One-Time Computable Self-Erasing Functions[⋆]

Stefan Dziembowski[⋆⋆] and Tomasz Kazana[⋆⋆⋆] and Daniel Wichs[†]

Abstract. This paper studies the design of cryptographic schemes that are secure even if implemented on untrusted machines that fall under adversarial control. For example, this includes machines that are infected by a software virus.

We introduce a new cryptographic notion that we call a *one-time computable pseudorandom function (PRF)*, which is a PRF $F_K(\cdot)$ that can be evaluated *on at most one input*, even by an adversary who controls the device storing the key K, as long as: (1) the adversary cannot "leak" the key K out of the device completely (this is similar to the assumptions made in the *Bounded-Retrieval Model*), and (2) the local read/write memory of the machine is restricted, and not too much larger than the size of K. In particular, the *only way* to evaluate $F_K(x)$ on such device, is to overwrite part of the key K during the computation, thus preventing all future evaluations of $F_K(\cdot)$ at any other point $x' \neq x$. We show that this primitive can be used to construct schemes for *password protected storage* that are secure against dictionary attacks, even by a virus that infects the machine. Our constructions rely on the random-oracle model, and lower-bounds for *graphs pebbling* problems.

We show that our techniques can also be used to construct another primitive, called *uncomputable hash functions*, which are hash functions that have a short description but require a large amount of space to compute on any input. We show that this tool can be used to improve the communication complexity of *proofs-of-erasure* schemes, introduced recently by Perito and Tsudik (ESORICS 2010).

1 Introduction

A recent trend in cryptographic research is to construct cryptographic schemes that have some provable security properties, even when they are implemented on devices that are not fully trusted. In general, two types of adversarial models are considered in this area. In the *passive* one, the adversary can get some partial information ("leakage") about the internal data stored on a cryptographic machine $\mathcal{M}$. This line of research, motivated by various *side-channel attacks* [24] was initiated in the seminal papers of Ishai et al. [28] and Micali and Reyzin [32], and followed by many recent works [22,1,35,29,33,12,37,13,14,25,8,7]. Some

[⋆] This work was supported by the ERC Starting Grant, agreement CNTM-207908.

[⋆⋆] University of Warsaw and Sapienza University of Rome

[⋆⋆⋆] BioInfBank Research Institute and University of Warsaw

[†] New York University, part of this work was done while this author was visiting Sapienza University of Rome.

papers have also been motivated by the attacks of the malicious software (like viruses) against computers [11,18,17,10,21,3,2]. What all these works have in common is that they provide a formal model for reasoning about the adversary that can obtain some information about the cryptographic secrets stored on $\mathcal{M}$. It is easy to see that some restrictions on the information that the adversary can learn is necessary, as the adversary that has an unlimited access to the internal data of $\mathcal{M}$ can simply "leak" the internals in their entirety, which is usually enough to completely break any type of security. A common assumption in this area is the *bounded-retrievability property*, which states that the adversary can retrieve at most some *input-shrinking* function f of the secret K stored on the device, i.e. he can learn a value $f(K)$ such that $|f(K)| \ll |K|$. The second class of models considered in the literature [27,26,23,30] are those where the adversary is *active*, which corresponds to the so-called *tampering attacks*. In these models the adversary is allowed to maliciously modify the internals of the device. For example, in the model of [27] the adversary that can tamper a restricted number of wires of the circuit that performs the computation (in a given time-frame), and in [26] it is assumed that a device is equipped with a small tamper-free component.

The above discussion motivates the following question:

> Can we achieve security against an adversary that has *complete active/passive control* over a device $\mathcal{M}$ storing cryptographic secrets, by only relying on simple physical characteristics of the device? For which cryptographic primitives can this be achieved and what characteristics are needed?

In this work, we focus on answering the above question for new primitive called a *one-time computable pseudorandom function*. That is, we consider a device that stores a key K for a function $F_K(\cdot)$ and allows the user to evaluate the function at a single arbitrary input x. Moreover, even if an adversary gains complete control of the device, he should be unable to learn any information, beyond a single value $F_K(x)$ at a single point x. We rely on the following two physical characteristics of the device $\mathcal{M}$ on which the secret key K is stored: (1) $\mathcal{M}$ satisfies the bounded-retrievability property, and (2) the read/write memory of $\mathcal{M}$ is restricted in size and not much larger than the size of the key K. Intuitively, the first property ensures that the attacker cannot leak the key K out of the device, while the second property will prevent the attacker from evaluating $F_K(\cdot)$ at multiple inputs using the resources of the device itself. The main application of this primitive is a scheme for password-protected storage. We also construct another, related primitive, that we call *uncomputable hash functions*, and use it construct an improved protocol for *proofs-of-erasure*, a concept recently introduced in [34].

1.1 One-Time Computable Functions

In this section we informally define the concept of a one-time computable PRF $F_K(\cdot)$ implemented on a resource-constrained device $\mathcal{M}$. Firstly, for correctness/functinality, we need to ensure that the key K of the PRF can be stored on the device $\mathcal{M}$ and that there is a method for honestly evaluating $F_K(\cdot)$ on a single arbitrary input x, using the resources of the device $\mathcal{M}$. Secondly, for security, we consider an attacker that gains control of the device $\mathcal{M}$. Such an adversary may learn the value $F_K(x)$ for some arbitrary point x, but should not have any other information about $F_K(x')$ for any other point $x' \neq x$.

So far we have not been very specific about the type of constraints placed on the resources of the device $\mathcal{M}$, and the type of control that the adversary gets over the device. One could imagine settings in which the above would be easy to implement. For example, if the adversary only gets black-box access to $\mathcal{M}$ then we can use an arbitrary PRF to achieve the above goal; simply have the device only perform only one evaluation of the PRF and then set a flag to stop responding to all future inputs. However, if the adversary can perform even relatively simple tampering attacks, it may be possible for it to "reset" the flag on the device and thus break security of the above solution.

In this work, we consider an adversary that has *complete control* over the device $\mathcal{M}$. That is, for the purpose of security, we can consider the device $\mathcal{M}$ itself to be a resource-constrained adversarial entity that gets the key K, and can communicate with an external unconstrained adversary $\mathcal{A}$. In this case, we must place some constraints on the resources of $\mathcal{M}$. Firstly, we must bound the amount of outgoing communication available to the device $\mathcal{M}$, as otherwise the $\mathcal{M}$ can just "leak" the entire key K to the external adversary $\mathcal{A}$, who can then evaluate $F_K(\cdot)$ at arbitrarily many points. Secondly, we must also place some limits on the computational resources available to $\mathcal{M}$, to prevent it from e.g. evaluating $F_K(x_0), F_K(x_1)$ at two points $x_0 \neq x_1$ and leaking the first bit of each output to the external adversary $\mathcal{A}$. In this work, we will assume that the amount of read/write memory available to the device $\mathcal{M}$ is bounded, and not much larger than the size of the key K. (The device can have arbitrary additional read-only or write-only memory).

