University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Tomasz Kazana

Security against space-restricted physical attacks

PhD dissertation

Supervisor

dr hab. Stefan Dziembowski

Institute of Informatics
University of Warsaw

December 2012

Author's declaration:
aware of legal responsibility I hereby declare that I have written this dissertation myself and all the contents of the dissertation have been obtained by legal means.

December, 2012 .....

date *mgr Tomasz Kazana*

Supervisor's declaration:
the dissertation is ready to be reviewed

December, 2012 .....

date *dr hab. Stefan Dziembowski*

Abstract

The dissertation introduces a new, defined by the author, model of cryptographic computation, called SBA–model. Characteristic features of the model are: space boundary, leakage and random oracle. The dissertation studies three schemes (one-time computable pseudorandom functions, key-evolution schemes and one-time programs) of cryptographic primitives. We show existence of these schemes in SBA–model. The results of this dissertation were presented in [19, 24, 25].

Key words: pebble, one-time program, key-evolution, time-memory trade-off

AMS Classification: 94A60, 68P25.

Streszczenie

Rozprawa doktorska wprowadza nowy, zdefiniowany przez autora model obliczeń kryptograficznych, nazwany SBA–modelem. Charakterystyczne cechy tego modelu to ograniczona pamieć, wycieki oraz użycie losowej wyroczni. W rozprawie badane sa trzy schematy kryptograficzne: *Jednorazowe funkcje pseudolosowe*, *Schemat ewolucji klucza* oraz *Funkcje jednorazowe* (ang. *One-time computable pseudorandom function*, *Key-evolution schemes* oraz *One-time programs*). Pokazujemy istnienie ww. schematów w SBA–modelu. Wyniki z roprawy zostały zaprezentowane w następujących pracach: [19, 24, 25].

Słowa kluczowe: pebble, one-time program, key-evolution, time-memory trade-off

Klasyfikacja AMS: 94A60, 68P25.

Contents

List of Figures

Chapter 1

Introduction

This dissertation constructs several protocols secure against physical attacks. Traditionally, most of the security proofs in cryptography assume the existence of devices that are completely safe. For example, we can strictly prove the security of electronic car keys, when we assume that the adversary can eavesdrop just what is *on the air*: all the communication between the car and the remote control. However, what is *inside of the key*, remains secret. In practice, however, it is very difficult to construct a physical device which is entirely resilient to information leakage. In particular, leakage on the internals can often be obtained using *side-channel attacks*, that exploit physical phenomena such as electromagnetic radiation [44,55], timing [10], power consumption [43], acoustic emanations [57], and many others (see e.g. [51] for an overview). Another case in which the adversary may obtain leakage from cryptographic protocols is the situation when the protocols are implemented on PCs on which the adversary can install malicious software, like the computer viruses.

The above concerns give motivation for the research on protocol on untrusted machines. The first papers proposing algorithmic countermeasures against the side-channel attacks came from the practitioners' community (e.g. [13]). Later this area attracted a lot of attention also from the theoreticians, starting from [38, 49]. The theoretical countermeasures against the virus attacks are known under the name *bounded-retrieval model* [14,22]. Theoretical work offers rigor and provable security, at the cost of having to make strong restrictions on the type of leakage and designing complicated schemes to make standard reduction-based proof techniques go through (an example of such an assumption is that only the data actually used in computation can leak to the adversary). On the other hand, practical work focuses on simple and efficient schemes, often at the cost of achieving only an intuitive notion of security without formal well-specified guarantees.

In this dissertation, we complement the two tracks via a middle-of-the-road approach. On one hand, we rely on the random-oracle model, acknowledging the usefulness of this methodology in practice despite its theoretical shortcomings. On the other hand, we show that even in the random-oracle model, designing secure leakage-resilient schemes with clear and meaningful guarantees requires great care and is susceptible to pitfalls. For example, just assuming that leakage "cannot evaluate the random oracle" can be misleading. Instead, we define a new model in which we assume that the "leakage" can be any arbitrary *space bounded* computation that can make random oracle calls itself. Our security proofs do not rely on the assumption that only data used in the computation can leak.

In general, two types of adversarial models are considered in this research area. In the *passive* one, the adversary can get some partial information about the internal data stored on a cryptographic machine $\mathcal{M}$. This line of research, motivated by various side-channel attacks was initiated in the seminal papers of Ishai et al. [38] and Micali and Reyzin [49], and followed by many recent works [1,8,9,16–18,28,31,41,50,53,58]. Some papers have also been motivated by the attacks of the malicious software (like viruses) against computers [2, 3, 12, 14, 22, 23, 27]. 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 [30, 33, 37, 46] 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 [37] the adversary that can tamper a restricted number of wires of the circuit that performs the computation (in a given time-frame), and in [33] it is assumed that a device is equipped with a small tamper-free component. So far, all results that provided solutions against such attacks were secure under the assumption that the virus can download the data from the machine, but he cannot modify any information stored on it. This dissertation provides the first schemes were the adversary in the BRM can also modify the data stored on the machine.

We achieve security against an adversary that has *complete active/passive control* over a device $\mathcal{M}$ storing cryptographic secrets, by only relying on a

simple physical characteristic of the device: we assume that local memory in the device is bounded.

1.1 Model of Computation (SBA–model)

To make our statements precise, we must fix a model of computation. We will 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 $\mathcal{A}_{big}$. The described model will be called SBA–model. [1]

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* [7]. 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).

We model our adversary $\mathcal{A} = (\mathcal{A}_{big}, \mathcal{A}_{small})$ as a pair of interactive algorithms[2] 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.[3]

- c-bounded communication: The total number of outgoing bits communicated by $\mathcal{A}_{small}$ is bounded by c. [4]

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 $H(\cdot)$.

[1] The abbreviation comes from *Small and Big Adversary*.

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

[3] 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.

[4] 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...

1.2 Remark

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).

1.3 Our Results

We define the following schemes in SBA–model:

1. One-time Pseudorandom Functions.

2. Key-Evolution Scheme.

3. One-time Programs.

For details see Sections (respectively): 3.1, 4.2.1, 5.3. The main Theorems are (respectively): 3.10, 4.1, 5.2. Introduction and informal intuitions to each of these schemes are written in the beginings of Chapters (respectively): 3, 4, 5.

Most parts of this dissertation are covered by the following papers: [19,24,25]. These papers were supported by The European Research Council Starting Grant, agreement CNTM-207908 (European Community's Seventh Framework Programme FP7/2007-2013).

Acknowledgements

Most importantly I would like to thank my advisor, dr hab. Stefan Dziembowski, for constant motivation and invaluable help during the process of writing this dissertation and for co-authoring all research papers included in it. I am also really grateful to prof. Damian Niwinski, for many kinds of guidance and support during the whole period of my PhD studies. I would like to express my gratitude to: Yevgeniy Dodis, Konrad Durnoga, Zdzisław Kręcina, Filip Mazowiecki, Maciej Obremski, Daniel Wichs and Michał Zając.

Finally I would like to thank my friends and family for strong support during my PhD studies.

Chapter 2

Preliminaries

In this chapter we present some basic technical definitions and lemmas. The concept of graph labeling is similar to concept of Dwork, Naor and Wee [21]. Here we present formal definition. Intuitions will follow in Chapters 3 and 4.

2.1 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 $M \leqslant N$ 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 $Eval(G, H, (R_1, \ldots, R_M))$ 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 $\mathcal{A}_{small}$.

2.2 Hint Lemma

Lemma 2.1. *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 $\frac{|\mathcal{H}|}{2^k}$. □

Chapter 3

One-Time Computable Pseudorandom Functions

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. In this chapter we give explicite construction for SBA–model.

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).

In this chapter, 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 [52].

3.1 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 $F_{2,K}(x_2)$ ...). 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 3.6. The maximal T that we can have is approximately equal to $\frac{c+s}{2(c+s-m)}$ (cf. (3.5)).

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 chapter 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 [11]. Here, a computation of H requires the user to solve the CAPTCHA puzzles [59], 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 chapter 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.

3.2 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 [52]. 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 [52] 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 [52]. What remains is to show a construction of such an H. We do it in Section 3.7.

3.3 Previous work

Most of the related work was already described in the previous sections. In our work we will use a technique called *graph pebbling* (see e.g. [56]). This technique has already been used in cryptography in an important work of [21], 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* [5, 26, 47]. As similar assumption was also used in [20]. The proof of

erasures can be viewed as a special case of the *remote attestation protocols* (see [52] for a list of relevant references).

3.3.1 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 $1 \leqslant i \leqslant j \leqslant N$, we define $R[i, \ldots, j] = (R_i, \ldots, R_j)$.

3.4 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 $(c, s, m, q, \epsilon, T)$-*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 $((F^H_{i,R}(x), (i, x)), \texttt{out})$ 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.

3.5 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 [21].

3.5.1 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 3.1. *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 $Eval(G, H, (R_1, \ldots, R_M))$ 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.

3.5.2 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.[1]

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.[2]

[1] 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.

[2] Note that the set of black-necessary vertices can be different at different points in the

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.

Theorem 3.2. *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 $\lambda \geqslant 0$, 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).*

Proof of Theorem 3.2. 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

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.

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 $H(\texttt{label}_1, \ldots, \texttt{label}_d, v)$. By assumption, for at least one of the parents v_i of v, the oracle was never queried at the point $\texttt{preLabel}(v_i)$, yet $\texttt{label}_i = \texttt{label}(v_i)$. Moreover, it is easy to figure out i, by computing $\texttt{preLabel}(v_j)$ for each parent v_j of v, without querying the oracle on input $\texttt{preLabel}(v_i)$.

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

But, by Lemma 2.1, 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 $\lambda \geqslant 0$ for which the red-pebble complexity of the ex-post-facto pebbling is $r \geqslant \frac{c+\lambda}{w-\log(q)}$ 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 $r' \leqslant r$ 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 2.1, the probability of the above succeeding is at most $\frac{q^r 2^c}{2^{rw}} \leqslant 2^{-(r(w-\log(q))-c)} \leqslant 2^{-\lambda}$, leading to a contradiction.

Lastly, let us turn to part 3 of the theorem. Again, assume otherwise, that there is some $\lambda \geqslant 0$ for which the sum of the red-pebble and black-pebble complexities of the ex-post-facto pebbling is $z \geqslant \frac{c+s+\lambda}{w-\log(q)}$ 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 $z' \leqslant z$ 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 2.1, the probability of the above succeeding is at most $\frac{q^z 2^c 2^s}{2^{zw}} \leqslant 2^{-(r(w-\log(q))-c-s)} \leqslant 2^{-\lambda}$, leading to a contradiction.

□

Lemma 3.3. *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 $\texttt{preLabel}(v)$. 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 $\texttt{label}$ 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 3.2 (Points 1 and 3) we get that $P(\mathcal{G}) \geqslant 1 - q \cdot 2^{-w} - 2^{-\lambda}$. 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. □

3.6 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. 3.1). Its set of vertices is equal to $V_0 \cup V_1$, where

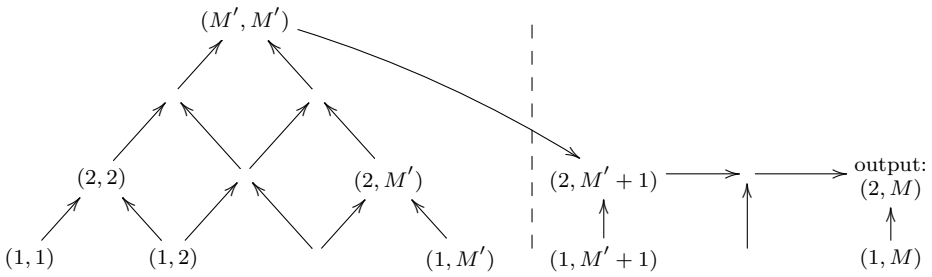


Figure 3.1: An (M, M')-lambda graph for $M' = 4$ and $M = 7$. The sub-graph on the left-hand side of the dashed line is an M'-pyramid.

$V_0 = \{(i, j) : 1 \leqslant i \leqslant j \leqslant M'\}$ 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:

$$\begin{aligned}
&\{((i-1, j-1), (i, j)) : (i-1, j-1), (i, j) \in V_0\} && (3.1)\\
&\cup\{((i-1, j), (i, j)) : (i-1, j), (i, j) \in V_0\} && (3.2)\\
&\cup\{((M', M'), (2, M'+1))\}\\
&\cup\{((1, j-1), (1, j)) : (1, j-1), (1, j) \in V_1\}\\
&\cup\{((1, j), (2, j)) : (1, j), (2, j) \in V_1\}.
\end{aligned}$$

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 (3.1) and (3.2) above. Its output vertex is (M, M). Such a graph is also called an M-*pyramid graph*.

Lemma 3.4. *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. □ □

3.6.1 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 3.5. *For $M \geqslant 2$ 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 $|\mathcal{X}| - 1 \geqslant M - 1$, which implies that $|\mathcal{X}| \geqslant M$. □ □

Lemma 3.6. *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 3.5. 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 3.5 (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 3.7. *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:

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

$E := \{((k,v),(k,v')) : v, v' \in V'_k \text{ and } (v,v') \in E_k\} \cup \{(v,(k,v')) : v \in V_I \text{ and } v' \in V'_k \text{ and } (v,v') \in E_k\}$. 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 3.8. *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 3.6 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 3.5 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 3.6 the vertex (M',M') is not yet pebbled in this configuration. Hence, by Lemma 3.7, 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 $M+M'-2$. □

□

Combining Lemma 3.3 with Lemma 3.8 we get the following.

Corollary 3.9. *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 $(q+1)\cdot 2^{-w}$.*

3.6.2 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

$$T < \frac{U-1}{2\Delta} \tag{3.3}$$

where $\Delta := U - M$.

We now construct an interactive algorithm $COMP^H_{U,M,T,w}$ 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

$$Eval(Lam^{M-(i-1)\Delta}_{i\Delta+2}, H^{(i,x_i)}, (R[1+(i-1)\Delta, \ldots, M]). \tag{3.4}$$

The algorithm $COMP^H_{U,M,T,w}(R)$ simply computes each (3.4) 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 3.1 from Section 3.5.1. Observe that from Lemma 3.4 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 3.1 we get that there is an algorithm that computes (3.4) 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 3.10. *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

$$T \approx \frac{c+s}{2(c+s-m)}. \tag{3.5}$$

of Theorem 3.10. 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.

$$P(\mathcal{E}) \leqslant T \cdot (q+1) \cdot 2^{-w}. \tag{3.6}$$

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

$$\left\{ Lam_{\tilde{i}\Delta+2}^{M-(\tilde{i}-1)\Delta} \right\}_{x \in \{0,1\}^*}.$$

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:

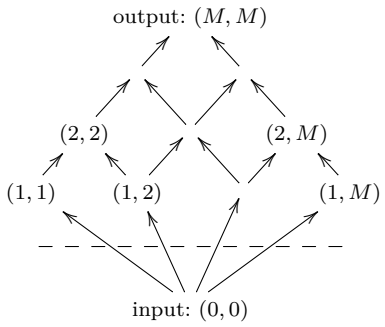
- Q has a form $((\tilde{i}, x), (\texttt{label}_1, \ldots, \texttt{label}_d, v))$ (for some $x, \texttt{label}_1, \ldots, \texttt{label}_d$) — in this case we translate it into a a query $(\texttt{label}_1, \ldots, \texttt{label}_d, ((i, x), v)$,
- Q has a form or $((\tilde{i}, x), (\texttt{label}, v, \texttt{out}))$ (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 3.9 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 $P(\mathcal{E}) \leqslant T \cdot ((q+1)\cdot 2^{-w})$. Therefore (3.6) is proven. □

3.7 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 $V = \{(0,0)\} \cup \{(i,j) : 1 \leqslant i \leqslant j \leqslant M$. 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 3.11. *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 3.1 it suffices to show a strategy that pebbles Arr_M using $M + 1$ pebbles. 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. □

3.7.1 Impossibility of pebbling

We now show the optimality of the strategy given in Lemma 3.11. The proof of the following lemma is very similar to the proof of Lemma 10.2.1 in the book of John Savage ([56]).

Lemma 3.12. *Every strategy that pebbles the output of Arr_M, and does not use the red pebbles, must use at least $M - 1$ black pebbles.*

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 3.2). 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 3.2). 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. □

Combining Lemma 3.12 with Corollary 3.3 we get the following.

Corollary 3.13. *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 3.14. *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.*

3.8 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 chapter.

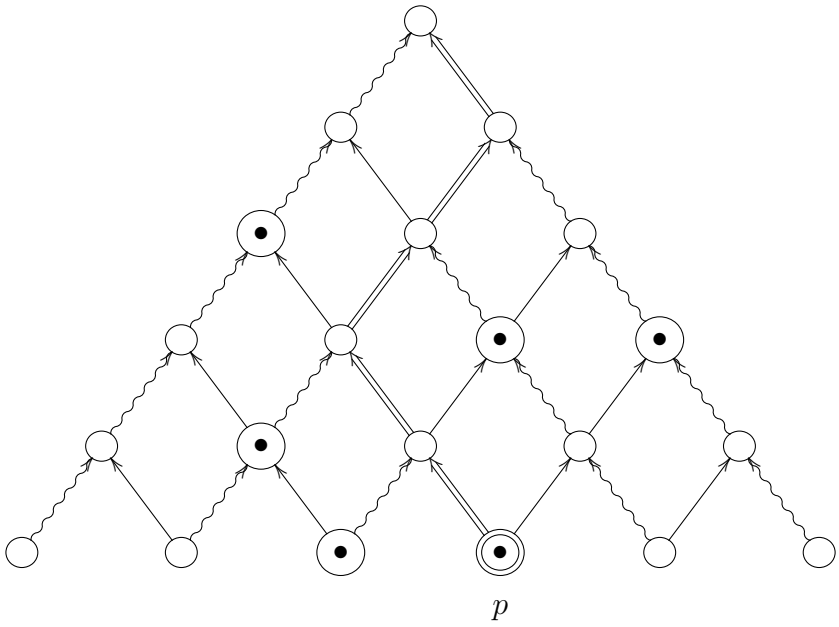


Figure 3.2: Illustration of the proof of Lemma 3.12. Double arrows indicate the path π, and the disjoint paths starting from other vertices from first row and connected to path π are indicated with the wave arrows.

