On the random-oracle methodology as applied to length-restricted signature schemes

Ran Canetti[*] Oded Goldreich[†] Shai Halevi[*]

July 30, 2003

Abstract

In earlier work, we described a "pathological" example of a signature scheme that is secure in the Random Oracle Model, but for which no secure implementation exists. For that example, however, it was crucial that the scheme is able to sign "long messages" (i.e., messages whose length is not a-priori bounded). This left open the possibility that the Random Oracle Methodology is sound with respect to signature schemes that sign only "short" messages (i.e., messages of a-priori bounded length, smaller than the length of the keys in use), and are "memoryless" (i.e., the only thing kept between different signature generations is the initial signing-key). In this work, we extend our negative result to address such signature schemes. A key ingredient in our proof is a new type of interactive proof systems, which may be of independent interest.

Keywords: Signature Schemes, The Random Oracle Model, Interactive Proof systems.

[*] IBM Watson, P.O. Box 704, Yorktown Height, NY 10598, USA. {canetti,shaih}@watson.ibm.com

[†] Department of Computer Science, Weizmann Institute of Science, Rehovot, Israel. oded@wisdom.weizmann.ac.il.

1 Introduction

A popular methodology for designing cryptographic protocols consists of the following two steps. One first designs an *ideal* system in which all parties (including the adversary) have oracle access to a truly random function, and proves the security of this ideal system. Next, one replaces the random oracle by a "good cryptographic hashing function" such as MD5 or SHA, providing all parties (including the adversary) with a succinct description of this function. Thus, one obtains an *implementation* of the ideal system in a "real-world" where random oracles do not exist. This methodology, explicitly formulated by Bellare and Rogaway [1] and hereafter referred to as the *random oracle methodology*, has been used in many works (see some references in [4]).

In our earlier work [4] we investigated the relationship between the security of cryptographic schemes in the Random Oracle Model, and the security of the schemes that result from implementing the random oracle by so called "cryptographic hash functions". In particular, we demonstrated the existence of "pathological" signature schemes that are secure in the Random Oracle Model, but for which no secure implementation exists. However, one feature of these signature schemes was that they were required to sign "long messages", in particular messages that are longer than the length of the public verification-key. Thus, that work left open the possibility that the Random Oracle Methodology may still be sound with respect to limited schemes that only sign "short messages" (i.e., messages that are significantly shorter than the length of the public verification-key). In this work we extend the negative result of [4] and show that it holds also with respect to (memoryless) signature schemes that are only required to sign "short messages". That is:

Theorem 1 (sketch) *There exists an ordinary* (i.e., memoryless) *signature scheme that is secure in the Random Oracle Model, but has no secure implementations by function ensembles. Furthermore, insecurity is demonstrated by an attack in which the scheme is only applied to messages of polylogarithmic length (in the security parameter).*

Indeed, the improvement of Theorem 1 over the corresponding result of [4] is merely in the "furthermore" clause.

Our proof extends the technique from [4] of constructing these "pathological" signature schemes. Intuitively, in these schemes the signer first checks whether the message to be signed contains a "proof of the non-randomness of the oracle". If the signer is convinced it performs some highly disastrous action, and otherwise it just employs some secure signature scheme. Such a scheme will be secure in the Random Oracle Model, since the the signer is unlikely to be convinced that its oracle is not random. In a "real world implementation" of the scheme, on the other hand, the oracle is completely specified by a portion of the public verification-key. The attacker, who has access to this specification, can use it to convince the signer that this oracle is not random, thus breaking the scheme. The "proof of non-randomness" that was used in [4] was non-interactive, and its length was longer than the verification-key, which is the reason that it is not applicable to "short messages". The crux of our extension is a new type of interactive proof systems, employing a stateless verifier and short messages, which may be of independent interest.

To prove "non-randomness" of a function, we would like to show that there exists a program that can predict the value of this function at "sufficiently many" points. However, it seems that such proof must be at least as long as said program. In our application, the proof needs to predict a function described in a portion of the verification-key, hence it needs to be of length comparable to that portion. But we want a signature scheme that only signs short messages, so the attacker (prover) cannot submit to the signer (verifier) such a long proof in just one message. It follows that we must use many messages to describe the proof, or in other words, we must have a long interaction. But recall that in our application, the proof has to be received and verified by the signing device, which by standard definitions is stateless.[1] Thus, the essence of what we need is an interactive proof with a stateless verifier.

At a first glance, this last notion may not seem interesting. What good is an interaction if the verifier cannot remember any of it? If it didn't accept after the prover's first message, why would it accept after the second? What makes this approach workable is the observation that *the verifier's state can be kept by the prover*, as long as the verifier has some means of authenticating this state. What we do is let the verifier (i.e., signer) emulate a computation of a Turing machine M (which in turn verifies a proof provided by the prover), and do so in an authenticated manner. The messages presented to the verifier will have the form $(\mathtt{cc}, \sigma, \mathtt{aux})$, where $\mathtt{cc}$ is a compressed version of an instantaneous configuration of the machine, σ is a "signature on $\mathtt{cc}$", and $\mathtt{aux}$ is an auxiliary information to be used in the generation of a compressed version of the next configuration. If the signature is valid then the verifier will respond with the triple $(\mathtt{cc}', \sigma', \mathtt{aux}')$, where $\mathtt{cc}'$ is a compressed version of the next configuration, σ' is a "signature on $\mathtt{cc}'$", and $\mathtt{aux}'$ is an auxiliary information regarding its update.

Relation to the adversarial-memory model. Our approach of emulating a computation by interaction between a memoryless verifier and an untrusted prover, is reminiscent of the interaction between a CPU and an adversarially-controlled memory in the works of Goldreich and Ostrovsky [6] and Blum et al. [2]. Indeed, the technique that we use in this paper to authenticate the state is very close to the "on line checker" of Blum et al. However, our problem still seems quite different than theirs. On one hand, our verifier cannot maintain state between interactions, whereas the CPUs in both the works from above maintain a small (updated) state. On the other hand, our authenticity requirement is weaker than in [6, 2], in that our solution allows the adversary to "roll back" the memory to a previous state. (Also, a main concern of [6], which is not required in our context, is hiding the "memory-access structure" from the adversary.)

Organization. We first present our interactive proof with stateless verifier while taking advantage of several specific features of our application: We start with an overview (Section 2), and provide the details in Section 3. In Section 4 we then sketch a more general treatment of this kind of interactive proofs.

