machina* convenient witnesses of their validity (whic, as we have argued in several places, may cause that the much desired feasible completeness cease to hold).

The idea of using a random oracle to avoid interacting with a Verifier was suggested by Manuel Blum for the purpose of removing interaction from a zero-knowledge proof of membership in $\mathcal{NP}$. This suggestion was in fact the starting point of the non-interactive zero-knowledge proofs of [16] and [17]. This suggestion, however, did not enter the final version of these papers: in fact, for the specific task of zero-knowledge proofs of membership in $\mathcal{NP}$ languages, random oracles were successfully replaced by short, random strings and quite-standard complexity assumptions. As a result, the idea of using random oracles for replacing interaction has, as often and unfairly happens, become nameless folklore. This "folklore," however, as can be seen in the proof that the construction works, refers to a *much simpler* setting than the one at hand.

3.1.3 The Construction

A SIMPLIFICATION. According to our definition, in a guaranteed CS proof-system 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 exempligy a guaranteed CS proof-system $(\mathcal{P}, \mathcal{V})$, where $\mathcal{P}$ and $\mathcal{V}$ have oracle-access to two distinct functions: f_1 and f_2, where feasible completeness holds for any possible choice of f_1 and f_2, while computational soundness when f_1 and f_2 are random and independent.

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

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

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

If N is a power of two, we let $\mathcal{T}_N$ denote the complete binary tree with N leaves, labeled so that v_ε is the root, v_0 and v_1 are, respectively, the left and right children of v_ε, and, $\forall \alpha \in \{0,1\}^i$ and $\forall i < \log N$, $v_{\alpha 0}$ and $v_{\alpha 1}$ are, respectively, the left and right children of node v_α. (For example, $\mathcal{T}_8$ is illustrated in Figure 3.1.3.) Consequently, $v_{\alpha[1\cdot\cdot j]}$ and $v_{\alpha[1\cdot\cdot\bar{j}]}$ are siblings whenever $0 < |\alpha| \leq \log N$ and $0 < j \leq |\alpha|$. The leaves of $\mathcal{T}_N$ are thought to be ordered "from left to right." Within the context of a tree $\mathcal{T}_N$, we denote by $[j]$ the $\log N$-bit binary representation of integer j (thus, $[j]$ consists of the binary representation of j with the right number of leading zeros). Accordingly, the jth leaf of $\mathcal{T}_N$ is node $v_{[j]}$.

In the following protocol we associate values to the nodes of $\mathcal{T}_N$. In so doing, we shall consistently denote the value of v_α as R_α. Thinking of each node of $\mathcal{T}_N$ as having its own memory location, we may also say that "R_α has been stored in node v_α."

ALGORITHMS $\mathcal{P}$ AND $\mathcal{V}$.

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

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

Common subroutines: SP and SV,

respectively prover and verifier of a given samplable proof-system (SP, SV) with fundamental lengths ℓ and L.

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

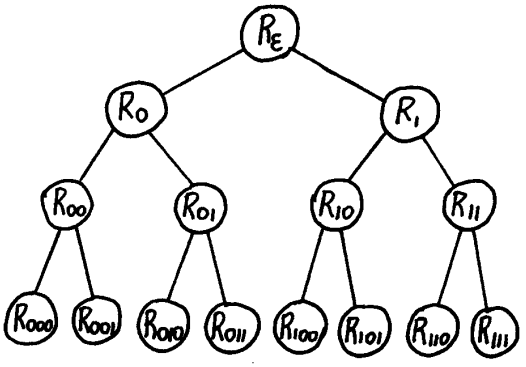

Figure 1: The 8-leaf complete binary tree $\mathcal{T}_8$.

(*Comment:* When randomly selected, oracle f_1 is used by $\mathcal{P}$ to commit to a samplable proof of $\vec{q} \in \mathcal{L}$, and oracle f_2 to commit to k random tapes for SV.)

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

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

Algorithm $\mathcal{P}$

*P*1. (Commit to a samplable proof of $\vec{q} \in \mathcal{L}$.)

*P*1.1 (Find a proof σ of $x \in L$.)
Run machine M on input x so as to generate the history, σ, of the $\leq t$-step accepting computation of $\mathcal{M}$.
(*Comment:* σ can be considered a proof that $x \in L$)

*P*1.2 (Put σ in a samplable form τ.)
Run SP on σ so as to obtain a samplable proof τ. Then, Run algorithm SP on input σ so as to obtain a samplable proof τ.
(*Comment:* Since $|(\bar{M}, x, t)| = n$, $|\tau| \leq \ell(n)$.)

*P*1.3 (Commit to τ by means of a k-bit value R_ε.)
Assume, for simplicity, that τ's length equals the product of k, the security parameter, and N, an integral power of 2. Then, sub-divide τ into the concatenation of N k-bit strings, $\tau = \tau_1 \cdots \tau_N$, and compute a value R_ε by associating to the vertices of tree $\mathcal{T}_N$ the following values. For $0 \leq j < N$, assign to the jth leaf, $v_{[j]}$, the k-bit value

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

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

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

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

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

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

$P2.2$ (Choose k random tapes for SV.)
Call $TAPE$ the random $kL(n)$-bit string obtained by computing $f_2(\vec{q}|R_\epsilon)$; divide this string in k disjoint segments, each $L(n)$-bit long, and call $TAPE_j$ the jth such segment.
(*Comment:* Because L is the second fundamental parameter of samplable proof-system (SP, SV), because $\vec{q} = (M, x, t)$ is n-bit long, $L(n)$ upper-bounds the length of the random tape of SV needed for verifying with probability at least $1/2$ that τ is a samplable proof of $\vec{q} \in \mathcal{L}$.)

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

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

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

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

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

Algorithm $\mathcal{V}$

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

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

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

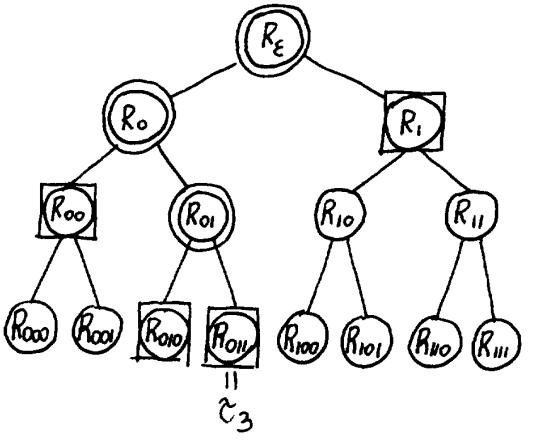

Figure 2: Squared values denote quantities supplied as part of the CS certificate, while circled values are computed by the Verifier.