Chapter 4

Key-Evolution Schemes Resilient to Space-Bounded Leakage

In this chapter we consider the following natural problem. Suppose a cryptographic key K_0 is stored on a device that leaks information. If leakage occurs continuously, then the adversary may obtain more and more information about the key, and eventually learn it entirely. A natural idea to prevent this from happening is to periodically update the key i.e. to repeatedly apply some *key evolution function* f to it, obtaining a sequence of keys $K_0, K_1, \ldots$, where each $K_{i+1} := f(K_i)$. The key evolution function should be constructed in such a way that the evolved key K_i used in period i is indistinguishable from uniform, even if the adversary can leak from the entire evolution process $K_1 \to K_2 \to \ldots \to K_{i-1}$. We will assume that the key evolution is deterministic (i.e. it does not depend on any external randomness) hence these keys K_i will be shared by synchronized devices, which evolve their keys simultaneously but independently (without communication). As we will see, the design of key-evolution functions is also intimately related to the design of leakage-resilient stream-ciphers, and pseudo-random generators. The problem of designing such primitives been studied before, both by the practitioners and by the theoreticians. Next, we look at several models of leakage-resilience and their results. In this chapter we give explicite construction for SBA–model.[1]

[1] Slightly modified; details in Section 4.2.1

4.1 Leakage Resilient Key Evolution: Theory vs. Practice

Theoretical work on leakage-resilience usually included a formal model for reasoning about leakage. Usually, side-channel leakage is modeled as a family of functions where the attacker can choose a function from the family and learn the output of this function applied to the secret key. For example, a popular and powerful model of Akavia et al. [1], allows the adversary to compute arbitrary poly-time leakage functions of the internal secret key, subject only to the constraint that the amount of data retrieved (the output-length of the function) is bounded. Unfortunately, when it comes to key-evolution, it is clear that security *cannot* be achieved in this model, even if the adversary is restricted to leaking a *single bit*! In fact, just given the ability to leak on the initial key K_1, the adversarial leakage-function can *pre-compute* any future key K_i and output (say) the first bit of it. If the adversary can leak even 1 bit in several consecutive rounds, the adversary can eventually recover any future key K_i in full! This example (called a *key-precomputation attack* in [28]) shows that, when considering the security of the key-evolution schemes, the sole restriction that the output of the leakage function is bounded does not suffice. However, it is also easy to see that the leakage-functions used in the above counter-examples are extremely artificial, and are very unlikely to model natural side-channel attacks that occur in real life. Hence, it is natural to look for different (weaker) models for the leakage, that still cover all the *realistic* attacks, but in which key-evolution schemes may exist. We survey two such key-evolution schemes in their corresponding models of leakage. The two schemes come from two very different point views: one theoretical with an emphasis on models and proofs, and the other practical with an emphasis toward efficiency and simplicity. Therefore, it is interesting to compare their advantages and disadvantages.

4.2 Previous work

The scheme of Dziembowski and Pietrzak [28]. On the theoretical side, [28] constructed a stream cipher (and, implicitly, a key evolution scheme) in a formal model called "only computation leaks information", first proposed by Micali and Reyzin [49] and refined in [28]. In this model, the internal memory of the device is separated into two or more segments, and all computation is divided into simpler sub-computations that access only some small subset of these segments. The assumption is that, during each sub-computation, the adversary can leak an arbitrary bounded-length function of only the memory

segments accessed by the sub-computation. In other words, during any computational step, data can leak if and only if it is accessed. Pre-computation attacks can therefore be prevented, since the adversary can never get any global leakage of the entire state of the system needed to compute a future key.

The actual scheme of [28] (and a related scheme of [54]) uses an alternating structure with two memory segments accessed in alternating rounds. The main drawback of this model is that it relies on the highly controversial assumption that data which is not accessed cannot leak. Also, the security of solutions in this model is highly dependent on "implementation details" such what data is accessed when.

The scheme of Kocher [42]. On the practical side, Kocher [42] proposes a simple and efficient solution to just use a "sufficiently complicated" cryptographic hash function for the key-evolution function f. This scheme seems intuitively secure since it's unlikely that any natural and therefore "sufficiently simple" leakage could learn anything about $K_{i+1} = f(K_i)$ from K_i (or even from the entire key-evolution process used to derive K_i), if f is "sufficiently complicated". However, [42] does not offer any meaningful model in which to analyze the above intuition. The main idea is to assume that the adversarial leakage on the ith update $K_i \to K_{i+1}$ can be an *arbitrary function* of the entire state of that update subject to the constraints: (1) the leakage function cannot make any random-oracle calls, (2) the output-length of the leakage function is bounded to be just slightly smaller than the key-length $|K|$. At first, it may seem that constraint (1) offers a meaningful way of capturing "sufficiently simple" leakage. Unfortunately, it is not clear what this constraint means in practice, and can lead to counter-intuitive consequences, described next.

Since the amount of leakage tolerated by the scheme should be close to $|K|$, it is natural to try to increase $|K|$ if one would want to achieve more leakage. Of course, that means using a key-evolution function, and therefore hash function, with sufficiently large input-size and output-size. Assume we start with a compression function $H : \{0,1\}^{2\ell} \to \{0,1\}^{\ell}$, and we want to allow key-size $|K| = t\ell$ to allow for more leakage. The standard technique for domain-extension is the Merkle-Damgård transformation [15] (and variants of it in the indifferentiability framework) which gives a function $H' : \{0,1\}^* \to \{0,1\}^{\ell}$. Now, to increase the output-size, we can define $\tilde{H} : \{0,1\}^{t\ell} \to \{0,1\}^{t\ell}$ by $\tilde{H}(K) = H(H'(K)||1), \ldots, H(H'(K)||t)$. But, it is clear that, if one leaks the ℓ-bit value $H'(K_1)$ used as sub-component of the key-evolution computation $K_2 = \tilde{H}(K_1)$, then one can compute all future keys K_j and

thus completely break the key-evolution scheme! Therefore, the scheme is not secure with respect to even ℓ bits of leakage when instantiated with real-world hash functions. Notice that the leakage-function does not perform any complicated computation; it just leaks several consecutive bits of the internal state, corresponding to $H'(K)$, which is computed as an intermediate value during the key-evolution computation.

So we see that, although the initial scheme of Kocher provides some intuitive leakage-resilience properties, one runs into pitfalls when trying to model and quantify them. In particular, by relying on the random-oracle model in an unintended way (assuming that simple functions cannot make random oracle calls), and assuming that leakage on a computation making random-oracle calls only gets the input and output of such calls, one reaches a model that doesn't correspond to reality in a meaningful way.

The scheme of Yu Yu et al. [62] In a recent important work Yu Yu et al. [62] propose a practical scheme whose security is based on the assumptions that (1) the leakage functions cannot be chosen adaptively, and (2) the leakage function cannot evaluate the hash function (which is modeled as a random oracle). The security of the scheme that we construct in this work does not require these assumptions.

Other Models of Leakage-Resilience. We note that several other models of leakage resilience, with restricted leakage functions, have appeared in the literature. For example, [38] assumes leakage functions that leak individual wires from a circuit that performs a computation. Alternatively, Faust et al. [32] assume that leakage-function is an AC^0 circuit of the internal state. Several works consider computation that uses some small/simple leak-free components [32, 34, 36, 40].

4.2.1 Our Model: Space-Bounded Leakage in the Random Oracle Model

In this chapter we propose a method for the key evolution that combines the advantages of the Kocher's practical scheme (efficiency and simplicity of model, scheme) with some of the advantages of the theoretical scheme of [28] (provable security and scalability). On a high level, we restrict the class of leakage functions to ones that are bounded in the amount of "auxiliary work space" used during the computation of the output. In particular, this should model natural leakage which is unlikely to be sufficiently complex to require much space to compute. We will analyze a variant of the Kocher key-evolution

scheme in the random-oracle model, but give all parties (including the leakage functions) the ability to compute the random oracle. In particular, we define a deterministic key-evolution function f that makes random-oracle calls to compute $K_{i+1} = f(K_i)$. On a high level, pre-computation attacks will not be possible because the space allowed to the leakage function is not sufficient to pre-compute K_{i+1} from K_i.

In greater detail, we model the leakage process on the key evolution as follows. We consider an adversary $\mathcal{A} = (\mathcal{A}_{small}, \mathcal{A}_{big})$ consisting of two parts: the adversary $\mathcal{A}_{big}$ corresponds to the external real-world attacker that tries to break the scheme, and $\mathcal{A}_{small}$ corresponds to the "space-bounded" device which is storing and evolving the secret key while leaking partial information to the external attacker $\mathcal{A}_{big}$. We do not assume anything about how the computation is implemented on the device, and thus allow the computation $\mathcal{A}_{small}$ itself to be adversarial. Initially, $\mathcal{A}_{small}$ is given the random starting key K_1. Since key evolution is deterministic, this will completely specify all future keys $K_2, K_3, \ldots$. Of course, we need somehow to "force" $\mathcal{A}_{small}$ to perform the key evolution (if $\mathcal{A}_{small}$ can completely "halt" the key evolution, then he can simply keep K_1 on $\mathcal{M}$ for a long time, and slowly retrieve it bit-by-bit). In the passive case we could simply assume that, in time period i, the key K_i is fully stored on the machine and therefore there is not enough memory on the machine to store much information about any of the prior keys from earlier time periods. However, if $\mathcal{A}_{small}$ is active then this assumption could be completely unreasonable as the attacker could just keep K_1 on the machine for arbitrarily many time periods and leak it entirely. Therefore, we will introduce a special procedure that we call Verify_i, and assume that this procedure is called in each time period i to ensure that the device is storing the full key K_i at that point in time.

During the entire key-evolution process, $\mathcal{A}_{small}$ can communicate with $\mathcal{A}_{big}$ and can perform arbitrary computation (in addition to / instead of computing K_i honestly) *including* the ability to make random oracle calls. We only make three restrictions: (1) the amount of data that $\mathcal{A}_{small}$ can send to $\mathcal{A}_{big}$ *in each period* is bounded (2) the space-complexity of $\mathcal{A}_{small}$ is bounded and not much larger then the space-complexity of the honest computation of f, (3) the number of oracle-queries made by all parties is polynomial (no other computational assumptions are made). (4) At the end of round i, the machine $\mathcal{A}_{small}$ stores the correct key K_i. We will allow unlimited communication in the other direction $\mathcal{A}_{big}$ to $\mathcal{A}_{small}$.

Let us elaborate on the restrictions in more detail. Restriction (1) models the fact that natural leakage is too simple to reveal too much data about any single computational step. Restriction (2) models that fact that the *complexity* of natural leakage functions is rather simple. In particular, the space-

complexity of leaking on the internals of a computation should not be much larger then the amount of space actually used by the computation itself! This seems to be a rather conservative assumption. Lastly, restriction (3) models that all parties run in polynomial time, as is standard in cryptography, and restriction (4) models the fact that the device itself correctly computes the key K_i in round i.

Now that we have explained the model, let us give some intuition why key-evolution schemes are achievable in it. Firstly, note that, in round i, the adversary can (in principle) pre-compute any future key K_{i+t} in the space it is allotted. However, we will ensure that such computation would necessarily require the adversary to erase some data about K_i (since it cannot store all of K_i *and* compute K_{i+1} simultaneously with limited space) and hence it will be unable to satisfy the requirement that K_i is stored on the system at the end of round i.

4.2.2 Our Results

We construct a key-evolution function f that is secure in the model described above. Let c be the amount of bits that the adversary can retrieve in each round, and let s be the space that the adversary can use to compute the leakage function (including the $|K|$ bits needed to store the key K). We show that our scheme is secure as long as

$$4c + s \leqslant 3 \cdot |K|/2 \tag{4.1}$$

(cf. Theorem 3.10). Let us mention two applications of our construction.

Security against passive leakages Firstly, suppose that the evolving key K_i is stored on some device (say, a smart-card) that may leak some information. Imagine that the device is used for (message or entity) authentication with a trusted server that has his own copy of K_i. Suppose the adversary can get a temporary access to the device, observe the process of key evolution and learn some partial information about the keys. At some later point the adversary looses access to the device. The properties of our function f will guarantee that the future keys are unknown to the adversary assuming that the leakage is bounded in the way described above. Since in this case the adversary is only passive, there is no need to perform the procedure Verify_i. It may look like the model described above is stronger than what we need for this application, since it seems too pessimistic to assume that the adversary fully controls how the keys K_i are computed on the device. It may seem tempting to consider a weaker model, where the computation is done is

some honest way, and the adversary can apply the leakage functions during the evolution process. We believe that such a restriction would not make the proof simpler (while it would make the model more complicated). Moreover, going to the extreme and allowing the adversary to control the computation has the advantage that it protects us (to a certain extent) against implementation errors.

Security against active attacks in the BRM The second application of our construction concerns the bounded-retrieval model (BRM) [14, 23]. In this model one constructs schemes where the cryptographic key K is very large. The idea is that K can be stored on a PC that can be infected by viruses, and, as long as the virus does not retrieve a large portion of K, the scheme should remain secure. So far, all the work in the BRM considered only the passive attacks, where the virus was not allowed to modify the data on the machine. Now, consider the following problem: suppose we are using the BRM scheme for the session-key agreement [12, 23] (where a pair of users share a secret key K), and we want to evolve the secret key K stored on the machine, so that in total over a long period of time we can tolerate a leakage of more than $|K|$ bits from the machine. We show that f can be used as such a key-evolution function. The details of the model are as follows. Suppose we store the keys K_i on the machine $\mathcal{M}$, and assume that the size of the local memory on $\mathcal{M}$ is s, and the amount of bits that can be retrieved in each key-evolution round is c, and c and s are such that (4.1) holds. Suppose $\mathcal{M}$ wants to authenticate to a trusted server that has his own copy of K_i. If there is a virus on $\mathcal{M}$ then we can even allow him to modify the data stored on $\mathcal{M}$. The restrictions that we impose are as follows. First, we assume that any computation that the virus performs has to be done within $\mathcal{M}$'s memory (of size s). Second, we somehow need to guarantee that the verification procedure Verify_i can be performed. We do it by assuming that $\mathcal{M}$ is equipped with a small tamper-free component $\mathcal{D}$ that can periodically check if the contents of the memory is "correct". For example, $\mathcal{D}$ could store the values of some hash function of $K_1, K_2, \ldots,$, and the Verify_i procedure would just consist of hashing the contents of the memory where K_i is supposed to be stored, and comparing the result with $H(K_i)$.

4.2.3 Some implementation details

In this work we do not define formally the security of the concrete schemes (like the message authentication), since we are more interested in considering the key-evolution as an abstract procedure. Every key K_i will consist of $N-1$ blocks: $K_i = (K_i^0, \ldots, K_i^{N-1})$, each of the blocks being an output of a hash

function modeled as a random oracle. Hence, we can simply say that K_i is secret if none of its blocks has every been calculated by the random oracle (in our model calculating such a block will correspond to "labeling" a vertex in some graph).

Of course, the key evolution scheme would be useless, if we could not use the evolved key in some other application. In other words, after each round i, the key evolution function should output some key κ_i, and in the formal model this κ_i should be given to $\mathcal{A}_{big}$ "for free". In our case we simply assume that κ_i is equal to one of the blocks of K_i. Note, that it does not require any modification of the model, since we can as well assume that κ_i is sent to $\mathcal{A}_{big}$ by $\mathcal{A}_{small}$.

The Verify_i procedure (that verifies the knowledge of K_i) can be implemented as follows: (1) the verifier (that knows K_i) sends a random value c to the device, and (2) the device replies with $v = \mathsf{MAC}(c, K_i)$ (where MAC is a tagging function of some message authentication code scheme, c is treated as a key for the MAC, and K_i is treated as a message), (3) the verifier checks if $v = \mathsf{MAC}(c, K_i)$, and if not then he aborts. Note, that we cannot hope for more than only verifying the correctness, since in the worst case an active adversary can anyway completely destroy the contents of the device. In our model we will not assume anything about how $\mathsf{MAC}(c, K_i)$ is computed by $\mathcal{A}_{small}$. For example, it will be possible that he partially "pre-computes" it before learning c. The only property of MAC that we use is that $\mathcal{A}_{small}$ can compute the value of $\mathsf{MAC}(c, K_i)$ only if each of the blocks of K_i were computed by him at some point earlier.

4.2.4 Organization

Our key-evolution function is defined using a special type of a graph, that we call a *tower graph*. The method of translating graphs into functions is described in Section 4.4.2. The tower-graphs and the function f are defined in Section 4.3. The main theorem is stated in Section 4.4, which also introduces most of the tools needed for the proof. The proof itself appears in Section 4.4.6.

