One-time Programs with Limited Memory [⋆]

K. Durnoga[1], S. Dziembowski[1,2], T. Kazana[1], and M. Zając[1]

[1] University of Warsaw
[2] Sapienza University of Rome

Abstract. We reinvestigate a notion of *one-time programs* introduced in the CRYPTO 2008 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 does not rely on the hardware gadgets. Instead, it 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 essentially the model used recently by Dziembowski *et al.* (TCC 2011, CRYPTO 2011) to construct one-time computable *pseudorandom* functions and key-evolution schemes.

Keywords: pseudorandom functions, one-time device, one-time program, circuit garbling.

1 Introduction

A notion of *one-time programs* was introduced by Goldwasser *et al.* [13]. Informally speaking, a one-time program is a device 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 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 D. Indeed, if we assume that the entire contents $\mathcal{P}$ of D can be read freely, then an adversary can create his own copies

[⋆] This work was partly supported by the WELCOME/2010-4/2 grant founded within the framework of the EU Innovative Economy (National Cohesion Strategy) Operational Programme. The European Research Council has provided financial support for this work under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement no CNTM-207908.

of 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 D is fully-trusted, i.e. the adversary cannot read or modify its contents. Obviously, then one can simply put any program C on 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 [11] attacks have demonstrated that in real-life constructing 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.* [13] is based on the following physical assumption: they assume that D is equipped with special gadgets that they call *one-time memory (OTM)* devices. At the deployment of D an OTM can be initialized with a pair of values (K_0, K_1). The program $\mathcal{P}$ that is stored on 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.* [13] replace the unrealistic assumption that the whole device D is fully secure, with a much weaker one that the OTM gadgets on D are secure. Here, 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 D are secure, and it is natural to look for other, perhaps more realistic, models where the transformation similar to the one of [13] would be possible. In this paper we propose such an alternative model, inspired by recent work of Dziembowski *et al.* [8] 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 D, but rather impose "global" restrictions on what kind of attacks are possible.

To explain and motivate the use of the model of [8] 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 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 paper, 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 [17]. Later, they were used to define the memory leakage occurring during the virus attacks in the bounded-retrieval model [2, 5, 6]. In the context of the side-channel leakages they were first used by Dziembowski and Pietrzak [9] 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 [12], or they are so strong that they permit only very limited constructions [10].

As mentioned before, in this paper we follow the approach of [8], 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 D, that can be used by anybody who performs computations on 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 describing 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 [8] 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[3]. It can perform any computation on D subject to the constraint that it cannot use more memory than s. The fact that it can leak information to the outside is

[3] In the work of Dziembowski *et al.* [8] the adversary $\mathcal{A}_{small}$ is used to model the malicious code executed on the device (e.g. a computer virus), while in our case it models the tampering function. While in real life mallware is usually much more powerful than hardware tampering functions, we adopt the model of [8] since we do not see any other natural restriction on the tampering function that would lead to better parameters or the simpler proofs.

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 paper. Our main result (see Theorem 1) 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 Dziembowski *et al.* [8], our construction works in the Random Oracle Model, where we model as random oracles hash functions of fixed input lengths. For a complete statement of our result see Theorem 1. Let us only remark here that for a fixed circuit we get that the security holds as long as $s - 2nc \geq \gamma k$, 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, which may be sufficient for practical applications where n is small, e.g., if the input is a human-memorized PIN. 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.* [13] was analyzed by Jarvinen *et al.* [14]. The model of Dziembowski *et al.* [8] and related techniques were also used in a subsequent paper [7] to construct leakage-resilient key-evolution schemes. Finally, let us note that the main difference between [8] and our work is that in [8] 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.

It has been recently pointed out by Bellare *et al.* [4] that the original security proof of [13] had a gap. Informally speaking, this was due to the fact that the scheme of [13] was based on a statically-secure Yao garbled circuit, and hence did not provide security against the adversaries that can modify the input during the computation. We note that this problem does not affect the security of our construction. We elaborate more on this in Section 5.2.

Organization of the Paper. Some basic definitions we refer to later on are listed in Section 2. In Section 3, we give a formal statement of what we mean by a one-time device. Also, we announce the main theorem of the paper asserting that our construction produces programs compliant with this definition. Several tools we extensively use throughout the paper are synopsized in Section 4. These include: circuit garbling [16, 19], universal circuits [15, 18], and one-time computable pseudorandom functions [8]. In Section 5, we describe a compiler that converts a boolean circuit to a one-time device. A proof-sketch of the main theorem from Section 3 follows in Section 6.