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

$V2.3$ (Check whether the certificate contains the siblings of the path between leaf I and the root of $\mathcal{T}_N$, and remove them.)
For $m = 1$ to $\log N$, set $R_{\alpha[1..m]} = \mathcal{C}[1 \cdot \cdot k]$ and reset $\mathcal{C} \leftarrow \mathcal{C}[k+1 \cdot \cdot]$. Then, for $m = \log N, \ldots, 1$, compute $R_{\alpha[1..m-1]}$ as follows:

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

and check whether the computed value R_ϵ equals the read value $ALLEGED - ROOT$.
(For example, if $N = 8$ and $I = 3$, then the verifier computes

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

In this example, the values of τ_3, R_{010}, R_{00}, and R_1 are part of the certificate provided by the prover, and the values of R_{011}, R_{01}, R_0, and R_ϵ are computed by the verifier: See Figure 2.)

$V3$. (Accept if and only if the sibling path have always been correct and if SV has always accepted.)
If each check performed in Verification Step 2.3 has been passed, and if SV has output YES in each of its k runs, then output YES. Else, output NO.

3.1.4 A Flawed Proof That the Construction Works.

Every one would agree that proving the feasible completeness of $(\mathcal{P}, \mathcal{V})$ presents no difficulties. But though this might be

true also for the proof of computational soundness, the one we present in the next subsection is not particularly compact. We felt compelled to give a certain amount of details by the fact that an attractively simple (but flawed) alternative "proof" seems at hand.

Indeed, due to all previous uses of Merkle's tree-hashing construction, proving the soundness of $(\mathcal{P}, \mathcal{V})$ requires resisting the temptation of quickly "deriving" it from the following

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

The above statement is certainly true, and, once properly formalized, is not hard to prove. However, is not directly relevant to the scenario at hand. It applies, instead, to settings where one party wishes to secretly commit to a string τ that will be later revealed in its entirety. This is the case, for instance, of the following

> *Different Scenario:* Mathematician A tells her colleague B that she has secretly found a classical proof, p, of either Fermat's or Goldbach's conjecture, and encourages him to bet on which is the case. To make the bet fairer, A, with the help of a publicly-available random oracle, tree-hashes string p like in Proving Step 1.3, so as to obtain a k-bit value R_ϵ, which she gives B. (As it is intuitively clear, if B does not know p *exactly* and does not have much time before betting, R_ϵ is essentially useless to him for figuring out which conjecture A has actually solved.) After B announces his guess, A must de-commit R_ϵ by revealing p, since failure to do so is tantamount to her losing the bet. B then checks whether p, with the help of the same oracle, tree-hashes to R_ϵ, and, if so, whether p is a proof of the first or of the second conjecture. Because, technically speaking, it can be guaranteed that no string can be a proof of both conjectures (no matter how easily one may follow from the other) the above lemma guarantees B that it is practically impossible for A to change her decommitment of R_ϵ after learning his bet.

Unfortunately, the context of Theorem 1 is quite different: Prover $\mathcal{P}$ never de-commits R_ϵ by revealing the entire samplable proof τ. Indeed, because τ may be too long, and because $\mathcal{V}$'s feasible completeness requirement must be satisfied, $\mathcal{P}$ only reveals a few (i.e., $Poly(k \log |\tau|)$) bits of τ: those requested by the samplable verifier in its k simulated and virtual runs. It is thus not crucial that a cheating prover cannot feasibly find two strings τ and τ' that tree-hash to the same k-bit value. Rather, we must prove that a cheating prover, on input $q' \notin \mathcal{L}$, cannot feasibly compute a k-bit value R'_ϵ such that, after a few leafs of $\mathcal{T}_N$ have been randomly selected, he has a non-negligible chance of quickly computing values that

(a) appear to be stored in a sub-tree of $\mathcal{T}_N$ containing those leaves, where R'_ϵ is stored in $\mathcal{T}_N$'s root

(i.e., such that their corresponding sibling-paths correctly hash into each other and, ultimately, into R'_ϵ)

and

(b) mislead $\mathcal{V}$ into believing that $q' \in \mathcal{L}$.

This is what we are going to do below; and it will require some *ad hoc* efforts, because the contexts in which the cited prior ideas were developed are both different and simpler than the one at hand.[3]

3.1.5 The Construction Works

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

Proof of Feasible Completeness.

It is immediately seen that if $\vec{q} \in \mathcal{L}$, and the guaranteed CS prover is honest, the guaranteed CS verifier will accept any CS certificate of $\vec{q} \in \mathcal{L}$; that is, subproperty (ii) holds. Subproperty (i) follows as easily when one considers the running time of each of $\mathcal{P}$'s subroutines and the length of their respective inputs and outputs (and recalls, as we have already said, that each call to our f_1 and f_2 can be simulated very efficiently even if one were given access only to a single oracle $f : \Sigma^{3k+n} \to \Sigma^{k(L(n)+1)}$). ■

Proof of Computational Soundness.

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

In our proof we shall also consider oracle-calling algorithms that are probabilistic. The sequence of coin tosses of a probabilistic oracle-calling algorithm A shall be always denoted by $C(A)$. Notice that the notion of a N-call algorithm include probabilistic ones.[4]

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

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

Let $\mathcal{A}_{(\cdot)}$ be a N-call algorithm querying its oracle about strings in a finite set D, let R be another finite set, and let $f : D \to R$ be a function. Then, an execution of $\mathcal{A}_f$ identifies an element of the Cartesian product R^N; that is, the sequence

[3] For instance, the signer of Merkle's scheme wished to prevent someone else from cheating (by producing a short string relative to which anyone else could not easily authenticate different values). In our case, instead, we wish that the Prover produces a short string that prevents *himself* from cheating. As for another example, the interaction of the Verifier in a zero-knowledge proof simply consists of selecting at random one out of two alternatives (or a random subset out of a given set). In our context, instead, the Verifier's questions are both random and correlated.

