One-time Computable and Uncomputable Functions

Stefan Dziembowski, Tomasz Kazana and Daniel Wichs

October 22, 2010

Abstract

This paper studies the design of cryptographic schemes that are secure even if implemented on untrusted machines, whose internals can be partially observed/controlled by an adversary. For example, this includes machines that are infected with 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 *at most once* on a machine which stores the (long) key K, as long as: (1) the adversary cannot retrieve the key K out of the machine completely (this is similar to the assumptions made in the so-called *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, 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, that we call *uncomputable hash functions*, which are hash funcitons that cannot be computed if the local storage has some restricted size s, but *can* be computed if they are given slightly more storage than s. 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 theoretical cryptography is to construct cryptographic schemes that have some provable security properties, even if 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 about the internal data stored on a cryptographic device $\mathcal{M}$. This line of research, motivated by the so-called *side-channel attacks* [25], started with seminal papers of Ishai et al. [29] and Micali and Reyzin [33], that were followed by a number of recent works [23, 1, 37, 30, 35, 12, 39, 13, 14, 26, 8, 7]. Some papers have also been motivated by the attacks of the malicious software (like viruses) against computers [11, 20, 19, 10, 22, 2, 3]. 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 obviously necessary, as the adversary that has an unlimited access to the internal data of $\mathcal{M}$ can simply create his own copy of it, which is usually enough to completely break any type of security. The most common assumption that is currently used 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 [29, 27, 24, 31] 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 internal data of the device. Obviously if we do not put any restrictions on the adversary then he can substitute the internal data of the device by some data chosen by him. For example, the adversary can modify the device so that it just outputs all of its internals. Hence also in this case some restrictions on the power of the adversary are usually needed.

For example, in the model of [29] the adversary that can tamper a restricted number of wires of the circuit that performs the computation (in a given time-frame), and in [27] it is assumed that a device is equipped with a small tamper-free component. When the active attacks are considered the following natural question arises:

> Can we have a practical cryptographic primitive π, implemented on a machine $\mathcal{M}$, such that already the physical characteristics of $\mathcal{M}$ provide some meaningful security guarantees even if $\mathcal{M}$ falls in the hands of a tampering adversary; in a similar way as the "bounded retrievability property" guarantees security against the leakages?

Of course, one could just assume from the beginning that such a "physical characteristic" is: "$\mathcal{M}$ is tamper-free". We do not want to do it, since such an assumption is often too optimistic, and, moreover, it is hard to verify in practice. Hence, what we are interested in are the "physical characteristics" that are simple, and can be derived by some very general features of the device. One example of such a feature could be the above-mentioned "bounded-retrievability property".

In this paper we give an affirmative answer to this question, by constructing a new cryptographic primitive, which we call *one-time computable pseudorandom functions (PRFs)*. The physical characteristic of $\mathcal{M}$ on which the security is based are very natural and simple: we assume that (1) $\mathcal{M}$ satisfies the bounded-retrievability property, and (2) the memory of $\mathcal{M}$ has restricted size. 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*. An application of it is an improved protocol for the *proof-of-erasure* (a concept recently introduced in [36]). We explain these concept in the next sections.

Let us first clarify the following. Normally, people think of leakage functions as modeling various side-channels. So the memory used by the leakage function has nothing to do with the memory available on the device (is probably much smaller, but bigger makes sense too in theory). For us, we assume leakage runs on device itself and uses its resources. On the other hand, people think of tampering-attacks as modifying the internals of a device in the hope of getting useful information by interacting with the modified device honestly later. The information learned later is not really thought of as leakage (it can be large). In our model we will assume that the adversary is permanently controlling the device, and hence the leakage bound will also concern all the data transfered away from the device (including the data learned "after the adversary tampered the device") .

1.1 One-time computable functions

In this section we informally define the one-time computable PRFs. Let us start with describing the "ideal functionality", i.e. let us say what we mean by the "one-time computability" assuming for a moment that we have a completely secure (i.e. leakage- and tamper-proof) device $\mathcal{M}$. Intuitively, a function F is a one-time computable PRF if (1) it is a pseudorandom function, and (2) anybody who gets access to a machine where F is stored can

compute F on exactly one input. To be more precise, assume a key R for F is selected uniformly at random, and it is stored on $\mathcal{M}$. The device $\mathcal{M}$ can be queried on any input $x \in \{0,1\}^*$ and it outputs $F_R(x)$. Condition (2) means that during its life-time the device $\mathcal{M}$ can be queried on at most one input. Condition (1) means that (a) as long as the device $\mathcal{M}$ has not been queried on any input, the value of any $F_R(x)$ is indistinguishable from a random string, and (b) if the adversary knows some $(x, F_R(x))$ (for x chosen by him) all the other values $F_R(x')$ for $x' \neq x$ are indistinguishable from random. Of course, if $\mathcal{M}$ is completely secure one could implement this functionality by simply storing a flag f that indicates if H has been already been evaluated on some input.

Let us now consider the case when $\mathcal{M}$ is not completely secure. Obviously, the solution described above does not work in this case since the adversary can reset the value of the flag f. This is only an example of a tampering attack, and in general several other tampering attacks are possible. Our approach is to propose a model that covers a large class of realistic tampering attacks. To make our model as strong as possible we will assume that the adversary is simply permanently controlling $\mathcal{M}$, and he may freely use all its computational and memory resources (on the other hand, he does not have any additional resources as long as he wants to execute some local computation of the internal data of $\mathcal{M}$). Of course, if the adversary wants, he can also honestly execute the code on $\mathcal{M}$ on an input chosen by him. Hence, in our model there will be no distinction between an internal adversary that tampers the device, and an external adversary that asks the legitimate queries to $\mathcal{M}$.

Roughly, our main idea is to construct F in such a way that any computation of $F_R(x)$ has to destroy R, by partially overwriting it. On the first sight it may look like it is impossible to achieve it for the following reason. Assume that there exists a procedure γ that takes as input R and x, and computes $F_R(x)$ by overwriting part R. Since the basic property of the digital data is that it can be freely copied, therefore anybody who gets access to R can make a temporary local copy of it, and, after performing the computation γ on R and some x_0, refresh the description of R by copying it back from the temporary storage. After the description of R is refreshed he can apply γ to compute the value of R on some other x_1. Since this can be repeated over and over again, the adversary that controls $\mathcal{M}$ can evaluate F_R on unlimited number of different inputs $x_0, x_1, \ldots$.