Putting this together, our goal is to design a PRF $F_K(\cdot)$ which can be efficiently evaluated at any single point x on a memory-constrained device, but cannot be evaluated at any additional point $x' \neq x$ afterward. Roughly, our main idea is to construct F_K in such a way that any computation of $F_K(x)$ has to (at least partially) destroy K, by overwriting it, and thus prevents future computations of the function. That is, we assume that the key K itself is stored on the read/write memory of the device and takes up $m = |K|$ bits of space, which is a large fraction of the total. We design the PRF in such a way that there is an honest computation of $F_K(x)$ that uses (almost) no extra space beyond that on which K was originally stored, but overwrites K with various intermediate values during the computation. On the other hand, assuming the total memory on the device is $s < 2m$, we show that there is *no* method for computing of $F_K(x)$ at any single point x, without erasing part of the key K and preventing evaluation at any other input. Note that it is necessary for us to require that the key takes up more than half of the available read/write memory of the device, as otherwise it is possible to make a "copy" of the key that does not get damaged during the computation $F_K(x)$. In fact, we show a stronger result along these lines, where we also allow the adversarial memory-constrained device $\mathcal{M}$ to communicate up to $c < m$ bits to an external unconstrained adversary $\mathcal{A}$ (and we allow unbounded communication from $\mathcal{A}$ to the device).

One-time computable functions – a generalization. We also construct a generalization of the concept described above, where a single key K defines T different pseudorandom functions: $(F_{1,K}, \ldots . F_{T,K})$. Using the resources of the device, the honest user can evaluate each of the function $F_{i,K}$ at a single arbitrary point (i.e. the user first chooses an arbitrary x_1 and evaluates $F_{1,K}(x_1)$, then adaptively chooses x_2 and evaluates $F_{2,K}(x_2)$ $\ldots$). However, even if the device is under full adversarial control, the attacker cannot get information about any of the T functions at more than one point – i.e. the attacker cannot get information about $F_{i,K}(x), F_{i,K}(x')$ for any two distinct points $x \neq x'$ and the same index i. The construction is given in Section 5. The maximal T that we can have is approximately equal to $\frac{c+s}{2(c+s-m)}$ (cf. (3)).

Application: Password-protected storage Let us now describe an application of the primitives described above. Our application is related to *password-based cryptography*, which is an area that deals with the protocols where the secrets used by the parties are human-memorizable passwords. The crucial difference between a password and a cryptographic key is that the latter is usually assumed to be chosen uniformly at random from a large domain, while the former may come from some relatively small (polynomial sized) *dictionary set* $\mathcal{D}$. One of the main problems in constructing the password-based protocols is that one needs to consider the so-called *offline dictionary attacks*, where the adversary simply tries to break the scheme by analyzing all of the passwords from $\mathcal{D}$ one-by-one.

In this paper we are particularly interested in designing schemes for *password-protected storage*, which are schemes for secure encryption of data using passwords. A typical scheme of this type works as follows: let $\pi \in \mathcal{D}$ be a password. To encrypt a message X we apply a *key-derivation function* H to π and then encrypt X with $H(\pi)$ using some standard symmetric encryption scheme $(\mathit{Enc}, \mathit{Dec})$. To decrypt a ciphertext $C = \mathit{Enc}(H(\pi), X)$ one simply calculates $\mathit{Dec}(H(\pi), C)$.

A typical choice for H is a hash function. This solution is vulnerable to a following offline dictionary attack. An attacker simply tries, for every $\pi' \in \mathcal{D}$ to decrypt C until he finds π' such that $\mathit{Dec}(H(\pi'), C)$ "makes sense". Most likely there will be only one such π', and hence, with a good probability, this will be the correct π that the user has chosen to compute C.

A common way to make this attack harder is to design H in such a way that it is moderately expensive to compute it. The time needed to compute H should be acceptable for a legitimate user, and to high for the adversary if he has to do it for all passwords in $\mathcal{D}$. A drawback of this solution is that it depends on the amount of computing power available to the adversary. Moreover, the algorithm of the adversary can be easily parallelized.

An interesting solution to this problem was proposed in [9]. Here, a computation of H requires the user to solve the CAPTCHA puzzles [38], which are small puzzles that are easy to solve by a human, and hard to solve be a machine. A disadvantage of this solution is that it imposes additional burden on the user (he needs to solve the CAPTCHAs when he wants to access his data). Moreover, experience shows that designing secure CAPTCHAs gets increasingly difficult.

In this paper we show an alternative solution to this problem. Our solution works in a model where the data is stored on some machine that can be infected by a virus. In this model, the virus can get a total control over the machine, but he can retrieve only c bits from it. The main idea is that we will use a one-time computable function F (secure against an adversary with c-bounded communication and s-bounded storage) as the key-derivation function. To encrypt a message X with a password π we first choose randomly a key R for a one-time computable PRF. We then calculate $K = F_R(\pi)$. The ciphertext stored on the machine is $Enc(K, X)$. It is now clear that the honest user can easily compute K in space bounded by $c - \delta$. On the other hand, the adversary can compute K only once, even if he has space c. Of course, the adversary could use a part of the ciphertext $Enc(K, X)$ as his additional storage. This is not a problem if X is short (shorter than δ). If X is long, we can solve this problem by assuming that $Enc(K, X)$ is stored on a read-only memory.

A problem with this solution is that if an honest user makes an error and types in a wrong password then he does not have a chance to try another password. This can be solved by using the generalized version of the one-time computable functions. The scheme works as follows. First, we choose a key K for symmetric encryption. Then, we choose randomly R and for each $i = 1, \ldots, T$ we calculate $K_i = F_{R_i}(\pi) \oplus K$ (where the keys R_i are derived from R). The values that are stored on the machine are $(R, (K_1, \ldots, K_T), Enc(K, M))$. Now, to decrypt the message, the user first calculates $K = F_{R_1}(\pi) \oplus K_1$, and then decrypts $Enc(K, M)$ using K. If a user makes an error and calculates K_1 using a wrong π he still has a chance to calculate K_2, and so on.

1.2 Uncomputable functions

We also introduce a notion of *uncomputable* hash functions, which we explain here informally. A hash function H is (s, ϵ)*-uncomputable*, if any machine that uses space s and takes a random input $x \in \{0, 1\}^*$ outputs $H(x)$ with probability at most ϵ. We say that H is s'*-computable* if it *can* be computed in space s'. Note that in this case we assume that the adversary cannot use any external help to compute H (using the terminology from the previous sections: his communication is 0-bounded). Informally, we are interested in constructing (s, ϵ)-uncomputable, s'-computable functions for a small ϵ and s' being only slightly larger than s.

This notion can be used to construct an improved scheme for the *proof of erasure*, a concept recently introduced in [34]. Essentially the proof of erasure is a protocol between two parties: a powerful verifier $\mathcal{V}$ and a weak prover $\mathcal{P}$ (that can be, e.g., an embedded device). The goal of the verifier is to ensure that the prover has erased all the data that he stores in his RAM (we assume that $\mathcal{P}$ can also have a small ROM). This is done by forcing $\mathcal{P}$ to overwrite his RAM. Let m be the size of RAM. Then, a simple proof of erasure consists of $\mathcal{V}$ sending to $\mathcal{P}$ a random string R such that $|R| = m$, and then $\mathcal{V}$ replying with R. In [34] the authors observe that the communication from $\mathcal{P}$ to $\mathcal{V}$ can be reduced in the following way: instead of asking $\mathcal{P}$ to send the entire R, we can just verify his knowledge of R using a protocol for the "proof of data possession" (see, e.g., [4]). Such a protocol still requires the verifier to send a large string R to the prover, hence the communication from the verifier to the prover is m. Using our uncomputable functions we can reduce this communication significantly.