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

$T = r_1, \ldots, r_N$, where r_i is f's answer to $\mathcal{A}$'s ith query. Viceversa, because $\mathcal{A}$ does not, within the same execution, query twice its oracle about the same string, given an R-valued sequence $T = r_1, \ldots, r_N$, we can envisage executing $\mathcal{A}$ so that the ith query to the oracle is answered by r_i. The ensemble of executions so generated will be denoted by $\mathcal{A}_T$. It should be noticed that the set of ensembles $\mathcal{A}_T$ and the set of ensembles $\mathcal{A}_f$ may be different. (If $\mathcal{A}$ is deterministic, then $\mathcal{A}_T$ consists of a single execution, in which case it is immediate to see that there exist functions f such that $\mathcal{A}_T = \mathcal{A}_f$. However, if $\mathcal{A}$ is probabilistic, it may happen that in one execution of $\mathcal{A}_T$, the ith query, q, is answered by a value v, while in a differnt execution the same string q, asked as the jth query, where $j \neq i$, is answered by a value other than v. In this case for no function f is $\mathcal{A}_f$ equal to $\mathcal{A}_T$.) Nonetheless, it is simple to notice that, whether or not $\mathcal{A}$ is probabilistic, for any event E, the following identity holds:

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

The above concept, notation, and identity naturally extend to more "complex" cases.[5]

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

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

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

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

To proceed, we need to elaborate a bit on the well-known "birthday paradox" (namely, we need a reasonable upper-bound for the probability of a birthday when a class has very few students).

[5] For instance, Let T be a N-long sequence whose values belong to a finite set R, and let $\mathcal{A}_{(\cdot,\cdot)}(\cdot)$ be a N-call algorithm that always queries its first oracle about a string in a finite set D. Then, $\mathcal{A}_{T,f_2}(x)$ denotes an execution of $\mathcal{A}$, where x is the input, f_2 the second oracle, and where the ith query of $\mathcal{A}$ to the first oracle is answered by the ith string of a sequence T. It is immediately seen that, for all possible choices of f_2, x, D, and R,

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

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

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

Proof of Lemma 1: See Reference [1]. (The slick proof of Lemma 1 presented there was suggested by Ray Sidney.) ■

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

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

Proof of Corollary 1: Algorithm $\mathcal{A}$ has no greater a chance of finding a f-collision in the present setting than in that of Lemma 1. In fact, if it makes two queries α and β both in the set $X = \{x_1, \ldots, x_m\}$, then we are guaranteed that $f(\alpha) \neq f(\beta)$; on the other side, if α and β are not both in X, then the probability that $f(\alpha) = f(\beta)$ still is 2^{-k}. ■

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

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

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

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

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

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

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

We shall solely focus on the non-empty outputs of a cheating prover $\mathcal{P}'$; thus, when writing $\mathcal{P}'_{f_1,f_2}(q', k') = \mathcal{C}'$, we mean that $\mathcal{P}'$ has output a non-empty string, and that this string is $\mathcal{C}'$. If this occurs, given that a cheating prover always verifies its own non-empty outputs, we must have

$\mathcal{V}_{f_1,f_2}(q',k',\mathcal{C}') = YES$. Assume now that, during the verification of $\mathcal{C}'$, samplable verifier V virtually asks about a bit-location i of the virtual samplable proof; then, we write $\mathcal{C}' \ni i$.

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

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

Lemma 2: Proposition $\wp 1$ implies the following proposition

$\wp 2$: *There exist a k'-bit value R'_ϵ; a $\leq 2^{c_5 k'}$-long sequence S extendible from $\Sigma^{2k'}$ to $\Sigma^{k'}$; an integer $m \in [1, 2^{c_5 k'}]$; and a sequence $T = \{\sigma_i \in \Sigma^{k'L(n')} : i \in [1, 2^{c_5 k'}] - \{m\}\}$, such that*

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

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

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

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

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

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

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

and

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

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

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

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

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

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

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

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

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

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

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

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

A1. For $j = 1$ to $4 \cdot 2^{\ell(n')} \cdot 2^{(c_5+c_6)k'+2}$, randomly select $\sigma_j \in \Sigma^{k'L(n')}$ and execute $\mathcal{P}'_{f_1,T_m=\sigma_j}(q',k')$.

A2. If in Step $A1$ algorithm $\mathcal{P}'$ has output $< 2 \cdot 2^{\ell(n')}$ pseudo-certificates (of $q' \in \mathcal{L}$) whose prefix is R'_ϵ, HALT without any output. Else,

A3. Compute BL, the set of bit-locations i such that, for some execution j,

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

A4. If, for some $i \in BL$, there is no unique bit b_i such that all questions of V about bit-location i have been consistently answered with b_i, HALT without any output. Else,

A5. HALT outputing the $2^{\ell(n')}$-long string τ whose ith character is b_i, if $i \in BL$, and $*$ otherwise.

Lemma 3: There exists $f_1' \in ext(S)$ such that

$$Prob_{C(\mathcal{A})}(\mathcal{A}_{f_1'} \text{halts in Step } A5) > 1/2$$
and
$$Prob_{\sigma \in \Sigma^{k'L(n')}}(\mathcal{P}'_{f_1', T_m=\sigma}(q', k') = R'_\epsilon \cdots) \geq 2^{-(c_5+c_6)k'-2}.$$

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

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

Then, a simple counting argument shows that

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

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

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

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

and

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

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

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

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

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

but

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

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

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

Second,

$$R^1_{\alpha_1 \cdots \alpha_{\log N}} | R^1_{\alpha_1 \cdots \bar{\alpha}_{\log N}} \neq R^2_{\alpha_1 \cdots \alpha_{\log N}} | R^2_{\alpha_1 \cdots \bar{\alpha}_{\log N}}.$$

In fact, their k'-bit prefixes are, respectively, $R^1_{[I]}$ and $R^2_{[I]}$.