2 Preliminaries

Across the paper, 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. 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 $|C|$.

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 $\mathcal{A}$ (modeled 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. Although the declaration of $\mathcal{H}$ assumes that $\mathcal{H}$ accepts arguments of an arbitrary length, we only apply $\mathcal{H}$ to inputs not longer that a fixed multiple of k except for one case. In this particular case, however, the long input can be split into smaller chunks which allows cascading of $\mathcal{H}$. Overall, we can invoke the oracle only for short inputs.

When typesetting algorithms, we write $R \overset{\$}{\leftarrow} S$ for sampling a uniformly random value from some set S and assigning it to a variable R. We assume that every such a 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 Section 1, the model we adopt in the paper 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 the 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.

3 One-time Programs

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. Additionally, the black-box should not leak any information about C whatsoever. As explained in Section 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. Definition 1 makes this property formal in terms of a simulator that is permitted to call an oracle evaluating C only once.

Definition 1. *Let c, s, δ, q, and ϵ be parameters. Let $C\colon \{0,1\}^n \to \{0,1\}^m$ be a boolean circuit with positive integers n and m. Write $\mathcal{O}$ for an oracle that computes $C(x)$ given $x \in \{0,1\}^n$. Consider an algorithm $\mathcal{A} = (\mathcal{A}_{big}, \mathcal{A}_{small})$ which is $(s+\delta)$-bounded in space, c-bounded in communication, and is allowed at most q calls to the random oracle $\mathcal{H}$. A string $\mathcal{P}$ is called a $(c, s, \delta, q, \epsilon)$*–one-time program *for C if both of the following conditions hold:*

- *there exists a probabilistic polynomial-time decoder $\mathcal{D}$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 restricted to at most q oracle calls to $\mathcal{H}$ can distinguish $\mathcal{S}(1^n, 1^m, 1^{|C|}, \mathcal{A})$ and $\mathcal{A}(\mathcal{P})$ with a probability greater than ϵ.*

Basically, the definition states that a user can honestly execute a device containing a one-time program on a single input of his choice. Yet, even for a computationally unbounded adversary $\mathcal{A} = (\mathcal{A}_{big}, \mathcal{A}_{small})$, with $\mathcal{A}_{small}$ having extra δ bits of memory, it is infeasible to break the device. We note that the one-time property formulated above is slightly stronger than what one may need for the applications. For instance, it could be safe to give the adversary some partial information about the circuit (e.g. information about a single boolean gate). In our definition, we disallow adversary to find out anything more than n, m, $|C|$, and $C(x)$ for a single x. We also remark that the definition provides adaptive security, i.e., the adversary can freely choose x depending on the contents of $\mathcal{P}$.

Shortly, in Section 5.3, we construct a compiler $\text{Compile}_{k,s}(C)$ that, for some parameter k, converts any boolean circuit C to a one-time program $\mathcal{P}$ that can be organized into a device with s bits of memory. The main result of this paper is stated in the below Theorem 1 about $\text{Compile}_{k,s}(C)$. The theorem contains a reference to circuits of uniform topology. A uniform version of C, denoted $\widetilde{C}$, is produced by the algorithm of Kolesnikov and Schneider [15], which is discussed in Section 4.2. Transforming C to such a form introduces a small blow-up factor (see (4) below) so that $\widetilde{C}$ is slightly larger than C.

Theorem 1. *Let k be a security parameter and let $\mathcal{H}\colon \{0,1\}^* \to \{0,1\}^k$ be modeled as a random oracle. Then, for any boolean circuit $C\colon \{0,1\}^n \to \{0,1\}^m$ and $\mathcal{P} \leftarrow \text{Compile}_{k,s}(C)$, the string $\mathcal{P}$ is a $(c, s, \delta, q, \epsilon)$–one-time program for C with $\epsilon = O(q|\widetilde{C}|2^{-k})$, provided that $k \geq \max(m, 4n^2 \log q)$ and*

$$s - 2nc \geq 2n\delta + 6k(2|\widetilde{C}|\log|\widetilde{C}| + 5n^2 + 4nm) \ , \tag{1}$$

where $|\widetilde{C}|$ denotes the number of gates in $\widetilde{C}$ – a version of C with uniform topology.