We bypass this problem in the following way: we construct F in such a way that the R is too large to be copied locally. To be more precise, we use the following notation. Let m denote the length of R, and let s denote the size of $\mathcal{M}$'s memory (including the memory needed to store the description of R). In our construction it will always be the case that $2m > s > m$. We will say that an adversary that has memory size s has *s-bounded storage*.

Another obvious thing that the adversary can do is: once he gets access to $\mathcal{M}$ he can retrieve the key R and perform the computation of $F_R(x_0), F_R(x_1), \ldots$ on his own storage that is external to $\mathcal{M}$. Therefore we need also to assume that the total amount of data that the adversary can download from $\mathcal{M}$ is limited. Denote this amount by c (we will also say that the adversary has *c-bounded communication*). We obviously need to have that $m > c$. Note that it is essentially the bounded-retrievability property that was mentioned above. In this paper we construct the one-time computable functions that are secure assuming that $\mathcal{M}$ has c-bounded communication and s-bounded storage. Note, that in particular we do not need to assume any bound on the amount of information that the adversary can *upload* on $\mathcal{M}$.

Recall that in our model there is essentially no distinction between the resources (the storage and the computing power) available to the legitimate program executed on $\mathcal{M}$, and the adversary that controls $\mathcal{M}$. From the practical point of view it is wiser to be conservative, and assume that the adversary has slightly *more* storage than the legitimate program. This is because the adversary may, overwrite some part of the memory of $\mathcal{M}$ that an honest user would not like to overwrite (e.g. the part containing the operating system). Therefore we will require that the legitimate program has to use at most $s - \delta$ space on $\mathcal{M}$ (for some parameter δ).

One-time computable functions – a generalization We also construct the following generalization of the concept described above. Suppose we have a sequence of T randomly and independently chosen keys $R_1, \ldots, R_T$ stored on $\mathcal{M}$ and we want that the user can pick some inputs $x_1, \ldots, x_T$ and evaluate $F_{R_1}(x_1), \ldots, F_{R_T}(x_T)$ (the inputs can be chosen adaptively, depending on the previous outputs). On the other hand, the adversary that controls the machine should not be able to compute any of the functions F_{R_i} on more than one input (the previously described concept is a special case of this notion with $T = 1$). We show how to implement it using the same assumptions as above. Observe that a naive solution of using the previous scheme T times independently on the same machine does not work, since the adversary could use the space where R_1, say, is stored as an extra memory to compute F_{R_2} more than once. We show a solution for this problem. In our solution, instead of storing $R_1, \ldots, R_T$ we will just store one R and derive the R_i's from it. The construction is given in Section 5. The maximal T that we can have is approximately equal to $\frac{2m-s}{2\delta}$ (cf. 3).

1.1.1 Application: Password-protected storage

Let us now describe an application of the primitives described above. Our application is related to the *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 is only guaranteed to have a large entropy, and the *dictionary set* $\mathcal{D}$ from which it is chosen may be not very large. 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 one-by-one all the passwords from $\mathcal{D}$.

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 (Enc, Dec). To decrypt a ciphertext $C = Enc(H(\pi), X)$ one simply calculates $Dec(H(\pi), C)$.

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 $Dec(H(\pi'), C)$ "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 [40], 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 [36]. 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 [36] 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 [36]. 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. [38]). This technique has already been used in cryptography in an important work of [17], 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* [32, 5, 21]. As similar assumption was also used in [16]. The proof of erasures can be viewed as a special case of the *remote attestation protocols* (see [36] for a list of relevant references).

1.4 Notation

In our constructions we will assume that the memory is divided into blocks of length w. We will use the following convention: the length of the strings *in bits* will be denoted with lower-case letters (n, c, s and δ) and the lengths of the strings *in blocks* will be denoted with the corresponding upper-case letters (N, C, S and Δ), where, e.g., we will have $n = w \cdot N$.

For a sequence $R = (R_1, \ldots, R_N)$ and for indices i, j such that $1 \le i \le j \le N$, 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 $H : \{0,1\}^{\xi w} \rightarrow \{0,1\}^w$, 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]

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 $(F^H_{1,R}, \ldots F^H_{T,R})$ 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 $((F^H_{i,R}(x_i), (i, x_i)), \texttt{out})$. We say that $W^{H(\cdot)}$ is a $(c, s, m, q, \delta, \epsilon, T)$-*onetime computable PRF* if:

- $W^{H(\cdot)}$ has $(s - \delta)$-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 $((F^H_{i,R}(x), (i, x)), \texttt{out})$ and $((F^H_{i,R}(x'), (i, x')), \texttt{out})$, 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.

[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...

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 [18].

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 $\texttt{label}(v_i) \stackrel{\text{def}}{=} R_i$.
- The label of every other vertex v is defined recursively by

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

  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 $\texttt{preLabel}(v) \stackrel{\text{def}}{=} (\texttt{label}(v_1), \ldots, \texttt{label}(v_d), v)$, 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 4.1. *Let G be a graph with N vertices and M input vertices. Let v be an output vertex of G and let $v_{i_1}, \ldots, v_{i_d}$ 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 $v_{i_1}, \ldots, v_{i_d}$, 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 $\mathit{Eval}(G, H, (R_1, \ldots, R_M))$ 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 $(R_1, \ldots, R_M)$, however, it does not overwrite the input corresponding to the vertices $v_{i_1}, \ldots, v_{i_d}$, i.e.: $(R_{v_1}, \ldots, R_{v_d})$.

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. We establish such a connection in the next section.

4.3 Connection Between Random-Oracle Labeling and the Pebbling Game

We now connect the random-oracle labeling of a graph G in our model of computation to the red-black pebbling game on G. In particular, we will show that the black-pebble complexity of the pebbling will correspond to the space-complexity of $\mathcal{A}_{small}$ and the red-pebble complexity corresponds to the communication-complexity of $\mathcal{A}_{small}$.

Let $H : \{0,1\}^* \to \{0,1\}^w$ be a random oracle, and let $R = (R_1, \ldots, R_M)$ be a labeling of the input-vertices of G. For any algorithms $\mathcal{A} = (\mathcal{A}_{big}, \mathcal{A}_{small})$ we can use the execution $\mathcal{A}^{H(\cdot)}(R) = \left(\mathcal{A}_{big}^{H(\cdot)} \leftrightarrows \mathcal{A}_{small}^{H(\cdot)}(R)\right)$ to construct a red-black pebbling of the graph G. In particular, we get a transcript listing all oracle calls made during its entire execution, and whether they were made by $\mathcal{A}_{small}$ or $\mathcal{A}_{big}$.