4.2.5 Related Work

The theoretical countermeasures against the side-channel attacks were considered in [1, 8, 9, 16–18, 28, 31, 41, 50, 53, 58]. The schemes in the Bounded Retrieval Model were constructed in [3, 12, 14, 22, 23, 27]. We will use the technique called "graph pebbling" (cf. e.g. [56]), that was already used in cryptography in [21]. Some of our techniques (esp. those used in Sections

4.4.1 and 4.4.5) were introduced in [21] and recently extended in [25]. We note that, although our techniques are quite similar, the application is completely different (the main application of [25] is a construction of a scheme for the password-protected local storage). Unfortunately, it is impossible to use the theorems from [21, 25] in a black-box way, and therefore we needed to non-trivially extend them. On a technical level, the main difference comes from the fact that in [25] the total amount of leakage was bounded globally, and in our work we need to consider continual leakage over an unbounded number of round.

4.3 Our Key-Evolution Scheme

In this section we define our key-evolution function f. We start with defining a special type of graphs, that we call the "tower graphs". A graph $G = (V, E)$ is called an (N, M)-*tower graph* if $V = \{0, \ldots, M-1\} \times \{0, \ldots, N-1\}$ and $E = \{((i,j),(i+1,j) : i \in \{0,1,\ldots,\}, j \in \{0,\ldots,N-1\}\} \cup \{((i,j),(i+1,(j-1) \bmod N) : i \in \{0,1,\ldots\}, j \in \{0,\ldots,N-1\}\}$ (cf. Fig. 4.1 in the appendix). For $i = 0, \ldots, t$ the set $V_i = \{(i,0), \ldots, (i, N-1)\}$ is called the *ith line of* G. Let $V_{\geqslant i}$ denote the set $V_i \cup V_{i+1} \cup \cdots$. Note that the set of the input vertices of G is equal to V_0. We will say that an (infinite) graph G is an N-*tower graph* if it is an (N, ∞)-tower graph.

We are now ready to define f. If we fix a hash function $\mathcal{H}$ and label length w then the (N, M)-tower graph G defines a function $f : \{0,1\}^{Nw} \to \{0,1\}^{Nw}$ in the following way. On an input K the function f computes the $(\mathcal{H}, K)$-labeling of G and it outputs $(K'_1, \ldots, K'_N)$, where each K_i is the label of $(M-1, i)$. The procedure for computing $f(K_0, \ldots, K_{N-1})$ simply computes the labels bottom-up row-by-row in the following way:

- Set $(K^0_0, \ldots, K^0_{N-1}) := (K_0, \ldots, K_{N-1})$.
- For $j = 1, \ldots, M-1$ do
  - For $i = 0, \ldots, N-1$ do $K^j_i := \mathcal{H}(K^{j-1}_i, K^{j-1}_{i+1 \bmod N}, (i,j))$

Observe that the time needed to compute f is roughly equal to $N \cdot M$ times the time needed to compute $\mathcal{H}$, and the space needed to compute f is only slightly larger than the space needed to store K, since we can overwrite each $(K^j_0, \ldots, K^j_{N-1})$ with $(K^{j+1}_0, \ldots, K^{j+1}_{N-1})$ and hence re-use the space.

It is also easy to see that iterating the computation of f on the same input a times, i.e. computing $K' = f^a(K)$ can be seen as computing the labeling of an (N, aM)-tower graph, and in particular, if we want to evolve

the key K^0 using the procedure $K^{i+1} = f(K^i)$ (for $i = 0, 1, \ldots$) then we can look at it as a labeling the tower infinite N-tower graph, where the keys $K^1, K^2, \ldots$ appear as labels of the lines $V_{1 \cdot M}, V_{2 \cdot M}, \ldots$. We will call such lines the *round-switching lines*.

4.4 Games on Tower Graphs

We will show a connection between an adversary computing a "random oracle graph" and a pebbling game for the corresponding graph. A similar connection appears in [21] (and in [25], see Section 4.2.5 for more on relation between this work and [25]).

4.4.1 Model of Computation

Our main goal is to show that computing the labeling of a tower graph G requires a large amount of resources in the random-oracle model, and is therefore difficult. To do so, we must fix a model of computation in which we can make statements of the above form precise. Recall that we will usually consider an adversary that consists of two parts: a "space-bounded" component which gets access to the internals of an attacked device and has "bounded communication" to an external, and otherwise unrestricted, adversary.

We model such a adversary $\mathcal{A} = (\mathcal{A}_{big}, \mathcal{A}_{small})$ as a pair of interactive algorithms[2] with oracle-access to a random-oracle $\mathcal{H}(\cdot)$. Let M be some natural number that we will call the *round length*. While executing the algorithms the time is divided into rounds. Initially the computation is in a round 1. The adversary $\mathcal{A}_{small}$ is responsible for switching to next round. Namely: the round is changed to k when $\mathcal{A}_{small}$ calls special function $\texttt{nextRound}_k(\texttt{label}(a_1) \ldots \texttt{label}(a_n))$, where $\{a_1, \ldots, a_n\}$ is the kth *round-switching line* $V_{k \cdot M}$. A round k can be switched only to round $k+1$, in other words the order of the round-changing calls has to be $\texttt{nextRound}_1, \texttt{nextRound}_2, \ldots$. The period between the calls $\texttt{nextRound}_i$ and $\texttt{nextRound}_{i+1}$ will be called *the ith round*. 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.[3]

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

[3] This is somewhat different than standard space-complexity considered in complexity

- c-bounded communication: The total number of outgoing bits communicated by $\mathcal{A}_{small}$ in each round is bounded by c. [4]

Note that these restrictions imply that he total number of outgoing bits communicated by $\mathcal{A}_{small}$ in every round is bounded by c and there is no global bound for communication. We use the notation $\mathcal{A}^{\mathcal{H}(\cdot)}(K) = \left(\mathcal{A}_{big}^{\mathcal{H}(\cdot)}() \leftrightarrows \mathcal{A}_{small}^{\mathcal{H}(\cdot)}(K)\right)$ to denote the interactive execution of $\mathcal{A}_{big}$ and $\mathcal{A}_{small}$, where $\mathcal{A}_{small}$ gets input K and both machines have access to the oracle $\mathcal{H}(\cdot)$. In particular, 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 $\mathcal{A}^{H(\cdot)}(K)$ *labels* a vertex v, if a random-oracle call to $\texttt{preLabel}(v)$ is made by either $\mathcal{A}_{big}$ or $\mathcal{A}_{small}$. We are now ready to state our main theorem.

Theorem 4.1. *Let G be a N-tower graph and $\lambda > 0$. Suppose c, s and q are such that $\frac{4c+s+\lambda}{w-\log(q)} \leqslant N + N/2$, and let T be an arbitrary natural number. Let $\mathcal{A} = (\mathcal{A}_{big}, \mathcal{A}_{small})$ be an adversary with c-bounded communication and s-bounded storage that makes at most q queries to H. The probability p (taken over the choice of $(\mathcal{H}, K)$) that there exists $i = 1, \ldots, T-1$ such that $\mathcal{A}$ labels the line $V_{(i+1)\cdot M}$ of G in round i is at most*

$$q \cdot 2^{-w} + T \cdot 2^{1-\lambda} \tag{4.2}$$

The proof appears in Section 4.4.6. The necessary machinery is introduced in the next sections.

4.4.2 Pebbling Games on Tower Graphs

We will consider a variant of the pebble game that we call the "red-black" pebble game over an N-tower graph $G = (V, E)$. Each vertex of the graph G can either be empty, contain a red pebble, contain a black pebble, or contain both types of pebbles. More precisely, if G is a tower graph, then a *pebbling configuration on G* is a function $\gamma : V \to \mathcal{P}(\{\texttt{red}, \texttt{black}\})$. Define $Red(\gamma) := \{v : \texttt{red} \in \gamma(v)\}$, and $Black(\gamma) := \{v : \texttt{black} \in \gamma(v)\}$. If $V' \subseteq V$ then define $proj(V') := (|V' \cap V_1|, \ldots, |V' \cap V_t|)$.

For a set $V' \subseteq V$ denote by $[V']$ the *closure of V* defined recursively as follows:

---

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.

[4] 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...

- if $v \in V'$ then $v \in [V']$,
- if all the children of v' are in $[V']$ then $v' \in [V']$.

An initial configuration γ_1 consists of (only) a black pebble placed on each input vertex of G. The game proceeds in steps where, in the ith step, the configuration γ_i is transformed into γ_{i+1} using one of the following four actions:

1. A red pebble can be placed on any vertex already containing a black pebble.
2. If both children of a vertex v have a red pebble on them, a red pebble can be placed on v.
3. If both children 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.

A *pebbling game* is a sequence $\gamma_1 \to \gamma_2 \to \cdots \to_\ell$ of configurations. The game is — similarly to real computational model — divided into rounds. One starts a game in round 1. A round may be switched to u in a configuration γ_i if all vertices from a line $V_{u \cdot M}$ in γ_i are pebbled by some pebble (technically, a round is switched to u by issuing a request $\texttt{nextRound}_u$). This switch is not obligatory when the specific row is pebbled. However, we require that the order of the request is $\texttt{nextRound}_0, \texttt{nextRound}_1, \ldots,$.

We define the *black-pebble complexity* of a round k of a pebbling game to be the maximum number of *black pebbles* on vertices in $V_{\geqslant k \cdot M}$ in use at any time of round k. More precisely if $\gamma_i \to \cdots \to \gamma_j$ are the configurations in round k. Then the black pebble complexity of k is equal to $\max_{\ell=i}^{j} |Black(\gamma_\ell) \cap V_{\geqslant k \cdot M}|$. If $\gamma_i, \ldots, \gamma_j$ are as above then the red pebble complexity of k is equal to the number of times in round k in which Step 1 was applied. For a parameter X a pebbling game is *X-bounded* if for every round k we have $2R_k + B_k < X$, where R_k and B_k denote the red- and the black-pebble complexities (resp.) of round k.

4.4.3 Auxiliary lemmata

We need some auxiliary definitions and lemmas. For $(a_0, \ldots, a_t) \in \{0, \ldots, N\}^{t+1}$ define the *optimistic width of* $(a_0, \ldots, a_t)$ as: $OptWidth(a_0, \ldots, a_t) := (b_0, \ldots, b_t)$, where

- $b_0 := a_0$, and

- for every $i = 1, \ldots, t$ we set

$$b_i := \begin{cases} N & \text{if } b_{i-1} = N \\ \min(N, b_{i-1} - 1 + a_i) & \text{otherwise} \end{cases}$$

Intuitively, the idea is that if $(b_1, \ldots, b_t) := \mathit{OptWidth}(a_0, \ldots, a_t)$ then b_i's give an upper bound on the number of pebbles in the ith line of $[V']$ (for any $V' \subseteq V$), assuming that a_i is the number of pebbles in the ith line of V'. Formally, this is shown in the following lemma.

Lemma 4.2. *Take any set $V' \subseteq V$ and let $(a_0, \ldots, a_t) := \mathit{proj}([V'])$ and $(b_0, \ldots, b_t) := \mathit{OptWidth}(\mathit{proj}(V'))$. For every i we have that $a_i \leqslant b_i$.*

Proof. The proof goes by induction on $i = 0, \ldots, t$. Case $i = 0$ follows immediately from the fact that the closure operation does not change the configuration of the pebbles on the bottom row (V_0), and hence $a_0 = b_0$.

Now suppose the lemma holds for some i. The set of pebbles in the $(i+1)$st line of $[V']$ is equal to the sum of V'_{i+1} and the pebbles P that were derived (using the closure operation) from the pebbles in the ith line of $[V']$. By the induction hypothesis we get that the number of pebbles in the ith line of $[V']$ is at most b_{i-1}. Now, consider the case when $b_{i-1} \neq N$. From the definition of the closure operation it follows that $|P| \leqslant b_{i-1} - 1$. Therefore $|V'_{i+1} \cup P| \leqslant |V'_{i+1}| + |P| \leqslant a_i + b_{i-1} - 1$. Since the maximal value of $a_i + b_{i-1} - 1$ cannot be greater than N we get $|V'_{i+1} \cup P| \leqslant \min(N, a_i + b_{i-1} - 1)$.

The second case ($b_{i-1} = N$) follows easily from the fact that in this case $b_i = N$. □

A sequence $(a_0, \ldots, a_t)$ is called *wide* if for some i we have $a_i = N$. A set $V' \subseteq V$ will be called *wide* if $\mathit{proj}([V'])$ is wide. We have the following simple observation.

Lemma 4.3. *For $U, W \subseteq V$ such that $U \cup W$ is not wide define $(a_0, \ldots, a_t) := \mathit{OptWidth}(\mathit{proj}([U \cup W]))$ and $(b_0, \ldots, b_t) := \mathit{OptWidth}(\mathit{proj}([W]))$. Then, for every i we have $b_i \leqslant a_i - |W|$. In other words: adding $|W|$ elements to U cannot increase the values on the coordinates in $\mathit{OptWidth}(\mathit{proj}([U]))$ by more than $|W|$ (as long as the resulting set $U \cup W$ is not wide).*

Proof. Suppose we add the elements of W to U one-by-one. From the definition of the closure operation it easily follows that adding one element cannot increase $\mathit{OptWidth}(\mathit{proj}([W]))$ by more than on each coordinate (as long as the resulting set is not wide). Hence the statement of the lemma follows. □

A subgraph G' of a tower graph is a *pyramid graph.* if it is induced by the set of vertices: $\{(i + x \bmod N - 1, j + y \bmod N) : 0 \leqslant x + y \leqslant N - 1\}$ for some i and j. The vertex $(i + N, j)$ will be call the *root of* G'. We now have the following lemma:

Lemma 4.4. *Consider a pebbling game for initially empty pyramid graph. If the root vertex is pebbled at the end of the game then there exist a configuration γ of the considered game with sum of red pebbles in the first row and the black pebbles is at least N.*

Proof. The argument is similar to the one appearing in the proof of Lemma 10.2.1 in [56]. We consider all paths from the root to the bottom row. We say that a path carries a pebble if at least one vertex of the path has some pebble on it. If a path does not carry a pebble we say it is empty.

Initially all paths are empty. At the end of strategy, all paths must carry a pebble (because there is a pebble in the root, and root is a vertex in every path). Putting a pebble on a pyramid according to the pebbling rules can increase number of paths carrying a pebble only by one (this happens obviously only when the pebble is put on the first row). Therefore, there must exist the first configuration in time when all paths are carrying a pebble. This must happen when a pebble p is put on the first row of some path and the path is empty besides this one particular pebble at the bottom (on Figure 4.2 this path is indicated with double arrows). All other paths are carrying a pebble. Let us look at the paths starting from other vertices from first row and connected to the last pebbled path (see figure: wave arrows). Each of this paths must carry a pebble. There are N such paths so we have at least N pebbles on graph (at least one on every highlighted fragment of path). Moreover: there are at least N black and red pebbles in the first row: When a red pebble is on graph then both children are also on graph (they are red and we do not remove the red pebbles), so when choosing representatives for the paths we can always choose pebble from first row if only red pebbles remained. This finishes the proof. □

4.4.4 The impossibility of pebbling

Our goal is to show that — with some restrictions on red and black pebble complexity — it is impossible to pebble any vertex in $V_{\geqslant (u+2)\cdot M}$ in round u. Intuitively, it means that we cannot get any information about pebbles from any line $V_{\geqslant (u+2)\cdot M}$, before switching to round $u + 1$. More precisely, the following theorem holds:

Theorem 4.5. *Let N, T and X be arbitrary natural numbers such that*

$$X < \frac{3N}{2}.$$

Set $M := \frac{3N}{2}$. Suppose G is an N-tower graph. Then, for any X-bounded pebbling game for G with round length M and any configuration γ that belongs to the uth round, we have that in γ there are no pebbles on $V_{\geqslant(u+2)\cdot M}$.

Proof. In an execution of a pebbling game a pebble will be called *heavy* if it is a black pebble, or a red pebble placed on the graph using rule 1 (cf. Page 54). For a round u let γ_{i_u} be the last configuration of this round. We claim that in γ_{i_u} there is no pebble on $V_{\geqslant(u+2)\cdot M}$. Let A_u be the set of all pebbled vertices in the configuration γ_{i_u} and let Q_u be the set of all pebbles in A_u except of the black pebbles lying on the line $u \cdot M$. More precisely: $Q_u = A_u \setminus (A_u \cap V_{u\cdot M} \cap \mathit{Black}(\gamma))$. Set $Y_u := V_{\geqslant M\cdot u} \setminus V_{\geqslant M\cdot(u+1)}$

Lemma 4.6. *For every u we have:*

1. $A_u \cap V_{\geqslant(u+2)\cdot M} = \emptyset$
2. $\mathit{OptWidth}(\mathit{proj}([Q_u])) = (a_0, \ldots, a_{T\cdot M}) < (\underbrace{N, \ldots, N}_{u\cdot M}, \underbrace{N/2, \ldots, N/2}_{M}, 1, \ldots, 1)$, *in particular $[Q_u]$ is not wide.*

After showing this we will be done with the proof since Point 1 of Lemma 4.6 clearly implies that Theorem 4.5 holds.