Remark 1. 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 $\mathcal{S}$. We also mention that with a minor modification of our construction we can replace the factor $2n$ on the left-hand side of (1) with $2n/\log n$. We do not present this modification here as it would make the proofs considerably more complicated.

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.

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 [19] 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. Since the circuit garbling seems to be well-known, we skip its description here and only give the minimal relevant excerpt just to fix the notation.

Let k be a security parameter. For a boolean circuit C the garbling procedure $\mathrm{Garble}_k(C)$ associates two random strings K_0^w and K_1^w of length k with each wire w of C. These two keys correspond to bits 0 and 1, respectively, that could appear on the wire w when evaluating C in its plain form. The mapping between input and output keys for each gate is masked using an auxiliary encryption scheme (E, D). We call it *a garbling encryption* scheme. It enjoys some extra properties, given by Pinkas and Lindell [16], going a little beyond the 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 instantiate E_K using the following setting, compliant with the requirements listed by Pinkas and Lindell, based on the oracle $\mathcal{H}$:

$$E_K(M) \coloneqq (\mathcal{H}(K), r, \mathcal{H}(K, r) \oplus M) \quad \text{where } r \xleftarrow{\$} \{0,1\}^k. \tag{2}$$

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 by Pinkas and Lindell for technical reasons, we specify $E_{K_1;K_2}(\cdot)$ separately extending (2) with:

$$E_{K_1;K_2}(M) \coloneqq (\mathcal{H}(K_1, K_2), r, \mathcal{H}(K_1, K_2, r) \oplus M) \quad \text{for } r \xleftarrow{\$} \{0,1\}^k. \tag{3}$$

In the remainder of this paper we assume that ciphertexts in a garbling encryption scheme are all of length $3k$ as implied by (2) and (3).

Below, we assume that the garbling procedure outputs a triple $(I, \mathfrak{C}, O)$, where $\mathfrak{C}$ is the actual garbled circuit, while I and O are arrays mapping plain bits to keys for input and output wires. $\mathfrak{C}$ is just a list of encrypted keys and each ciphertext on that list was produced using the double-encryption (3). Closely related to $\mathrm{Garble}_k(C)$ is the procedure for evaluating the garbled circuit $\mathfrak{C}$ on a given input x. We write $\mathrm{Eval}(\mathfrak{C}, O, K_x)$ to name this procedure.

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.* [3] 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. There are several works describing methods for hiding circuit topology [15, 18, 20]. The most recognized one, which is asymptotically optimal in terms of additional overhead it incurs, comes from Valiant [18]. Recently, Vladimir Kolesnikov has pointed out to us that his construction [15], while being slightly worse asymptotically than Valiant's, achieves a better implied constant and thus performs better for small circuits. We recall his joint result on topology erasing algorithm UniformCircuit(C) below.

Theorem 2 (Kolesnikov and Schneider [15]). *Let* $C\colon \{0,1\}^n \to \{0,1\}^m$ *be a boolean circuit. Then, the topology erasing algorithm* UniformCircuit(C) *constructs a circuit* $\widetilde{C}$ *with*

$$|\widetilde{C}| = \big(1 + o(1)\big) \cdot |C| \log^2 |C| \ . \tag{4}$$

such that $\widetilde{C}(x) = C(x)$ *for all* $x \in \{0,1\}^n$ *and the topology of* C *(i.e., the connectivity graph of* C *where each gate is stripped of information about what functionality it actually implements) discloses (in the information-theoretic sense) nothing more than* n*,* m*, and* $|C|$*.*

The below Proposition 1 follows from the analysis given by Kolesnikov and Schneider.

Proposition 1. *The algorithm* UniformCircuit(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.*

4.3 One-Time Computable Pseudorandom Functions (PRFs)

A notion of the one-time computable pseudorandom functions was introduced by Dziembowski *et al.* [8]. 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 can 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. Below, we borrow some basic definitions from the original paper to formalize 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 PRFs* 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 2 (Dziembowski *et al.* **[8]).** *An algorithm $\mathcal{W}^{\mathcal{H}}$ with at most q queries to the oracle $\mathcal{H}$ defines $(c,\mu,\sigma,q,\epsilon,n)$*–one-time computable PRFs *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 PRFs, is at most ϵ.*

Basically, what this definition states is that no adversary with σ-bounded storage and c-bounded communication can viably compute a value of any $F^{\mathcal{H}}_{i,R}$ on two distinct inputs. Dziembowski *et al.* [8] prove the existence of the one-time computable PRFs in the Random Oracle model under some plausible assumption on parameters c, μ, σ, q, ϵ, and n.