Our idea as follows. Suppose we have a function H that is m-computable and $(m - \delta, \epsilon)$-uncomputable (for some small $\delta \in \mathcal{N}$ and a negligible $\epsilon \in [0, 1]$). Moreover, assume that H has a short domain and co-domain, say: $H : \{0,1\}^w \to \{0,1\}^w$ for some $w \ll m$. We can now design the following protocol:

1. $\mathcal{V}$ selects $X \leftarrow \{0,1\}^w$ at random and sends it to $\mathcal{P}$,
2. $\mathcal{P}$ calculates $Y = H(X)$ and sends it back to $\mathcal{V}$,
3. $\mathcal{V}$ accepts if $Y = H(X)$.

Clearly, an honest prover can calculate Y, since he has enough memory for this. On the other hand, from the $(m - \delta, \epsilon)$-uncomputability of H we get that a cheating prover cannot calculate Y with probability greater than ϵ without overwriting $m - \delta$ bits. The total communication between $\mathcal{P}$ and $\mathcal{V}$ has length $2w$. Note, that we need to assume that an adversary that controls the prover cannot communicate any data outside of the machine (therefore we are interested only in protocols with 0-bounded communication). This is because otherwise he could simply forward X to some external party that has more memory. The same assumption needs to be made in the protocols of [34]. What remains is to show a construction of such an H. We do it in Section 6.

1.3 Related work

Most of the related work was already described in the previous sections. In our paper we will use a technique called *graph pebbling* (see e.g. [36]). This technique has already been used in cryptography in an important work of [16], some of our methods were inspired by this paper. The assumption that the adversary is memory-bounded has been used in the so-called *bounded-storage model* [31,5,20]. As similar assumption was also used in [15]. The proof of erasures can be viewed as a special case of the *remote attestation protocols* (see [34] for a list of relevant references).

1.4 Notation

For a sequence $R = (R_1, \ldots, R_N)$ and for indices i, j such that $1 \leq i \leq j \leq N$, we define $R[i, \ldots, j] = (R_i, \ldots, R_j)$.

2 Model of Computation

To make our statements precise, we must fix a model of computation. We will usually consider an adversary that consists of two parts: a "space-bounded" component $\mathcal{A}_{small}$ which gets access to the internals of an attacked device and has "bounded communication" to an external, and otherwise unrestricted, adversary $\mathcal{A}_{big}$.

Since the lower bounds on the computational complexity of functions are usually hard to prove, it seems difficult to show any meaningful statements in this model using purely complexity-theoretic settings. We will therefore use the *random-oracle model* [6]. Recall, that in this case a hash function is modeled as an external oracle containing a random function, and the oracle can be queried by all the parties in the protocol (including the adversary).

Using the random-oracle model in our case is a little bit tricky. To illustrate the problem consider a following protocol for the proof of erasure: (1) $\mathcal{V}$ sends to $\mathcal{P}$ a long random string R, (2) $\mathcal{P}$ replies with $H(R)$, where H is a hash function. Now, this protocol is obviously not secure for most of the real-life hash functions. For example, if H is designed using the Merkle-Damgård paradigm, then it can be computed "on fly", and hence there is no need to store the whole R before starting the computation of H.

On the other hand, if we model H as a random oracle, then the protocol described above can be proven secure, as the adversary has to wait until he gets the complete R before sending it to the oracle. We solve this problem in the following way: we will require that the only way in which the hash function is used is that it is applied to small inputs, i.e. if w is the length of the output of a hash function (e.g.: $w = 128$) then the hash function will have a type $H : \{0,1\}^{\xi w} \to \{0,1\}^w$, for some small ξ. Observe that if $\xi = 2$ then the function H can simply be a *compression function* used to construct the hash function).

We model our adversary $\mathcal{A} = (\mathcal{A}_{big}, \mathcal{A}_{small})$ as a pair of interactive algorithms[1] with oracle-access to a random-oracle $H(\cdot)$. The algorithm $\mathcal{A}_{big}$ will only be restricted in the number of oracle calls made. On the other hand, we impose the following additional restrictions on $\mathcal{A}_{small}$:

- s-bounded space: The total amount of space used by $\mathcal{A}_{small}$ is bounded by s. That is, we can accurately describe the entire configuration of $\mathcal{A}_{small}$ at any point in time using s bits.[2]
- c-bounded communication: The total number of outgoing bits communicated by $\mathcal{A}_{small}$ is bounded by c. [3]

[1] Say ITMs, interactive RAMs, ... The exact model will not matter.

[2] This is somewhat different than standard space-complexity considered in complexity theory, even when we restrict the discussion to ITMs. Firstly, the configuration of $\mathcal{A}_{small}$ includes the value of *all* tapes, including the input tape. Secondly, it includes the current state that the machine is in and the position of all the tape heads.

[3] To be precise, we assume that we can completely describe the patters of outgoing communication of $\mathcal{A}_{small}$ using c bits. That is, $\mathcal{A}_{small}$ cannot convey additional information in when it sends these bits, how many bits are sent at a given time and so on...

We use the notation $\mathcal{A}^{H(\cdot)}(R) = \left(\mathcal{A}_{big}^{H(\cdot)}() \leftrightarrows \mathcal{A}_{small}^{H(\cdot)}(R)\right)$ to denote the interactive execution of $\mathcal{A}_{big}$ and $\mathcal{A}_{small}$, where $\mathcal{A}_{small}$ gets input R and both machines have access to the oracle $H(\cdot)$.

3 Definitions

Let $W^{H(\cdot)}$ be an algorithm that takes as input $R \in \{0,1\}^m$ and has access to the oracle H. Let $(F_{1,R}^H, \ldots . F_{T,R}^H)$ be sequence of functions that depend on H and R. Assume that $W^{H(\cdot)}$ is interactive, i.e. it may receive queries from the outside. Let $x_1, \ldots, x_T$ be the sequence of queries that $W^{H(\cdot)}$ received. The algorithm $W^{H(\cdot)}$ replies to such a query by issuing a special *output query* to the oracle H. We assume that after receiving each $x_i \in \{0,1\}^*$ the algorithm $W^{H(\cdot)}$ always issues an output query to H of a form $((F_{i,R}^H(x_i), (i, x_i)), \texttt{out})$. We say that $W^{H(\cdot)}$ is a $(c, s, m, q, \epsilon, T)$-*onetime computable PRF* if:

- $W^{H(\cdot)}$ has m-bounded storage, and 0-bounded communication.
- for any $\mathcal{A}^{H(\cdot)}(R)$ that makes at most q queries to H and has s-bounded storage and c-bounded communication, the probability that $\mathcal{A}^{H(\cdot)}(R)$ (for a randomly chosen $R \leftarrow \{0,1\}^m$) issues two queries $((F_{i,R}^H(x), (i, x)), \texttt{out})$ and $((F_{i,R}^H(x'), (i, x')), \texttt{out})$, for $x \neq x'$, is at most ϵ.