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\mathcal{P}_{f'_1,T_m=\sigma}(q',k') = (R'_e \cdots).$$

Then, in order for $\mathcal{A}_{f'_1}$ to output $\tilde{\tau}$, for each of these σ, the event

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

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

Now, because any string outputable by $\mathcal{A}_{f'_1}$ is $2^{\ell(n')}$-long, there may be at most $3^{2^{\ell(n')}}$ strings τ such that $P'_\tau \leq 1/2$. Thus,

$$Prob_{C(\mathcal{A})}(\mathcal{A}_{f'_1} = \tau \ \wedge \ P'_\tau \leq 1/2) < \sum_{j=1}^{3^{2^{\ell(n')}}} 2^{-2\cdot 2^{\ell(n')}} < 1/2.$$

But because, as shown in Lemma 3, $\mathcal{A}_{f'_1}$ halts in Step $A5$ (and thus outputs a non-empty string τ) with probability $> 1/2$, $\mathcal{A}_{f'_1}$ must output a string $\tilde{\tau}$ such that $P'_{\tilde{\tau}} > 1/2$. ■

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

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

and

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

Then,

$$(\tilde{P}_{\tilde{\tau}})^{k'} = P_{\tilde{\tau}} \geq P'_{\tilde{\tau}} \cdot Prob_{\sigma\in\Sigma^{k'L(n')}}(\mathcal{P}'_{f'_1,T_m=\sigma}(q',k') = R'_e \cdots) \geq 2^{-(c_5+c_6)k'-3}.$$

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

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

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

4 Cryptographic CS Proofs

In dealing with cryptographic CS proofs, we shall be more informal than we have been for guaranteed CS proofs. Actual details will be given in the final paper.

4.1 Basic Definitions

As in the "guaranteed case," a cryptographic CS proof-system still is a pair of algorithms, a honest prover P and a honest verifier V, designed to handle membership in our language $\mathcal{L}$. But cheating provers may no longer have arbitrarily-long descriptions: they now are poly-size circuits. Also, P and V no longer share access to a common random oracle $\mathcal{R}$, but simply share a short random string r.

Definition: Let (P,V) be a pair of Turing machines, the second of which runs in polynomial-time, executable according to the following simple mechanics: on common inputs a security parameter k, an auxiliary string r, and an alleged member of $\mathcal{L}$, $\vec{q}$, P produces a string $\mathcal{C}$; then, V computes on inputs k, r, $\vec{q}$, and $\mathcal{C}$ in order to decide whether to accept or reject.

We say that such a pair (P,V) is a *cryptographic computationally-sound proof-system* (*cryptographic CS proof-system* for short) if there exist a sequence of 6 positive constants, $c_1,\ldots,c_6$ (refered to as the *fundamental constants* of the system), such that the following two properties are satisfied:

1''. *Feasible Completeness.* $\forall$ integers $n > 1$, $\forall$ n-bit input $\vec{q} = (\bar{M},x,t) \in \mathcal{L}$, $\forall$ unary integers k, and $\forall$ $r \in \Sigma^{k^{c_1}}$,

(i) P halts within $(nkt)^{c_2}$ computational steps, and outputs a binary string $\mathcal{C} = P(k,r,\vec{q})$ whose length is $\leq (nk\log t)^{c_3}$, and

(ii) $V(k,r,\vec{q},\mathcal{C}) = YES$.

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

$$V(k,r,\vec{q},P'(k,r,\vec{q})) = YES$$

for at most a fraction $2^{-c_6 k}$ of the strings r in $\{0,1\}^{k^{c_1}}$.

If (P,V) is a cryptographic CS proof-system, the process of running P on an input $(\bar{M},x,t)$ with a security parameter k and a random string r will be refered to as a *cryptographic CS proof*, and the output of P will be refered to as a *cryptographic CS witness* or *cryptographic CS certificate* (of security k) of $M(x) = YES$. When it is clear from context that we are dealing with cryptographic CS proof-systems, we may drop the adjective "cryptographic." Alternatively, we shall speak of CS proof-systems, CS proofs, CS witnesses, etc., when what is said applies both to the guaranteed and the cryptographic scenario.

Remarks:

- Cryptographic CS proof-systems are more practical than guaranteed ones, because sharing a short random string is more practical than sharing access to a random oracle. One may envisage a yet more practical version of CS proofs, where Prover and Verifier share no auxiliary string at all. (I.e., their only inputs are the security parameter and the alleged member of $\mathcal{L}$.) However, it is our opinion that such super-practical version of CS proofs do not actually exist; that is, we believe that any assumption on which they can be explicitly constructed is actually false.

- The assumptions necessary for the construction of cryptographic CS proofs may be substantially weakened if, rather than the short random string r, Prover and Verifier shared $r' = A(r)$, where A is a polynomial-time algorithm (and thus r' is still short) and r is kept secret. We will not discuss this any further in this extended abstract.

- There actually is a polynomial-time amount of "slackness" in the notion of a cryptographic CS proof, in that, as it will become clearer from their construction, the same random string r can handle n-bit inputs as well as, say, n^2 ones.

4.2 Constructing Cryptographic CS Proofs

For explicitly constructing a cryptographic CS proof-system we use the same code of the guaranteed CS proof-system $(\mathcal{P}, \mathcal{V})$ of Subsection 3.1.2, but evaluate an explicit program f on inputs x and r whenever the original code queried the random oracle on input x.

Based on current knowledge, we could safely replace the original random oracle with a variety of plausible candidates for (the pseudo-random function) f. In particular, f may be chosen from many well-known one-way hashing functions. For *conceptual clarity*, however, we prefer to choose f to be a pseudo-random function à la Goldreich, Goldwasser, and Micali [18] with a public seed. The proof that the so-modified construction works is derived from that of the guaranteed construction and from a special, but reasonable complexity assumption. To elucidate what this assumption may be, we need to recall the results of [18].

THE GGM CONSTRUCTION OF PSEUDO-RANDOM ORACLE-FUNCTIONS. In [18], Goldreich, Goldwasser, and Micali show that, given a cryptographically-strong generator [19] [20], one can explicitly construct an efficient algorithm for simulating oracle-access to a random function. Namely, their construction maps a secret, k-bit, random seed s to a easy-to-evaluate function $f_s : \Sigma^{k^c} \rightarrow \Sigma^{k^d}$ (where c and d can be fixed to be any positive constants). Any adversary who (1) does not have sufficient computational resources for breaking the underlying generator and who is not given the seed s, but (2) can, in an adaptive manner, request and obtain the values $f_s(x)$ at any points x of his own choosing, cannot distinguish accessing such a pseudo-random f_s from accessing a function f randomly selected from the space of all functions mapping Σ^{k^c} to Σ^{k^d}.