The use case we investigate in the paper requires a slightly stronger primitive than the PRFs of Definition 2. In this work, we introduce *extended one-time computable PRFs*. 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 PRFs 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 $((F^{\mathcal{H}}_{i,R}(x_i),(i,x_i)),\texttt{out})$ 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 Definition 3 strengthens the notion of one-time computable PRFs.

Definition 3. *An algorithm $\mathcal{W}^{\mathcal{H}}$ defines extended $(c,\mu,\sigma,q,\epsilon,n)$*–one-time computable PRFs *if:*

- *$\mathcal{W}^{\mathcal{H}}$ defines $(c,\mu,\sigma,q,\epsilon,n)$*–one-time computable PRFs*;*
- *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 PRFs, is at most ϵ.*

In full version of the paper, we verify that the theorem about the extended one-time computable PRFs holds with essentially the same parameters as in the base theorem by Dziembowski *et al.* [8]. Here, we present one more result about the existence of the extended PRFs that stems from the one proven in the full version and provides a condition which is more convenient to use in our particular application.

Theorem 3. *Let* c, μ, δ, q, *and* n *be positive integers. Then, for any* $\epsilon \leq q2^{-4n^2}$, *there exist extended* $(c, \mu, \mu+\tau, q, \epsilon, n)$*–one-time computable PRFs, provided that*

$$\mu \geq 2n \cdot (\tau + c + 4\log q + 6\log \epsilon^{-1} + 6) \ . \tag{5}$$

A proof of Theorem 3 appears in full version of the paper.

5 The Construction

In this section, we give a high-level description of what a one-time device is made up of. Our solution, in principle, combines the hardware-based construction [13] and the primitive developed by Dziembowski *et al.* [8]. We replace the OTM units present in the former work with the extended one-time computable PRFs to achieve a purely software-based construction. There are, however, certain subtleties that occur when attempting to compose these both worlds together. Before proceeding to the correct construction we ultimately propose, we detail why a security proof for the most straightforward solution simply does not work out of the box.

5.1 Naïve Approach

A simple composition of techniques outlined in Section 4 one might conceive of could be the following. Garble a circuit as per Yao's method in the same way as it is done by Goldwasser *et al.* [13], and conceal its input keys using one-time computable PRFs. That is, let $K_0^{\mathrm{in}_i}$ and $K_1^{\mathrm{in}_i}$ be two keys corresponding to the ith input wire of the garbled circuit. Pick a long random string R and calculate both $F_{i,R}^{\mathcal{H}}(0)$ and $F_{i,R}^{\mathcal{H}}(1)$ for each member function $F_{i,R}^{\mathcal{H}}$ of the one-time computable PRFs. Then, a one-time device can store just the garbled circuit, the key R together with both $K_0^{\mathrm{in}_i} \oplus F_{i,R}^{\mathcal{H}}(0)$ and $K_1^{\mathrm{in}_i} \oplus F_{i,R}^{\mathcal{H}}(1)$ for each i. Intuitively, since the one-time PRFs only allow any space restricted algorithm to discover each $F_{i,R}^{\mathcal{H}}(b_i)$ for a single bit $b_i = 0$ or 1, keeping its counterpart $F_{i,R}^{\mathcal{H}}(\overline{b_i})$ entirely random, we can guarantee that such an algorithm can learn at most one input key $K_0^{\mathrm{in}_i}$ or $K_1^{\mathrm{in}_i}$. In that way we simulate the OTM gadgets and the original reasoning [13] should apply from this point. This would seemingly satisfy the requirements of Definition 1.