Proof of Lemma 4.6. Induction on $u = 1, 2, \ldots$. The base of the induction holds trivially since in the initial configuration only the bottom line is pebbled. Let us now assume the statement holds for some u. Now we prove the following claims for next round $u + 1$:

Claim 4.7. *During the entire round $u + 1$ there must be at least $N/2$ heavy pebbles in the subgraph Y_u (i.e. the lines $u \cdot M, \ldots, (u + 1) \cdot M - 1$).*

Proof. Let us consider any configuration γ from this round and denote the set of heavy pebbles in X in γ by P. At the end of the round every vertex on the $(u + 1)$st round-switching line $V_{(u+1)\cdot M}$ will contain a pebble. Therefore the closure $[P]$ of heavy pebbles P from the current configuration and the pebbles from the previous rounds Q_u need to contain whole line $V_{(u+1)\cdot M}$. This is because otherwise one would never be able to pebble $V_{(u+1)\cdot M}$ in the future (this follows easily from the definition of closure and the pebbling game). Hence $[P \cup Q_u]$ has to be wide and therefore (from Lemma 4.2) $\mathit{OptWidth}(P \cup Q_u$ also has to be wide. On the other hand, by the induction

hypothesis we know that every coordinate of $OptWidth(Q_u)$ on positions $u \cdot M, \ldots$ is smaller than $N/2$. Now, by Lemma 4.3 adding to Q_u a set of cardinality $|P|$ cannot increase any of this coordinates by more than $|P|$. Hence $|P| \geqslant N/2$. □

Claim 4.8. *Through the whole round $u+1$ no vertex in $V_{\geqslant (u+2)\cdot M}$ is pebbled.*

Proof. For the sake of contradiction assume that the claim is not true. So, we have a configuration γ from round $u+1$ with a pebble on some vertex v from set $V_{\geqslant (u+2)\cdot M}$. Denote by $(V', E')'$ the subgraph forming the pyramid graph with root in vertex v. From Lemma 4.4 we have that before γ there was a configuration γ' that: had b black pebbles on V' and had r red pebbles in bottom line of V' (which is the line $V_{\geqslant (u+2)\cdot M - N + 1}$ of the tower graph) and $b + s \geqslant N$. However from Claim 4.7 there are at least $N/2$ heavy pebbles on Y_u, and Y_u is disjoint with the pyramid (V', E'). The number of all heavy pebbles is $A := B + R < (N - R) + (N/2)$. Therefore in the set $V_{\geqslant M\cdot(u+1)} \supset V'$ there are at most $(N - R)$ heavy pebbles. Since $b + s$ is bounded by the number of heavy pebbles so we have a contradiction with the fact that $b + s \geqslant N$. □

Claim 4.9. $OptWidth(proj([Q_{u+1}])) = (a_0, \ldots, a_{T\cdot M})$
$< (\underbrace{N, \ldots, N}_{u+1\cdot M}, N/2, \ldots, N/2)$

Proof. Denote the configuration at the end of this round by γ and the set of heavy pebbles in γ by P. Let P' denote P without black-pebbled vertices from $(u+1)$th finishing line. From definition, we have $[Q_{u+1}] = [Q_u \cup P']$. In γ the $(u+1)$th finishing line is pebbled. There at most R red pebbles, so at least $N - R$ black pebbles are on this line. So $|P'|$ is at most $A - (N - R) = B + 2R - N < N/2$. Similarly as at the end of proof of the Claim 4.7 adding to Q_u a set of cardinality $|P'| < N/2$ cannot increase any coordinate of $OptWidth$ by more then $|P'|$. This finishes the proof. □

Claims 4.8 and 4.9 prove inductive hypothesis for $u + 1$. Hence we are done. □

□

4.4.5 Connection Between Random-Oracle Labeling and the Pebbling Game

We now connect the random-oracle labeling of a tower graph G in our model of computation to the red-black pebbling game on G. The idea is to show

that from any execution of $\mathcal{A}$ (with space bounded by some s, and communication bounded by some c) we can construct some pebbling game for pebble game described before. Then, we show that (with high probability) this game respects the rules of the game and is (B, R)-bounded (for some B, R that will depend on c and s). We will then combine it with Theorem 4.5 to conclude that some specific oracle calls are impossible.

The main fact about the connection is that — in every round — 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 $\mathcal{A}_{small}$.

First, let us strictly define the method of translating an execution of $\mathcal{A}$ into a pebbling game. Let $\mathcal{H} : \{0,1\}^* \to \{0,1\}^w$ be a random oracle, and let $K = (K_1, \ldots, K_N)$ 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 $\mathcal{A}^{\mathcal{H}(\cdot)}(K) = \left(\mathcal{A}_{big}^{\mathcal{H}(\cdot)} \leftrightarrows \mathcal{A}_{small}^{\mathcal{H}(\cdot)}(K)\right)$ 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 $\mathcal{A}_{big}$ and all `nextRound` calls made by $\mathcal{A}_{small}$.

We fix some terminology about the transcript. Given $(\mathcal{H}, K)$, we say that an oracle call of the form $\mathcal{H}(\texttt{label}_1, \texttt{label}_2, v)$ is *correct* if $(\texttt{label}_1, \texttt{label}_2, v) = \texttt{preLabel}(v)$. We call the children v_1, v_2 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 $\mathcal{H}, K$) we define the *ex-post-facto* pebbling of the graph G. We do so by processing the random-oracle calls and `nextRound` calls in the transcript one-by-one starting with the earliest one, and, for each call, we take the following steps:

Change round: If $\mathcal{A}_{small}$ calls `nextRound`, change round in the pebble game.

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. If the call is taken in kth round and $v \in V_{\geqslant k \cdot M}$ then we say that v is *k-red-necessary.*

Go through all red-necessary vertices v one-by-one and, for each one check if that has a black pebble,but no red pebble. If so, put red pebble on v.[5]

[5]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 $\mathcal{A}_{big}$

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.[6]

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 $\mathcal{A}^{\mathcal{H}(\cdot)}(K)$ gets a (red or black) pebble placed on it in the corresponding ex-post-facto pebbling (although, of course, this pebble may have been removed at some later point). Moreover, the order in which vertices get red/black pebbles corresponds to the order in which the oracle calls are made by $\mathcal{A}$.

As mentioned before, 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.

Theorem 4.10. *Let G be an N-tower graph. Let $\mathcal{A} = (\mathcal{A}_{big}, \mathcal{A}_{small})$ be any adversarial labeling game in our restricted model of computation. Let $(\mathcal{H}, K)$ 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 the execution. Then, the ex-post-facto pebbling of G corresponding to an execution of $\mathcal{A}^{\mathcal{H}(\cdot)}(K)$ has the following properties (for any k):*

---

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.

[6] 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.

1. *It is a legal pebbling (i.e. follows the rules of the red-black pebbling game and changes round only when appropriate condition is hold) with probability* $1 - \frac{q}{2^w}$ *over the choice of* $(\mathcal{H}, K)$.

2. *Assuming that* $\mathcal{A}_{small}$ *has c-bounded communication and that in rounds* $0, \ldots, k-2$ *no vertices from* $V_{\geqslant k \cdot M}$ *were pebbled in the* ex-post-facto *game then, for any* $\lambda \geqslant 0$ *the red-pebble complexity of the round* k *is at most* $\frac{2c+\lambda}{w-\log(q)}$ *with probability* $1 - 2^{-\lambda}$ *over the choice of* $(\mathcal{H}, K)$.

3. *Assuming that* $\mathcal{A}_{small}$ *has s-bounded storage and c-bounded communication and that in rounds* $0, \ldots, k-2$ *no vertices from* $V_{\geqslant k \cdot M}$ *were pebbled then, for any* $\lambda > 0$*, the* sum *of the red-pebble complexity* and *the black-pebble complexity of the round* k *is at most* $\frac{2c+s+\lambda}{w-\log(q)}$ *with probability* $1 - 2^{-\lambda}$ *over the choice of* $(\mathcal{H}, K)$.

Proof of Theorem 4.10. 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 or `nextRound` 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 or `nextRound` 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 $B = (K, \mathcal{H})$ which:

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

Runs: Runs $\mathcal{A}^{\mathcal{H}(\cdot)}(K)$. Answers all queries of $\mathcal{A}$ honestly (using access to $\mathcal{H}, K$) until the ith query made by $\mathcal{A}$, which is of the form $\mathcal{H}(\texttt{label}(a_1), \texttt{label}(a_2), v)$, or $\texttt{nextRound}(a_1, \ldots, a_N)$. By assumption, for at least one of the arguments a_i, the oracle was never queried at the point $\texttt{preLabel}(a_i)$. Moreover, it is easy to figure out i, by computing $\texttt{preLabel}(a_j)$ for each argument, without querying the oracle on input $\texttt{preLabel}(a_i)$.

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

But, by Lemma 2.1, 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 k and some $\lambda \geqslant 0$ such that (a) in rounds $0, \ldots, k-2$ no vertices from $V_{\geqslant k \cdot M}$ were pebbled, and (b) the red-pebble complexity of round k of the ex-post-facto pebbling is $r \geqslant \frac{c+\lambda}{w-\log(q)}$ with probability (strictly) greater than $2^{-\lambda}$. The only way that the red-pebble complexity of round k could be r is if there are r distinct k-red-necessary vertices v. Recall that a vertex is k-red-necessary if the transcript includes correct oracle call in round k 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 k-red-necessary, and there are $r' \leqslant r$ of them (one oracle-call can make many of its input-vertices k-red-necessary). The intuition is that the algorithm $\mathcal{A}_{big}$ must then somehow predict the labels of these k-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 $B = (K, \mathcal{H})$, which works as follows:

Gets as hint: The value $h_{com} \in \{0,1\}^{2c}$ of all communication from $\mathcal{A}_{small}$ to $\mathcal{A}_{big}$ made during the execution $\mathcal{A}^{H(\cdot)}(K)$ of rounds $k-1$ and k. The indices $(i_1, \ldots, i_{r'}) \subseteq \{1, \ldots, q\}^{r'}$ of the r' k-red-necessary oracle-calls made by $\mathcal{A}_{big}^{\mathcal{H}(\cdot)}$ during the execution.

Runs: Runs $\mathcal{A}$ for rounds 1 to $k-2$, answering all queries honestly (using access to $\mathcal{H}, K$). Then runs only $\mathcal{A}_{big}^{\mathcal{H}(\cdot)}$ and feeds it the correct communication on behalf of $\mathcal{A}_{small}$ (without running $\mathcal{A}_{small}$) 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 k-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 children 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 $\mathcal{H}$ 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 k-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 $\mathcal{H}$ with these as inputs.

Outputs: The bits of $\mathcal{H}$ 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, $\mathcal{H}$ is never queried on the inputs $\texttt{preLabel}(v_i)$ for the k-red-necessary vertices v_i. (By assumption (a) we have that in rounds $1 \ldots k-2$ it is impossible to pebble any vertex from $V_{\geqslant k \cdot M}$.) Therefore, by Lemma 2.1, the probability of the above succeeding is at most $\frac{q^r 2^{2c}}{2^{rw}} \leqslant 2^{-(r(w-\log(q))-2c)} \leqslant 2^{-\lambda}$, leading to a contradiction.

Lastly, let us turn to part 3 of the theorem. Again, assume otherwise, that there is some $\lambda \geqslant 0$ for which the sum of the red-pebble and black-pebble complexities of the ex-post-facto pebbling of round k is $z \geqslant \frac{c+s+\lambda}{w-\log(q)}$ with probability (strictly) greater than $2^{-\lambda}$. The only way that this could happen is if r of the vertices are k-red-necessary and if at *at some point* there are there are b vertices that are k-black-necessary (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 $\mathcal{A}_{big}$ in rounds $k-1$ and k:

Gets as hint: The values $h_{com} \in \{0,1\}^{2c}, h_{state} \in \{0,1\}^{s}$.
The indices $(i_1, \ldots, i_{z'}) \subseteq \{1, \ldots, q\}^{z'}$ of the $z' \leqslant z$ distinct oracle-calls made by $\mathcal{A}_{big}^{\mathcal{H}(\cdot)}$ and $\mathcal{A}_{small}^{\mathcal{H}(\cdot)}$ which make some vertex k-red-necessary or k-black-necessary.

Runs: First run $\mathcal{A}$ for rounds $1 \ldots k-2$, then run $\mathcal{A}_{big}^{\mathcal{H}(\cdot)}$ 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 game 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 k-red-necessary and k-black-necessary vertices $v_1, \ldots, v_z$, and can compute $\texttt{preLabel}(v_i)$ as before.

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

By Lemma 2.1, the probability of the above succeeding is at most $\frac{(q)^z 2^{2c} 2^s}{2^{zw}} \leqslant 2^{-(z(w-\log(qt))-2c-s)} \leqslant 2^{-\lambda}$, leading to a contradiction.

□

4.4.6 Proof of Theorem 3.10

Consider an execution of $\mathcal{A}$ and a corresponding ex-post-facto pebbling game $\mathcal{G}$. Let $\mathcal{X}$ denote an event that $\mathcal{G}$ is legal. For $i = 1, \ldots, T$ let B_i and R_i denote the respective black- and red-pebble complexities of round i in $\mathcal{G}$. Moreover, let $\mathcal{Y}_i$ denote the event that in the round i we have that (1) $B_i + 2R_i < N + N/2$, and (2) no vertex in $V_{(i+2)\cdot M}$ has been pebbled. Recall that every vertex that is labeled gets also pebbled. Therefore we have

$$\begin{aligned} 1 - p &\geqslant P(\mathcal{Y}_1 \wedge \cdots \wedge \mathcal{Y}_{T-2}) \\ &\geqslant P(\mathcal{X} \wedge \mathcal{Y}_1 \wedge \cdots \wedge \mathcal{Y}_{T-2}) \\ &= P(\mathcal{X}) \cdot P(\mathcal{Y}_1|\mathcal{X}) \cdot P(\mathcal{Y}_2|\mathcal{Y}_1 \wedge \mathcal{X}) \cdot \cdots \wedge P(\mathcal{Y}_{T-2}|\mathcal{Y}_1 \wedge \cdots \wedge \mathcal{Y}_{T-3} \end{aligned} \quad (4.23)$$

Let us look at a term $P(\mathcal{Y}_i|\mathcal{Y}_1 \wedge \cdots \wedge \mathcal{Y}_{i-1} \wedge \mathcal{X})$. Suppose $\mathcal{Y}_1 \wedge \cdots \wedge \mathcal{Y}_{i-1} \wedge \mathcal{X}$ occurred. The events $\mathcal{Y}_1, \ldots \mathcal{Y}_{i-2}$ together imply that until round $i-2$ no pebble has been placed on any vertex in $V_{\geqslant i \cdot M}$. Hence:

- $R_i \leqslant \frac{2c+\lambda}{w-\log(q)}$ with probability at least $1 - 2^{-\lambda}$ (from Part 2 of Theorem 4.10), and
- $B_i + R_i \leqslant \frac{2c+s+\lambda}{w-\log(q)}$ with probability at least $1 - 2^{-\lambda}$ (from Part 3 of Theorem 4.10).

Therefore we get that $B_i + 2R_i \leqslant \frac{4c+s+\lambda}{w \log(q)} \leqslant N + N/2$ with probability at least $1 - 2^{1-\lambda}$. Since the events $\mathcal{Y}_1, \ldots \mathcal{Y}_{i-1}$ also imply that $B_j + 2R_j \leqslant N + N/2$ holds for every $j < i$, therefore we can apply Theorem 4.5 and get that with probability $1 - 2^{1-\lambda}$ no pebble is put on any vertex in $V_{\geqslant (i+1)\cdot M}$ in round i. Since we also know that the pebbling is legal (because we assumed that $\mathcal{X}$ occurred), a vertex can be labeled by $\mathcal{A}$ only if it is pebbled. Hence $\mathcal{Y}_i$ holds (with probability at least $1 - 2^{1-\lambda}$). From Part 1 of Theorem 4.10 we have that $P(\mathcal{X}) \geqslant 1 - \frac{q}{2^w}$. Putting things together we get $P(\mathcal{Y}_i|\mathcal{Y}_1 \wedge \cdots \wedge \mathcal{Y}_{i-1} \wedge \mathcal{X}) \geqslant$

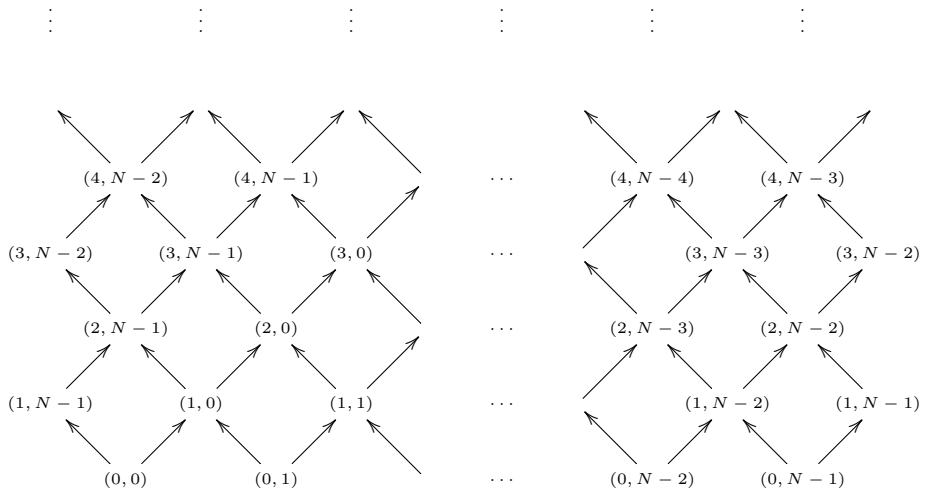


Figure 4.1: An N-tower graph. Note that the vertices $(1, N-1), (3, N-1)$ are duplicated on this picture. An (N, M)-tower graph is a subgraph of the N-tower graph induced by its bottom M lines.