We fix some terminology about the transcript. Given (H, R), we say that an oracle call of the form $H(\texttt{label}_1, \ldots, \texttt{label}_d, v)$ is *correct* if $(\texttt{label}_1, \ldots, \texttt{label}_d, v) = \texttt{preLabel}(v)$. An oracle is also correct if it has a form $H((\texttt{label}, v), \texttt{out})$, where $\texttt{out}$ is some special symbol, v is an output vertex, and $\texttt{label}(v) = \texttt{label}$. We call the children $v_1, \ldots, v_d$ of v the input-vertices of the oracle call, and v is the output-vertex of the oracle call.

Using the transcript (along with the description of H, R) we define the *ex-post-facto* pebbling of the graph G. We do so by processing the random-oracle calls in the transcript one-by-one starting with the earliest one, and, for each call, we take the following steps:

Place all necessary red pebbles: A vertex v is *red-necessary* if, looking at the *entire transcript* of all oracle calls, there exists some correct oracle call made by $\mathcal{A}_{big}$ with v as an input-vertex, which *precedes* all correct oracle calls made by $\mathcal{A}_{big}$ with v as an output-vertex.

Go through all red-necessary vertices v one-by-one and, for each one that has a black pebble but no red pebble, put a red pebble.[4]

Delete all unnecessary black pebbles: A vertex v is *black-necessary* if it is *not* red-necessary and, *in the remainder of the transcript* of oracle-calls that have not yet been processed (including the current call), there exists some correct oracle call made by $\mathcal{A}_{small}$ with v as an input-vertex such that:

- In the *remainder of the transcript*, there is no *earlier* correct oracle call made by $\mathcal{A}_{small}$ with v as an output-vertex.
- In the *entire transcript*, there is no *earlier* correct oracle call made by $\mathcal{A}_{big}$ with v as an output-vertex.

Go through all vertices v which are *not* black-necessary but have a black pebble on them, one-by-one, and remove the black pebble.[5]

Process oracle call: If the current oracle call is correct and made by $\mathcal{A}_{small}$ (respectively $\mathcal{A}_{big}$) with output vertex v, we put a black (respectively red) pebble on v.

We notice that every vertex that is labeled by the execution of $\mathcal{A}^{H(\cdot)}(R)$ gets a (red or black) pebble placed on it in the corresponding ex-post-facto pebbling. Moreover, the order in which vertices get red/black pebbles corresponds to the order in which the oracle calls are made by $\mathcal{A}$.

We now show that, for any adversary $\mathcal{A} = (\mathcal{A}_{small}, \mathcal{A}_{big})$ which is space/communication bounded, and which makes a bounded number of oracle calls, the ex-post-facto pebbling is legal and has small space/communication complexity.

[4] Note that the set of red-necessary vertices does not change throughout the process. Intuitively, these are the vertices whose labels must be communicated by $\mathcal{A}_{small}$ to $\mathcal{A}_{big}$ at some point in time, and correspondingly for which we need to take pebbling-action 1 to place a red pebble on them. We choose to take this action as early as legally possible, since it might allow us to remove related black pebbles early.

[5] Note that the set of black-necessary vertices can be different at different points in the process. Intuitively, at any point in time, a black-necessary vertex is one whose label must be stored in the memory of $\mathcal{A}_{small}$ since it will not be re-computed by $\mathcal{A}_{small}$ via oracle calls, it was never communicated to $\mathcal{A}_{big}$, nor will it be computed by $\mathcal{A}_{big}$ in time.

Theorem 4.2. *Let $G = (V, E)$ be a DAG and let $\mathcal{A} = (\mathcal{A}_{big}, \mathcal{A}_{small})$ be any adversarial labeling strategy in our restricted model of computation. Let (H, R) define a random-oracle labeling of the graph G, with label-length w. Assume that $\mathcal{A}$ makes at most q random-oracle queries during any execution. Then, the ex-post-facto pebbling of G corresponding to an execution of $\mathcal{A}^{H(\cdot)}(R)$ has the following properties:*

1. *It is a legal pebbling (i.e. follows the rules of the red-black pebbling game) with probability $1 - \frac{q}{2^w}$ over the choice of (H, R).*
2. *Assuming that $\mathcal{A}_{small}$ has c-bounded communication then, for any $\lambda \geq 0$, the red-pebble complexity is at most $\frac{c+\lambda}{w-\log(q)}$ with probability $1 - 2^{-\lambda}$ over the choice of (H, R).*
3. *Assuming that $\mathcal{A}_{small}$ has s-bounded storage and c-bounded communication then, for any $\lambda > 0$, the all-pebble complexity is at most $\frac{c+s+\lambda}{w-\log(q)}$ with probability $1 - 2^{-\lambda}$ over the choice of (H, R).*

For the lack of space the proof of Theorem 4.2 is moved to Appendix B.

Corollary 4.3. *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 $\alpha \in \{1, \ldots, K\}$).*

Then, for any c, s, w and λ such that $\frac{c+s+\lambda}{w-\log(q)} < a$, 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 $q \cdot 2^{-w} + 2^{-\lambda}$ (where the probability is taken over the randomness of $\mathcal{A}$ and the random choice of H and R).

Proof. Take 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. Recall that "labeling a vertex v" means that the adversary makes an oracle call to `preLabel`(v). Note that it does not necessarily mean that the corresponding pebbling strategy will every pebble v, as pebbling a vertex occurs only when the label of v is given as an input to the oracle. Therefore we assume that $\mathcal{A}$ always after learning the label `label` of any output vertex v issues an oracle call $H((\texttt{label}, v), \texttt{out})$. This guarantees that such a v will always be pebbled. Clearly adding this instruction does not increase the space and communication complexity of $\mathcal{A}$.

Suppose we choose randomly R and H and run the experiment. Let $\mathcal{E}$ denote the event that $\mathcal{A}$ labels more than $\alpha - 1$ output vertices. For the sake of contradiction suppose that $P(\mathcal{E}) > q \cdot 2^{-w} + 2^{-\lambda}$. Let $\mathcal{G}$ denote the event that the corresponding pebbling strategy is legal and its all-pebble complexity is at most $\frac{c+s+\lambda}{w-\log(q)}$. From Theorem 4.2 (Points 1 and 3) we get that $P(\mathcal{G}) \geq 1 - q \cdot 2^{-w} - 2^{-\lambda}$. Therefore the probability that $\mathcal{E}$ and $\mathcal{G}$ occurred simultaneously is positive. Hence there needs to exist an execution of $\mathcal{A}$ such that the corresponding pebbling strategy that pebbles α vertices and has all-pebble complexity at most $\frac{c+s+\lambda}{w-\log(q)}$, which, by our assumption, is less than $c + s$. This yields a contradiction. □

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. also Fig. 1, Appendix A). Its set of vertices is equal to $V_0 \cup V_1$, where $V_0 = \{(i, j) : 1 \leq i \leq j \leq M'\}$ 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 as the following sum:

$$\begin{aligned}
&\{((i-1, j-1), (i, j)) : (i-1, j-1), (i, j) \in V_0\} && (1)\\
&\cup\{((i-1, j), (i, j)) : (i-1, j), (i, j) \in V_0\} && (2)\\
&\cup\{((M', M'), (2, M' + 1))\}\\
&\cup\{((1, j-1), (1, j)) : (1, j-1), (1, j) \in V_1\}\\
&\cup\{((1, j), (2, j)) : (1, j), (2, j) \in V_1\}.
\end{aligned}$$

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 set in Equations (1) and (2) above. Its output vertex is (M, M). Such a graph is also called an M-*pyramid graph*.