2 Overview of our approach

On a high level, the negative result in our earlier work [4] can be described as starting from a secure signature scheme in the Random Oracle Model, and modifying it as follows: The signer in the original scheme was interacting with some oracle (which was random in the Random Oracle Model, but implemented by some function ensemble in the "real world"). In the modified scheme, the signer examines each message before it signs it, looking for a "proof" that its oracle is not random. If it finds such a convincing "proof" it does some obviously stupid thing, like outputting the secret key. Otherwise, it reverts to the original (secure) scheme. Hence, the crucial step in the construction is to exhibit a "proof" as above. Namely, we have a prover and a verifier, both polynomial-time interactive machines with access to an oracle, such that the following holds:

[1] Indeed, the statelessness condition is the reason that a non-interactive information transfer seems a natural choice, but in the current work we are unwilling to pay the cost in terms of message length.

- When the oracle is a truly random function, the verifier rejects with overwhelming probability, regardless of what the prover does. (The probability is taken also over the choice of the oracle.)
- For any polynomial-time function ensemble,[2] there is a polynomial-time prover that causes the verifier to accept with noticeable probability, when the oracle is implemented by a random member of that ensemble. In this case, the prover receives a full description of the function used in the role of the oracle. (In our application, this description is part of the verification-key in the corresponding implementation of the signature scheme.)

In [4] we used correlation-intractable functions to devise such a proof system.[3] However, simpler constructions can be obtained. For example, when the oracle is implemented by a polynomial-time function ensemble, the prover could essentially just send to the verifier the description of the function that implements the oracle. The verifier can then evaluate that function on several inputs, and compare the outputs to the responses that it gets from the oracle. If the outputs match for sufficiently many inputs (where sufficiently many means more that the length of the description), then the verifier concludes that the oracle cannot be a random function. Indeed, roughly this simplified proof was proposed by Holenstein, Maurer, and Renner [9]. We remark that both our original proof and the simplified proof of Holenstein et al., are *non-interactive* proofs of non-randomness: The prover just sends one string to the verifier, thus convincing it that its oracle is not a random function.

However, implementing the proof in this manner implies that the attacker must send to the verifier a complete description of the function, which in our application may be almost as long as the verification-key. In terms of the resulting "pathological example", this means that the signature scheme that we construct must accept long enough messages.

Clearly, one can do away with the need for long messages, if we allow the signature scheme to "keep history" and pass some evolving state from one signature to the next. In that case the attacker can feed the long proof to the scheme bit by bit, and the scheme would only act on it once its history gets long enough. In particular, this means that the signature scheme will not only maintain a state (between signatures) but rather maintain a state of a-priori unbounded length. Thus, the negative result will refer only to such signature schemes, while we seek to present a negative result that refers to any signature scheme (even stateless ones), and in particular to ones that only sign "short messages".

In this work we show how such a result can be obtained. Specifically, we present a signature scheme that operates in the random-oracle model, with the following properties:

- The scheme is *stateless*: the signer only keeps in memory the secret key, and this key does not evolve from one signature to the next.
- The scheme is only required to *sign short messages*: On security parameter k, the scheme can only be applied to messages whose length is less than k. Furthermore, one could even restrict it to messages of length sub-linear in k (e.g., polylog(k)).
- The scheme is *secure in the Random Oracle Model*: When the oracle is implemented by a truly random function, the scheme is existentially unforgeable under an adaptive chosen-message attack.

[2] A polynomial-time function ensemble is a sequence $\mathcal{F} = \{F_k\}_{k \in N}$ of families of functions, $F_k = \{f_s : \{0,1\}^* \to \{0,1\}^{\ell_{\rm out}(k)}\}_{s \in \{0,1\}^k}$, such that there exists a polynomial-time algorithm that given s and x returns $f_s(x)$. In the sequel we often call s the description or the seed of the function f_s.

[3] We used (non-interactive) CS-proofs (cf. [12]) to make it possible for the verifier to run in fixed polynomial time, regardless of the polynomial that bounds the running time of the ensemble.

- The scheme *has no secure implementation*: When the oracle is implemented by any function ensemble (even one with functions having description length that is polynomially longer than k), the scheme is completely breakable under an adaptive chosen-message attack. We remark that in this case the function's description is part of the verification-key.[4]

To construct such a scheme we need to design a "proof system" that only uses very short messages. As opposed to previous examples, we will now have an *interactive* proof system, with the proof taking place during the attack. Each communication-round of the proof is being "implemented" by the attacker (in the role of the prover) sending a message to be signed, and the signer (in the role of the verifier) signing that message.

The ideas that make this work are the following: We start from the aforementioned non-interactive proof (of "non-randomness"), where the verifier is given the description of a function, and compares that function to its own oracle (i.e., compares their values at sufficiently many points). Then, instead of having the verifier execute the entire test on its own, we feed the execution of this test to the verifier "one step at a time" (and, in particular, the input function is fed "one step at a time"). Namely, let M be the oracle Turing machine implementing the aforementioned test. The adversary provides the verifier with the relevant information pertaining to the current step in the test (e.g., the state of the control of M and the character under the head) and the verifier returns the information for the next step. This requires only short messages, since each step of M has a succinct description.

To keep the security of the scheme in the Random Oracle Model, we need to make sure that the adversary can only feed the verifier with "valid states" of the machine M. (Namely, states that can indeed result from the execution of this machine on some input.) To do that, we have the verifier authenticate each step of the computation. That is, together with the "local information" about the current step, the verifier also compute an authentication tag for the "global state" of the machine in this step, which is done using Merkle trees [11]. Such authentication has the property that it can be computed and verified using only the path from the root to the current leaf in the tree, and the authentication tag itself is very short. A little more precisely, the current configuration of the machine M (using some standard encoding) is viewed as the leaves of a Merkle tree, and the verifier provides the prover with an authentication tag for the root of this tree. Then a typical step in the proof proceeds as follows:

1. The attacker sends to the verifier the "relevant leaf" of the tree (i.e., the one containing the head of M), together with the entire path from the root to that leaf (and the siblings for that path), and the authentication tag for the root.

2. The verifier checks the authentication tag of the root and the validity of the root–leaf path (using the siblings). If everything is valid, then the verifier executes the next step of M, and returns to the attacker the updated path to the root, and an authentication tag for the new root.

If the machine M ever enters an accept state, then the verifier accepts. This proof can still be implemented using only short messages, since the root-leaf path has only logarithmic depth. As for security, since it is infeasible for the attacker to "forge a state" of M, then the verifier will accept only if the machine M indeed has an accepting computation.

[4] In contrast, if the function's description is only part of the signing-key then using any pseudorandom function [5] would yield a secure signature scheme. However, this would not be an application of the Random Oracle Methodology, which explicitly refers to making the function's description public.

3 The details

We now flesh out the description from Section 2. We begin in §3.1 with the basic test that we are going to implement step-by-step. In §3.2 we describe the Merkle-tree authentication mechanism that we use, and in §3.3 we describe the complete "interactive proof system". Finally, we show in §3.4 how this proof system is used to derive our counter-example.