$1 - 2^{1-\lambda}$. Hence (4.3) is at least equal to

$$\begin{aligned} & \left(1 - \frac{q}{2^w}\right) \cdot \left(1 - 2^{1-\lambda}\right)^{T-2} \\ \geqslant \; & \left(1 - q \cdot 2^{-w}\right) \cdot \left(1 - T \cdot 2^{1-\lambda}\right) \\ \geqslant \; & 1 - q \cdot 2^{-w} - T \cdot 2^{1-\lambda} \end{aligned}$$

Therefore p is at most (4.2).

4.5 Figures

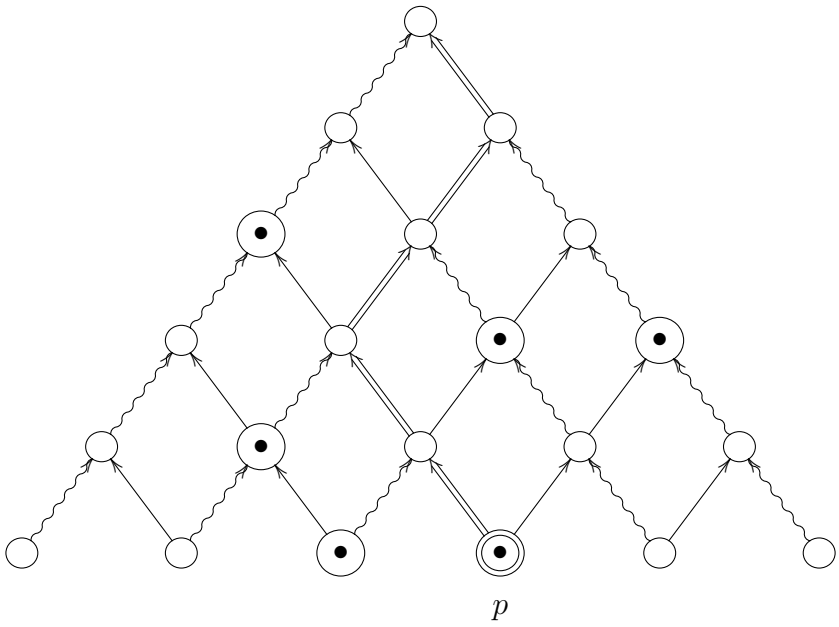


Figure 4.2: Pyramid graph with N vertices at the bottom

Chapter 5

One time programs

In this chapter we reinvestigate a notion of *one-time programs* introduced in the CRYPTO'08 paper by Goldwasser et al. A one-time program is a device containing a program C, with the property that the program C can be executed on at most one input. Goldwasser et al. show how to implement one-time programs on devices equipped with special hardware-gadgets called *one-time memory* tokens. We provide an alternative construction that is based on the following assumptions: (1) the total amount of data that can leak from the device is bounded, and (2) the total memory on the device (available both to the honest user and to the attacker) is also restricted, which is SBA–model desribed before.

5.1 Introduction

A notion of *one-time programs* was introduced by Goldwasser et al. [35]. Informally speaking, a one-time program is a device $\mathcal{D}$ containing a program C, that comes with the following property: the program C can be executed on at most one input. In other words, any user, even a malicious one, that gets access to $\mathcal{D}$, should be able to learn the value of $C(x)$ for exactly one x at his choice. As argued by Goldwasser et al., one-time programs have vast potential applications in software protection, electronic tokens and electronic cash.

It is a simple observation that one-time programs cannot be solely software-based, or, in other words, one always needs to make some assumptions about the physical properties of the device $\mathcal{D}$. Indeed, if we assume that the entire contents $\mathcal{P}$ of $\mathcal{D}$ can be read freely, then an adversary can create his own copies of $\mathcal{D}$ and compute C on as many inputs as he wishes. Hence, it is natural to ask what kind of "physical assumptions" are needed to construct

the one-time programs. Of course, a trivial way is to go to the extreme and assume that $\mathcal{D}$ is fully-trusted, i.e. the adversary cannot read or modify its contents. Obviously, then one can simply put any program C on $\mathcal{D}$, adding an extra instruction to allow only one execution of C. Unfortunately, it turns out that such assumption is often unrealistic. Indeed, a number of recent works on side-channel leakage and tampering attacks have demonstrated that in real-life constructing a leakage- and tamper-proof devices is hard, if not impossible.

Therefore it is desirable to base the one-time programs on weaker physical assumptions. The construction of Goldwasser et al. [35] is based on the following physical assumption: they assume that $\mathcal{D}$ is equipped with special gadgets that they call *one-time memory (OTM)* devices. At the deployment of $\mathcal{D}$ an OTM can be initialized with a pair of values (K_0, K_1). The program $\mathcal{P}$ that is stored on $\mathcal{D}$ can later ask the OTM for the value of exactly one K_i. The main security feature of the OTMs is that the OTM under no circumstances releases both K_0 and K_1. Technically, it can be implemented by (a) storing on each OTM a flag f initially set to 0, that changes its value to 1 after the first query to this OTM, and (b) adding a requirement that if $f = 1$ then an OTM answers $\perp$ to every query. Under this assumption one can construct a general complier that transforms any program C (given as a Boolean circuit) into a one-time program that uses the OTMs. Hence, in some sense, Goldwasser et al. [35] replace the unrealistic assumption that the whole device $\mathcal{D}$ is fully secure, with a much weaker one that the OTM gadgets on $\mathcal{D}$ are secure. Actually, by *secure*' we mean that they are leakage-proof (in particular: they never leak both K_0 and K_1) and tamper-proof (and hence the adversary should not be allowed to tamper with f).

Our Contribution. One can, of course, still ask how reasonable it is to assume that all the OTMs placed on $\mathcal{D}$ are secure, and it is natural to look for other, perhaps more realistic, models where the transformation similar to the one of [35] would be possible. In this chapter we propose such an alternative model, inspired by recent work of Dziembowski et al. [25] on one-time computable self-erasing functions. In contrast to the assumption used by Goldwasser et al., in our model we do not assume security of individual gadgets on $\mathcal{D}$, but rather impose *global* restrictions on what kind of attacks are possible.

To explain and motivate the use of the model of Dziembowski et al. [25] in our context, let us come back to the observation that a "physical assumption" that is obviously needed is that the adversary cannot copy the entire contents $\mathcal{P}$ of $\mathcal{D}$, or more precisely, that the amount of information $f(\mathcal{P})$ about $\mathcal{P}$ that

leaked to the adversary is bounded. There has been lot of work recently on modeling such bounded leakage. A common approach, that we follow in this work, is to model it as an *input shrinking function*, i.e. a function f whose output is much shorter than its input (the length c of the output of f is a parameter called the *amount of leakage*). Such functions were first proposed in cryptography in the so-called bounded-storage model of Maurer [48]. Later, they were used to define the memory leakage occurring during the virus attacks in the bounded-retrieval model [3, 14]. In the context of the side-channel leakages they were first used by Dziembowski and Pietrzak [29] with an additional restriction that the memory is divided into two separate parts that do not leak information simultaneously, and in the full generality in the paper of Akavia et al. [1].

Obviously, if we want to incorporate the tampering attacks into our model then we also need some kind of a formal way to define the class of admissible tampering attacks. To see that some kind of limitations on tampering attacks are always needed let us first consider the broadest possible class of such attacks, i.e. let us assume that we allow the adversary to transform the contents $\mathcal{P}$ of the device in an arbitrary way. More precisely, suppose the adversary is allowed to substitute $\mathcal{P}$ with some $g(\mathcal{P})$, where g is an arbitrary function chosen by him. Obviously in this case there is no hope for any security, as the adversary can design a function g that simply calculates "internally" (i.e.: on the device) the values of the encoded program on two different inputs, and leaks them to the adversary (if these values are short then this can be done even if the amount of leakage is small). Hence, some limitations on g are always needed. Unfortunately, it is not so obvious what kind of restrictions to use here, as currently, unlike in the case of leakage attacks, there does not seem to be any widely-adopted model for tampering attacks. In fact, most of the anti-tampering models either assume that some part of the device is tamper-proof [33], or they are so strong that they permit only very limited constructions [30].

As mentioned before, in this work we follow the approach of Dziembowski et al. [25], where the authors model the tampering attacks by restricting the size of memory available to the tampering function g. More precisely, we assume that there is a general bound s on the space available on $\mathcal{D}$, that can be used by anybody who performs computations on $\mathcal{D}$, including the honest program $\mathcal{P}$ and the adversary. This assumption can be justified by the following observations: (1) it is reasonable to assume that in practice the bound on the memory size of the device is known, and no adversary can "produce" additional space on it by tampering with it, and (2) in general it is also reasonable to assume that the tampering function is "simple", and hence it cannot have a large space-complexity.

What remains is the describe the way in which the restrictions c on leakage and s on communication are combined into a single model. The way it is done by Dziembowski et al. [25] is as follows: they model the adversary as two entities: a *big* adversary $\mathcal{A}_{big}$ and a *small* adversary $\mathcal{A}_{small}$. The small adversary represents the tampering function, and hence it has a full access to the contents $\mathcal{P}$ of the device. It can perform any computation on $\mathcal{D}$ subject to the constraint that it cannot use more memory than s. The fact that it can leak information to the outside is modeled by allowing him to communicate up to c bits outside of the device. This leakage information can later be processed by the big adversary $\mathcal{A}_{big}$ that has no restrictions on his space complexity. In order to make the model as strong as possible we actually allow $\mathcal{A}_{big}$ to communicate with $\mathcal{A}_{small}$ in several rounds (and we do not impose any restriction on the amount of information communicated by $\mathcal{A}_{big}$ back to $\mathcal{A}_{small}$). We apply exactly the same approach in our work. Our main result (see 5.2) is a generic compiler that takes any circuits C and transforms it into a one-time program $\mathcal{P}$ secure in the model described above. As in the case of [25] , our result holds in the Random Oracle Model (where we model as random oracles hash functions of fixed input lengths). For the concrete parameters see 5.2, and 5.4 below it. Let us only remark here, that for a fixed circuit we get that the security holds as long as $s - 2nc \geqslant \gamma k$, where γ is some constant, and n is the number of input bits of the circuit. Hence, the leakage size c has to be inversely-proportional to n. We remark that it is probably ok for small n's (e.g. if the input is a PIN that has for decimal digits). In any case, for any realistic values of other parameters it is super-logarithmic, and hence covers all attacks where the leaking value is a scalar (e.g. the Hamming weight of the bits on the wires).

Related Work. Some related work was already described above. The feasibility of implementing the scheme of Goldwasser et al. [35] was analysed by Jarvinen et al. [39]. The model of Dziembowski et al. [25] and the pebbling technique were also used in a subsequent paper [24] to construct leakage-resilient key-evolution schemes. Finally, let us note that the main difference between [25] and our work is that in [25] the authors construct a one-time scheme for a concrete cryptographic functionality (i.e., a pseudorandom function), while here we show a generic way to implement *any* functionality as a one-time program.

5.2 Preliminaries

Across this chapter, we often make use of boolean circuits. We use a capital C to label such a circuit. If C has n inputs and m outputs then we identify C with a function $C\colon \{0,1\}^n \to \{0,1\}^m$. For simplicity, we confine the analysis to the case where every gate of C has a fan-in of 2. Also, we assume that no input wire of C is also an output wire of the circuit, and that output of any gate cannot be simultaneously attached to input of some other gate and to output wire of C. Each wire, including the input and output ones, and every gate is assigned a unique label. A size of C, defined as a number of gates in C, is denoted by $|C|$. Following the work of Yu et al. [61], we use $\mathrm{Topo}(C)$ to represent just a connectivity graph (a topology) of C where each gate is stripped of information about what functionality it actually implements.

We write $C(x)$ for a result of evaluating C on a given input x, and, more generally, $\mathcal{A}(x)$ for an outcome of running an algorithm $\mathcal{A}$ (modelled as a Turing machine, possibly a non-deterministic one) on x. Occasionally, we add a superscript $\mathcal{H}$ to $\mathcal{A}$ and write $\mathcal{A}^{\mathcal{H}}$ to signify that $\mathcal{A}$ is given access to an oracle that computes some function $\mathcal{H}$. Everywhere below it is assumed that there exists a (programmable) random oracle $\mathcal{H}\colon \{0,1\}^* \to \{0,1\}^k$ for a parameter k to be specified later. Phrases: the random oracle $\mathcal{H}$ and the function $\mathcal{H}$ are then used interchangeably.

When typesetting algorithms, we write $R \xleftarrow{\$} S$ for sampling a uniformly random value from some set S and assigning it to a variable R. We assume that every such sample is independent of other choices. We conform to the common bracket notation $T[i]$ for accessing the ith element of an array T.

We say that a function is *negligible* in k if it vanishes faster than the inverse of any polynomial of k. In particularly, we use this expression to indicate that certain event can only occur with a small, i.e. negligible, probability in some security parameter k. Also, we often write just: *a negligible probability* and omit k when this parameter is clear from context.

As announced in 5.1, the model we adopt in the work assumes splitting an adversary $\mathcal{A}$ into two components: $\mathcal{A}_{small}$ and $\mathcal{A}_{big}$. Both parts are interactive algorithms with access to $\mathcal{H}$, where a total number of oracle calls made is limited. Additionally, $\mathcal{A}_{small}$, which can see the internals of an attacked device, has:

- s-bounded space – a total amount of memory used by $\mathcal{A}_{small}$ does not exceed s bits, i.e., an entire configuration of $\mathcal{A}_{small}$ (contents of all tapes, a current state, and positions of all the tape heads), at any point of execution, can be described using s bits;

- c-bounded communication – a total number of outgoing bits sent by

$\mathcal{A}_{small}$ does not exceed c, assuming that $\mathcal{A}_{small}$ cannot convey any extra information when communicating with $\mathcal{A}_{big}$ (e.g. by abstaining from sending anything during some period of time).

Note that $\mathcal{A} = (\mathcal{A}_{small}, \mathcal{A}_{big})$ can have an unbounded computational power. Also, the amount of bits uploaded by $\mathcal{A}_{big}$ to $\mathcal{A}_{small}$ is not restricted.

We write $\mathcal{A}^{\mathcal{H}}(R) = \left(\mathcal{A}^{\mathcal{H}}_{big}() \leftrightarrows \mathcal{A}^{\mathcal{H}}_{small}(R)\right)$ to denote the interactive execution of $\mathcal{A}_{big}$ and $\mathcal{A}_{small}$, where $\mathcal{A}_{small}$ gets R as an input. We settle on a simplifying arrangement that the contents of memory (e.g. the data on all the tapes) of $\mathcal{A}_{big}$ after it finishes its run form a result of this execution. In particular, any information computed by $\mathcal{A}_{small}$ needs to be transmitted to $\mathcal{A}_{big}$ (contributing to the communication quota) in order to be included as a part of the result. Such an approach is justified by our real-world interpretation of $\mathcal{A}_{small}$ and $\mathcal{A}_{big}$ as a virus and a remote adversary controlling the virus. Here, only the data that the external adversary can receive is considered valuable.

5.3 One-time Program

In this section, we give a strict definition of one-time programs/devices. Intuitively, an ideal one-time program should mimic a black-box that internally calculates a value of some boolean circuit C. It should allow only one execution on an arbitrary input after which it self-destructs. To boot, the black-box should not leak any information about C whatsoever. As explained in 5.4.2, there are theoretical obstacles that make this goal impossible to achieve in its full generality. So instead, we show that any adversary that operates a one-time device can evaluate it on a single argument x and can hardly learn anything more about the underlying circuit C but n, m, and $|C|$. It therefore gains some additional knowledge that goes beyond $C(x)$, namely the size of the circuit. Admittedly, that information is not considered substantial in practice. 5.1 makes this property formal in terms of a simulator that is permitted to call an oracle evaluating C only once.

Definition 5.1. *Suppose that $\mathcal{K}$ is a fixed class of algorithms. Let $C\colon \{0,1\}^n \to \{0,1\}^m$ be a boolean circuit with positive integers n and m. Let $\mathcal{O}$ denote an oracle that computes $C(x)$ given $x \in \{0,1\}^n$. Consider an algorithm $\mathcal{A} \in \mathcal{K}$ which is additionally s-bounded in space and c-bounded in communication. A protocol $\mathcal{P}$ is called a* (c, s, ϵ)–one-time program *for C if both of the following conditions hold:*

- *there exists a probabilistic polynomial-time decoder $\mathcal{D}\mathrm{ec}$ that given $x \in \{0,1\}^n$ executes $\mathcal{P}$ using at most s bits of memory, so that $\mathcal{D}\mathrm{ec}(x, \mathcal{P}) =$*

$C(x)$, except for probability ϵ (where the probability is taken over all possible choices of x and $\mathcal{P}$);

- *there exists a simulator $\mathcal{S}$ with one-time oracle access to $\mathcal{O}$, such that, for any adversary $\mathcal{A}$, no algorithm belonging to $\mathcal{K}$ can distinguish $\mathcal{S}(1^n, 1^m, 1^{|C|}, \mathcal{A})$ and $\mathcal{A}(\mathcal{P})$ with a probability greater than ϵ.*

We note that the one-time property formulated in this way is even a bit stronger than one might expect. For instance, it could be believed that discovering $C(x)$ alongside with a functionality of a single boolean gate inside C does not grant an adversary any significant advantage. If $|C|$ is large then this additional knowledge barely helps to perceive how C actually looks like as a whole. In our definition, we disallow adversary to find out anything more than n, m, $|C|$, and $C(x)$ for a single x.