Lemma 5.1. *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 5.2. *For $M \geq 1$ 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 on the input row (i.e. on the vertices $\{0\} \times \{0, 1, \ldots, M - 1\}$) 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"). Let Y be the number of the heavy pebbles on G_{M-1}. Of course G_{M_1} is Lam^{M-1}_{M-1}.

Now, put black pebbles on the vertices of the input raw 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). Clearly the number of black pebbles created this way 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 $Y + |\mathcal{X}| - 1 \geq M - 1$, which implies that $Y \geq M$. □

Lemma 5.3. *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 5.2. Let γ be the first configuration in which (M, M) is input-independent, and let γ' be the configuration that directly proceeds 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 5.2 (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 5.4. *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 $j \in \{M' + 1, \dots, M\}$).*

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 $j \in \{M' + 1, \dots, M\}$ 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:

$$V_I \cup \bigcup_{k=1}^{\ell} \{k\} \times V_k$$

The set E of edges is defined as:

$$\begin{aligned} E \quad := \quad & \{((k, v), (k, v')) : v, v' \in V_k' \text{ and } (v, v') \in E_k\} \\ & \cup \{(v, (k, v')) : v \in V_I \text{ and } v' \in V_k' \text{ and } (v, v') \in E_k\} \end{aligned}$$

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 5.5. *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 5.3 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 5.2 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 5.3 the vertex (M', M') is not yet pebbled in this configuration. Hence, by Lemma 5.4, 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 $j \in \{M' + 1, \dots, M\}$. 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 $M + M' - 2$.

□

Combining Lemma 5.5 with Corollary 4.3 we get the following.

Corollary 5.6. *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, for any s, c, λ and q, such that $\frac{c+s+\lambda}{w-\log(q)} < M + M' - 2$, 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 $q \cdot 2^{-w} + 2^{-\lambda}$.*

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, S, Δ, and T be some positive integer parameters such that

$$2M > 2(T-1)\Delta + S + 1, \tag{3}$$

and $S - M - \Delta \geq 0$. The intuition is that M denotes the length of the key R, the symbol S denotes the total space on the device, and Δ denotes that extra space that the adversary has on the device (the length is measured in blocks of length w). Let

$$E := S - M - \Delta + 1. \tag{4}$$

We now construct an interactive algorithm $COMP^H_{\Delta,S,M,T,w}$ that has access to a random oracle H and has s-bounded storage for $s = (S - \Delta + 1) \cdot w$, and takes as input $R = (R_1, \ldots, R_M)$, and 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

$$Eval(Lam^{M-(i-1)\Delta}_{E+i\Delta+1}, H^{(i,x_i)}, (R[1 + (i-1)\Delta, \ldots, M]). \tag{5}$$

The algorithm $COMP^H_{\Delta,S,M,T,w}(R)$ simply computes each (5) 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 $E_i := E + (i-1)\Delta$. The algorithm in the i-th step is just a simple application of Remark 4.1 from Section 4.2. Observe that from Lemma 5.1 we get a pebbling strategy that pebbles output vertex of $Lam^{M-(i-1)\Delta}_{E+i\Delta+1}$ using E_i extra pebbles and removes the first $(E+i\Delta+1)-1-E_i$ input pebbles. From the definition of E_i we have $(E+i\Delta+1)-1-E_i = \Delta$. So, from Remark 4.1 we get that there is an algorithm that computes (5) 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 5.7. *Suppose Δ, S, M and T satisfy (3). Then, for any q, λ, c, s such that $\frac{c+s+\lambda}{w-\log(q)} < S$ the algorithm $COMP^H_{\Delta,S,M,T,w}$ is a $(c, s, M \cdot w, q, \Delta \cdot w, q \cdot 2^{-w} + 2^{-\lambda}, T)$-one-time computable PRF.*

Proof. 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 $Lam^{M-(i-1)\Delta}_{E+i\Delta+1}$ in the $(H^{(i,x)}, R)$-labeling and $(H^{(i,x')}, R)$-labeling (where E is defined in (4)). To prove the theorem we need to show the following.

$$P(\mathcal{E}) \leq T \cdot (q \cdot 2^{-w} + 2^{-\lambda}). \tag{6}$$

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

$$\{Lam^{M-(\tilde{i}-1)\Delta}_{E+\tilde{i}\Delta+1}\}_{x \in \{0,1\}^*}.$$

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 $Lam^{M-(\tilde{i}-1)\Delta}_{E+\tilde{i}\Delta+1}$. Therefore the probability that $\tilde{\mathcal{A}}$ labeled two output vertices of G is equal to $P(E_{\tilde{i}})$. Now, by Corollary 5.6 we get that this probability is at most $q\cdot 2^{-w}+2^{-\lambda}$ as long as $\frac{s+c+\lambda}{w-\log(q)} < M-(\tilde{i}-1)\Delta+E+\tilde{i}\Delta+1-2 = M+\Delta+E-1 = S$, 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 $P(\mathcal{E}) \leq T\cdot(q\cdot 2^{-w}+2^{-\lambda})$. Therefore (6) is proven. ☐

Assuming that the parameters c, s and δ are fixed, Theorem 5.7 implies that the maximal value T_{max} of T, that we can achieve is

$$T_{max} \approx \frac{c+s}{2\delta}. \tag{7}$$

To see why it is the case, choose the minimum S possible, i.e: $S := \lfloor \frac{c+s+\lambda}{w-\log q}+1 \rfloor (\approx \frac{c+s}{w})$. Let $\Delta := \lceil \frac{\delta}{w} \rceil (\approx \frac{\delta}{w})$. Moreover, assume that $M := S-\Delta+1 (\approx \frac{c+s-\delta}{w})$. Hence (3) is satisfied, and therefore, by Theorem 5.7 we get that $COMP^H_{\Delta,S,M,T,w}$ is a $(c,s,m,q,\delta',\epsilon,T)$-one-time computable PRF, for $m \approx c+s-\delta$, and $\delta' \approx \delta$ and a negligible ϵ.