Basically, what this definition states is that no adversary with s-bounded storage and c-bounded communication can compute the value of any $F_{i,R}$ on two different inputs. It may look suspicious that we defined the secrecy of a value in terms of the hardness of guessing it, instead of using the indistinguishability paradigm. We now argue why our approach is ok. There are two reasons for this. The first one is that in the schemes that we construct that output of each F^H is always equal to some output of H (i.e. the algorithm F simply outputs on the the responses he got from H). Hence $\mathcal{A}$ cannot have a "partial knowledge" of the output (either he was lucky and he queried H on the "right" inputs, or not – in the latter case the output is indistinguishable from random, from his point of view).

The second reason is that, even if it was not the case — i.e. even if F^H outputted some value y that is a more complicated function of the responses he got from H — we could modify F^H by hashing y with H (and hence if y is "hard to guess" then $H(y)$ would be completely random, with a high probability).

Now, suppose that $V^{H(\cdot)}$ is defined identically to $W^{H(\cdot)}$ with the only difference that it receives just one query $x \in \{0,1\}^*$, and afterwards it issues one output query $((F^H(x), x), \texttt{out})$ (for some function F that depends on H). We say that $V^{H(\cdot)}$ is an $(s, w, q, \delta, \epsilon)$-*uncomputable hash function* if:

- $V^{H(\cdot)}$ has s-bounded storage, and 0-bounded communication.
- for any $\mathcal{A}^{H(\cdot)}(R)$ that makes at most q queries to H and has $(s-\delta)$-bounded storage and c-bounded communication, the probability that $\mathcal{A}^{H(\cdot)}(R)$ (for a randomly chosen $R \leftarrow \{0,1\}^w$) issues a query $((F^H(x), x_i), \texttt{out})$ is at most ϵ.

4 Random Oracle Graphs and the Pebbling Game

We show a connection between an adversary computing a "random oracle graph" and a pebbling strategy for the corresponding graph. A similar connection appears in [16].

4.1 Random-Oracle Labeling of a Graph.

Let $G = (V, E)$ be a DAG with $|V| = N$ vertices. Without loss of generality, we will just assume that $V = \{1, \ldots, N\}$ (we will also consider infinite graphs, in which case we will have $N = \infty$). We call vertices with no incoming edges *input vertices*, and will assume there are $M \leq N$ of them. A *labeling* of G is a function $\texttt{label}(\cdot)$, which assigns values $\texttt{label}(v) \in \{0,1\}^w$ to vertices $v \in V$. We call w the *label-length.* For any function $H : \{0,1\}^* \to \{0,1\}^w$ and input-labels $R = (R_1, \ldots, R_M)$ with $R_i \in \{0,1\}^w$, we define the (H, R)-labeling of G as follows:

- The labels of the M distinct input vertices $v_1 < v_2 < \ldots < v_M$ are given by $\texttt{label}(v_i) \overset{\text{def}}{=} R_i$.
- The label of every other vertex v is defined recursively by

$$\texttt{label}(v) \overset{\text{def}}{=} H(\texttt{label}(v_1), \ldots, \texttt{label}(v_d), v)$$

  where $v_1 < \ldots < v_d$ are the d parents of v.

A *random oracle labeling* of G is an (H, R)-labeling of G where H is a random-function and R is chosen uniformly at random.

For convenience, we also define $\texttt{preLabel}(v) \overset{\text{def}}{=} (\texttt{label}(v_1), \ldots, \texttt{label}(v_d), v)$, where $v_1 < \ldots < v_d$ are the parents of v, so that $\texttt{label}(v) = H(\texttt{preLabel}(v))$.

The *output vertices* of G are the vertices that have no children. Let $v_1, \ldots, v_K$ be the output vertices of G. Let $\mathit{Eval}(G, H, (R_1, \ldots, R_M))$ denote the sequence of labels $(\texttt{label}(v_1), \ldots, \texttt{label}(v_K))$ of the output vertices calculated with the procedure described above (with $R_1, \ldots, R_M$ being the labels of the input vertices $v_1, \ldots, v_M$ and H being the hash function).

Our main goal is to show that computing the labeling of a graph G requires a large amount of resources in the random-oracle model, and is therefore difficult. We will usually (only) care about the list of random-oracle calls made by $\mathcal{A}_{big}$ and $\mathcal{A}_{small}$ during such an execution. We say that an execution $\mathcal{A}^{H(\cdot)}(R)$ *labels* a vertex v, if a random-oracle call to $\texttt{preLabel}(v)$, is made by either $\mathcal{A}_{big}$ or $\mathcal{A}_{small}$.

4.2 Pebbling Game

We will consider a new variant of the pebble game that we call the "red-black" pebble game over a graph G. Each vertex of the graph G can either be empty, contain a red pebble, contain a black pebble, or contain both types of pebbles. An initial configuration consists of (only) a black pebble placed on each input vertex of G. The game proceeds in steps where, in each step, one of the following four actions is taken:

1. A red pebble can be placed on any vertex already containing a black pebble.
2. If all the parents of a vertex v have a red pebble on them, a red pebble can be placed on v.
3. If all the parents of v have *some* pebble on them (red or black), a black pebble can be placed on v.
4. A black pebble can be removed from any vertex.

We define the *black-pebble complexity* of a pebbling strategy to be the maximum number of *black pebbles* in use at any given time. We define the *red-pebble complexity* of a pebbling strategy to be the total number of steps in which action 1 is taken. We also define the *all-pebble complexity* of a pebbling strategy to be the sum of its black- and red-pebble complexities. By *heavy-pebbles* we will mean the black pebbles, or the red-pebbles that appeared on the graph because of action 1. Note, that these are exactly the pebbles that count when we calculate the all-pebble complexity of a strategy.

Remark 1. Let G be a graph with N vertices and M input vertices. Let v be an output vertex of G and let $v_{i_1}, \dots, v_{i_d}$ be a subset of the set of input vertices. Suppose there exists a pebbling strategy that (1) pebbles v while keeping the pebbles on the vertices $v_{i_1}, \dots, v_{i_d}$, and (2) has black-pebble complexity b and it does not use the red pebbles, i.e. its red-pebble complexity 0. Then the value of $\mathit{Eval}(G, H, (R_1, \dots, R_M))$ can be computed by a machine with bw-bounded storage and an access to a random oracle that computes H. This is because the only thing that the machine needs to remember are the labels of at most b vertices (each of those labels has length at most w). The computation may overwrite some part of the input $(R_1, \dots, R_M)$, however, it does not overwrite the input corresponding to the vertices $v_{i_1}, \dots, v_{i_d}$, i.e.: $(R_{v_1}, \dots, R_{v_d})$.

It is more complicated to show a connection in the opposite direction, namely to prove that if a graph cannot be pebbled with a strategy with low black- and red-complexities, then it cannot be computed by a machine with a restricted storage and communication. The following lemma establishes such a connection (its proof, that appears in [19], is an extension of the proof of Lemma 1 in [16].)