Shortly, in 5.5, we construct a compiler $\mathrm{Compile}_{k,s}(C)$ that, for some parameter k, converts any boolean circuit C to a one-time program $\mathcal{P}$ that can be organised into a device with s bits of memory. The main result of this chapter is stated in the below 5.2 about $\mathrm{Compile}_{k,s}(C)$. The Theorem contains a reference to circuit obfuscation. An obscured form of C, denoted $\widetilde{C}$, is produced by the algorithm of Yu et al. [61], which is discussed in 5.4.2. Making C uniform introduces a small blow-up factor (see (5.4) below) so that $\widetilde{C}$ is slightly larger than C.

Theorem 5.2. *Let k be a security parameter. For a boolean circuit $C\colon \{0,1\}^n \to \{0,1\}^m$ write $|\widetilde{C}|$ to denote a number of gates of $\widetilde{C}$ – an obfuscated version of C of uniform topology. Suppose there exists an oracle $\mathcal{H}\colon \{0,1\}^* \to \{0,1\}^k$ computing a random hash function. Assume that every algorithm in the class $\mathcal{K} = \mathcal{K}_q$ is allowed at most q calls to $\mathcal{H}$, where $k \geqslant 4n^2 \log q$. Then, for any C and $\mathcal{P} \leftarrow \mathrm{Compile}_{k,s}(C)$, the protocol $\mathcal{P}$ is a (c, s, ϵ)–one-time program for C with $\epsilon = O(q|\widetilde{C}|2^{-k})$, provided that $k \geqslant m$ and*

$$s - 2nc \geqslant 16k(|\widetilde{C}| \log |\widetilde{C}| + 3n^2 + nm) + O\Big((n + k + \log |\widetilde{C}|)n\Big)\ . \qquad (5.1)$$

Remark 5.3. *We note that the above Theorem holds even if a potential distinguisher is given C. Also, we impose no limits on its running time as well as on time-complexity of the adversary. $\mathcal{A}$ can be computationally unbounded but he merely subjects to restriction on the number of oracle calls made. The construction of $\mathcal{S}$ is universal, i.e., it is independent of C and $\mathcal{P}$ so no information about C is hardwired in $\mathcal{S}$, as opposed to the solution by Goldwasser et al. [35]. We also mention with a minor modification of our construction we can replace the factor $2n$ on the left-hand side of (5.1) with $2n/\log n$.*

Remark 5.4. *To interpret 5.2 for concrete parameters we can assume that in (5.1) the big O term has only a small contribution compared to the other parameters and thus can be neglected. Take* $n = m = 128$ *(the key and the block size of AES),* $k = 50$ *and* $|\widetilde{C}| = 256$*. With these parameters we obtain, by* (5.1)*, that* $s - 256c$ *has to be greater than approximately* $7MB$*. Hence, e.g., for* $s = 20MB$ *we get that the leakage* c *that we can tolerate is approximately* $51KB$*.*

If we consider smaller input and output size, $n = m = 14$*, say (*2^{14} *is an approximate number of 4-decimal digits PINs), and* $k = 50$ *and* $|\widetilde{C}| = 128$*, then we get that* $s - 28c$ *has to be greater than approximately* $200KB$*. Hence, e.g., for* $s = 1MB$ *we get that the leakage* c *that we can tolerate is approximately* $30KB$*.*

5.4 Tools

For completeness of the exposition, we outline several existing constructions the architecture of one-time devices builds upon – circuit obfuscation techniques and one-time computable pseudorandom functions.

5.4.1 Circuit Garbling

An important landmark in the theory of multi-party computations was set up by Yao in mid '80s. His seminal work [60] provided the first general protocol that enabled two honest-but-curious users to jointly evaluate a function f without disclosing their respective private inputs x. A so called *circuit garbling* process accounted for an essential part of this method. Its role was to conceal all intermediate values that occur on internal wires (in particular: on certain input wires) of a boolean circuit representing f during computation. Below, we follow an expository paper [45] by Pinkas and Lindell who gave the first rigorous treatment of Yao's protocol.

Let k be a security parameter to be determined shortly. Given a boolean circuit C, the garbling procedure $\mathrm{Garble}_k(C)$ converts it to a somewhat encrypted form and specifies how to conduct computations on the latter. First, we associate each wire w of C (including input and output wires of C) with two random strings K_0^w and K_1^w of length k. These two keys represent binary values 0 and 1, respectively, that would appear on w if a computation took place on C in its plain, unencrypted state. Instead, a user that evaluates a garbled circuit, can only see either $K = K_0^w$ or $K = K_1^w$, but he is not able to tell which binary value K corresponds to. This way, all actual intermediate values on internal wires can be hidden effectively.

To make the description of $\mathrm{Garble}_k(C)$ complete, we need to specify how the above strings for input wires of any gate map to values on output wires. Pinkas and Linell [45] introduce an auxiliary encryption scheme, possibly a non-deterministic one, (E, D) to mask this mapping. We call it *a garbling encryption* scheme. It enjoys some extra properties going a little beyond standard semantic security. In what follows, $E_K(\cdot)$ denotes the encryption under a key K (similarly, $D_K(\cdot)$ stands for the decryption using K). We adopt the following definition of E_K based on $\mathcal{H}$, which satisfies the requirements listed by Pinkas and Lindell:

$$E_K(M) \coloneqq (\mathcal{H}(K), r, \mathcal{H}(K, r) \oplus M) \quad \text{where } r \xleftarrow{\$} \{0,1\}^k. \tag{5.2}$$

A double-encryption under two keys, say K_1 and K_2, each of length k, which is written as $E_{K_1;\,K_2}(\cdot)$ with $D_{K_1;\,K_2}(\cdot)$ being the complementary double-decryption, is a paramount ingredient of the garbling process. Departing from the original solution [45] for technical reasons, we specify $E_{K_1;\,K_2}(\cdot)$ separately extending (5.2) with:

$$E_{K_1;\,K_2}(M) \coloneqq (\mathcal{H}(K_1, K_2), r, \mathcal{H}(K_1, K_2, r) \oplus M) \quad \text{for } r \xleftarrow{\$} \{0,1\}^k. \tag{5.3}$$

In the remainder of this work we assume that ciphertexts in a garbling encryption scheme are all of length $3k$ as implied by (5.2) and (5.3).

Now, consider a boolean gate g of C with two input wires w_1, w_2, and an output wire w_3. We begin with writing down an ordinary truth table for g. We replace each bit b appearing in the table for every wire $w \in \{w_1, w_2, w_3\}$ with K_b^w. Next, we double-encrypt each entry K_3 in the last column (corresponding to w_3) to $E_{K_1;\,K_2}(K_3)$ using keys K_1 and K_2 from the two preceding columns. Finally, we randomly permute 4 rows of the table and restrict attention the right-most column. These four entries constitute the result of garbling g. 5.1 depicts how this procedure works when applied to a NAND gate.

w_1	w_2	w_3
0	0	1
0	1	1
1	0	1
1	1	0

(a) The truth table for a NAND gate

w_1	w_2	w_3
$K_0^{w_1}$	$K_0^{w_2}$	$E_{K_0^{w_1};\,K_0^{w_2}}(K_1^{w_3})$
$K_0^{w_1}$	$K_1^{w_2}$	$E_{K_0^{w_1};\,K_1^{w_2}}(K_1^{w_3})$
$K_1^{w_1}$	$K_0^{w_2}$	$E_{K_1^{w_1};\,K_0^{w_2}}(K_1^{w_3})$
$K_1^{w_1}$	$K_1^{w_2}$	$E_{K_1^{w_1};\,K_1^{w_2}}(K_0^{w_3})$

garbled table

(b) A garbled table for a NAND gate

Figure 5.1: Garbling of a NAND gate

The garbling routine $\text{Garble}_k(C)$ processes C gate by gate. It constructs and collects garbled tables for each gate as described above. In overall, $\text{Garble}_k(C)$ outputs the following information: a mapping I between plain bits $b = 0, 1$ and $K_b^{\text{in}_i}$ for each input wire in_i, a reverse map O from $K_b^{\text{out}_i}$ to b for each output wire out_i, and a topology $\text{Topo}(C)$ of C with garbled tables attached. This data is required to make evaluations of C in a garbled form possible. Later, however, we switch to circuits of uniform topology. This allows to embed information about underlying graph in a program that computes a garbled circuit basing on a flat array of garbled tables. A summary of $\text{Garble}_k(C)$ is given in 1.

Algorithm 1 Garbling procedure $\text{Garble}_k(C)$

INPUT: a boolean circuit $C\colon \{0,1\}^n \to \{0,1\}^m$, a security parameter k
OUTPUT: a circuit topology $\mathfrak{C}$ with garbled tables, I, O – bit-to-key mappings

1: **procedure** $\text{Garble}_k(C)$
2: **for each** wire w of C **do**
3: $K_0^w, K_1^w \xleftarrow{\$} \{0,1\}^k$
4: **for each** gate g of C **do**
5: construct a truth table for g
6: replace each bit b with K_b^w for every wire w connected to g
7: $T^g \leftarrow \{E_{K_1;K_2}(K_3) \mid K_1, K_2, K_3 \text{ form a row of the truth table of } g\}$
8: **end for each**
9: **for** $i \leftarrow 1$ **to** n **do**
10: $I[i] \leftarrow (K_0^{\text{in}_i}, K_1^{\text{in}_i})$ ▷ in_i is the ith input wire of C
11: $\mathfrak{C} \leftarrow \text{Topo}(C) \cup \{(g, T^g) \mid g \text{ is a gate of } C\}$
12: **for** $i \leftarrow 1$ **to** m **do**
13: $O[i] \leftarrow (K_0^{\text{out}_i}, K_1^{\text{out}_i})$ ▷ out_i is the ith output wire of C
14: **return** $(I, \mathfrak{C}, O)$
15: **end procedure**

The protocol for evaluating a garbled circuit on any input x is quite straightforward. First, we use the mapping I to translate each bit b of x appearing on an input wire w to the appropriate key K_b^w. Let K_x be a vector composed of these K_b^w. Subsequently, we apply the following algorithm $\text{Eval}(\mathfrak{C}, O, K_x)$. We traverse $\mathfrak{C}$ in the same way as in the case of ordinary boolean circuit, that is: when we obtain keys K_1 and K_2 for both input wires of any garbled gate, then we can put a proper key K_3 on its output wire. To this end, we test which of 4 entries of the garbled table is a double-encryption under K_1 and K_2. If more than one ciphertext matches then the whole computation aborts reporting an error (this can, however, happen with

a negligible probability). Otherwise, we decrypt that entry with $D_{K_2;\,K_1}(\cdot)$ getting K_3 as a result. In the end, it suffices to consult O to map keys from output wires of $\mathfrak{C}$ back to plain bits.

It can be proven that Eval correctly evaluates a garbled circuit given K_x and the computation reveals nothing more about x than $C(x)$.

5.4.2 Uniform Circuit Topology

One of the requirements a one-time program has to stand up to is ensuring that no eavesdropping into program's internals is possible. It is also a common problem in practical computer science to create software invulnerable to reverse engineering. Usually, satisfactory results can be achieved by ad-hoc techniques that decrease readability of a program (e.g. by obscuring a source code syntactically or inserting NOOPs). From a theoretical point of view, however, an ideal obfuscator cannot exist. Barak et al. [6] provide an artificial example of a family of functions that are inherently unobfuscatable. That is, there always exists a predicate which leaks when we are given a function in its plain form but cannot be reliably guessed if the function is implemented as a black-box. Fortunately, some *partial* obfuscation is attainable. Yu et al. [61] describe a method for hiding a topology of a boolean circuit C. We recall a result on their obfuscating algorithm Obfuscate(C) below.

Theorem 5.5 (cf. Theorem 1 and Theorem 2 of Yu et al. [61])**.** *Let* $C\colon \{0,1\}^n \to \{0,1\}^m$ *be a boolean circuit with* $m = 1$*. Then, the obfuscating algorithm* Obfuscate(C) *constructs a circuit* $\widetilde{C}$ *with* $|\widetilde{C}| = O(|C| \log^3 |C|)$ *such that* $\widetilde{C}(x) = C(x)$ *for all* $x \in \{0,1\}^n$ *and* Topo($\widetilde{C}$) *discloses (in the information-theoretic sense) nothing more than* n*,* m*, and* $|C|$*.*

Remark 5.6. *A naĂŻve application of the above Theorem leads to an obfuscated circuit of uniform topology for the case where* $m > 1$*. The additional overhead the construction then incurs is*

$$|\widetilde{C}| = O(m|C| \log^3 |C|)\ . \tag{5.4}$$

The below 5.4.2 follows from the analysis given by Yu et al. [61].

Proposition The algorithm Obfuscate(C) uses at most $4|\widetilde{C}| \log |\widetilde{C}|$ bits of memory. Put differently, given n, m, and $|C|$ it is possible to generate a uniform topology that is common for all circuits with n-bit input, m-bit output, and $|C|$ gates, within space of $4|\widetilde{C}| \log |\widetilde{C}|$ bits.

5.4.3 One-Time Computable Pseudorandom Function (PRF)

A notion of the one-time computable pseudorandom functions was introduced by Dziembowski et al. [25]. A salient development of this work is a construction of a pseudorandom function, or, more generally, a set of n such functions, where each function can be calculated for a single argument in the computation model with $\mathcal{A}_{big}$ and $\mathcal{A}_{small}$ having limited space and communication. Dziembowski et al. assume the existence of the random oracle $\mathcal{H}$. The underlying idea is to store a long random key, say R, on a device that $\mathcal{A}_{small}$ operates on. Now, R and $\mathcal{H}$ determine n distinct pseudorandom functions $(F^{\mathcal{H}}_{1,R}, \ldots, F^{\mathcal{H}}_{n,R})$. It is possible to evaluate each one on any input but the computation forces an erasure of R so that no one viably compute both $F^{\mathcal{H}}_{i,R}(x)$ and $F^{\mathcal{H}}_{i,R}(x')$ for any two points $x \neq x'$ and the same index i. The reasoning by Dziembowski et al. [25] includes establishing a strict connection between this model and a certain game, called *a pebble game*, on graphs of a special form. That is why we occasionally use the word *pebble* in the context of the one-time computable PRFs. Below, we borrow some basic definitions from the original paper to formalise the mentioned properties.

Consider an algorithm $\mathcal{W}^{\mathcal{H}}$ that takes a key $R \in \{0,1\}^{\mu}$ as an input and has access to the oracle $\mathcal{H}$. Let $(F^{\mathcal{H}}_{1,R}, \ldots, F^{\mathcal{H}}_{n,R})$ be a sequence of functions depending on $\mathcal{H}$ and R. Assume that $\mathcal{W}^{\mathcal{H}}$ is interactive, i.e., it may receive queries, say $x_1, \ldots, x_n$, from the outside. The algorithm $\mathcal{W}^{\mathcal{H}}$ replies to such a query by issuing a special *output query* to $\mathcal{H}$. We assume that after receiving each $x_i \in \{0,1\}^*$ the algorithm $\mathcal{W}^{\mathcal{H}}$ always issues an output query of a form $((F^{\mathcal{H}}_{i,R}(x_i), (i, x_i)), \texttt{out})$. We say that an adversary *breaks pebbles* if a transcript of oracle calls made during its entire execution contains two queries $((F^{\mathcal{H}}_{i,R}(x), (i, x)), \texttt{out})$ and $((F^{\mathcal{H}}_{i,R}(x'), (i, x')), \texttt{out})$, appearing at any point, for some index i and $x \neq x'$.

Definition 5.7. *$\mathcal{W}^{\mathcal{H}}$ is a $(c, \mu, \sigma, q, \epsilon, n)$–*one-time computable PRF *if:*

- *$\mathcal{W}^{\mathcal{H}}$ has μ-bounded storage and 0-bounded communication;*
- *for any $\mathcal{A}^{\mathcal{H}}(R)$ that makes at most q queries to $\mathcal{H}$ and has σ-bounded storage and c-bounded communication, the probability that $\mathcal{A}^{\mathcal{H}}(R)$ (for a randomly chosen $R \xleftarrow{\$} \{0,1\}^{\mu}$) breaks pebbles, is at most ϵ.*

Dziembowski et al. [25] prove the existence of the one-time computable PRFs under some plausible assumption on parameters c, μ, σ, q, ϵ, and n.

The use case we investigate in the work requires a slightly stronger primitive than a PRF of 5.7. In this work, we introduce an *extended one-time computable PRF*. An observation we come out with is that the limits on memory

available to an adversary can be relaxed moderately. Namely, once all $F^{\mathcal{H}}_{i,R}$ are computed on some arguments, an adversary might be given unrestricted space, yet it still gains no advantage in breaking pebbles in the remainder of its execution. Now, the *computing phase* is a time interval between the beginning of an execution and the moment when all output queries of the form $((F^{\mathcal{H}}_{i,R}(x_i), (i, x_i)), \mathtt{out})$ were made (for some x_i and every $i = 1, \ldots, n$), provided that no i appears twice in that part of transcript. The below 5.8 strengthens the notion of a one-time computable PRF.

Definition 5.8. *An algorithm $\mathcal{W}^{\mathcal{H}}$ is a $(c, \mu, \sigma, q, \epsilon, n)$–*one-time computable extended PRF *if:*

- *$\mathcal{W}^{\mathcal{H}}$ is a $(c, \mu, \sigma, q, \epsilon, n)$–*one-time computable PRF*;*
- *for any adversary $\mathcal{A}^{\mathcal{H}}(R)$ that makes at most q queries to $\mathcal{H}$, has σ-bounded storage and c-bounded communication* during the computing phase*, but is not bounded on space afterwards, the probability that $\mathcal{A}^{\mathcal{H}}(R)$ breaks pebbles, is at most ϵ.*

The theorem about the existence of the one-time computable extended PRFs holds with essentially the same parameters as in the base theorem by Dziembowski et al. [25]. Here, we present one more result about the existence of the extended PRFs that provides a condition which is more convenient to use in our particular application.