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 $V = \{(0,0)\} \cup \{(i,j) : 1 \leq i \leq j \leq M$ (cf. Figure 2, Appendix A). A graph Arr_M consists of one input vertex $(0,0)$ and one output vertex (M,M):

Lemma 6.1. *There exists a strategy that pebbles Arr_M using $M+1$ pebbles.*

Proof. The strategy works as follows.

pebble the bottom row For $j = 1,\ldots,M$ put a pebble on $(1,j)$.

pebble the rest of the graph For $i = 2,\ldots,M$ do the following:

for every $j = i,\ldots,M$ put a pebble on (j,i) and then remove a pebble from $(j-1,i-1)$.

It is easy to see that this pebbling strategy is correct, and uses $M+1$ pebbles. ☐

Hence, using Remark 4.1, we get the following.

Corollary 6.2. *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.*

6.1 Impossibility of pebbling

We now show the optimality of the strategy given in Lemma 6.1. The proof of the following lemma is essentially very similar to the proof of Lemma 10.2.1 appearing in the book of John Savage ([38]). For the lack of space we move it to Appendix C.

Lemma 6.3. *Every strategy that pebbles the output of Arr_M, and does not use the red pebbles, must use at least $M-1$ black pebbles.*

Combining Lemma 6.3 with Corollary 4.3 we get the following.

Corollary 6.4. *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 $q\cdot 2^{-w}+2^{-\lambda}$.*

The corollaries and the lemma above imply the following.

Theorem 6.5. *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 Open problems

It would be very interesting to show any non-trivial schemes for one-time computable and uncomputable functions without the need of the random oracle assumption. Another interesting research direction is to find more applications for the notions introduced in this paper.

References

[1] Adi Akavia, Shafi Goldwasser, and Vinod Vaikuntanathan. Simultaneous hardcore bits and cryptography against memory attacks. In Omer Reingold, editor, *Theory of Cryptography, 6th Theory of Cryptography Conference, TCC 2009, San Francisco, CA, USA, March 15-17, 2009. Proceedings*, volume 5444 of *Lecture Notes in Computer Science*, pages 474–495. Springer, 2009.

[2] Joël Alwen, Yevgeniy Dodis, and Daniel Wichs. Leakage-resilient public-key cryptography in the bounded-retrieval model. In Shai Halevi, editor, *CRYPTO*, volume 5677 of *Lecture Notes in Computer Science*, pages 36–54. Springer, 2009.

[3] Joel Alwen, Yevgeniy Dodis, and Daniel Wichs. Leakage-resilient public-key cryptography in the bounded-retrieval model. Cryptology ePrint Archive, Report 2009/160, 2009. `http://eprint.iacr.org/`.

[4] Giuseppe Ateniese, Roberto Di Pietro, Luigi V. Mancini, and Gene Tsudik. Scalable and efficient provable data possession. In *SecureComm '08: Proceedings of the 4th international conference on Security and privacy in communication netowrks*, pages 1–10, New York, NY, USA, 2008. ACM.

[5] Yonatan Aumann, Yan Zong Ding, and Michael O. Rabin. Everlasting security in the bounded storage model. *IEEE Transactions on Information Theory*, 48(6):1668–1680, 2002.

[6] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM Conference on Computer and Communications Security*, pages 62–73, 1993.

[7] Zvika Brakerski and Shafi Goldwasser. Circular and leakage resilient public-key encryption under subgroup indistinguishability (or: Quadratic residuosity strikes back). Cryptology ePrint Archive, Report 2010/226, 2010. `http://eprint.iacr.org/`, accepted to CRYPTO'10.

[8] Zvika Brakerski, Yael Tauman Kalai, Jonathan Katz, and Vinod Vaikuntanathan. Cryptography resilient to continual memory leakage. Cryptology ePrint Archive, Report 2010/278, 2010. `http://eprint.iacr.org/`, accepted to FOCS'10.

[9] Ran Canetti, Shai Halevi, and Michael Steiner. Mitigating dictionary attacks on password-protected local storage. In Dwork [15], pages 160–179.

[10] David Cash, Yan Zong Ding, Yevgeniy Dodis, Wenke Lee, Richard J. Lipton, and Shabsi Walfish. Intrusion-resilient key exchange in the bounded retrieval model. In Salil P. Vadhan, editor, *TCC*, volume 4392 of *Lecture Notes in Computer Science*, pages 479–498. Springer, 2007.

[11] Giovanni Di Crescenzo, Richard J. Lipton, and Shabsi Walfish. Perfectly secure password protocols in the bounded retrieval model. In Halevi and Rabin [28], pages 225–244.

[12] Francesco Davì, Stefan Dziembowski, and Daniele Venturi. Leakage-resilient storage. Cryptology ePrint Archive, Report 2009/399, 2009. `http://eprint.iacr.org/`, accepnted to SCN'10.

[13] Yevgeniy Dodis, Shafi Goldwasser, Yael Tauman Kalai, Chris Peikert, and Vinod Vaikuntanathan. Public-key encryption schemes with auxiliary inputs. In Daniele Micciancio, editor, *TCC*, volume 5978 of *Lecture Notes in Computer Science*, pages 361–381. Springer, 2010.

[14] Yevgeniy Dodis, Kristiyan Haralambiev, Adriana Lopez-Alt, and Daniel Wichs. Cryptography against continuous memory attacks. Cryptology ePrint Archive, Report 2010/196, 2010. http://eprint.iacr.org/, accepted to FOCS'10.

[15] Cynthia Dwork, editor. *Advances in Cryptology - CRYPTO 2006, 26th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2006, Proceedings*, volume 4117 of *Lecture Notes in Computer Science*. Springer, 2006.

[16] Cynthia Dwork, Andrew Goldberg, and Moni Naor. On memory-bound functions for fighting spam. In Dan Boneh, editor, *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 426–444. Springer, 2003.

[17] Cynthia Dwork, Moni Naor, and Hoeteck Wee. Pebbling and proofs of work. In Victor Shoup, editor, *CRYPTO*, volume 3621 of *Lecture Notes in Computer Science*, pages 37–54. Springer, 2005.

[18] Cynthia Dwork, Moni Naor, and Hoeteck Wee. Pebbling and proofs of work. In Victor Shoup, editor, *Advances in Cryptology – CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 37–54. Springer Berlin / Heidelberg, 2005.