Lemma 1. *Let G be a DAG with M input vertices and K output vertices. Let r and b be arbitrary parameters. Suppose that G is such that there does not exist a pebbling strategy such that (1) its all-pebble complexity is at most a, and that (2) pebbles at least α output vertices (for some $\alpha \in \{1, \dots, K\}$).*

Then, for any c, s, w such that $\frac{c+s+w}{w-\log(q)} < a$, and for any $\mathcal{A} = (\mathcal{A}_{big}, \mathcal{A}_{small})$ that makes at most q oracle calls and has s-bounded space and c-bounded communication the probability that $\mathcal{A}$ labels more than $\alpha - 1$ output vertices is at most $(q+1) \cdot 2^{-w}$ (where the probability is taken over the randomness of $\mathcal{A}$ and the random choice of H and R).

5 One-time computable functions

In this section we show specific examples of graphs that are hard to pebble in limited space and bounded communication. Let M, M' be a parameters such that $M' < M$. The (M, M')-*lambda graph* (denoted $Lam^M_{M'}$) is defined in the following way (cf. Fig. 1). Its set of vertices is equal to $V_0 \cup V_1$, where $V_0 =$

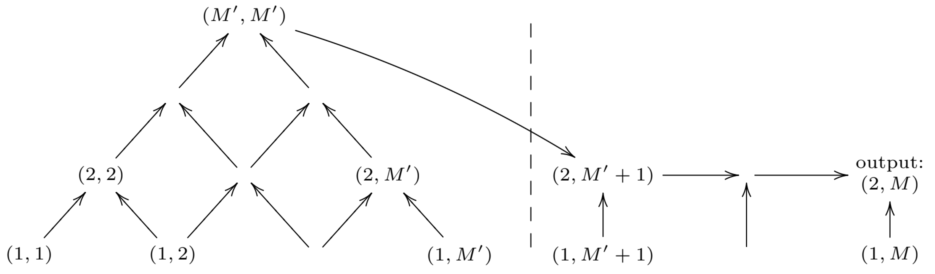


Fig. 1. An (M, M')-lambda graph for $M' = 4$ and $M = 7$. The sub-graph on the left-hand side of the dashed line is an M'-pyramid.

$\{(i, j) : 1 \leq i \leq j \leq M'\}$ and $V_1 = \{1, 2\} \times \{M' + 1, \ldots, M\}$. The set of input vertices is equal to $\{1\} \times \{1, \ldots, M\}$. The output vertex is $(2, M)$. The set of edges is equal to the following sum: $\{((i-1, j-1), (i, j)) : (i-1, j-1), (i, j) \in V_0\} \cup \{((i-1, j), (i, j)) : (i-1, j), (i, j) \in V_0\} \cup \{((M', M'), (2, M'+1))\} \cup \{((1, j-1), (1, j)) : (1, j-1), (1, j) \in V_1\} \cup \{((1, j), (2, j)) : (1, j), (2, j) \in V_1\}$. If $M' = M$ then a (M, M)-lambda graph is defined as above, with $V_1 = \emptyset$ and with the set of edges consisting only of the first two summands of the sum above. Its output vertex is (M, M). Such a graph is also called an *M-pyramid graph.*

Lemma 2. *For any $X < M' - 1$ there exists a strategy that pebbles the output vertex of $Lam^M_{M'}$ that satisfies the following:*

- *it uses $M + M' - 1 - X$ black pebbles (remember that all the M input vertices are initially pebbled with a black pebble, and therefore using $M + M' - 1 - X$ means having $M' - 1 - X$* extra *pebbles),*
- *it uses no red pebbles, and*
- *at the moment when the output vertex is pebbled there are still pebbles on the last $M - X$ input vertices, i.e.: vertices from the set $\{1\} \times \{X + 1, \ldots, M\}$.*

Proof. The pebbling strategy consists of the following steps:

pebble the second row of the pyramid In this step we pebble the second row of the pyramid, i.e. the vertices from the set $\{2\} \times \{2, \ldots, M'\}$. We do it by removing X pebbles from the input of the pyramid, and by using the $M' - 1 - X$ extra pebbles that we have. The procedure is as follows:

1. First, we put pebbles on the vertices from the set $\{2\} \times \{2, \ldots, X + 1\}$. We do it in the following way: for $j = 2, \ldots, X' + 1$ we put a pebble on $(2, j)$ and remove it from $(1, j - 1)$.

2. We then put pebbles on the vertices from the set $\{2\} \times \{X+2, M'\}$. We do it just using the extra pebbles, without removing any pebble from the input. Clearly we have enough extra pebbles, since $|\{2\} \times \{X+2, M'\}| = M' - 1 - X$.

pebble the rest of the pyramid In this step we pebble the pyramid row-by-row, starting from the third row, and ending with the top of the pyramid (M', M'). We do it in the by executing the following procedure for $i = 3, \ldots, M'$:
  - for $j = i, \ldots, M'$ do the following: put a pebble on (i, j) and remove it from $(i-1, j-1)$.

pebble the rest of the graph We now pebble the rest of the graph in the following way. First, we put a pebble on $(2, M'+1)$ and remove it from (M', M'). Then, for $j = M'+2, \ldots, M$ we put a pebble on $(2, j)$ and remove it from $(2, j-1)$. At the end of this loop there output vertex is pebbled.

It is easy to see that the above procedure results in a correct pebbling strategy. Moreover, it uses only $M' - 1 - X$ extra pebbles, and it removes the pebbles only from the first X vertices of the input. □

5.1 Hardness of pebbling

Consider a configuration of the red and black pebbles on some DAG G. Let v be a vertex of G. We say that v *is input-dependent in this configuration* if, after removing all the pebbles from the input it is impossible to pebble the vertex v. If v is not input-dependent then we say that it is *input-independent*.

Lemma 3. *For $M \geq 2$ consider an M-pyramid graph Lam_M^M and some configuration of pebbles on it. If the output vertex (M, M) is input-dependent then the number of heavy pebbles is at least M.*

Proof. We prove it by induction on $M = 2, 3, \ldots$. To root the induction we first consider the case when $M = 2$. In this case the graph consists of 3 vertices only: 2 input vertices, and 1 output vertex. If it is input-dependent then the output vertex is not pebbled. Hence both input vertices need to have a pebble.

Now, let us assume the hypothesis for $M-1$ and consider $G_M = Lam_M^M$. Take some configuration γ of pebbles. Denote the set of heavy pebbles in γ by $\mathcal{X}$. Let G_{M-1} be a subgraph of G_M induced by all the vertices of G_M except of the input row (in other words: G_{M-1} is equal to G_M with the bottom row "cut"). Of course G_{M-1} is Lam_{M-1}^{M-1}.

Now, put black pebbles on the vertices of the input row of G_{M-1} in the following way: put a pebble on a vertex v whenever v has both parents in $\mathcal{X}$ (and keep the old pebbles from the configuration γ). It is easy to see that the number of black pebbles in this new configuration is at most $|\mathcal{X}| - 1$.