Thanks to the results of [24] and [25], cryptographically-strong genertors can now be based on any one-way function. Thus, given a function easy to evaluate but that needs, say, $O(2^k)$ time to be inverted, one can construct pseudo-random functions f_s that pass, when the seed s is hidden, all, say $2^{\epsilon k}$-time statistical tests for functions, for some $\epsilon \in (0,1)$.

In sum, the GGM construction shows that,

> based on the weakest cryptographic assumption (i.e., the existence of at least one one-way function), random oracles can be, for all practical purposes, replaced by pseudo-random functions f_s that are secretly-evaluatable, because the secrecy of the seed s must be preserved.

Oracle-access to a GGM pseudo-random function f_s could be achieved in a variety of ways. For instance, by having one trusted party (e.g., a computer in a computer network) randomly select, and keep secret, a seed s; receive (electronic) requests x; and answer them by sending back (again, electronically) the proper (and, given s, easily computable) strings $f_s(x)$. Alternatively, f_s could be realized by tamper-proof devices, containing the same, secret, random seed s in a protected portion of their memory, that are made universally available, so that everyone can *de facto* share a random oracle with everyone else.

A DIFFERENT NEED. In cryptographic CS proofs, however, we wish to avoid (for reasons that will become clear in the next section) any type of oracle-access, no matter how plausible and practical this may be. Thus we do not need secretly-evaluatable pseudo-random functions, but publicly-evaluatable pseudo-random functions.

To this end, we envisage replacing the random oracle of the guaranteed CS proof construction with a GGM pseudo-random function f_s whose seed, s, has been made public, so as to enable everyone to evaluate f_s on any input string x directly. In other words, we envisage letting the short and shared random string r of the definition of a cryptographic CS proof-system to be s, the seed for the GGM construction. Now, publishing the seed of the GGM construction certainly makes the corresponding f_s publicly evaluatable, but: will it be still "sufficiently random?"

THE PROBLEM OF PUBLICLY-EVALUATABLE PSEUDO-RANDOM FUNCTIONS. Let s be a sufficiently-long random string, and let f_s be a GGM pseudo-random function with seed s. Consider now the following informal statements:[6]

- $S1$: Given an input string x and the first half of $f_s(x)$, it is had to predict the next bit of $f_s(x)$ with probability of success essentially greater than $1/2$.

- $S2$: For any given input string x, the number of the 0's in $f_s(x)$ roughly equals that of the 1's.

- $S3$: It is hard to find (i.e., to compute) an input string x together with the prime factorization of $f_s(x)$.

For what we have said so far, assuming that the GGM construction is based on a true one-way function, all the above statements hold when s is kept secret, and thus f_s is accessible as an oracle. But what if s is made public? Statement $S1$ clearly becomes false. Statement $S2$ remains true. (In fact, once we have established, based on the secrecy of s, that it holds, publishing s cannot make such a property disappear!) As for Statement $S3$, it is far from clear whether it will continue to hold or not. (That is, assuming that factoring remains, as it currently seems, a computationally intractable problem.)

This discussion somehow emphasizes that constructing a publicly-evaluatable pseudo-random function is not easy; at least, if one aims at achieving a construction that provably passes the largest possible number of statistical tests

[6] Indeed, rather than talking about individual seeds and strings, we should talk of *ensembles*.

for functions of this type. Indeed, even expressing what this largest possible set of tests may be (without, of course, ruling out right away the existence of publicly-evaluatable pseudo-random functions) is hard. Personally, we do not expect that a characterization of the statistical properties enjoyed by a publicly-evaluatable pseudo-random function will be as clean and powerful as that of [18] for the case of secretly-evaluatable paseudo-random functions (namely, all tests running in time substantially smaller than the time needed to invert the underlying one-way funtion —e.g., all polynomial-time tests).

Finding a good construction and characterization of publicly-evaluatable pseudo-random functions is very important problem, but, fortunately, not one that needs to be solved before constructing cryptographic CS proof-systems.

WHAT SUFFICES FOR CONSTRUCTING CS PROOF-SYSTEMS. Going through the proof of the guaranteed-CS-proof construction, one sees that we rely on a handful of statistical properties of a random oracle, and only these properties should be preserved by a publicly-evaluatable pseudo-random function for implementing cryptographic CS proofs.

Some of these properties can be guaranteed by *ad hoc* constructions based on traditional complexity assumptions, such as the hardness of factoring or computing discrete logarithms. (For instance, one of the property of a random oracle $f-1$ we relied upon in the proof that the pair $(\mathcal{P},\mathcal{V})$ of Section 3 is a guaranteed proof system is that it is hard to find f_1-collision. Now, it is possible, based on standard cryptographic assumptions, to construct publicly-evaluatable functions f_1 for which it is hard to find collisions.)

Some others of these properties can be satisfied by the GGM construction with a public and randomly chosen seed s, provided (which is easy to do) that this seed is also chosen sufficiently longer than both the inputs and outputs of f_s.

Only for one property, in essence, do we need to make a new assumption, $\mathcal{A}$, stating that a *specific* statistical property of a GGM pseudo-random function keeps on holding when its seed is made public. For clarity purposes, however, rather than describing the minimal such assumption $\mathcal{A}$, we prefer to state it, informally, as a bigger assumption:

> $\mathcal{A}$: *There exists at least a one-way function on which to base the GGM construction such that, though its random seed s is made public, it is computationally hard for a cheating prover (who cannot break the one-way hash function of the authentication-tree of a CS proof) to find an alleged-root R'_ϵ such that*
>
> - *he could answer the questions generated by the sampling verifier SV (run with random tapes taken from $f_s(R'_\epsilon)$) about an input $\vec{q}' \notin \mathcal{L}$, and*
> - *he could provide authenticating paths for these answers that "properly hash" to R'_ϵ.*

Notice that trying several values R'_ϵ until one is found for which the cheating prover can properly answer the relevant questions about $\vec{q}' \notin \mathcal{L}$, does not work. Indeed, we do not need assumption $\mathcal{A}$ to rule out this possible attack: this attack uses f_s as an oracle, and given that this attack fails with a random oracle, the GGM construction guarantees that it will fail with f_s used as an oracle too.