[19] Stefan Dziembowski. Intrusion-resilience via the bounded-storage model. In Halevi and Rabin [28], pages 207–224.

[20] Stefan Dziembowski. On forward-secure storage. In Dwork [15], pages 251–270.

[21] Stefan Dziembowski and Ueli M. Maurer. Optimal randomizer efficiency in the bounded-storage model. *J. Cryptology*, 17(1):5–26, 2004.

[22] Stefan Dziembowski and Krzysztof Pietrzak. Intrusion-resilient secret sharing. In *FOCS*, pages 227–237, 2007.

[23] Stefan Dziembowski and Krzysztof Pietrzak. Leakage-resilient cryptography. In *FOCS '08: Proceedings of the 49th Annual IEEE Symposium on Foundations of Computer Science*, Washington, DC, USA, 2008. IEEE Computer Society.

[24] Stefan Dziembowski, Krzysztof Pietrzak, and Daniel Wichs. Non-malleable codes. In *Proceedings of the First Symposium on Innovations in Computer Science, ICS2010*, pages 434–452, 2010.

[25] ECRYPT. *The Side Channel Cryptanalysis Lounge* http://www.crypto.rub.de/en_sclounge.html.

[26] Sebastian Faust, Eike Kiltz, Krzysztof Pietrzak, and Guy N. Rothblum. Leakage-resilient signatures. In Daniele Micciancio, editor, *Theory of Cryptography, 7th Theory of Cryptography Conference, TCC 2010, Zurich, Switzerland, February 9-11, 2010. Proceedings*, volume 5978 of *Lecture Notes in Computer Science*, pages 343–360. Springer, 2010.

[27] Rosario Gennaro, Anna Lysyanskaya, Tal Malkin, Silvio Micali, and Tal Rabin. Algorithmic tamper-proof (atp) security: Theoretical foundations for security against hardware tampering. In Naor [34], pages 258–277.

[28] Shai Halevi and Tal Rabin, editors. *Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings*, volume 3876 of *Lecture Notes in Computer Science*. Springer, 2006.

[29] Yuval Ishai, Amit Sahai, and David Wagner. Private Circuits: Securing Hardware against Probing Attacks. In *CRYPTO*, pages 463–481, 2003.

[30] Jonathan Katz and Vinod Vaikuntanathan. Signature schemes with bounded leakage resilience. In *ASIACRYPT*, pages 703–720, 2009.

[31] Feng-Hao Liu and Anna Lysyanskaya. Algorithmic tamper-proof security under probing attacks. In Juan A. Garay and Roberto De Prisco, editors, *SCN*, volume 6280 of *Lecture Notes in Computer Science*, pages 106–120. Springer, 2010.

[32] Ueli M. Maurer. Conditionally-perfect secrecy and a provably-secure randomized cipher. *J. Cryptology*, 5(1):53–66, 1992.

[33] Silvio Micali and Leonid Reyzin. Physically observable cryptography (extended abstract). In Naor [34], pages 278–296.

[34] Moni Naor, editor. *Theory of Cryptography, First Theory of Cryptography Conference, TCC 2004, Cambridge, MA, USA, February 19-21, 2004, Proceedings*, volume 2951 of *Lecture Notes in Computer Science*. Springer, 2004.

[35] Moni Naor and Gil Segev. Public-key cryptosystems resilient to key leakage. In *Advances in Cryptology - CRYPTO*, August 2009.

[36] Daniele Perito and Gene Tsudik. Secure code update for embedded devices via proofs of secure erasure. In Dimitris Gritzalis, Bart Preneel, and Marianthi Theoharidou, editors, *ESORICS*, volume 6345 of *Lecture Notes in Computer Science*, pages 643–662. Springer, 2010.

[37] Krzysztof Pietrzak. A leakage-resilient mode of operation. In Antoine Joux, editor, *Advances in Cryptology - EUROCRYPT 2009, 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings*, volume 5479 of *Lecture Notes in Computer Science*, pages 462–482. Springer, 2009.

[38] John E. Savage. *Models of Computation: Exploring the Power of Computing.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[39] François-Xavier Standaert, Tal Malkin, and Moti Yung. A unified framework for the analysis of side-channel key recovery attacks. In *EUROCRYPT*, pages 443–461, 2009.

[40] Luis von Ahn, Manuel Blum, Nicholas J. Hopper, and John Langford. Captcha: Using hard ai problems for security. In Eli Biham, editor, *EUROCRYPT*, volume 2656 of *Lecture Notes in Computer Science*, pages 294–311. Springer, 2003.

A Figures

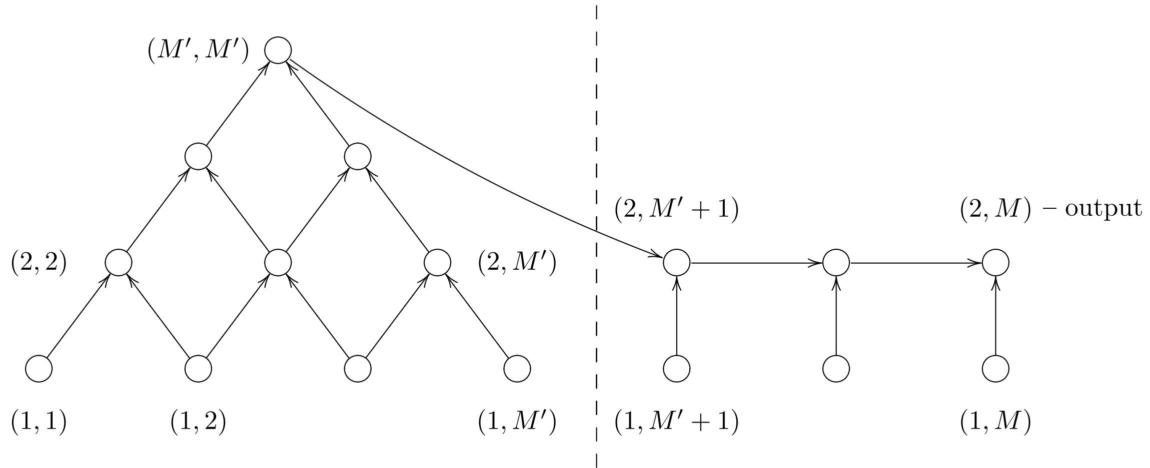

Figure 1: An (M, M')-lambda graph for $M' = 4$ and $M = 7$. The sub-graph on the left-hand side of the dashed line is an M'-pyramid.