Theorem 5.9. *Let c, μ, δ, q, and n be positive integers. Then, for any $\epsilon \leqslant q2^{-4n^2}$, there exists a $(c, \mu, \mu + \delta, q, \epsilon, n)$–one-time computable extended PRF, provided that*

$$\mu \geqslant 2n(\delta + c + 4\log q + 6\log \epsilon^{-1} + 6)\ . \tag{5.5}$$

5.5 One-time Compiler

In this section, we give a high-level description of what a one-time device is made up of. A purpose of a one-time compiler is to transform an arbitrary boolean circuit $C\colon \{0,1\}^n \to \{0,1\}^m$ into a deliberately obscured form accompanied with some additional logic (a procedure) that enables evaluations of the circuit on every single n-bit input.

The compiler routine $\mathrm{Compile}_{k,s}$ constructs a one-time program deployable on a device with a grand total of s bits of writable memory (including registers, RAM, flash memory, and any other persistent storage). We, however, introduce no extra assumptions on the amount of read-only memory

available. $\text{Compile}_{k,s}$ is allowed unrestricted use of a source of random bits, as well as access to the aforementioned random oracle $\mathcal{H}\colon \{0,1\}^* \to \{0,1\}^k$ with k being a security parameter. 2 presents a listing of the one-time compiler procedure.

Algorithm 2 One-time compiler $\text{Compile}_{k,s}(C)$

INPUT: a boolean circuit $C\colon \{0,1\}^n \to \{0,1\}^m$, a security parameter $k \geqslant m$, a total amount of memory on the device s
OUTPUT: a one-time program $\mathcal{P} = (m, R, L, K, \mathfrak{C}, O, \mathcal{B})$

procedure $\text{Compile}_{k,s}(C)$
  for $i \leftarrow 1$ **to** n **do**
    $L^{\text{in}_i} \xleftarrow{\$} \{0,1\}^k$
  $\text{Latch} \leftarrow L^{\text{in}_1} \oplus \cdots \oplus L^{\text{in}_n}$
  $\text{Mask} \leftarrow \mathcal{H}(\text{Latch})_{|m}$
  $\widetilde{C} \leftarrow \text{Obfuscate}(C \oplus \text{Mask})$
  $(I, \mathfrak{C}, O) \leftarrow \text{Garble}_k(\widetilde{C})$
  $\text{Master} \leftarrow \mathcal{H}(\text{Latch}, \mathfrak{C})$
  $\mu \leftarrow s - (12|\widetilde{C}| + 8n + 2m)k - \log m - \texttt{EXEC\_SIZE}$
  round μ down to the largest multiple of k
  $R \xleftarrow{\$} \{0,1\}^\mu$
  for each input wire in_i of C **do** ▷ in_i is the ith input wire of C
    $(K_0^{\text{in}_i}, K_1^{\text{in}_i}) \leftarrow I[i]$
    compute $F_{i,R}^{\mathcal{H}}(0)$ and $F_{i,R}^{\mathcal{H}}(1)$
    $L[i] \leftarrow \left(E_{F_{i,R}^{\mathcal{H}}(0)}(L^{\text{in}_i}), E_{F_{i,R}^{\mathcal{H}}(1)}(L^{\text{in}_i})\right)$
    $K[i] \leftarrow (K_0^{\text{in}_i} \oplus F_{i,R}^{\mathcal{H}}(0) \oplus \text{Master}, K_1^{\text{in}_i} \oplus F_{i,R}^{\mathcal{H}}(1) \oplus \text{Master})$
  end for each
  construct a boot program $\mathcal{B}$ that fits into space of `EXEC_SIZE` bits
  return $(m, R, L, K, \mathfrak{C}, O, \mathcal{B})$
end procedure

Firstly, the compiler prepares (2 of 2) a set of random keys L^{in_i}, which we refer to as *latchkeys*. A value L^{in_i} corresponds to the ith input wire in_i of C. A string $\text{Latch} \coloneqq L^{\text{in}_1} \oplus \cdots \oplus L^{\text{in}_n}$ combines all the latchkeys into a single key. From Latch we derive (5), by means of the oracle, one more random value, denoted Mask, trimming the output of $\mathcal{H}$ to the leading $m \leqslant k$ bits. The exact role that all these auxiliary components play should become clear later, in 5.6. Having calculated these values, the compiler enters its main phase in 6. There, the obfuscation algorithm is run, yet on a biased version of C, say C^*, defined as $C^*(x) \coloneqq C(x) \oplus \text{Mask}$. At this point Mask is merely a constant that does not depend on x. Obviously, C^* can be viewed as a

boolean circuit and implemented in such a way that $|C^*| = |C|$ (it suffices to flip, if needed, a functionality of each gate an output wire of C is attached to, depending on the corresponding bit of Mask). The reason behind switching to C^* instead of working with C directly is that the simulator from 5.2 needs to alter an output of a circuit when interacting with an adversary. This trick can be exercised by changing the value of Mask in a transparent way, which is done by $\mathcal{S}$ in 5.6.

The obfuscated circuit is garbled (7) using Yao's method given in 1. Next, a one-time computable extended PRF (in the sense of 5.8) is set up (10). Actually, this step boils down to picking a random string R that determines (together with $\mathcal{H}$) said pseudorandom functions $F^{\mathcal{H}}_{i,R}$. The embedded one-time computable extended PRF is a primitive that protects input keys of the garbled circuit. Namely, in order to evaluate a one-time program on some input $x = b_1 b_2 \dots b_n$, one has to compute each $F^{\mathcal{H}}_{i,R}(b_i)$ for $i = 1, \dots, n$. By virtue of the property of extended PRFs, this computation erases an essential portion of memory available on the device and makes evaluations of $F^{\mathcal{H}}_{i,R}(\bar{b}_i)$ infeasible. The compiler, however, needs to find both: $F^{\mathcal{H}}_{i,R}(0)$ and $F^{\mathcal{H}}_{i,R}(1)$ for all i's (this requires a larger amount of memory than just s bits but still $\mathrm{Compile}_{k,s}$ is clearly polynomial in space and time).

Stored on the device are two encryptions of each latchkey L^{in_i} under $F^{\mathcal{H}}_{i,R}(0)$ and $F^{\mathcal{H}}_{i,R}(1)$ as encryption keys. For this purpose, in 15 where these ciphertexts are accumulated in array L, we use the garbling encryption scheme as given by (5.2). The input keys for $\mathfrak{C}$ generated by the garbling procedure get encoded too before being placed on the device. That is: the ith entry of K contains, for $b = 0$ and 1, simple one-time pad encryptions of $K^{\mathrm{in}_i}_b$ under a key $F^{\mathcal{H}}_{i,R}(b) \oplus \mathrm{Master}$. Here, $\mathrm{Master} := \mathcal{H}(\mathrm{Latch}, \mathfrak{C})$ is a value that depends on all the latchkeys and the garbled circuit $\mathfrak{C}$. This all-or-nothing construction ensures that a user can no sooner determine $K^{\mathrm{in}_i}_b$ than he has computed *all* $F^{\mathcal{H}}_{i,R}(b_i)$'s. Also, this allows us to hold off the moment when an adversary can reclaim a part of memory occupied by $\mathfrak{C}$ and reuse it to enlarge space available for computing (or breaking) the extended PRF. In this way we control the amount of free memory during the computing phase specified in 5.8.

A module that no physical computing device could do without and function properly is obviously its firmware. Thus, we include a machine code $\mathcal{B}$ responsible for performing a bootstrap and managing computations. The central part of $\mathcal{B}$ is an implementation of the decoder $\mathcal{D}\mathrm{ec}$ claimed by 5.1. The code as this one must be vastly architecture specific. That is why we omit the details of $\mathcal{B}$'s internals. The only constraint imposed on $\mathcal{B}$ is that it does not exceed a reserved space of `EXEC_SIZE` bits total, where

$$\texttt{EXEC\_SIZE} = O(k + n + \log|\widetilde{C}|) \ . \tag{5.6}$$

We stress that the memory $\mathcal{B}$ is deployed on does not necessarily need to be read-only but is fully accessible to an adversary. It can be also assumed that registers or CPU cache, if present on the device, count toward the size of $\mathcal{B}$.

Now that we have described the one-time compiler, we present a decoder $\mathcal{D}\mathrm{ec} = \mathcal{D}\mathrm{ec}_k$ which is capable of evaluating a program produced by $\mathrm{Compile}_{k,s}$ on an arbitrary input $x = b_1 b_2 \dots b_n$. As the first step, $\mathcal{D}\mathrm{ec}_k$ determines $F^{\mathcal{H}}_{i,R}(b_i)$ for each $i = 1, \dots, n$. This is accomplished by computing labels of output vertices under a *random oracle labeling* of a certain on-line constructed graph. The key R that settles a labeling of input vertices of this graph gets erased during the process, and the region of memory that contained R can be reused by $\mathcal{D}\mathrm{ec}$. Next, the decoder decrypts a matching entry of each $L[i]$ to find L^{in_i}. Based on these latchkeys, $\mathcal{D}\mathrm{ec}$ computes $\mathrm{Latch}, \mathrm{Mask} = \mathcal{H}(\mathrm{Latch})_{|m}$, $\mathrm{Master} = \mathcal{H}(\mathrm{Latch}, \mathfrak{C})$, and reveals, using $K[i]$, input garbled keys $K^{\mathrm{in}_i}_{b_i}$ that correspond to each bit b_i. Let K_x be a vector consisting of all $K^{\mathrm{in}_i}_{b_i}$'s. The decoder then executes $\mathrm{Eval}(\mathfrak{C}, O, K_x)$ subroutine and calculates a bitwise exclusive or of the result with Mask to obtain the final value, i.e., $C(x)$. As for evaluating $\mathfrak{C}$, the garbled circuit kept on the device only includes a list of garbled tables without its actual topology. Prior to running Eval, the decoder needs to generate the unique uniform topology distinctive for all circuits of n inputs, m outputs, and $|C|$ gates. That is, $\mathcal{D}\mathrm{ec}$ simulates the obfuscating algorithm on such an arbitrarily chosen circuit. A memory that has to be supplied by $\mathcal{D}\mathrm{ec}$ for this step is located exactly in the same region the key R was previously stored in. By 5.4.2, this space, which is considered free after computing the extended PRF, has a sufficient size if $\mu = |R| \geqslant 4|\widetilde{C}| \log |\widetilde{C}|$. The sizes of the remaining components of $\mathcal{P}$ can be easily counted: $|L| = 6nk$, $|K| = 2nk$, $|\mathfrak{C}| = 12|\widetilde{C}|k$, and $|O| = 2mk$. In total, the space that $\mathcal{P}$ occupies is

$$|\mathcal{P}| = \mu + (12|\widetilde{C}| + 8n + 2m)k + \log m + \texttt{EXEC\_SIZE}\ , \tag{5.7}$$

where we can replace `EXEC_SIZE` with the big O term arising from (5.6).

We state some straightforward facts about the decoder in 5.7.

5.6 Universal Simulator for One-time Programs

In this section, we focus on the more intricate part of 5.1 and describe an explicit simulator $\mathcal{S}$. We employ a similar approach to the one that appears in the work of Goldwasser et al. [35]. A notable difference, however, is that our construction includes a component, i.e, the extended one-time computable PRF, which does not offer a black-box security, in opposition to the aforementioned OTMs. The condition (5.1) of 5.2 ensures that our replacement

of the OTMs, i.e., the extended one-time computable PRF, performs nearly equally well. Namely, it is possible to achieve $\epsilon = (q+1)2^{-k}$ in 5.9 so that the corresponding pebbles can be broken with a small probability. By the analysis given in 5.8, the extra memory the adversary can retain in the computing phase (see 5.8) can be bounded by $\delta = (8n + 2m + 1)k + \texttt{EXEC\_SIZE}$. Now, combining (5.5) and (5.7) we get the following constraint

$$s - 2nc \geqslant 2n(\delta + 6k + 6) + (12|\widetilde{C}| + 8n + 2m)k + \log m + \texttt{EXEC\_SIZE}$$

. But (5.1) guarantees this condition is met.

Now, we give an outline of how the simulator $\mathcal{S}$ of 5.1 works given 1^n, 1^m, $1^{|C|}$, and an s-space bounded, c-communication bounded adversary $\mathcal{A}^{\mathcal{H}} \in \mathcal{K}$. Plus, $\mathcal{S}$ has access to $\mathcal{H}$. The simulator begins with assembling a uniformly random circuit $C' \colon \{0,1\}^n \to \{0,1\}^m$ of size $|C'| = |C|$. Then, it runs the one-time compiler $\mathrm{Compile}_{k,s}$ on C' obtaining a protocol $\mathcal{P}' = (m, R, L, K, \mathfrak{C}, O, \mathcal{B})$. The simulator maintains two exact copies of $\mathcal{P}'$. In the next step $\mathcal{S}$ starts executing $\mathcal{A}^{\mathcal{H}}$ on a copy of $\mathcal{P}'$, recording each oracle call to $\mathcal{H}$. Depending on what the resulting transcript contains, the simulator picks one of the following paths:

1. There exists at least one index i such that none of the associated values $F^{\mathcal{H}}_{i,R}(0)$ nor $F^{\mathcal{H}}_{i,R}(1)$ has been computed. Then, $\mathcal{S}$ simply outputs a result $\mathcal{A}^{\mathcal{H}}$ has returned.

2. $\mathcal{A}^{\mathcal{H}}$ has broken pebbles (in the sense given in 5.4.3). In this case an outcome of the simulation is again the same as the result $\mathcal{A}^{\mathcal{H}}$ has produced.

3. For each $i = 1, \ldots, n$, the adversary $\mathcal{A}^{\mathcal{H}}$ has issued an output query to $\mathcal{H}$ computing $F^{\mathcal{H}}_{i,R}(b_i)$ either for $b_i = 0$ or $b_i = 1$ (but not both – therefore $\mathcal{A}^{\mathcal{H}}$ has not broken pebbles). As $\mathcal{S}$ has learnt all these values in the process, it can decrypt each of the latchkeys L^{in_i} just to pinpoint for which b_i the function $F^{\mathcal{H}}_{i,R}$ has been computed. All the $F^{\mathcal{H}}_{i,R}(b_i)$'s correspond to a single value $x_{\mathcal{A}} \coloneqq b_1 b_2 \ldots b_n$ that $\mathcal{A}^{\mathcal{H}}$ has committed to by evaluating the extended one-time computable PRF. Thus, $\mathcal{S}$ is also able to find out $x_{\mathcal{A}}$, compute $C'(x_{\mathcal{A}})$ on its own, and query $\mathcal{O}$ on argument $x_{\mathcal{A}}$. Let $\Delta_x \coloneqq C'(x_{\mathcal{A}}) \oplus C(x_{\mathcal{A}})$. If Δ_x happens to be 0^m then $\mathcal{S}$ continues by returning the value $\mathcal{A}^{\mathcal{H}}$ has outputted. Otherwise, the simulator discards this result. Using the latchkeys and querying the oracle $\mathcal{H}$ on $\mathrm{Latch} = L^{\mathrm{in}_1} \oplus \cdots \oplus L^{\mathrm{in}_n}$, the simulator determines the genuine value of $\mathrm{Mask} = \mathcal{H}(\mathrm{Latch})_{|m}$. Then, it reprograms $\mathcal{H}$ so that $\mathcal{H}(\mathrm{Latch})_{|m} \coloneqq \mathrm{Mask} \oplus \Delta_x$. Next, $\mathcal{S}$ rewinds $\mathcal{A}^{\mathcal{H}}$ and runs it again on a

leftover copy of $\mathcal{P}'$ with substituted $\mathcal{H}$. No matter which of the above conditions 1-3 this second execution matches, an output of $\mathcal{A}^{\mathcal{H}}$ becomes the final result of the simulation.

In 5.8 we prove that the output of $\mathcal{S}$ is indistinguishable from a result of $\mathcal{A}^{\mathcal{H}}$ running on $\mathcal{P}$, except for $O(q|\widetilde{C}|2^{-k})$ probability.

5.7 Properties of the Decoder

The below 5.10 asserts that a one-time program can be indeed executed on a device with limited memory.

Lemma 5.10. *For every k, s, a boolean circuit $C\colon \{0,1\}^n \to \{0,1\}^m$, a one-time program $\mathcal{P}$ produced by $\mathrm{Compile}_{k,s}(C)$, and an input x, an execution $\mathcal{D}\mathrm{ec}(x,\mathcal{P})$ requires at most s bits of memory. Moreover, the decoder $\mathcal{D}\mathrm{ec}$ runs in polynomial time in $n \cdot s$, issuing $O(ns/k)$ queries to the random oracle $\mathcal{H}$.*

Over and above that, 5.11 declares that $\mathcal{D}\mathrm{ec}$ fails to evaluate $\mathcal{P}$ at most with a negligible probability. As a matter of fact, the compiler can produce a flawed program prone to ambiguities that may occur when decrypting garbled tables of $\mathfrak{C}$. Like in the case of garbled circuits, this can happen only rarely.

Lemma 5.11. *Let k, s, C, and $\mathcal{P}$ be the same as in 5.10. Then it holds that $\Pr(\mathcal{D}\mathrm{ec}(x,\mathcal{P}) \neq C(x)) = O(|\widetilde{C}|2^{-k})$, where the probability is taken over all possible choices of input $x \in \{0,1\}^n$ and one-time programs $\mathcal{P}$ for C.*

5.8 Proof of the Main Theorem

Below, we provide a pending proof of 5.2 from 5.3.