Clearly the resulting configuration of pebbles on G_{M-1} satisfies the following: (1) the output vertex can be pebbled from this configuration, and (2) the output vertex on G_{M-1} is input-dependent (if it was not input-dependent then also the configuration γ would not be input-dependent). Hence, by the induction hypothesis $|\mathcal{X}| - 1 \geq M - 1$, which implies that $|\mathcal{X}| \geq M$. □

Lemma 4. *Suppose $M > 2$. Consider a pebbling strategy for Lam_M^M that pebbles the vertex (M, M). In the first configuration in which (M, M) is input-independent we have that: (1) the total number of the heavy pebbles that are not on the input row is at least $M - 1$, and (2) there is no pebble on (M, M).*

Proof. Let $G_M = Lam_M^M$, and let G_{M-1} be defined as in the proof of Lemma 3. Let γ be the first configuration in which (M, M) is input-independent, and let γ' be the configuration that directly precedes γ, i.e. the last configuration that is input-dependent. Keep on the vertices of G_{M-1} all the pebbles from the configuration γ. We now show that in such a configuration of the pebbles on G_{M-1} the output of G_{M-1} is input-dependent. After showing it we will be done: part (1) will follow directly from Lemma 3 (applied to G^{M-1}), and part (2) will follow from the fact that (for $M - 1 > 1$) if the output vertex is input-dependent then it cannot be pebbled.

To finish the proof assume that the output of G_{M-1} is input-independent. We obtain contradiction by showing that in this case also G_M needs to be input-independent. Clearly the only way in which γ' was transformed into γ was that a pebble was added on the input row of G_{M-1}. However, by our assumption the output of G_{M-1} (and hence also of G_M) does not depend on this row. Therefore also in the configuration γ' the output cannot depend on the two bottom rows of G_M. This gives us a contradiction. □

Lemma 5. *Consider a pebbling strategy that pebbles the output of $Lam_{M'}^M$. As long as the vertex (M', M') has not been pebbled, there has to be a heavy pebble on every input vertex on the second part of $Lam_{M'}^M$ (i.e. the vertices $(1, j)$ such that $j \in \{M' + 1, \ldots, M\}$).*

Proof. This follows easily from the construction of the $Lam_{M'}^M$ graph: if one removes a pebble from any vertex $(1, j)$ such that $j \in \{M' + 1, \ldots, M\}$ then one cannot put a pebble on it in the future. Therefore it will never be possible to pebble $(2, j)$, and hence also $(2, M)$. □

For $\ell \in \mathcal{N} \cup \{\infty\}$ consider a family of ℓ DAGs $\{G_k = (V_k, E_k)\}_{k=1}^{\ell}$ such that every DAG in this family has the same set of input V_I of input vertices. Define $V_k' = V_k \setminus V_I$. The graph $G = (V, E)$ is a *sum of* $\{G_k = (V_k, E_k)\}_{k=1}^{\ell}$ if is defined as follows: the set of vertices V is equal to V_I plus the disjoint sum of the sets V_k'. More precisely: $V := V_I \cup \bigcup_{k=1}^{\ell} \{k\} \times V_k$. The set E of edges is defined as: $E := \{((k, v), (k, v')) : v, v' \in V_k' \text{ and } (v, v') \in E_k\} \cup \{(v, (k, v')) : v \in V_I \text{ and } v' \in V_k' \text{ and } (v, v') \in E_k\}$. The set of input vertices of G is equal to V_I, and the set of the output vertices is equal to $V_{O,L} \cup V_{O,R}$, where $V_{O,L}$ and $V_{O,R}$ are the sets of the output verices of G_L and G_R, respectively.

Lemma 6. *Consider a family $\{G_k\}_{k=1}^{\ell}$ of (M, M')-*lambda graphs*. Let G be a sum of the graphs in this family. Then there does not exist a pebbling strategy with all-pebble complexity bounded by $M + M' - 2$ that pebbles more than one output of G.*

Proof. For the sake of contradiction suppose that such a strategy exists. Pebbling the output of $Lam_{M'}^{M}$ requires first pebbling the top of the pyramid graph that is a part of $Lam_{M'}^{M}$. Therefore there has to exist a pebbling strategy with all-pebble complexity bounded by $M + M' - 2$ that pebbles two different vertices that are the tops of the pyramids in some G_k and G_h (i.e. vertices $(k, (M', M'))$ and $(h, (M', M'))$).

Clearly, at the beginning of any pebbling strategy the top of each pyramid is dependent on the input of this pyramid. Consider the first configuration where the top of one of the pyramids, the one belonging to G_k, say, gets independent from the input of this pyramid. In this moment, by Lemma 4 the total number of the heavy pebbles that are not on the input row of G_k is at least $M' - 1$. Since in this moment the top vertex of G_h is still dependent on the input, hence, by Lemma 3 the total number of the heavy primary red pebbles and the black pebbles on G_k is at least M'. Therefore the number of the heavy pebbles on the two pyramids is at least $2M' - 1$.

On the other hand, by the second part of Lemma 4 the vertex (M', M') is not yet pebbled in this configuration. Hence, by Lemma 5, there needs to be a heavy pebble on every vertex from the second part of the input of G_h and G_k, i.e. on the vertices $(1, j))$ such that $j \in \{M' + 1, \ldots, M\}$. Therefore altogether we have $2M' - 1 + (M - M') = M + M' - 1$ pebbles on the sum of G_h and G_k. This yields a contradiction with the assumption that the all-pebble complexity of the strategy is bounded by $M + M' - 2$. $\square$

Combining Lemma 1 with Lemma 6 we get the following.

Corollary 1. *Consider a family $\{G_k\}_{k=1}^{\ell}$ of (M, M')-lambda graphs. Let G be a sum of the graphs in this family. Then, for any s, c, w and q, such that $\frac{c+s+w}{w-\log(q)} < M + M' - 2$, and any adversary $\mathcal{A}$ that has s-bounded storage and c-bounded communication, and makes at most q queries to the oracle, the probability that $\mathcal{A}$ labels more than one output of G is at most $(q + 1) \cdot 2^{-w}$.*

5.2 The construction

In our construction the hash function will depend on an additional parameter a. Formally, let $H : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^w$ be a function that is modeled as a random oracle. For any $a \in \{0,1\}^*$ let H^a denote a function defined as $H^a(z) = H(a, z)$. Let M, U and T be some positive integer parameters such that

$$T < \frac{U - 1}{2\varDelta} \tag{1}$$

where $\varDelta := U - M$. We now construct an interactive algorithm $COMP_{U,M,T,w}^{H}$ that has access to a random oracle H and stores a key consisting of M blocks (of length w). That is, the input to the algorithm is $R = (R_1, \ldots, R_M)$, and it behaves as follows. Suppose it is queried on some inputs $x_1, \ldots, x_T$. Then, after receiving each x_i it computes the value of

$$Eval(Lam_{i\varDelta+2}^{M-(i-1)\varDelta}, H^{(i,x_i)}, (R[1 + (i-1)\varDelta, \ldots, M]). \tag{2}$$