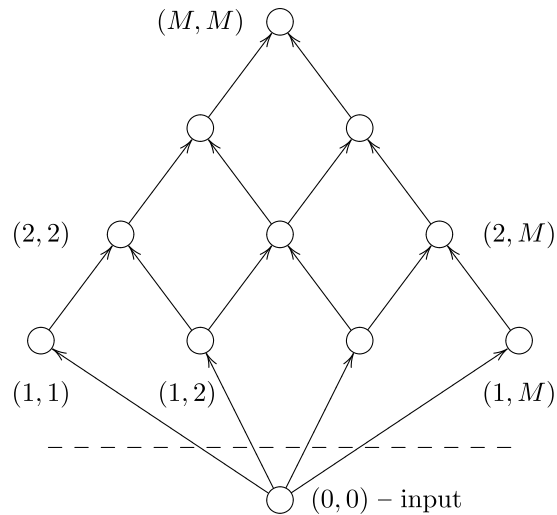

Figure 2: An M-arrowhead graph G_M for $M = 4$. The subgraph on the upper side of the dashed line is an M-pyramid.

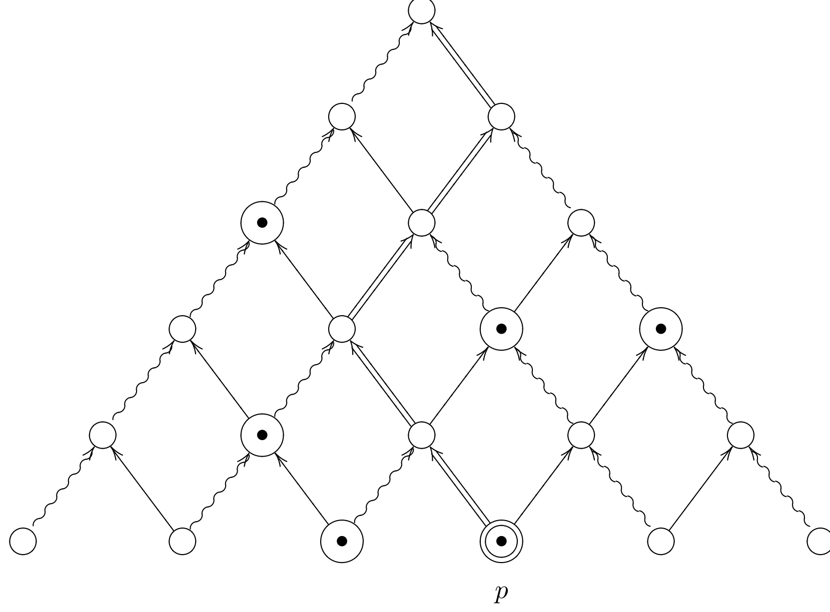

Figure 3: Illustration of the proof of Lemma 6.3. Double arrows indicate the path π, and the disjoint paths starting from other vertices from first row and connected to path π are indicated with the wave arrows.

B Proof of Theorem 4.2

As a precursor to the proof of Theorem 4.2, we will need the following Lemma.

Lemma B.1. *Let $B = b_1, \ldots, b_u$ be random bits. Let $\mathcal{P}$ be a randomized procedure which gets a hint $h \in \mathcal{H}$, and can adaptively query any of the bits of B by submitting an index i and receiving b_i. At the end of the execution $\mathcal{P}$ outputs a subset $S \subset \{1, \ldots, u\}$ of $|S| = k$ indices which were not previously queried, along with guesses for all of the bits $\{b_i | i \in S\}$. Then the probability (over the choice of B and randomness of $\mathcal{P}$) that there* exists *some $h \in \mathcal{H}$ for which $\mathcal{P}(h)$ outputs* all correct guesses *is at most $\frac{|\mathcal{H}|}{2^k}$.*

Proof. Fix any $h \in \mathcal{H}$ a-priori and independently of B. Then the probability that $\mathcal{P}(h)$ outputs all correct guesses is $1/2^k$. By the union-bound, the probability that there *exists* some h (a-posteriori, depending on B) is therefore at most $\frac{|\mathcal{H}|}{2^k}$. □

Proof of Theorem 4.2. First, let us show part 1 of the theorem, that the ex-post-facto pebbling is legal with probability at least $1 - \frac{q}{2^w}$. Assume otherwise. The only way that our pebbling could be illegal is if, during the processing of a correct oracle call (made by $\mathcal{A}_{big}$ or $\mathcal{A}_{small}$), one of the input-vertices v of the call does not have a pebble of the correct color (resp. red or any) on it. Since such a pebble would never have been deleted, this can only happen if it was never placed. That is, there must be a vertex v, which is not an input-vertex of G, such that the execution-transcript of $\mathcal{A}$ contains a correct oracle call (made by either $\mathcal{A}_{big}$ or $\mathcal{A}_{small}$) with v as an input vertex, which *precedes* all correct oracle calls made with v as an output-vertex. Therefore, the above must happen with probability greater than $\frac{q}{2^w}$. But then, we can define a predictor $\mathcal{P}$ for the values of $B = (R, H)$ which:

Gets as hint: The index $i \in \{1, \ldots, q\}$ of the oracle-call made by $\mathcal{A}^{H(\cdot)}(R)$ that satisfies the requirement.

Runs: Runs $\mathcal{A}^{H(\cdot)}(R)$. Answers all queries of $\mathcal{A}$ honestly (using access to H, R) until the ith oracle query made by $\mathcal{A}$, which is of the form $H(\mathtt{label}_1, \ldots, \mathtt{label}_d, v)$. By assumption, for at least one of the parents v_i of v, the oracle was never queried at the point $\mathtt{preLabel}(v_i)$, yet $\mathtt{label}_i = \mathtt{label}(v_i)$. Moreover, it is easy to figure out i, by computing $\mathtt{preLabel}(v_j)$ for each parent v_j of v, without querying the oracle on input $\mathtt{preLabel}(v_i)$.

Outputs: The bits of H corresponding to $\mathtt{label}(v_i)$ at "position" $\mathtt{preLabel}(v_i)$.

But, by Lemma B.1, the probability of the above succeeding is at most $\frac{q}{2^w}$, leading to a contradiction.