As we did in [4], we avoid making intractability assumptions by using the random oracle itself for various constructs that we need. For example, we implement the Merkle-tree authentication mechanism (which typically requires collision-resistant hash functions) by using the random oracle. We stress that we only rely on the security of this and other constructs in the Random Oracle Model, and do not care whether or not its implementation is secure (because we are going to demonstrate the insecurity of the implementation anyhow). Formally, in the context of the proof system, the security of the constructs only effects the soundness of the proof, which in turn refers to the Random Oracle Model.

In both the basic test and the authentication mechanisms we use access to an oracle (which will be a random function in the Random Oracle Model, and a random member in an arbitrary function ensemble in the "real world"). When we work in the Random Oracle Model, we wish these two oracles to be independent. Thus, we use the single oracle to which we have access to define two oracles that are independent if the original oracle is random (e.g., using the oracle $\mathcal{O}$, we define oracles $\mathcal{O}_i(x) \stackrel{\text{def}}{=} \mathcal{O}(i, x)$).

In the rest of this section, we assume that the reader is familiar with the notion of a polynomial-time function ensemble (as reviewed in Footnote 2).

3.1 The basic test

Our starting point is a very simple non-interactive "proof of non-randomness" of an oracle $\mathcal{O}$. (The basic idea for this proof is described by Holenstein et al. in [9].) The verifier is a (non-interactive) oracle Turing machine, denoted M, which is given a candidate proof, denoted π, as input. The input π is supposed to be a program (or a description of a Turing machine) that predicts $\mathcal{O}$. Intuitively, if $\mathcal{O}$ is random then no π may be successful (when we try to use it in order to predict the value of $\mathcal{O}$ on more than $|\pi|$ predetermined inputs). On the other hand, if $\mathcal{O}$ has a short description (as in case where it is taken from some function ensemble) then setting π to be the program that computes $\mathcal{O}$ will do perfectly well. The operation of M, on security parameter k, input π and access to an oracle $\mathcal{O}$, is given below:

```
Procedure $\mathsf{M}^{\mathcal{O}}(1^k, \pi)$:
 1.  Let $n = |\pi|$ be the bit length of $\pi$.
     ($\pi$ is viewed as a description of a Turing-machine.)
 2.  For $i = 1$ to $2n + k$, let $y_i \leftarrow \mathcal{O}(i)$ and $z_i \leftarrow \pi(i)$.
 3.  If $y_i$ and $z_i$ agree on their first bit for all $i \in [1..2n + k]$, then **accept**.
 4.  Else **reject**.
```

Below it will be convenient to think of the machine M as having one *security-parameter tape* (a read-only tape containing 1^k), one "regular" *work tape* that initially contains π, one oracle *query tape* and one oracle *reply tape* (the last having just a single bit, since we only look at the first bit of the answer). A configuration of this machine can therefore be described as a 4-tuple $\mathsf{c} = (q, r, w, sp)$ describing the contents of each tape (i.e., q describes the query, r the reply, w the contents of the work-tape and sp the security-parameter). By convention, we assume that the description of each tape include also the location of the head on this tape, and that the description of the work tape also includes the state of the finite control. Thus, for the above machine M, we always have $|q| = \log(2|\pi| + k) + \log\log(2|\pi| + k)$, $|r| = 1$, $|w| \leq |\pi| + s_k(\pi) + \log(2|\pi| + k) + \log(|\pi| + s(\pi) + \log(2|\pi| + k)) + O(1)$, $|sp| = k$, where $s_k(\pi)$ is the space require for computing $\pi(i)$ for the worst possible $i \in [2|\pi| + k]$. It follows that $|\mathsf{c}| = O(|\pi| + s_k(\pi) + k)$.

Note that M itself is not a "verifier in the usual sense", because its running time may depend arbitrarily on its input. In particular, for some inputs π (describing a non-halting program), the machine M may not halt at all. Nonetheless, we may analyze what happens in the two cases that we care about:

Proposition 2 (Properties of machine M):

1. Random oracle: *For security parameter k, if the oracle $\mathcal{O}$ is chosen uniformly from all the Boolean functions, then*

$$\Pr_{\mathcal{O}}\left[\exists\ \pi \in \{0,1\}^* \text{ s.t. } \mathsf{M}^{\mathcal{O}}(1^k, \pi) \textbf{ accepts}\right] \ <\ 2^{-k}$$

2. Oracle with succinct description: *For every function ensemble $\{f_s : \{0,1\}^* \to \{0,1\}\}_{s\in\{0,1\}^*}$* (having a polynomial-time evaluation algorithm)*, there exists an efficient mapping $s \mapsto \pi_s$ such that for every s and every k it holds that $\mathsf{M}^{f_s}(1^k, \pi_s)$ accepts in polynomial-time.*

Proof Sketch: In Item 1, we apply the union bound on all possible (i.e., infinitely many) π's. For each fixed $\pi \in \{0,1\}^*$, it holds that the probability that $\mathsf{M}^{\mathcal{O}}(1^k, \pi)$ accepts is at most $2^{-(2|\pi|+k)}$, where the probability is taken uniformly over all possible choices of $\mathcal{O}$. In Item 2, we use the program π_s obtained by hard-wiring the seed s into the polynomial-time evaluation algorithm associated with the function ensemble. □

3.2 Authenticating the configuration

We next describe the specifics of how we use Merkle trees to authenticate the configurations of the machine M. In the description below, we view the configuration $\mathsf{c} = (q, r, w, sp)$ as a binary string (using some standard encoding).

We assume that the authentication mechanism too has access to a random oracle, and this random oracle is independent of the one that is used by the machine M. Below we denote this "authentication oracle" by $\mathcal{A}$. To be concrete, on security parameter k, denote $\ell_{\text{out}} = \ell_{\text{out}}(k) = \lceil \log^2(k) \rceil$,[5] and assume that the oracle is chosen at random, from all the functions $\mathcal{A} : \{0,1\}^* \to \{0,1\}^{\ell_{\text{out}}}$. (Actually, we may consider the functions $\mathcal{A} : \{0,1\}^{3\ell_{\text{out}}} \to \{0,1\}^{\ell_{\text{out}}}$.) We stress again that we do not lose much generality by these assumptions, as they can be easily met in the Random Oracle Model. Also, when the security parameter is k, we use a random ℓ_{out}-bit string for $\mathsf{authentication\ key}$, which we denote by $\text{ak} \in_R \{0,1\}^{\ell_{\text{out}}}$.