The algorithm $COMP^H_{U,M,T,w}(R)$ simply computes each (2) one-by-one for $i = 1, \ldots, T$. Each of these steps destroys Δ values R_j from the input. Thus, before the i-th step we keep in the memory only $R[1 + (i-1)\Delta, \ldots, M]$. This means that the space used by the remaining part of the input ($R[1, \ldots, (i-1)\Delta]$) is now free and it can be used as additional storage for computation. So, just before the beginning of the i-th step the free storage (i.e. storage not including kept fragment of the input) is bounded by $e_i = E_i \cdot w$, where $E_i := 1 + (i-1)\Delta$. The algorithm in the i-th step is just a simple application of Remark 1 from Section 4.2. Observe that from Lemma 2 we get a pebbling strategy that pebbles output vertex of $Lam^{M-(i-1)\Delta}_{i\Delta+2}$ using E_i extra pebbles and removes the first $(i\Delta+2)-1-E_i$ input pebbles. From the definition of E_i we have $(i\Delta+2)-1-E_i = \Delta$. So, from Remark 1 we get that there is an algorithm that computes (2) overwriting $\Delta \cdot w$ first bits of remaining input. So, after this step the algorithm can keep $R[1 + i\Delta, \ldots, M]$ to be used in the next steps.

Theorem 1. *For any integers c, s, m, w, let $U \stackrel{def}{=} \lfloor \frac{c+s+w}{w-\log(q)} \rfloor$ and $M \stackrel{def}{=} \lfloor \frac{m}{w} \rfloor - 1$. Then, for any integer $T < \frac{U-1}{2(U-M)}$ the algorithm $COMP^H_{U,M,T,w}$ is a $(c, s, m, q, (q+1) \cdot 2^{-w}, T)$-one-time computable PRF.*

Asymptotically, as $c, s, m \gg w \gg \log(q)$, the maximal T becomes

$$T \approx \frac{c+s}{2(c+s-m)}. \tag{3}$$

Proof (of Theorem 1). Suppose $(R_1, \ldots, R_M)$ is chosen uniformly at random. Let $\mathcal{A} = (\mathcal{A}_{big}, \mathcal{A}_{small})$ be an arbitrary adversary with oracle access to H that has s-bounded space and c-bounded communication and makes at most q oracle calls. Consider an execution $\mathcal{A}^{H(\cdot)}(R)$. Let $\mathcal{E}$ be an event that for some i and for two different x and x' the adversary labeled the output vertex of $Lam^{M-(i-1)\Delta}_{i\Delta+2}$ in the $(H^{(i,x)}, R)$-labeling and $(H^{(i,x')}, R)$-labeling. To prove the theorem we need to show the following.

$$P(\mathcal{E}) \leq T \cdot (q+1) \cdot 2^{-w}. \tag{4}$$

Fix some $\tilde{i}$, and let $\mathcal{E}_{\tilde{i}}$ denote the event that $\mathcal{E}$ happened for $i = \tilde{i}$. Let G be equal to the sum of following infinite sequence of graphs $\left\{ Lam^{M-(\tilde{i}-1)\Delta}_{\tilde{i}\Delta+2} \right\}_{x \in \{0,1\}^*}$. We now show an adversary $\tilde{\mathcal{A}}$ with an s-bounded space and c-bounded communication that has access to an oracle $\tilde{H}$ and makes at most q queries to it, and satisfies the following: for a randomly-chosen $\tilde{R} = (R_{1+(\tilde{i}-1)\Delta}, \ldots, R_M) \in (\{0,1\}^w)^{M-(\tilde{i}-1)\Delta}$ in the execution $\tilde{\mathcal{A}}^{\tilde{H}(\cdot)}(\tilde{R})$ the probability that the adversary labels at least two different output vertices of G is equal to $P(\mathcal{E}_{\tilde{i}})$.

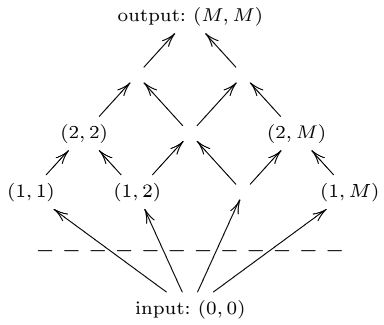
The adversary $\tilde{\mathcal{A}}$ simulates $\mathcal{A}$ in the following way. First, since $\mathcal{A}$ "expects" the input to have length M, it fills-in the "missing" elements of $\tilde{R}$, i.e. he selects randomly $(R_1, \ldots, R_{(\tilde{i}-1)\Delta})$ and sets $R = (R_1, \ldots, R_{(\tilde{i}-1)\Delta}) || \tilde{R}$. Next, it simply runs $\mathcal{A}$. The only thing that we need to take care of is to "translate" the oracle queries issued by $\mathcal{A}$ to H into oracle queries issued by $\tilde{\mathcal{A}}$ to $\tilde{H}$. Let Q be a query issued by $\mathcal{A}$. Consider the following cases:

- Q has a form $((\tilde{i}, x), (\texttt{label}_1, \ldots, \texttt{label}_d, v))$ (for some $x, \texttt{label}_1, \ldots, \texttt{label}_d$) — in this case we translate it into a a query $(\texttt{label}_1, \ldots, \texttt{label}_d, ((i, x), v),$
- Q has a form or $((\tilde{i}, x), (\texttt{label}, v, \texttt{out}))$ (for some x and $\texttt{label}$) — in this case we translate it into a a query $(\texttt{label}, ((\tilde{i}, x), v), \texttt{out})$.
- if Q does not have any of the forms above — we translate it in some arbitrary (deterministic and injective) way.

It is easy to see that $\tilde{\mathcal{A}}$ labels an output vertex $((\tilde{i}, x), v)$ of G if and only if his simulated copy of $\mathcal{A}$ labeled v in the graph $Lam_{\tilde{i}\Delta+2}^{M-(\tilde{i}-1)\Delta}$. Therefore the probability that $\tilde{\mathcal{A}}$ labeled two output vertices of G is equal to $P(E_{\tilde{i}})$. Now, by Corollary 1 we get that this probability is at most $(q+1) \cdot 2^{-w}$ as long as $\frac{s+c+w}{w-\log(q)} < M - (\tilde{i}-1)\Delta + 1 + \tilde{i}\Delta + 1 - 2 = M + \Delta = U$, which is exactly the assumption that we made in the statement of the lemma. Since $\mathcal{E} = \cup_{i=1}^{T} \mathcal{E}_i$, therefore, by the union-bound we get that $P(\mathcal{E}) \leq T \cdot ((q+1) \cdot 2^{-w})$. Therefore (4) is proven.

6 Arrowhead functions

In this section we define a class of DAGs that we call the *arrowhead graphs*. For every $M \in \mathcal{N}$ let Arr_M be a graph consisting of defined previously M-pyramid with one additional vertex $(0,0)$ and additional edge from $(0,0)$ to $(1,x)$ for $x \in 1, \ldots M$. More precisely, $Arr_M = (V_M, E_M)$, where $V = \{(0,0)\} \cup \{(i,j) : 1 \leq i \leq j \leq M$. A graph Arr_M consists of one input vertex $(0,0)$ and one output vertex (M,M). The follwoing figure shows an example of an M-arrowhead graph for $M = 4$. The subgraph on the upper side of the dashed line is an M-pyramid.