However, there are several problems arising in the above construction. Firstly, there is more space available on a device than just space needed to store the key R for the one-time computable PRFs. For instance, the garbled circuit resides in this additional memory. The extra space could be possibly used by an adversary to break PRFs, i.e., to compute both $F_{i,R}^{\mathcal{H}}(0)$ and $F_{i,R}^{\mathcal{H}}(1)$ for some i. There are several ways to fix this issue. Perhaps the most basic and the cleanest one is asserting that the garbled circuit is read-only. This is not a viable option for us as long as we aim at a solution that does not assume any tamper- nor even leakage-resistant components. Another way to circumvent the problem would be increasing the amount of free memory (cf. the parameter δ in Definition 1) available to an adversary, which would, however, worsen the parameters in Theorem 1 substantially. We take a different path and, in fact, ensure that an adversary may not erase the garbled circuit partially, reuse the claimed memory to break PRFs, and then still be able to evaluate the circuit. What makes establishing this property a bit tricky is an observation that a limited erasure is always possible, e.g., a constant number of bits from the circuit can be safely dropped and then guessed back correctly with large enough probability. A new element we introduce in the construction secures that an adversary cannot reliably do more than that. This is made formal in full version of the paper.

An important consequence of putting the garbled circuit into writable memory is that the basic one-time computable PRFs, as given by Definition 2, fall short of providing suitable security. The reason is that once an adversary computes each member function of the PRFs honestly for a single input and evaluates the circuit, then it can erase the circuit and, again, break the PRFs soon after using the increased amount of memory. This justifies turning to the extended one-time computable PRFs of Definition 3. Studying the details of the construction by Dziembowski *et al.* it is not hard to notice that their PRFs effectively self-destruct themselves when evaluating and thus prevent any further evaluations even by space unrestricted algorithms. This makes a transition to the extended PRFs rather straightforward.

Finally, it is not evident whether such a vague construction provides adaptive security or suffers from the same issue Bellare *et al.* [4] identified in the work of Goldwasser *et al.* [13].

5.2 One-time Device

Our concluding construction of one-time programs does not differ significantly from the basic idea sketched above. Therefore, it involves garbling a circuit and masking its input keys with the extended one-time computable PRFs. However, it features an additional layer between these two components which is needed to address the issues we have mentioned. This auxiliary element can be viewed as a simple all-or-nothing transform. The main purpose it serves is splitting each execution of a one-time program into two phases. In the first stage any user, an honest or a malicious one, has to commit himself to the entire input he intends to compute the program on. The garbled circuit can be evaluated in the second phase, yet this process cannot begin before input bits for *all* the input wires are decided.

The exact way how such a separation can be accomplished is not very complex. Each input wire of the circuit is associated with an additional random key. We refer to a set of these keys as to *latchkeys*. Then, every latchkey gets encrypted using the outputs of the extended PRFs. We ensure that the ith latchkey can be recovered given $F^{\mathcal{H}}_{i,R}(0)$ or $F^{\mathcal{H}}_{i,R}(1)$. Lastly, we apply n-out-of-n secret sharing, where each latchkey forms a share, and combine the resulting secret with the garbled circuit using the random oracle $\mathcal{H}$. This produces a new string which we call the *master key*. Intuitively, by the property of $\mathcal{H}$, this value cannot be determined without all the latchkeys and the circuit. We now require the master key to be known to anyone attempting to discover the actual input keys of the garbled circuit. Technically, we adjust the naïve approach and replace the one-time pad encryption keys $F^{\mathcal{H}}_{i,R}(0)$ and $F^{\mathcal{H}}_{i,R}(1)$ present there with the keys that also depend on the master key.

There are two goals we achieve with this transform. First, the garbled circuit cannot be partially erased during the first phase as this would make computing the master key impossible, block opening all its input keys and render the circuit unusable. Second, it makes our construction immune to the attack devised by Bellare *et al.* [4]. The fact that the one-time programs of Goldwasser *et al.* permit what Bellare *et al.* call *partial evaluations* is the fundamental reason that makes these programs susceptible to the attack. In our construction a user cannot attempt to evaluate the circuit if he has learnt keys corresponding only to a proper subset of input wires. In other words, partial evaluations are not feasible. Also, we note that Bellare *et al.* consider a family garbling schemes and construct an artificial scheme for which the proof of Goldwasser *et al.* fails. We, in turn, use only a single, explicitly defined garbling scheme (3). It leaves no room for attaching any security-exploiting superfluous data as Bellare *et al.* do. Finally, our garbled circuits look entirely random to any adversary who does not know the input keys. Seeing this random string does not help the adversary in choosing his input.