To authenticate a configuration c (on security parameter k, with access to an oracle $\mathcal{A}$, and with key ak), we first pad the binary encoding of c to length $2^d \cdot \ell_{\text{out}}$ (where d is an integer). We then consider a complete binary tree with 2^d leaves, where the i'th leaf contains the i'th ℓ_{out}-bit chunk of the configuration. Each internal node in this tree contains an ℓ_{out}-bit string. For a node at distance i from the root, this ℓ_{out}-bit string equals $\mathcal{A}(i, \texttt{left}, \texttt{right})$, where `left` and `right` are

[5] The choice of $\ell_{\text{out}}(k) = \lceil \log^2(k) \rceil$ is somewhat arbitrary. For the construction below we need the output length ℓ_{out} to satisfy $\omega(\log k) \leq \ell_{\text{out}}(k) \leq o(k/\log k)$, whereas the input length should be at least $2\ell_{\text{out}}(k) + \omega(\log k)$. (Note that $2\ell_{\text{out}}(k) + \omega(\log k) < 3\ell_{\text{out}}(k)$.)

the ℓ_{out}-bit strings in the left and right children of that node, respectively. The authentication tag for this configuration equals $\mathcal{A}(d, \text{ak}, \texttt{root})$, where `root` is the ℓ_{out}-bit string in the root of the tree.

The security property that we need here is slightly stronger than the usual notion for authentication codes. The usual notion would say that for an attacker who does not know the key ak, it is hard to come up with any valid pair (configuration,tag) that was not previously given to him by the party who knows ak. In our application, however, the verifier is only presented with root–leaf paths in the tree, never with complete configurations. We therefore require that it is hard even to come up with a single path that "looks like it belongs to a valid configuration", without this path being part of a previously authenticated configuration. We use the following notions:

Definition 3 (valid paths) *Let $\mathcal{A} : \{0,1\}^* \to \{0,1\}^{\ell_{\text{out}}}$ be an oracle and $\text{ak} \in \{0,1\}^{\ell_{\text{out}}}$ be a string as above. A* valid path with respect to $\mathcal{A}$ and ak *is a triple*

$$(\langle \sigma_1 \cdots \sigma_d \rangle,\ \langle (v_{1,0}, v_{1,1}), ..., (v_{d,0}, v_{d,1}) \rangle,\ t)$$

where the σ_i's are bits, and the $v_{i,b}$'s and t are all ℓ_{out}-bit strings, satisfying the following conditions:

1. *For every $i = 1, ..., d-1$, it holds that $v_{i,\sigma_i} = \mathcal{A}(i, v_{i+1,0}, v_{i+1,1})$.*
2. *$t = \mathcal{A}(d, \text{ak}, \mathcal{A}(0, v_{1,0}, v_{1,1}))$.*

This path is said to be consistent with the configuration c *if when placing* c *in the leaves and propagating values described above,*[6] *then for every $i = 1, ..., d-1$, the node reached from the root by following the path $\sigma_1 \cdots \sigma_i$ is assigned the value v_{i,σ_i}, and the sibling of that node is assigned the value $v_{i,\bar{\sigma}_i}$.*

In this definition, v_{i,σ_i} is the value claimed for the internal node reached from the root by following the path $\sigma_1 \cdots \sigma_i$. The value claimed for the root is $v_0 \overset{\text{def}}{=} \mathcal{A}(0, v_{1,0}, v_{1,1})$, and this value is authenticated by $\mathcal{A}(d, \text{ak}, v_0)$, which also authenticates the depth of the tree. Indeed, only the value of the root is directly authenticated, and this indirectly authenticates all the rest.

Fix some $\ell_{\text{out}} \in \mathsf{N}$, and let $\mathcal{A}$ be a random function from $\{0,1\}^*$ to $\{0,1\}^{\ell_{\text{out}}}$ and ak be a random ℓ_{out}-bit string. Consider a forger, F, that can query the oracle $\mathcal{A}$ on arbitrary strings, and can also issue authentication queries, where the query is a configurations c and the answer is the authentication tag on c corresponding to $\mathcal{A}$ and ak. The forger F is deemed successful if at the end of its run it outputs a path $(\alpha, \bar{v}, t)$ that is valid with respect to $\mathcal{A}$ and ak but is inconsistent with any of the authentication queries. One can easily prove the following:

Proposition 4 *For any $\ell_{\text{out}} \in \mathsf{N}$ and any forger F, the probability that F is successful is at most $q^2/2^{\ell_{\text{out}}}$, where q is the total number of queries made by F (i.e., both queries to the oracle $\mathcal{A}$ and authentication queries). The probability is taken over the choices of $\mathcal{A}$ and* ak*, as well as over the coins of the forger F.*

Proof Sketch: Intuitively, the authentication of the root's value makes it hard to produce a path that is valid with respect to $\mathcal{A}$ and (the unknown) ak but uses a different value for the root. Similarly for a path of a different length for the same root value. On the other hand, it is hard to form collisions with respect to the values of internal nodes (i.e., obtain two pairs (u, w) and (u', w') such that for some i it holds that $\mathcal{A}(i, u, w) = \mathcal{A}(i, u', w')$). □

[6] That is, an internal node at distance i from the root is assigned the value $\mathcal{A}(i, u, w)$, where u and w are the values assigned to its children.

3.3 An interactive proof of non-randomness

We are now ready to describe our interactive proof, where a prover can convince a "stateless" verifier that their common oracle is not random, using only very short messages.

The setting is as follows: We have a prover and a verifier, both work in polynomial time in their input, both sharing a security parameter $k \in \mathsf{N}$ (encoded in unary), and both having access to an oracle, say $\mathcal{O}' : \{0,1\}^* \to \{0,1\}^{\ell_{\text{out}}}$. (The parameter ℓ_{out} is quite arbitrary. Below we assume for convenience that this is the same parameter as we use for the authentication scheme, namely $\ell_{\text{out}} = \left\lceil \log^2(k) \right\rceil$.)[7] In this proof system, the prover is trying to convince the verifier that their common oracle in not random. Specifically, both prover and verifier interpret their oracle as two separate oracles, $\mathcal{A}$ and $\mathcal{O}$ (say, $\mathcal{O}(x) = \mathcal{O}'(0x)$ and $\mathcal{A}(x) = \mathcal{O}'(1x)$), and the honest prover has as input a description of a Turing machine that computes the function $\mathcal{O}$. However, we place some severe limitations on what the verifier can do. Specifically, the verifier has as private input a random string $\text{ak} \in \{0,1\}^{\ell_{\text{out}}}$, but other than this fixed string, it is not allowed to maintain any state between steps. That is, when answering a message from the prover, the verifier always begin the computation from a fixed state consisting only of the security parameter k and the string ak. In addition, on security parameter k, the verifier is only allowed to see prover-messages of length strictly smaller than k. (In fact, below we only use messages of size $\text{polylog}(k)$.)