Proof of 5.2. The existence of a decoder claimed in 5.2 follows from the construction in Sections 5.5 and 5.7.

Now, we argue that the output of $\mathcal{S}$ described in 5.6 is indistinguishable from a result of $\mathcal{A}^{\mathcal{H}}$ running on $\mathcal{P}$, except for a small probability:

$$|\Pr\Big(\mathcal{D}\mathrm{ist}(\mathcal{A}^{\mathcal{H}}(\mathcal{P})\Big) = 1) - \Pr\Big(\mathcal{D}\mathrm{ist}(\mathcal{S}(1^n, 1^m, 1^{|C|}, \mathcal{A}^{\mathcal{H}})) = 1\Big)| \leqslant \epsilon\ , \quad (5.8)$$

where $\mathcal{D}\mathrm{ist}$ is the distinguisher in question, $\epsilon = O(q|\widetilde{C}|2^{-k})$, and the above probabilities range over $\mathcal{P} \leftarrow \mathrm{Compile}_{k,s}(C)$ and all random choices of $\mathcal{H}$, $\mathcal{A}^{\mathcal{H}}$, $\mathcal{S}$, $\mathcal{D}\mathrm{ist}$. Here, we recall that $\mathcal{D}\mathrm{ist}$ is given access to the base circuit C and belongs to the class $\mathcal{K} = \mathcal{K}_q$, i.e., it has *a posteriori* access to $\mathcal{H}$ limited to q queries.

It can be easily seen that except for $O(q2^{-k})$ probability the second simulated execution of $\mathcal{A}^{\mathcal{H}}$ after the rewind is the same as the first one in that $\mathcal{A}^{\mathcal{H}}$ commits to the same input $x_{\mathcal{A}} = b_1 b_2 \ldots b_n$.

We define an auxiliary triple of events $\mathcal{E}_1$, $\mathcal{E}_2$, and $\mathcal{E}_3$, pertinent to the execution $\mathcal{A}^{\mathcal{H}}(\mathcal{P})$, in the following way:

1. $\mathcal{E}_1$ is an event that $\mathcal{A}^{\mathcal{H}}$ has not computed some of the pseudorandom functions of the extended PRF on any input;

2. $\mathcal{E}_2$ is an event that $\mathcal{A}^{\mathcal{H}}$ has broken pebbles;

3. $\mathcal{E}_3$ is an event that $\mathcal{A}^{\mathcal{H}}$ has computed each member function of the extended PRF for exactly one input.

We specify $\mathcal{E}'_1$, $\mathcal{E}'_2$, and $\mathcal{E}'_3$ analogously, with the only difference that primed events refer to the simulated execution of $\mathcal{A}^{\mathcal{H}}$ on $\mathcal{P}'$ (where actions after the rewind are not taken into account). These events clearly correspond to steps 1-3 of the description of $\mathcal{S}$.

The left-hand side of (5.8) can be now rewritten as

$$|\Pr(\mathcal{E}_{\mathcal{D}\mathrm{ist}}) - \Pr(\mathcal{E}'_{\mathcal{D}\mathrm{ist}})| \leqslant \sum_{i=1}^{3} |\Pr(\mathcal{E}_{\mathcal{D}\mathrm{ist}} \mid \mathcal{E}_i)\Pr(\mathcal{E}_i) - \Pr(\mathcal{E}'_{\mathcal{D}\mathrm{ist}} \mid \mathcal{E}'_i)\Pr(\mathcal{E}'_i)| \ ,$$

where we abbreviate the events $\mathcal{D}\mathrm{ist}(\mathcal{A}^{\mathcal{H}}(\mathcal{P})\big) = 1$ and $\mathcal{D}\mathrm{ist}(\mathcal{S}(1^n, 1^m, 1^{|C|}, \mathcal{A}^{\mathcal{H}})) = 1$ by $\mathcal{E}_{\mathcal{D}\mathrm{ist}}$ and $\mathcal{E}'_{\mathcal{D}\mathrm{ist}}$, respectively. We prove that each term of the above sum is $O(q|\widetilde{C}|2^{-k})$. To this end, we establish that

$$|\Pr(\mathcal{E}_i) - \Pr(\mathcal{E}'_i)| = O(q|\widetilde{C}|2^{-k}) \quad \text{for each } i = 1, 2, 3. \tag{5.9}$$

Basically, these formulæ follow from the fact that the adversary himself (note that we do not mention the distinguisher $\mathcal{D}\mathrm{ist}$ here) cannot tell apart $\mathcal{P}$ and $\mathcal{P}'$. Indeed, by the construction of one-time programs for two distinct circuits of equal size, 5.5, and properties of the garbling encryption scheme based on the random oracle $\mathcal{H}$, we have that distributions of both programs are close in statistical distance. Moreover, $\mathcal{P}$ and $\mathcal{P}'$ cannot be distinguished using q queries to $\mathcal{H}$ except for a probability implied by (5.9).

Intuitively, we would like to say even if the results of both executions $\mathcal{A}^{\mathcal{H}}(\mathcal{P})$ and $\mathcal{A}^{\mathcal{H}}(\mathcal{P}')$ are given to a $\mathcal{D}\mathrm{ist}$ then the distinguisher does not have any significant advantage over $\mathcal{A}^{\mathcal{H}}$ in telling the programs apart. By (5.9) the only thing that remains is to estimate $|\Pr(\mathcal{E}_{\mathcal{D}\mathrm{ist}} \mid \mathcal{E}_i) - \Pr(\mathcal{E}'_{\mathcal{D}\mathrm{ist}} \mid \mathcal{E}'_i)|$ for each i. Now, we note that (by applying 5.13 of 5.9) if either $\mathcal{E}_1$ or $\mathcal{E}_2$ (respectively, $\mathcal{E}'_1$ or $\mathcal{E}'_2$ for $\mathcal{A}^{\mathcal{H}}(\mathcal{P}')$) happens then $\mathcal{A}^{\mathcal{H}}$ can hardly learn the value of Master,

and therefore all the input keys of the garbled circuit are indistinguishable, except for a small probability, from random values. From this we can infer the indistiguishability of $\mathcal{P}$ and $\mathcal{P}'$. As for the case when $\mathcal{E}_3$ ($\mathcal{E}_3'$ respectively) happens – here, with a large probability, $\mathcal{A}^{\mathcal{H}}$ can compute only one key (either $K_0^{\text{in}_i}$ or $K_1^{\text{in}_i}$) corresponding to each input wire in_i of the garbled circuit. Having ensured this we obtain the required indistinguishability by 5.12 of 5.9.

□

5.9 Circuit Indistinguishability

The proof of 5.2 depends considerably on the following 5.12. The latter states that two garbled circuits cannot be distinguished if only one key per input wire is known. In that it resembles the proof for garbled circuits given by Pinkas and Lindell [45] who use a standard hybrid argument to show the required indistinguishability.

Lemma 5.12. *Let x be a fixed string. Let C and C' be two boolean circuits with the same number of inputs, outputs, and gates such that $C(x) = C'(x)$. Write $\mathfrak{C}$ and $\mathfrak{C}'$ for lists of garbled gates produced by 1 applied to C and C', respectively. Consider any algorithm $\mathcal{W}^{\mathcal{H}}$ which is allowed q oracle calls to $\mathcal{H}$. Then*

$$|\Pr(\mathcal{W}^{\mathcal{H}}(\mathfrak{C}, C) = 1 \mid \mathcal{E}) - \Pr(\mathcal{W}^{\mathcal{H}}(\mathfrak{C}', C) = 1 \mid \mathcal{E}')| = O(q|C|2^{-k})$$

, where $\mathcal{E}$ (and $\mathcal{E}'$) is an event that for each input wire in_i of $\mathfrak{C}$ ($\mathfrak{C}'$ respectively) the average-conditional min-entropy of at least one input key corresponding to in_i, i.e., $K_0^{in_i}$ or $K_1^{in_i}$, conditioned on a transcript of the random oracle calls made throughout the entire execution $\mathcal{W}^{\mathcal{H}}(\mathfrak{C}, C)$ (and $\mathcal{W}^{\mathcal{H}}(\mathfrak{C}', C)$, respectively), is at least $k - \log q$.

It can be proven that if $\mathfrak{C}$ and $\mathfrak{C}'$ are extended to one-time programs containing $\mathfrak{C}$ and $\mathfrak{C}'$, then the indistinguishability property still holds.

By the below 5.13 asserts that it is infeasible to break pebbles and discover both input keys $K_0^{\text{in}_i}$ or $K_1^{\text{in}_i}$ for any input wire in_i of a garbled circuit.

Lemma 5.13. *Consider any one-time program outputted by* $\text{Compile}_{k,s}$ *with fixed s and k for a circuit with n input wires and m output wires. Let μ be the length of a random key R for the one-time computable PRF included in this program. Set $\delta' = (8n + 2m)k +$ `EXEC_SIZE` and suppose that $\mathcal{A}^{\mathcal{H}}$ breaks pebbles within space of $\mu + \delta' + \delta''$ bits. Then, the probability that $\mathcal{A}^{\mathcal{H}}$ issues a proper call to the oracle computing* $\text{Master} = \mathcal{H}(\text{Latch}, \mathfrak{C})$ *is at most $2^{-\delta''}$.*

Bibliography

[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. D. 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] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In J. Kilian, editor, *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '01, pages 1–18, London, UK, UK, 2001. Springer-Verlag.

[7] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM Conference on Computer and Communications Security*, 1993.

[8] Z. Brakerski and S. Goldwasser. Circular and leakage resilient public-key encryption under subgroup indistinguishability (or: Quadratic residuosity strikes back). CRYPTO, 2010.

[9] Z. Brakerski, Y. T. Kalai, J. Katz, and V. Vaikuntanathan. Cryptography resilient to continual memory leakage. FOCS, 2010.

[10] D. Brumley and D. Boneh. Remote timing attacks are practical. *Comput. Netw.*, 48(5):701–716, 2005.

[11] R. Canetti, S. Halevi, and M. Steiner. Mitigating dictionary attacks on password-protected local storage. In *CRYPTO*, 2006.

[12] D. Cash, Y. Z. Ding, Y. Dodis, W. Lee, R. J. Lipton, and S. Walfish. Intrusion-resilient key exchange in the bounded retrieval model. In *TCC*, 2007.

[13] S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi. Towards sound approaches to counteract ower-analysis attacks. In M. J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999.

[14] G. D. Crescenzo, R. J. Lipton, and S. Walfish. Perfectly secure password protocols in the bounded retrieval model. In *TCC*, 2006.

[15] I. Damgård. A design principle for hash functions. In *CRYPTO*, pages 416–427, 1989.

[16] F. Davì, S. Dziembowski, and D. Venturi. Leakage-resilient storage. SCN, 2010.

[17] Y. Dodis, S. Goldwasser, Y. T. Kalai, C. Peikert, and V. Vaikuntanathan. Public-key encryption schemes with auxiliary inputs. In *TCC*, 2010.

[18] Y. Dodis, K. Haralambiev, A. Lopez-Alt, and D. Wichs. Cryptography against continuous memory attacks. FOCS, 2010.

[19] K. Durnoga, S. Dziembowski, T. Kazana, and M. Zajac. One-time programs with limited memory. In *(submitted to Financial Crypto)*, 2012.

[20] C. Dwork, A. Goldberg, and M. Naor. On memory-bound functions for fighting spam. In *CRYPTO*, 2003.

[21] C. Dwork, M. Naor, and H. Wee. Pebbling and proofs of work. In *CRYPTO*, 2005.

[22] S. Dziembowski. Intrusion-resilience via the bounded-storage model. In *TCC*, 2006.

[23] S. Dziembowski. On forward-secure storage. In *CRYPTO*, 2006.

[24] S. Dziembowski, T. Kazana, and D. Wichs. Key-evolution schemes resilient to space-bounded leakage. In *CRYPTO*, pages 335–353, 2011.

[25] S. Dziembowski, T. Kazana, and D. Wichs. One-time computable self-erasing functions. In *TCC*, pages 125–143, 2011.

[26] S. Dziembowski and U. M. Maurer. Optimal randomizer efficiency in the bounded-storage model. *J. Cryptology*, 17(1), 2004.

[27] S. Dziembowski and K. Pietrzak. Intrusion-resilient secret sharing. In *FOCS*, pages 227–237, 2007.

[28] S. Dziembowski and K. Pietrzak. Leakage-resilient cryptography. In *FOCS*, 2008.

[29] S. Dziembowski and K. Pietrzak. Leakage-resilient cryptography. In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008*, pages 293–302, 2008.

[30] S. Dziembowski, K. Pietrzak, and D. Wichs. Non-malleable codes. In *ICS*, 2010.

[31] S. Faust, E. Kiltz, K. Pietrzak, and G. N. Rothblum. Leakage-resilient signatures. In *TCC*, 2010.

[32] S. Faust, T. Rabin, L. Reyzin, E. Tromer, and V. Vaikuntanathan. Protecting circuits from leakage: the computationally-bounded and noisy cases. to appear in proc. Eurocrypt 2010.

[33] 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.

[34] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. One-time programs. In D. Wagner, editor, *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings*, volume 5157 of *Lecture Notes in Computer Science*, pages 39–56. Springer, 2008.

[35] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. One-time programs. In D. Wagner, editor, *CRYPTO*, volume 5157 of *LNCS*, pages 39–56, 2008.

[36] S. Goldwasser and G. N. Rothblum. Securing computation against continuous leakage. In *CRYPTO*, pages 59–79, 2010.

[37] Y. Ishai, M. Prabhakaran, A. Sahai, and D. Wagner. Private Circuits II: Keeping Secrets in Tamperable Circuits. In *EUROCRYPT*, pages 308–327, 2006.

[38] Y. Ishai, A. Sahai, and D. Wagner. Private Circuits: Securing Hardware against Probing Attacks. In *CRYPTO*, 2003.

[39] K. Järvinen, V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Garbled circuits for leakage-resilience: Hardware implementation and evaluation of one-time programs. In S. Mangard and F.-X. Standaert, editors, *Proceedings of the 12th International Conference on Cryptographic Hardware and Embedded Systems*, CHES'10, pages 383–397, 2010.

[40] A. Juma and Y. Vahlis. Protecting cryptographic keys against continual leakage. In *CRYPTO*, pages 41–58, 2010.

[41] J. Katz and V. Vaikuntanathan. Signature schemes with bounded leakage resilience. In *ASIACRYPT*, pages 703–720, 2009.

[42] P. Kocher. Design and validation strategies for obtaining assurance in countermeasures to power analysis and related attacks. *NIST Physical Security Testing Workshop*, 2005. Available at `www.smartcard.co.uk/DPAValidation.pdf`.

[43] P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *CRYPTO*, pages 388–397, 1999.

[44] M. G. Kuhn. *Compromising emanations: eavesdropping risks of computer displays*. PhD thesis, University of Cambridge, 2003. Technical Report UCAM-CL-TR-577.

[45] Y. Lindell and B. Pinkas. A proof of security of Yao's protocol for two-party computation. *J. Cryptol.*, 22(2):161–188, Apr. 2009.

[46] F.-H. Liu and A. Lysyanskaya. Algorithmic tamper-proof security under probing attacks. In *SCN*, 2010.

[47] U. M. Maurer. Conditionally-perfect secrecy and a provably-secure randomized cipher. *J. Cryptology*, 5(1), 1992.

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

[49] S. Micali and L. Reyzin. Physically observable cryptography (extended abstract). In M. Naor, editor, *TCC*, volume 2951 of *Lecture Notes in Computer Science*, pages 278–296. Springer, 2004.

[50] M. Naor and G. Segev. Public-key cryptosystems resilient to key leakage. In *Advances in Cryptology - CRYPTO*, August 2009.

[51] E. N. of Excellence (ECRYPT). The side channel cryptanalysis lounge. http://www.crypto.ruhr-uni-bochum.de/en_sclounge.html. retrieved on 7.04.2010.

[52] D. Perito and G. Tsudik. Secure code update for embedded devices via proofs of secure erasure. In *ESORICS*, 2010.

[53] K. Pietrzak. A leakage-resilient mode of operation. In *EUROCRYPT*, 2009.

[54] K. Pietrzak. A leakage-resilient mode of operation. In A. 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.

[55] J.-J. Quisquater and D. Samyde. Electromagnetic analysis (ema): Measures and counter-measures for smart cards. In *E-smart*, pages 200–210, 2001.

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

[57] A. Shamir and E. Tromer. Acoustic cryptanalysis. on nosy people and noisy machines. A webpage: http://people.csail.mit.edu/tromer/acoustic/ accessed on 27.05.2009.

[58] F.-X. Standaert, T. Malkin, and M. Yung. A unified framework for the analysis of side-channel key recovery attacks. In *EUROCRYPT*, 2009.

[59] L. von Ahn, M. Blum, N. J. Hopper, and J. Langford. Captcha: Using hard ai problems for security. In *EUROCRYPT*, 2003.

[60] A. C.-C. Yao. How to generate and exchange secrets. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, SFCS '86, pages 162–167, 1986.

[61] Y. Yu, J. Leiwo, and B. Premkumar. Hiding circuit topology from unbounded reverse engineers. In L. M. Batten and R. Safavi-Naini, editors, *Proceedings of the 11th Australasian conference on Information Security and Privacy*, ACISP'06, pages 171–182, 2006.

[62] Y. Yu, F.-X. Standaert, O. Pereira, and M. Yung. Practical leakage-resilient pseudorandom generators. In *CCS: ACM Conference on Computer and Communications Security.*, 2010. To Appear.