Assumption A essentially rules out that an enemy can easily compute such a special value R'_ϵ by exploiting, somehow, his knowledge of s and of the complex steps of the GGM construction.

Personally, we believe that assumption $\mathcal{A}$ is much safer than, say, assuming that the hardness of the discrete-log problem. The latter assumption, in fact, yields a *specific* one-way function, and one that is full of structure too. By contrast, in $\mathcal{A}$ we are free to choose any one-way function, and we win if at least one with the desired property exists.

We would like to add that assumptions of this type are by now widely believed, and form the basis of several cryptographic constructions such as the Fiat-Shamir Scheme [23], the Rabin's scheme [22], the Schnorr's scheme, and the RSA digital signature scheme [21], were one must rely on sufficiently "random function" to apply to the messages to be signed in order to (hopefully) destroy those algebraic relationships that otherwise make forging signatures extremely easy.

Finally, saying that we believe in assumption $\mathcal{A}$ more than in traditional ones, like factoring or discret logs, does not mean that one should not investigate whether these older assumptions suffice for exemplifying cryptographic CS proofs. This may be an important research effort, but a separate one altogether.

5 $\mathcal{NP}$-Complete Problems Are Cryptographically Checkable

As we all know, Blum [26] has recently put forward a very intriguing notion of program checking. Since then, many beautiful examples of checkers for various problems have been devised (for instance, see [27] and [28]). It is fair to say, however, that most of these problems have been algebraic or belonged to extremely high complexity classes.

In this section, we wish to outline that a slight and meaningful variant of Blum's definition suffices to handle a class of problems of great interest: namely, the $\mathcal{NP}$-complete problems. Below, for concreteness purposes only, we shall focus on the laguage of all Hamiltonian graphs, $\mathcal{H}$.

Informal Lemma: For any cryptographic CS proof-system $(\mathcal{P},\mathcal{V})$, there is fixed polynomial P such that, for any n-bit, non-Hamiltonian graph G, and any security parameter k, there is a $P(kn)$-bit long cryptographic CS certificate (of security k) for the non-Hamiltonicity of G.

Informal Proof: Consider the brute-force algorithm $\mathcal{B}$ that decides the non-Hamiltonicity of a graph by enumerating all possible $n!$ permutations of the vertices of G and verifies that none of then is a cycle in G. The history, σ, of this computation is at most $O(2^{2n})$-long and constitutes a classical proof that $G \notin \mathcal{H}$. Thus σ can be transformed into a $poly(kn)$ cryptographic certificate of $G \notin \mathcal{H}$, for any choice of the security parameter k. ∎

Informal Definition: We shall refer to the CS certificates for non-Hamiltonicity constructed above as *brute-force* certificates.

Consider now an alleged Hamiltonicity program, $\mathcal{HP}$, that, on input a graph G, outputs YES or NO, without

any explanations. $\mathcal{HP}$ may be a heuristic program, and there is no guarantee that $G \in \mathcal{H}$ if and only if $\mathcal{HP}(G) = YES$. Nonetheless, if $\mathcal{HP}(G) = YES$, because of the self-reducibility of $\mathcal{NP}$-complete problems, one can easily find either type of result: (1) an Hamiltonian circuit in G or (2) a proof that $\mathcal{HP}$ is wrong (possibly on inputs other than G), by running $\mathcal{HP}$ on a specific sequence of graphs, $\{G_i\}$, obtained by deleting properly selected edges from G.

This is a theoretically important and practically useful property. Notice, however, that if one-way functions existed and the heuristic program $\mathcal{HP}$ were polynomial-time, we could easily force a type-2 result. For instance, if factoring were hard, we could select two very large primes, multiply them together so as to obtain a composite integer N, express the decision problem "is the ith bit of the factorization of N equal to 0?" in terms of the Hamiltonicity of some other graph G'_i, and run $\mathcal{HP}$ on each such G'_i. It is clear that, if factoring were hard, we would quickly construct a proof that $\mathcal{HP}$ is wrong. However, *this does not mean that self-reducibility is a foolish property.* Indeed, if $\mathcal{HP}$, though polynomial-time, happens to "work well" on a class of graphs that include our original graph G and the original *specific sequence* $\{G_i\}$, self-reducibility lets us obtain the much desired tour in G.

The above foolish dismissal of self-reducibility only indicates that we are much in need of a complexity theory that is based on *individual inputs* rather than on input classes. It is indeed our goal to provide such as a theory as well as a new and better use of self-reducibility. Fully achieving these objectives exceeds, however, the scope of this paper, and will be done in a forthcoming one. Below, nontheless, we shall start making some headway by removing a current limitation of self-reducibility. Namely, when $\mathcal{HP}(G) = NO$, up to now we did not have any meaningful way for checking $\mathcal{HP}$'s output. Such a way is precisely what we provide below if cryptographic CS proofs exist (e.g., if assumption $\mathcal{A}$ holds).

Algorithm $\mathcal{C}$

Real Inputs: $\mathcal{HP}$, an alleged Hamiltonicity program available as an oracle (which makes our result stronger); n, a positive integer; and G, a n-vertex graph such that $\mathcal{HP}(G) = NO$.

Auxiliary Inputs: $(\mathcal{P}, \mathcal{V})$, a cryptographic CS proof-system; and r, a k-bit random string (whose length is a security parameter).

$C0$. Set $\sigma_0 = \varepsilon$.

$C1$. For $i = 1$ to $P(kn)$ do:

$C1.1$ Construct a graph $G_{\sigma_{i-1}}$ that is Hamiltonian if and only if following sentence is true

$S_{\sigma_{i-1}}$: There exists a brute-force cryptographic CS certificate (relative to $(\mathcal{P}, \mathcal{V})$ and r) that is $P(kn)$-bit long and whose prefix is $\sigma_{i-1}0$.

(*Comments:* (1) If $G \notin \mathcal{H}$, then there will be a corresponding brute-force certificate, and, in view of the above Informal Lemma, it will have the desired length. (2) Cryptographic CS proofs are purely syntactic objects. Indeed, they are a short string —in our case $P(kn)$-bit long— that are accepted by the —polynomial-time!— CS verifier $\mathcal{V}$. Thus, (3) each sentence S_σ, "belongs to $\mathcal{NP}$," and can thus be encoded as an instance of graph Hamiltonicity! (4) One of the main reasons for considering versions of CS proofs —like cryptographic CS proofs— that do not rely on any oracle access is precisely that of ensuring that decision problems such as "is there a short CS certificate for $G \notin \mathcal{H}$ whose ith bit is 0?" belong to $\mathcal{NP}$.)