The proof that we describe below consists of two phases. In the first (initialization) phase, the prover uses the verifier to authenticate the initial configuration of the machine $\mathsf{M}^{\mathcal{O}}(1^k, x)$, where k is the security parameter that they both share, and x is some input that the prover chooses. For the honest prover, this input x will be the description of the Turing-machine that implements the oracle $\mathcal{O}$. In the second (computation) phase, the prover takes the verifier step-by-step through the computation of $\mathsf{M}^{\mathcal{O}}(1^k, x)$. For each step, the prover gives to the verifier the relevant part from the current authenticated configuration, and the verifier returns the authentication tag for the next configuration. The verifier is convinced if the machine M ever reaches the accepting state.

For notational convenience, we assume below that on security parameter k, the verifier only agrees to authenticate configurations of M whose length is less than $2^{\ell_{\text{out}}(k)}$. Indeed, in our application the honest prover will never need to use larger configuration (for large enough k).

3.3.1 Initialization phase

This phase consists of two steps. In the first step, the prover will use the verifier in order to authenticate "blank configuration" (lacking a real input) for the computation, whereas in the second step the prover will feed an input into this configuration and obtain (via interaction with the verifier) an initial configuration fitting this input.

First step. The *prover begins this phase by sending a message of the form* ('Init', 0, $\mathtt{sb}$) *to the verifier*, where the integer $\mathtt{sb} < 2^{\ell_{\text{out}}(k)}$ is an upper bound on the length of the configurations of M in the computation to come, and it is encoded in binary. In response, the *verifier computes a* $\mathsf{blank\ configuration}$*, denoted* c_0*, of length* $\mathtt{sb}$ *and sends the authentication tag for this configuration, with respect to oracle* $\mathcal{A}$ *and key* ak. The blank configuration c_0 consists of the security-parameter tape filled with 1^k, all the other tapes being "empty" (e.g., filled with $\star$'s), the heads being at the beginning of each tape, and the finite control being in a special blank state. Specifically, the

[7] Note that even a *binary* oracle (i.e., $\ell_{\text{out}} = 1$) suffices, since in the Random Oracle Model it is easy to convert one output length to another.

work-tape consists of $\mathtt{sb}$ blanks (i.e., $\star$'s), and the query-tape consists of $\ell_{\rm out}(k)/2 = \omega(\log k)$ blanks.[8]

We note that authenticating the blank configuration in a straightforward manner (i.e., by writing down the configuration and computing the labels of all nodes in the tree) takes time $O(\mathtt{sb})$, which may be super-polynomial in k. Nonetheless, it is possible to compute the authentication tag in time polynomial in k, because the configuration c_0 is "highly uniform". Specifically, note that the work tape is filled with $\star$'s, and all the other tapes are of size polynomial in k. Thus, in every level of the configuration tree, almost all the nodes have the same value (except, perhaps, a polynomial number of them). Hence, the number of queries to $\mathcal{A}$ and total time that it takes to compute the authentication tag is polynomial in k.

Conventions. For simplicity, we assume that the contents of the query-tape as well as the machine's state are encoded in the first $\ell_{\rm out}(k)$-bit long block of the configuration. Typically, in all subsequent modifications to the configuration, we will use this block as well as (possibly) some other block (in which the "actual action" takes place). We denote by $\langle i \rangle$ the bit-string describing the path from the root to the leaf that contains the i'th location in the work-tape. Needless to say, we assume that the encoding is simple enough such that $\langle i \rangle$ can be computed efficiently from i.

Second step. After obtaining the authentication tag for the blank configuration, the prover may fill in the input in this configuration by sending messages of the form ('Init', $i, b, \bar{p_1}, \bar{p_i}, t$) to the verifier. Upon receiving such a message, the verifier checks that $(\langle 1 \rangle, \bar{p_1}, t)$ and $(\langle i \rangle, \bar{p_i}, t)$ are valid paths w.r.t. $\mathcal{A}$ and ak, that path $\bar{p_1}$ shows the heads at the beginning of their tapes and the control in the special "blank state", and that path $\bar{p_i}$ shows the i'th location in the work-tape filled with a $\star$. In case all conditions hold, the verifier replaces the contents of the i'th location in the work-tape with the bit b, recomputes values along the path from that tape location to the root, and returns the new authentication tag to the prover. That is, the values along that path as recorded in $\bar{p_i}$ correspond to a setting of the i'th location to $\star$, and setting this location to b typically yields new values that propagate from this leaf up-to the root.

Thus, using $|x|$ rounds of interaction, the honest prover can obtain (from the verifier) the authentication tag on the initial configuration of $\mathsf{M}(1^k, x)$, where x is a string of the prover's choice. Note that a cheating prover may obtain from the prover an authentication tag that does not correspond to such an initial configuration. (In fact, even the honest prover obtains such tags in all but the last iterations of the current step.)

3.3.2 Computation phase

This phase begin with a message of the form ('Comp', $\bar{p_1}, t$) that the prover sends. The verifier checks that $(\langle 1 \rangle, \bar{p_1}, t)$ is a valid path, and that $\bar{p_1}$ shows the heads at the beginning of their tapes and the control in the special "blank state". If these conditions hold, the verifier changes the state to the initial state of M, recomputes the values on the path $\bar{p_1}$ from the initial tape location to the root, and returns the new authentication tag to the prover. (In fact, one may view this step as belonging to the initialization step.)

Thereafter, upon receiving a message of the form ('Comp', $i, j, \bar{p_1}, \bar{p_i}, \bar{p_j}, t$), where $j \in \{i-1, i, i+1\}$ (and indeed when $j = i$ it holds that $\bar{p_i} = \bar{p_j}$), the verifier checks that $(\langle 1 \rangle, \bar{p_1}, t)$, $(\langle i \rangle, \bar{p_i}, t)$, $(\langle j \rangle, \bar{p_j}, t)$, are all valid paths. Furthermore, it checks that $\bar{p_i}$ contains the head position and $\bar{p_j}$

[8] On input $(1^k, x)$, the query tape of M is of size $\log_2(2|x| + k)$. For ensemble F, the honest prover will use $|x| \leq \text{poly}(k) + O(1)$, and so the length of the query tape would be $O(logk)$.

describes a legal contents of the position that the head will move to after the current step. That is, $\bar{p_1}$ and $\bar{p_i}$ provide sufficient information to determine the single-step modification of the current configuration (which may include a movement of some heads and a change in the contents of a single symbol in some of the tapes). In case all conditions hold, then the verifier executes the current step (making a query to its oracle $\mathcal{O}$ if this step is an oracle query), recomputes the values on the three paths to the root, and returns the new authentication tag to the prover. If after this step the machine M enters its accept state, the verifier accepts.

It can be seen that the honest prover can use these interaction steps to take the verifier step-by-step through the computation of M. It follows that if the input to the honest prover is indeed a polynomial-time machine that computes the function $\mathcal{O}$, then the verifier will halt and accept after polynomially many steps. We conclude this subsection by showing that the above constitutes a proof system for non-randomness (satisfying additional properties that we will need in the next subsection).