Next let us show part 2 of the theorem. Again, assume otherwise, that there is some $\lambda \geq 0$ for which the red-pebble complexity of the ex-post-facto pebbling is $r \geq \frac{c+\lambda}{w-\log(q)}$ with probability (strictly) greater than $2^{-\lambda}$. The only way that the red-pebble complexity could be r is if there are r distinct red-necessary vertices v. Recall that a vertex is red-necessary if the transcript includes correct oracle call made by $\mathcal{A}_{big}$ with v as one of the input-vertices, which *precedes* all correct oracle calls made by $\mathcal{A}_{big}$ with v as an output-vertex. We call the corresponding oracle-calls red-necessary, and there are $r' \leq r$ of them (one oracle-call can make many of its input-vertices red-necessary). The intuition is that the algorithm $\mathcal{A}_{big}$ must then somehow predict the labels of these red-necessary vertices without querying the appropriate input to the oracle, given the communication from $\mathcal{A}_{small}$ as a hint. That is, we define a predictor $\mathcal{P}$ for the bits of $B = (R, H)$, which works as follows:

Gets as hint: The value $h_{com} \in \{0,1\}^c$ of all communication from $\mathcal{A}_{small}$ to $\mathcal{A}_{big}$ made during the execution $\mathcal{A}^{H(\cdot)}(R)$. The indices $(i_1, \ldots, i_{r'}) \subseteq \{1, \ldots, q\}^{r'}$ of the r' red-necessary oracle-calls made by $\mathcal{A}_{big}^{H(\cdot)}$ during the execution.

Runs: Runs $\mathcal{A}_{big}^{H(\cdot)}$ and feeds it the correct communication on behalf of $\mathcal{A}_{small}$ (without running $\mathcal{A}_{small}$ at all) using the hint. For the random-oracle queries corresponding to the indices $(i_1, \ldots, i_{r'})$, record the labels of all the input-vertices of such calls (we do not yet know which ones are red-necessary). To answer any oracle calls of $\mathcal{A}_{big}$, with output-vertex v:

- Determine if the call is correct. A call is correct iff (1) it corresponds to one of the stored indices i_j, or (2) correct oracle calls were previously made by $\mathcal{A}_{big}$ on all parents of v (having them as an output-vertex) and the provided input to the current call matches the output of all these previous calls. Note that correctness can therefore be checked recursively without making any new oracle calls.
- If the call is correct *and* the label of v is one of the recoded labels, output it. Otherwise query H to answer the call.

At the end, use the transcript of all oracle calls made by $\mathcal{A}_{big}$ to determine which r vertices $v_1, \ldots, v_r$ are red-necessary. The labels $\texttt{label}(v_1), \ldots, \texttt{label}(v_r)$ are among the recorded labels. Compute $\texttt{preLabel}(v_1), \ldots, \texttt{preLabel}(v_r)$, which can be done without querying H with these as inputs.

Outputs: The bits of H corresponding to $\texttt{label}(v_1), \ldots, \texttt{label}(v_r)$, at positions $\texttt{preLabel}(v_1), \ldots, \texttt{preLabel}(v_r)$.

It is easy to check that, in the above process, H is never queried on the inputs $\texttt{preLabel}(v_i)$ for the red-necessary vertices v_i. Therefore, by Lemma B.1, the probability of the above succeeding is at most $\frac{q^r 2^c}{2^{rw}} \leq 2^{-(r(w-\log(q))-c)} \leq 2^{-\lambda}$, leading to a contradiction.

Lastly, let us turn to part 3 of the theorem. Again, assume otherwise, that there is some $\lambda \geq 0$ for which the sum of the red-pebble and black-pebble complexities of the ex-post-facto pebbling is $z \geq \frac{c+s+\lambda}{w-\log(q)}$ with probability (strictly) greater than $2^{-\lambda}$. The only way that this could happen is if r of the vertices are red-necessary and if at *at some point* there are there are b black-necessary vertices (note that these sets are disjoint by definition). As the hint, we will store the value h_{state} which encodes the entire state of $\mathcal{A}_{small}$ corresponding to that point in the transcript and h_{com} which encodes all of the communication from $\mathcal{A}_{small}$ to $\mathcal{A}_{big}$:

Gets as hint: The values $h_{com} \in \{0,1\}^c, h_{state} \in \{0,1\}^s$. The indices $(i_1, \ldots, i_{z'}) \subseteq \{1, \ldots, q\}^{z'}$ of the $z' \leq z$ distinct oracle-calls made by $\mathcal{A}_{big}^{H(\cdot)}$ and $\mathcal{A}_{small}^{H(\cdot)}$ which make some vertex red-necessary or black-necessary.

Runs: First, run $\mathcal{A}_{big}^{H(\cdot)}$ *from the beginning* by feeding it the correct communication on behalf of $\mathcal{A}_{small}$ (without running $\mathcal{A}_{small}$) using the hint. Answer oracle queries as before. Once this is done, run $\mathcal{A}_{small}$ starting in the state encoded by h_{state} and pass it the communication on behalf of $\mathcal{A}_{big}$ that was produced by the earlier run. We can use the same strategy as in the last case to determine if oracle calls made by $\mathcal{A}_{small}$ are correct, and how to respond to them. At the end, we will have recorded the labels of all of the red-necessary and black-necessary vertices $v_1, \ldots, v_z$, and can compute $\texttt{preLabel}(v_i)$ as before.

Outputs: The bits of H corresponding to $\texttt{label}(v_1), \ldots, \texttt{label}(v_r)$, at positions $\texttt{preLabel}(v_1), \ldots, \texttt{preLabel}(v_z)$.

By Lemma B.1, the probability of the above succeeding is at most $\frac{q^z 2^c 2^s}{2^{zw}} \leq 2^{-(r(w-\log(q))-c-s)} \leq 2^{-\lambda}$, leading to a contradiction.

$\square$

C Proof of Lemma 6.3

Proof. We consider all paths from (M, M) to the first row (i.e. the set of vertices $\{(1,1), \ldots, (1, M)\}$). We say that a path *carries a pebble* if at least one vertex of the path has some pebble on it. If a path is not carrying a pebble we say it is *empty*.

Initially all paths are empty. At the end of a game, all paths must carry a pebble (because there is a pebble in (M, M), and (M, M) is a vertex in every path). Therefore, there must be a first moment t in time when all paths are carrying a pebble. Putting a pebble into graph can increase number of paths carrying a pebble only when putting pebble on the first row. So moment t must happen when a pebble p is put on the first row of some path π and π is empty except of the pebble p at the bottom (cf. Figure 3, Appendix A). Let us look at the disjoint paths starting from other vertices from first row (there are $M-1$ of them) and connected to path π. It is always possible to find such disjoint paths (cf. Figure 3). Each of this $M-1$ disjoint paths must carry a pebble. Additionally, we need to count a pebble p, and a pebble on $(0,0)$ (it is impossible to put a pebble on any vertex of the first row, without having a pebble on $(0,0)$). Altogether we have $M+1$ pebbles. This finishes the proof. □