$C1.2$ If $\mathcal{HP}(G_{\sigma_{i-1}}) = YES$, then set $\sigma_i := \sigma_{i-1}0$; else, set $\sigma_i := \sigma_{i-1}1$.

$C2$. If $\mathcal{V}(G, k, \sigma_{P(kn)}) = YES$, output $\sigma_{P(kn)}$. Else, output $G_{\sigma_1}, \ldots, G_{\sigma_{P(kn)}}$.

(*Comment:* $\sigma_{P(kn)}$ is a cryptographic CS certificate — of security k— that $G \notin \mathcal{H}$. $G_{\sigma_1}, \ldots, G_{\sigma_{P(kn)}}$ is a proof that $\mathcal{HP}$ does not decide $\mathcal{H}$.)

Informal Theorem: Given any cryptographic CS proof-system, checker $\mathcal{C}$ runs in polynomial-time and, on input any non-Hamiltonian graph G, outputs either (1) a CS certificate of security k that $G \notin \mathcal{H}$, or (2) a proof that $\mathcal{HP}$ does not decide $\mathcal{H}$. Moreover, if $\mathcal{HP}$'s answers to each of the polynomially many (in n and k) graphs G_{σ_i} about which $\mathcal{C}$ queries $\mathcal{HP}$ is correct, then $\mathcal{C}$'s output will NEVER be of type 2.

The proof of the above theorem will be given in the final paper. Below, we wish instead to drive home some important points.

5.1 Remarks and Coming Attractions

- It should be noted that though a graph G is Hamiltonian, a cryptographic CS certificate of its non-Hamiltonicity may exist, though very hard to find: finding it would in fact entail breaking the cryptographich problem underlying the given CS proof-system. Thus, one may worry that the heuristic program $\mathcal{HP}$, rather than tryig to solve the Hamiltonicity problem, tries to solve the underlying cryptographic problem, and, if successful, concocts CS certificates of non-Hamiltonicity for Hamiltonian graphs. It should be also noted, however, that this worry may be dismissed by properly selecting the security parameter. For instance, assume that we knew that the running time of $\mathcal{HP}$ on a n-vertex graph is upperbounded by —say— $n!$, and that the underlying cryptographic problem could be solved with probability grater than $2^{-\sqrt{k}}$ only if one runs for more than $2^{\sqrt{k}}$ steps. Then, it is easy to choose k so as to be essentially guaranteed that $\mathcal{HP}$, though designed to attack the underlying problem, will not succeed in solving it.
- Notice also that, in practice, we know sharp upperbounds to the running time of any possible $\mathcal{HP}$. Indeed, if checker $\mathcal{C}$ calls $\mathcal{HP}$ and their joint computation ends within our life time, then we *know* that, each time it was called, $\mathcal{HP}$ did not make more than 2^{300} steps of computation! This fact makes CS proofs much more

meaningful in the context of our program-checking application. Let us explain.

As we have always said, a CS proof (of security k) of a given statement S offers an overwhelming probabilistic evidence of the verity of S, provided that we believe that no cheating prover may perform 2^k steps of computation (a quite reasonable belief when k is sufficiently large, but a belief nonetheless). But in the application described above, things are even better. In fact, if a computer on your desk implements $\mathcal{HP}$ and another implements $\mathcal{C}$, and in two hours (or days) of computing, you have output a certificate that G is not Hamiltonian, you KNOW that 2^k steps of computation have not been taken, and thus the statement "G is not Hamiltonian" has been verified in a *purely probabilistic sense;* that is, no more and no less than when an integer has been "declared prime" by a probabilistic algorithm run sufficiently many times.

- As we have already had occasion to remark in passing, rather than using directly a short random string r as a common input to Prover and Verifier, we could construct cryptographic CS proofs by keeping r itself secret, and use in its place the public string $A(r)$, where A is a given polynomial-time algorithm. The attractiveness of this replacement is that it enable us to weaken the needed cryptographic assumption. In general, however, such a replacement may cause problems of a quite different nature. Indeed, for CS proofs to be universally accepted, it must be universaly believed that r or r' have been properly chosen. However, it appears easier to obtain a universal agreement on the fact that a given string r is (sufficiently) random, than to convince everyone that a a given public string r' has been obatined by running a given algorithm A on input a secret but sufficiently random string r. But, fortunately, in the context of checking $\mathcal{NP}$-complete problems, using $r' = A(r)$ instead of r does not create any problems. Indeed, in this context, tere is no need to reach any universl consensus: the person who wishes to check the alleged Hamiltonicity program $\mathcal{HP}$ on input G may easily select, randomly and secretly, r and run A on it so as to produce the needed r', which he will then pass along to algorithm $\mathcal{C}$ for the purpose of generating the right graph sequence$\{G_i\}$, but he needs not to convince any other person that r' was obtain in the prescribed way.

- Finally, let us raise an issue that will be properly resolved in a forthcoming paper. Can our checker be really useful in practice? Let us try to be more concrete. Assume that our given heuristic program $\mathcal{HP}$ runs in $n^{\log n}$ time and actually performs very well on a substantial subset of graphs, to which our original input G belong, and, not tomake things trivial, let us further assume that G is non-Hamiltonian. Then: will our $\mathcal{C}$ succeed in constructing a cryptographic CS certificate of $G \notin \mathcal{H}$? This will depend on the answers that $\mathcal{HP}$ will give in response to the *specific* Hamiltonicity challenges $\{G_i\}$ it receives from $\mathcal{C}$. Now the problem is that these specific graphs G_i's may not belong to the subset on which $\mathcal{HP}$ performs quite well; for example, they might be much harder instances of graph Hamiltonicity than G is.