Proposition 5 *The above construction constitutes a proof system with these properties:*

Efficiency. *Each verifier step can be computed in time polynomial in the security parameter k.*

Stateless verifier. *The verifier is stateless in the sense that it begins every step from the same state, consisting only of the security parameter k and its private input* $\mathrm{ak} \in \{0,1\}^{\ell_{\mathrm{out}}}$. *Formally, the verifier replies to each incoming message m with $V(1^k, \mathrm{ak}, m)$, where V is a fixed (efficiently computable) function.*[9]

Soundness. *If $\mathcal{O}'$ is chosen as a random function $\mathcal{O}' : \{0,1\}^* \to \{0,1\}^{\ell_{\mathrm{out}}(k)}$ and* ak *is chosen at random in $\{0,1\}^{\ell_{\mathrm{out}}(k)}$, then for every* (possibly cheating) *prover P it holds that*

$$\Pr_{\mathcal{O}',\mathrm{ak}}\left[\textit{The verifier } V^{\mathcal{O}'}(1^k, \mathrm{ak}) \textit{ accepts when talking to } P^{\mathcal{O}'}\right] \le (q + \ell_{\mathrm{out}}(k)\cdot m)^2 \cdot 2^{-\ell_{\mathrm{out}}} + 2^{-k}$$

where q is the total number of queries that P makes to the oracle $\mathcal{A}$ and m is the total number of messages that it sends to the verifier V.

Completeness with short messages. *For every polynomial-time-computable function ensemble $\mathcal{F}$, there exists a polynomial-time prover $P_{\mathcal{F}}$ such that:*

1. *For every choice of $s \in \{0,1\}^{\mathrm{poly}(k)}$ and $\mathrm{ak} \in \{0,1\}^{\ell_{\mathrm{out}}(k)}$, the verifier $V^{f_s}(1^k, \mathrm{ak})$ always accepts when talking to $P_{\mathcal{F}}(s)$.*
2. *On security parameter k, the prover $P_{\mathcal{F}}(s)$ only sends to the verifier* $\mathrm{poly}(k)$ *many messages, each of length $O((\log k)\cdot \ell_{\mathrm{out}}(k)) = O(\log^3 k)$.*

Proof Sketch: The only assertions that are not obvious are the soundness bound and the size of the messages. For the soundness bound, recall that (by Proposition 2) when $\mathcal{O}'$ is a random function (and therefore also $\mathcal{O}$ is a random function), the probability that there exists an input x that makes $\mathsf{M}^{\mathcal{O}}(1^k, x)$ accept is at most 2^{-k}. If there is no such input, then the only way to make $V^{\mathcal{O}'}$ accept is "forge" some valid paths, and (by Proposition 4) this can only be done with probability at most $q^2 2^{-\ell_{\mathrm{out}}}$. Slightly more formal, consider the transcript of a proof in which $P^{\mathcal{O}'}$ causes $V^{\mathcal{O}'}$ to accept. Considering all the messages that P sent to V in this transcript, one can easily define a "depend-on" relation among then (namely, when one message contains an authentication tag that was

[9] We slightly abuse notations here, and use V for both the verifier and the functions that it implements.

obtained in a previous message). This, in turn, allows us to define the complete configurations that were "rightly authenticated" during this transcript (namely, those configurations that correspond to a computation that starts from the initial configuration of M_k on some input x.) Hence, we either find an initial configuration from which $\mathsf{M}(1^k, x)$ accepts (a probability 2^{-k} event), or we find a computation that begins from some non-initial configuration. Since the verifier $V^{\mathcal{O}'}$ never authenticates a non-initial configuration unless it sees a valid path belonging to a configuration that directly precedes it, the valid path belonging to the first non-initial configuration must be a forgery. (By making suitable oracle calls before sending each message to the verifier, we can convert the cheating prover to a forger that makes at most $q + \ell_{\text{out}} \cdot m$ queries, and soundness follows.)

As for the size of the messages sent by the honest prover, let $\mathcal{F}$ be any polynomial-time-computable functions ensemble. This means that there is a polynomial $p(\cdot)$ such that on security parameter k, specifying any function $f_s \in \mathcal{F}_k$ can be done using at most $p(k)$ bits, and moreover, computing $f_s(x)$ for any $|x| < k$ takes at most $p(k)$ time. (Below we assume for convenience that $p(k) \geq k$.) For any $f_s \in \mathcal{F}_k$, let π_s be a description of a Turing machine computing f_s. By the above, $|\pi_s| = |s| + O(1) < p(k) + O(1)$. This implies that for any $f_s \in \mathcal{F}_k$, the non-interactive verifier $\mathsf{M}(1^k, \pi_s)$ runs in time at most $O(k + 2p(k)) \cdot p(k) = O(p^2(k))$, and therefore it only has configurations of length at most $O(p^2(k))$.

The honest prover $P_{\mathcal{F}}$, having access to s, can compute the description π_s and take the verifier step-by-step through the execution of $\mathsf{M}^{f_s}(1^k, \pi_s)$, which consists only of $O(p^2(k))$ steps. It begins by sending a message ('Init', 0, $\mathtt{sb}$), with the bound $\mathtt{sb}$ being set to $\mathtt{sb} = O(p^2(k))$, and in each step thereafter it only needs to send a constant number of paths in the tree, each of length $\log(\mathtt{sb})$. Since each node in the tree contains a string of length $\ell_{\text{out}}(k)$, it follows that the total length of the prover's queries is $O(\log(\mathtt{sb}) \cdot \ell_{\text{out}}(k)) = O(\ell_{\text{out}}(k) \cdot \log(p^2(k))) = O(\ell_{\text{out}}(k) \cdot \log k)$. □

3.4 The signature scheme

Combining the proof system from the previous section with the ideas outlined in Sections 1 and 2, it is quite straightforward to construct the desired signature scheme (summarized in the next theorem).

Theorem 6 (Theorem 1, restated) *There exists a signature scheme $\mathcal{S}$ that is existentially unforgeable under a chosen message attack in the Random Oracle Model, but such that when implemented with any efficiently computable function ensemble, the resulting scheme is totally breakable under chosen message attack. Moreover, the signing algorithm of $\mathcal{S}$ is stateless, and on security parameter k, it can only be applied to messages of size poly-logarithmic in k.*

Proof: Let $(P_{\text{pf}}, V_{\text{pf}})$ be the proof system for "non-randomness" described in Section 3.3. Let $\mathcal{S} = (G_{\text{sig}}, S_{\text{sig}}, V_{\text{sig}})$ be any stateless signature scheme that is existentially unforgeable under a chosen message attack in the Random Oracle Model (we know that such schemes exist, e.g., using Naor-Yung [13] with the random oracle used in the role of a universal one-way hash function)). We view all the machines P_{pf}, V_{pf}, G_{sig}, S_{sig}, and V_{sig} as oracle machines (although G_{sig}, S_{sig}, or V_{sig} may not use their oracle). We modify the signature scheme to obtain a different signature scheme $\mathcal{S}' = (G', S', V')$.