Overall, one-time devices we propose contain the following data:

– a garbled circuit $\mathfrak{C}$ together with a table O mapping output keys of the circuit back to plain bits;
– a random key R that determines the extended one-time computable PRFs;
– an array L of encrypted latchkeys;
– an array K consisting of one-time pad encrypted input keys for the garbled circuit $\mathfrak{C}$ – the encryption keys depend on all the latchkeys and $\mathfrak{C}$;
– the number m of output wires of the original circuit.

5.3 One-time Compiler

The 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. $\mathrm{Compile}_{k,s}$ 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. Algorithm 1 presents a listing of the one-time compiler procedure.

Firstly, the compiler prepares (Lines 2 and 3 of Algorithm 1) a set of random latchkeys L^{in_i}. A value L^{in_i} corresponds to the ith input wire in_i of C. A string

Algorithm 1 One-time compiler $\mathrm{Compile}_{k,s}(C)$

```
    INPUT: a boolean circuit $C\colon \{0,1\}^n \to \{0,1\}^m$, a security parameter $k \geq m$,
    a total amount of memory on the device $s$
    OUTPUT: a one-time program $\mathcal{P} = (m, R, L, K, \mathfrak{C}, O)$
 1: procedure $\mathrm{Compile}_{k,s}(C)$
 2:     for $i \leftarrow 1$ to $n$ do
 3:         $L^{\mathrm{in}_i} \xleftarrow{\$} \{0,1\}^k$
 4:     $\mathrm{Latch} \leftarrow L^{\mathrm{in}_1} \oplus \cdots \oplus L^{\mathrm{in}_n}$
 5:     $\mathrm{Mask} \leftarrow \mathcal{H}(\mathrm{Latch})_{|m}$
 6:     $\widetilde{C} \leftarrow \mathrm{UniformCircuit}(C \oplus \mathrm{Mask})$
 7:     $(I, \mathfrak{C}, O) \leftarrow \mathrm{Garble}_k(\widetilde{C})$
 8:     $\mathrm{Master} \leftarrow \mathcal{H}(\mathrm{Latch}, \mathfrak{C})$
 9:     $\mu \leftarrow s - (12|\widetilde{C}| + 8n + 2m)k - \log m$
10:     round $\mu$ down to the largest multiple of $k$
11:     $R \xleftarrow{\$} \{0,1\}^\mu$
12:     for each input wire $\mathrm{in}_i$ of $C$ do          ▷ $\mathrm{in}_i$ is the $i$th input wire of $C$
13:         $(K_0^{\mathrm{in}_i}, K_1^{\mathrm{in}_i}) \leftarrow I[i]$
14:         compute $F_{i,R}^{\mathcal{H}}(0)$ and $F_{i,R}^{\mathcal{H}}(1)$
15:         $L[i] \leftarrow \big(E_{F_{i,R}^{\mathcal{H}}(0)}(L^{\mathrm{in}_i}), E_{F_{i,R}^{\mathcal{H}}(1)}(L^{\mathrm{in}_i})\big)$
16:         $K[i] \leftarrow \big(K_0^{\mathrm{in}_i} \oplus \mathcal{H}(F_{i,R}^{\mathcal{H}}(0), \mathrm{Master}), K_1^{\mathrm{in}_i} \oplus \mathcal{H}(F_{i,R}^{\mathcal{H}}(1), \mathrm{Master})\big)$
17:     end for each
18:     return $(m, R, L, K, \mathfrak{C}, O)$
19: end procedure
```

$\mathrm{Latch} := L^{\mathrm{in}_1} \oplus \cdots \oplus L^{\mathrm{in}_n}$ combines all the latchkeys into a single key. From Latch we derive (Line 5), by means of the oracle, one more random value, denoted Mask, trimming the output of $\mathcal{H}$ to the leading $m \leq k$ bits. The exact role that all these auxiliary components play should become clear later, in Section 6. Having calculated these values, the compiler enters its main phase in Line 6. There, the obfuscation algorithm is run, yet on a biased version of C, say C^*, defined as $C^*(x) := C(x) \oplus \mathrm{Mask}$. 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 Theorem 1 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 Section 6.