Lemma 7. *For any a and $R = (R_1, \ldots, R_M)$ the value of $Eval(Arr_M, H, R)$ can be computed by an algorithm that has access to a random oracle H and has $(M+1) \cdot w$-bounded storage.*

Proof. Using Remark 1 it suffices to show a strategy that pebbles Arr_M using $M+1$ pebbles. Such a strategy is straighforward and appears in the full version of this paper [19].

The following lemma shows the optimality of the algorithm given in Lemma 7.

Lemma 8. *For any s, λ and q, such that $\frac{s+\lambda}{w-\log(q)} < M+1$, and any adversary $\mathcal{A}$ that has s-bounded storage and 0-bounded communication, and makes at most q queries to the oracle, the probability that $\mathcal{A}$ labels the output of Arr_M is at most $q \cdot 2^{-w} + 2^{-\lambda}$.*

This lemma follows from the fact that every strategy that pebbles the output of Arr_M, and does not use the red pebbles, must use at least $M-1$ black pebbles. The proof of this fact is very similar to the proof of Lemma 10.2.1 in the book of John Savage ([36]), and it appears in the full version of this paper [19]. Lemma 7 and 8 imply the following.

Theorem 2. *The hash function that takes as input R and outputs $Eval(Arr_M, H, R)$ is $((M+1) \cdot w, w, q, \log(q)(M+1) + \lambda, q \cdot 2^{-w} + 2^{-\lambda})$-uncomputable.*

7 Acknowledgments

The authors wish to thank Yevgeniy Dodis for helpful discussions.

References

1. A. Akavia, S. Goldwasser, and V. Vaikuntanathan. Simultaneous hardcore bits and cryptography against memory attacks. In *TCC*, 2009.
2. J. Alwen, Y. Dodis, M. Naor, G. Segev, S. Walfish, and D. Wichs. Public-key encryption in the bounded-retrieval model. In *EUROCRYPT*, 2010.
3. J. Alwen, Y. Dodis, and D. Wichs. Leakage-resilient public-key cryptography in the bounded-retrieval model. In *CRYPTO*, 2009.
4. G. Ateniese, R. Di Pietro, L. Mancini, and G. Tsudik. Scalable and efficient provable data possession. In *SecureComm*, 2008.
5. Y. Aumann, Y. Z. Ding, and M. O. Rabin. Everlasting security in the bounded storage model. *IEEE Transactions on Information Theory*, 48(6), 2002.
6. M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM Conference on Computer and Communications Security*, 1993.
7. Z. Brakerski and S. Goldwasser. Circular and leakage resilient public-key encryption under subgroup indistinguishability (or: Quadratic residuosity strikes back). CRYPTO, 2010.
8. Z. Brakerski, Y. Tauman Kalai, J. Katz, and V. Vaikuntanathan. Cryptography resilient to continual memory leakage. FOCS, 2010.
9. R. Canetti, S. Halevi, and M. Steiner. Mitigating dictionary attacks on password-protected local storage. In *CRYPTO*, 2006.
10. D. Cash, Yan Zong Ding, Y. Dodis, W. Lee, R. J. Lipton, and S. Walfish. Intrusion-resilient key exchange in the bounded retrieval model. In *TCC*, 2007.
11. G. Di Crescenzo, R. J. Lipton, and S. Walfish. Perfectly secure password protocols in the bounded retrieval model. In *TCC*, 2006.
12. F. Davì, S. Dziembowski, and D. Venturi. Leakage-resilient storage. SCN, 2010.

13. Y. Dodis, S. Goldwasser, Y. Tauman Kalai, C. Peikert, and V. Vaikuntanathan. Public-key encryption schemes with auxiliary inputs. In *TCC*, 2010.
14. Y. Dodis, K. Haralambiev, A. Lopez-Alt, and D. Wichs. Cryptography against continuous memory attacks. FOCS, 2010.
15. C. Dwork, A. Goldberg, and M. Naor. On memory-bound functions for fighting spam. In *CRYPTO*, 2003.
16. C. Dwork, M. Naor, and H. Wee. Pebbling and proofs of work. In *CRYPTO*, 2005.
17. S. Dziembowski. Intrusion-resilience via the bounded-storage model. In *TCC*, 2006.
18. S. Dziembowski. On forward-secure storage. In *CRYPTO*, 2006.
19. S. Dziembowski, T. Kazana, and D. Wichs. One-Time Computable Self-Erasing Functions, 2010 Cryptology ePrint Archive
20. S. Dziembowski and U. M. Maurer. Optimal randomizer efficiency in the bounded-storage model. *J. Cryptology*, 17(1), 2004.
21. S. Dziembowski and K. Pietrzak. Intrusion-resilient secret sharing. In *FOCS*, 2007.
22. S. Dziembowski and K. Pietrzak. Leakage-resilient cryptography. In *FOCS*, 2008.
23. S. Dziembowski, K. Pietrzak, and D. Wichs. Non-malleable codes. In *ICS*, 2010.
24. ECRYPT. *The Side Channel Cryptanalysis Lounge* `http://www.crypto.rub.de/en_sclounge.html`.
25. S. Faust, E. Kiltz, K. Pietrzak, and G. N. Rothblum. Leakage-resilient signatures. In *TCC*, 2010.
26. R. Gennaro, A. Lysyanskaya, T. Malkin, S. Micali, and T. Rabin. Algorithmic tamper-proof (atp) security: Theoretical foundations for security against hardware tampering. In *TCC*, 2004.
27. Y. Ishai, M. Prabhakaran, A. Sahai, and D. Wagner. Private Circuits II: Keeping Secrets in Tamperable Circuits. In *EUROCRYPT*, 2006.
28. Y. Ishai, A. Sahai, and D. Wagner. Private Circuits: Securing Hardware against Probing Attacks. In *CRYPTO*, 2003.
29. J. Katz and V. Vaikuntanathan. Signature schemes with bounded leakage resilience. In *ASIACRYPT*, 2009.
30. Feng-Hao Liu and A. Lysyanskaya. Algorithmic tamper-proof security under probing attacks. In *SCN*, 2010.
31. U. M. Maurer. Conditionally-perfect secrecy and a provably-secure randomized cipher. *J. Cryptology*, 5(1), 1992.
32. S. Micali and L. Reyzin. Physically observable cryptography (extended abstract). In *TCC*, 2004.
33. M. Naor and G. Segev. Public-key cryptosystems resilient to key leakage. In *CRYPTO*, 2009.
34. D. Perito and G. Tsudik. Secure code update for embedded devices via proofs of secure erasure. In *ESORICS*, 2010.
35. K. Pietrzak. A leakage-resilient mode of operation. In *EUROCRYPT*, 2009.
36. John E. Savage. *Models of Computation: Exploring the Power of Computing.* 1997.
37. F.-X. Standaert, T. Malkin, and M. Yung. A unified framework for the analysis of side-channel key recovery attacks. In *EUROCRYPT*, 2009.
38. L. von Ahn, M. Blum, N. J. Hopper, and J. Langford. Captcha: Using hard ai problems for security. In *EUROCRYPT*, 2003.