- On input 1^k (k being the security parameter), the key generation algorithm G' first runs G_{sig} to obtain a private/public key-pair of the original scheme, $(\text{sk}, \text{vk}) \leftarrow G_{\text{sig}}^{\mathcal{O}}(1^k)$. Then it chooses a random ℓ_{out}-bit "authentication key" $\text{ak} \in_R \{0,1\}^{\ell_{\text{out}}(k)}$ (to be used by V_{pf}). The public verification key is just vk, and the secret signing key is the pair (sk, ak). (We assume that the security parameter k is implicit in both vk and sk.)

- On message m, signing key $(\mathrm{sk}, \mathrm{ak})$ and access to oracle $\mathcal{O}$, the signature algorithm S' works as follows: If the message m is too long (i.e., $|m| > \log^4 k$) then it outputs an empty signature $\perp$.[10] Otherwise, it invokes both the proof-verifier V_{pf} and the signer S_{sig} on the message m to get $\sigma_{\mathrm{pf}} \leftarrow V_{\mathrm{pf}}^{\mathcal{O}}(\mathrm{ak}, m)$, and $\sigma_{\mathrm{sig}} \leftarrow S_{\mathrm{sig}}^{\mathcal{O}}(\mathrm{sk}, m)$.

  If the proof-verifier accepts (i.e., $\sigma_{\mathrm{pf}} =$ "accept") then the signature consists of the secret key $\sigma = (\sigma_{\mathrm{sig}}, (\mathrm{sk}, \mathrm{ak}))$. Otherwise, the signature is the pair $\sigma = (\sigma_{\mathrm{sig}}, \sigma_{\mathrm{pf}})$.

- The verification algorithm V', on message m, alleged signature $\sigma = (\sigma_1, \sigma_2)$, verification key vk and access to oracle $\mathcal{O}$, just invokes the original signature-verifier V_{sig} on the first part of the signature, outputting $V_{\mathrm{sig}}^{\mathcal{O}}(\mathrm{vk}, m, \sigma_1)$.

It is clear from the description that this scheme is stateless, and that it can only be used to sign messages of length at most $\log^4 k$. It is also easy to see that with any implementation via function ensemble, the resulting scheme is totally breakable under adaptive chosen message attack. When implemented using function ensemble $\mathcal{F}$, an attacker uses the prescribed prover $P_{\mathcal{F}}$ (of Proposition 5). Recall that the seed s for the function f_s that is used to implement the oracle is included in the public key, so the attacker can just run $P_{\mathcal{F}}(s)$. The attacker sends the prover's messages to the signer S', and the size of these messages is $O(\log^3 k) < \log^4 k$, where the constant in the O-notation depends on the ensemble $\mathcal{F}$. The second component of the signatures on these messages are the replies from the proof-verifier $V_{\mathrm{pf}}^{f_s}$. From Proposition 5 we conclude that after signing polynomially many such messages, the proof-verifier accepts (with probability one), at which point the signing algorithm will output the secret signing key. Thus, we totally break the scheme's implementation (by any function ensemble).

Next we show that the scheme $\mathcal{S}'$ is existentially unforgeable under a chosen message attack in the Random Oracle Model. Informally, the reason is that in the Random Oracle Model a forger will not be able to cause the proof-verifier to accept, and thus it will be left with the task of forging a signature with respect to the original (secure) signature scheme.

Formally, consider a polynomial-time forger F', attacking the scheme $\mathcal{S}'$, let $\epsilon = \epsilon(k)$ denote the probability that F' issues a forgery, and assume – toward contradiction – that ϵ is non-negligible. Consider the invocations that the signing algorithm makes to the proof-verifier V_{pf} during the attack. Let $\delta = \delta(k)$ be the probability that V_{pf} replies to some query with "accept". Since we can view the combination of F' and the signing algorithm as a (cheating) prover $P^{\mathcal{O}}$, Proposition 5 tells us that $\delta \leq q^2/2^{\ell_{\mathrm{out}}} + 2^{-k}$ where q is bounded by the running time of F' (which is polynomial in k). Hence δ is negligible.

Next we show a polynomial-time forger F_{sig} against the original scheme $\mathcal{S}$ that issues a forgery with probability at least $\epsilon - \delta$, contradicting the security of $\mathcal{S}$. The forger F_{sig} is given a public key vk that was generated by $G_{\mathrm{sig}}(1^k)$, it has access to the signing oracle $S_{\mathrm{sig}}^{\mathcal{O}}(\mathrm{sk}, \cdot)$ for the corresponding signing key sk, and also access to the random oracle $\mathcal{O}$. It picks at random an "authentication key" $\mathrm{ak} \in \{0,1\}^{\ell_{\mathrm{out}}(k)}$, and then invokes the forger F' on the same public key vk.

When F' asks for a signature on a message m, the forger F_{sig} behaves much like the signature algorithm S'. Namely, if $|m| > \log^4 k$ it returns $\perp$. Otherwise, it computes $\sigma_{\mathrm{pf}} \leftarrow V_{\mathrm{pf}}^{\mathcal{O}}(\mathrm{ak}, m)$, and it queries its signing oracle on m to get $\sigma_{\mathrm{sig}} \leftarrow S_{\mathrm{sig}}^{\mathcal{O}}(\mathrm{sk}, m)$. If the proof-verifier accepts, $\sigma_{\mathrm{pf}} =$ "accept", then F_{sig} aborts. Else it returns the pair $\sigma = (\sigma_{\mathrm{sig}}, \sigma_{\mathrm{pf}})$. If F' issues a forged message m' with signature (σ_1', σ_2') then F_{sig} issues the same forged message m' and signature σ_1'. It is clear that F_{sig} succeeds in forging a signature if and only if F' forges a signature without causing the proof-verifier V_{pf} to accept, which happens with probability at least $\epsilon - \delta$. ■

[10] Alternatively, S' may return $(S_{\mathrm{sig}}^{\mathcal{O}}(\mathrm{sk}, m), \perp)$, and we should note that in the ("real world") attack described below only short messages are used.

Remark 7 (Message length) Tracing through the arguments in this section, it can be seen that the message-length can be decreased from $\log^4 k$ to $\omega(\log^2 k)$: It suffices to use a space-bound $SB = \omega(\log k)$, which yields a (prescribed) proof system with prover message of length $\omega(\log^2 k)$, for any function ensemble. However, achieving poly-logarithmic message length relies heavily on the fact that we use the random oracle for authentication, and on the fact that the random oracle yields authentication with "exponential hardness". In Section 4 below, we instead use standard collision-intractable functions and message-authentication codes, that only enjoy "super-polynomial hardness". In this case, the achievable message length would be $O(k^\epsilon)$ for any desired (fixed) $\epsilon > 0$.