  To begin with, let us observe that our informal theorem does not say "there exists a cryptographic checker for $\mathcal{NP}$-complete problems." (If so, as we have remarked earlier in this section, one could easily construct a checker whose computations on input G and $\mathcal{HP}$ invariably end with a proof that $\mathcal{HP}$ has "bugs;" namely, such a checker calls $\mathcal{HP}$ on graphs $\{G_i'\}$ that encode a problem too hard for our $n^{\log n}$ heuristics.) Rather, our informal theorem says "the *specific algorithm* $\mathcal{C}$ described above is a cryptographic checker for $\mathcal{NP}$-complete problems;" and our $\mathcal{C}$ does not challenge $\mathcal{P}$ with, say, graphs whose Hamiltonicity encodes the factorization of a long and artificially generated composite integer. Indeed, $\mathcal{HP}$ is not asked to solve any cryptographic problem, but a given underlying cryptographic problem, CP, is used to "prevent $\mathcal{HP}$ from cheating."

  Nonetheless, the specific sequence of graphs about which our $\mathcal{C}$ challenges $\mathcal{HP}$ on input G, not only directly depends on G, but also, though quite indirectly (i.e., through the encoding of the existence of a cryptographic CS witness), on CP itself. Thus the worry may still exist that while "graphs like G" may be correclty handled by $\mathcal{HP}$, $\mathcal{HP}$ may fail on the graph sequences $\{G_i\}$ produced by $\mathcal{C}$. Fortunately, as we shall demonstrate in the near future, it is possible to modify $\mathcal{C}$ so that, for any input graph G, the corresponding graphs G_i's about which $\mathcal{HP}$ will be challenged are guaranteed to be of "the same difficulty as G." Part of the forthcoming effort, of course, will be giving a precise meaning to the latest expression between quotation marks.

6 Certified Computation vs. Program Checking

CS proofs can be used to transform any algorithm A into an equivalent form A'. The new algorithm A' runs in essentially the same time as A does, it receives the same inputs x that A does, and provides the same outputs y that A would output on input x, but also produces a short and easily inspectable certificate, $\mathcal{C}(A, x, y)$, vouching that indeed $y = A(x)$. (The idea essentially consists of giving a CS proof of the statement "$y = A(x)$," that of course could be decided by running A on input x, and thus, due to our feasible completeness property, the corresponding CS certificate can be output in a time that is comparable to that needed to run A.)

While this result is of great generality, it should be noticed that certified computation focuses on a purely *syntactic* property; that is, whether $y = A(x)$, no matter what A should accomplish. It should be appreciated instead that program checking in the sense of Blum also has a main *semantic* component; for instance, whether a given algorithm A really "multiplies ingeters."

For further discussion of certified computation and program checking, see Section 5 of Reference [1].

7 Conclusions

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

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

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

Let me emphasize, however, that, though very natural, the computational boundedness of a cheating prover is not, *per se*, a goal of the notion of a proof. Rather, it is the Trojan horse that lets us sneak in and achieve our desiderata. It is actually very important to establish whether CS proofs (after a slight modification of their definition) exist when a cheating Prover can compute for an unbounded amount of time.

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

References

[1] S. Micali. *CS Proofs*. Technical Memo MIT/LCS/TM-510. (Some of the material in this reference was submitted to STOC 93 and FOCS 93.)

[2] S. Goldwasser and S. Micali and C. Rackoff. *The Knowledge Complexity of Interactive Proof Systems*. SIAM J. Comput., 18, 1989, pp. 186-208.

[3] L. Babai and S. Moran. JCSS 1988

[4] M. Ben-or and S. Goldwasser and J. Kilian and A. Wigderson. *Multi Prover Interactive Proofs: How to Remove Intractability*. Proc. 20th ACM Symp. on Theory of Computing, 1988, pp. 113-131.

[5] C. Lund and L. Fortnow and H. Karloff and N. Nisan. *Algebraic Methods for Interactive Proof Systems*. Proc. 22nd STOC, 1990.

[6] A. Shamir. *IP = PSPACE*. Proc. 31st IEEE Foundation of Computer Science Conference, 1990, pp. 11-15.

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

[8] L. Babai and L. Fortnow and L. Levin and M. Szegedy. *Checking Computation in Polylogarithmic Time*. Proc. of STOC91.

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

[10] S. Arora and M. Safra. *Probabilistic checking of proofs*. Proc. 33rd. IEEE Conference on Foundation of Computer Science, 1992, pp. 2-13.

[11] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. *Proof verification and hardness of approximation problems*. Proc. 33rd. IEEE Conference on Foundation of Computer Science, 1992, pp. 14-23.

[12] M. Sudan. *Efficient checking of polynomials and proofs and the hardness of approximation problems*. Ph.D. Thesis, University of California at Berkeley, 1992.

[13] A. Polishchuk and D. Spielman. *Nearly-linear Size Holographic Proofs*. Proc. STOC 1994.

[14] R. Merkle. *A Certified Digital Signature*. Proc. Crypto 1989. Springer Verlag, 1990.

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

[16] M. Blum, P. Feldman, and S. Micali. *Non-Interactive Zero-Knowledge Proof Systems and Applications*. STOC 1988.

[17] M. Blum, A. De Santis, S. Micali, and G. Persiano. *Non-Interactive Zero-Knowledge*. SIAM J. on Comp. 1991.

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

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

[20] A. Yao. *Theory and Applications of Trap-Door Functions.* Proc. 23rd IEEE on Foundations of Computer Science, 1982.

[21] R. Rivest, A. Shamir, and L. Adleman. *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems.* Comm. ACM, Vol. 21, 1978, pp. 120-126.

[22] M. Rabin. *Digitalized Signatures as Intractable as Factorization.* MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-212, Massachusetts Institute of Technology, Cambridge, MA, January 1979.

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

[24] R. Impagliazzo, L. Levin, and M. Luby. *Pseudo-Random Generation From one-way functions.* STOC 1989

[25] *Pseudo-Random Generation under uniform Assumptions.* STOC 1990.

[26] M. Blum and S. Kannan. *Designing Programs that check their work.* STOC 1988

[27] R. Lipton. *New Directions in Testing.* Distributed Computing and Cryptography. (J. Feigembaum and M. Merritt Ed.) Vol. 2 of Dimacs Series in Discrete Mathematics and Theory of Computer Science. (Preliminary version: manuscript 1989.)

[28] M. Blum, M. Luby, and R. Rubinfeld. *Self-Testing and Self-Correcting Programs, With Applications to Numerical Problems.* Proc. 22nd ACM Symp. on Theory of Computing, 1990, pp. 73-83.