4 A proof system for any NP-language

The description in Section 3 combined the specifics of our application (i.e., proving non-randomness of an oracle) with the general ideas underlying the construction of the new proof system. In this section, we apply the latter ideas in order to derive a new type of proof systems for any language in $\mathcal{NP}$.

The model is similar to ordinary interactive proofs as in GMR [8] (and arguments as in BCC [3]), except that *the verifier is stateless.* That is, the verifier is represented by a randomized process that given the verifier's input and the current in-coming message, determines the verifier's next message. This process is probabilistic polynomial-time, but it cannot effect the verifier's state. In particular, the verifier's decision to accept or reject (or continue in the interaction) will be reflected in its next message. (In a sense, the verifier will not even remember its decision, but merely notify the world of it.)

The above model, per se, allows to prove membership in any NP-set, by merely having the prover send the corresponding NP-witness. However, we are interested in such proof systems in which *the prover only sends short messages.* This rules out the simple solution just suggested. But, as stated, this model does not allow to do much beyond using short NP-witnesses whenever they exist. The reason being that, from the verifier's point of view, there is no "relation" between the various communication rounds, and the only function of the multiple interactions is to provide multiple attempts of the same experiment. The situation changes once we *provide the verifier with an auxiliary secret input.* This input is chosen uniformly from some domain and remains fixed throughout the run of the protocol. The goal of this auxiliary input is to model some very limited form of state that is kept between sending a message and receiving the response.

To summarize, we are interested in proof systems (or arguments) that satisfy the following three conditions:

1. In addition to the common input, denoted x, the verifier receives an auxiliary secret input, denoted s, that is chosen uniformly from some domain. As usual, we focus on a probabilistic polynomial-time prover that also receives an auxiliary input, denoted y.

2. The verifier employs a stateless strategy. That is, there exists a probabilistic polynomial-time algorithm V such that the verifier answers the current message m with $V(x, s, m)$.

3. The prover can only send short messages. That is, it can only send messages of length $\ell(|x|)$, where $\ell(n) \ll n$ (e.g., $\ell(n) = \sqrt{n}$).

One may think of such proofs as proving statements to a child: The verifier's attention span limits us to sending it only $\ell(n)$ bits at a time, after which its attention is diverted to something else. Moreover, once we again capture the verifier's attention, it has already forgotten everything that had happened before.

Assuming the existence of collision-resistant hash functions, we can show that such a proof system can emulate any proof system (having an efficient prescribed prover strategy).[11] The emulation will only be *computationally-sound* (i.e., it is possible but not feasible to cause the verifier to accept false statements). In fact, we have already shown such a proof system: It is implicit in the description of Section 3, when one replaces the two different roles of $\mathcal{A}$ (see proof of Proposition 4) by a collision-resistant hash function and a message-authentication scheme, respectively. Indeed, the description in Section 3 referred to the emulation of a specific test, but it applies as well to the emulation of any ordinary verifier strategy (i.e., one that does maintain state between communication rounds). Specifically, one may first transform the original interactive proof to one in which the prover sends a single bit in each communication round, and then emulate the interaction of the resulting verifier by following the description in Section 3. Note that what we need to emulate in a non-trivial manner is merely the state maintained by the (resulting) verifier between communication rounds.

Comments: Since anyhow we are obtaining only a computationally-sound interactive proof (i.e., an argument system), we may as well emulate argument systems of low (total) communication complexity (cf. Kilian [10]), rather than interactive proofs or NP-proofs.[12] This way, the resulting proof system will also have low (total) communication complexity (because the length of the state maintained by the original verifier between communication rounds need not exceed the length of the total communication). (We stress that the original argument systems of low communication complexity cannot be executed, per se, in the current model, because its soundness relies on the verifier's memory of a previous message.) We also comment that (like in the description of Section 3), we can handle the case where the actual input (i.e., x) or part of it is sent to the verifier during the proof process (rather than being handed to it at the very start).

References

[1] M. Bellare and P. Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *1st Conference on Computer and Communications Security*, pages 62–73. ACM, 1993.

[2] M. Blum, W. S. Evans, P. Gemmell, S. Kannan and M. Naor, Checking the Correctness of Memories, *Algorithmica*, 12(2/3), pages 225–244, 1994. Preliminary version in *32nd FOCS*, 1991.

[3] G. Brassard, D. Chaum and C. Crépeau. Minimum Disclosure Proofs of Knowledge. *JCSS*, Vol. 37, No. 2, pages 156–189, 1988. Preliminary version by Brassard and Crépeau in *27th FOCS*, 1986.

[4] R. Canetti, O. Goldreich and S. Halevi. The Random Oracle Methodology, Revisited. Preliminary version in *Proceedings of the 30th Annual ACM Symposium on the Theory of Computing*, Dallas, TX, May 1998. ACM. TR version(s) available on-line from `http://eprint.iacr.org/1998/011` and `http://xxx.lanl.gov/abs/cs.CR/0010019`.

[11] In fact, the existence of one-way functions suffices, but this requires a minor modification of the argument used in Proposition 4. Specifically, instead of using a tree structure to hash configurations into short strings, we use the tree as an authentication tree, where collision-resistant hashing is replaced by (length-decreasing) MACs.

[12] Recall that interactive proof systems are unlikely to have low (total) communication complexity; see the work of Goldreich Vadhan and Wigderson [7].

[5] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *Journal of the ACM*, 33(4):210–217, 1986.

[6] O. Goldreich and R. Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *JACM*, Vol. 43, 1996, pages 431–473.

[7] O. Goldreich, S. Vadhan and A. Wigderson. On interactive proofs with a laconic provers. In *Proc. of the 28th ICALP*, Springer's LNCS 2076, pages 334–345, 2001.

[8] S. Goldwasser, S. Micali and C. Rackoff. The Knowledge Complexity of Interactive Proof Systems. *SICOMP*, Vol. 18, pages 186–208, 1989. Preliminary version in *17th STOC*, 1985.

[9] C. Holenstein and U. Maurer, and R. Renner. manuscript, 1998.

[10] J. Kilian. A Note on Efficient Zero-Knowledge Proofs and Arguments. In *24th STOC*, pages 723–732, 1992.

[11] R.C. Merkle. A certified digital signature. Advances in cryptology—CRYPTO '89, Vol. 435 of Lecture Notes in Computer Science, pages 218–238, Springer, New York, 1990.

[12] S. Micali. Computationally Sound Proofs. *SICOMP*, Vol. 30 (4), pages 1253–1298, 2000. Preliminary version in *35th FOCS*, 1994.

[13] M. Naor and M. Yung. Universal one-way hash functions and their cryptographic applications. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pages 33–43, 1989.