The obfuscated circuit is garbled (Line 7) using Yao's method. Next, extended one-time computable PRFs (in the sense of Definition 3) are set up (Line 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 extended one-time computable PRFs are a primitive that protects input keys of the garbled circuit. Namely, in order to evaluate a one-time program on some input $x = b_1 b_2 \ldots b_n$, one has to compute each $F^{\mathcal{H}}_{i,R}(b_i)$ for $i = 1, \ldots, n$. 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 $\text{Compile}_{k,s}$ 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 Line 15 where these ciphertexts are accumulated in array L, we use the garbling encryption scheme as given by (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 $K^{\text{in}_i}_b$ under a key $\mathcal{H}(F^{\mathcal{H}}_{i,R}(b), \text{Master})$. Here, $\text{Master} \coloneqq \mathcal{H}(\text{Latch}, \mathfrak{C})$ 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 $K^{\text{in}_i}_b$ than he has computed *all* $F^{\mathcal{H}}_{i,R}(b_i)$. 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 PRFs. In this way we control the amount of free memory during the computing phase specified in Definition 3.

Now that we have described the one-time compiler, we present a decoder $\mathcal{D}\text{ec} = \mathcal{D}\text{ec}_k$ which is capable of evaluating a program produced by $\text{Compile}_{k,s}$ on an arbitrary input $x = b_1 b_2 \ldots b_n$. As the first step, $\mathcal{D}\text{ec}_k$ determines $F^{\mathcal{H}}_{i,R}(b_i)$ for each $i = 1, \ldots, n$. This is accomplished by computing labels of output vertices under a *random oracle labeling* of a certain on-line constructed graph (the exact method follows from the work of Dziembowski *et al.* [8]). 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 $\mathcal{D}\text{ec}$. Next, the decoder decrypts a matching entry of each $L[i]$ to find L^{in_i}. Based on these latchkeys, $\mathcal{D}\text{ec}$ computes Latch, $\text{Mask} = \mathcal{H}(\text{Latch})_{|m}$, $\text{Master} = \mathcal{H}(\text{Latch}, \mathfrak{C})$, and reveals, using $K[i]$, input garbled keys $K^{\text{in}_i}_{b_i}$ that correspond to each bit b_i. Let K_x be a vector consisting of all $K^{\text{in}_i}_{b_i}$. The decoder then executes $\text{Eval}(\mathfrak{C}, O, K_x)$ 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, $\mathcal{D}\text{ec}$ simulates the topology erasing algorithm on such an arbitrarily chosen circuit. A memory that has to be supplied by $\mathcal{D}\text{ec}$ for this step is located exactly in the same region the key R was previously stored in. By Proposition 1, this space, which is considered free after computing the extended PRFs, has a sufficient size if $\mu = |R| \geq 4|\widetilde{C}| \log |\widetilde{C}|$. 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

$$|\mathcal{P}| = \mu + (12|\widetilde{C}| + 8n + 2m) \cdot k + \log m \ . \tag{6}$$

6 Universal Simulator for One-time Programs

In this section, we focus on the more intricate part of Definition 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.* [13]. A notable difference, however, is that our construction includes a component, i.e., the extended one-time computable PRFs, which does not offer a black-box security, in opposition to the aforementioned OTMs. The condition (1) of Theorem 1 ensures that our replacement of the OTMs performs nearly equally well. Namely, it is possible to achieve $\epsilon = (q+1)2^{-k}$ in Theorem 3 so that the corresponding extended one-time computable PRFs can only be broken with a small probability. By the analysis given in full version of the paper, the extra memory the adversary can retain in the computing phase (see Definition 3) can be bounded above by $\tau = \delta + (8n + 2m + 3)k$. Now, combining (5) and (6) we get the following constraint

$$s - 2nc \geq 2n(\delta + \tau + 6k + 6) + (12|\widetilde{C}| + 8n + 2m)k + \log m \tag{7}$$

But (1) guarantees this condition is met.

Now, we give an outline of how the simulator $\mathcal{S}$ of Definition 1 works given 1^n, 1^m, $1^{|C|}$, and an $(s+\delta)$-space bounded, c-communication bounded adversary $\mathcal{A}^{\mathcal{H}}$. 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 $\mathsf{Compile}_{k,s}$ on C' obtaining a protocol $\mathcal{P}' = (m, R, L, K, \mathfrak{C}, O)$. 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 the PRFs (in the sense given in Section 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 PRFs). 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 PRFs. 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 Latch $= L^{\text{in}_1} \oplus \cdots \oplus L^{\text{in}_n}$, the simulator determines the genuine value of Mask $= \mathcal{H}(\text{Latch})_{|m}$. Then, it reprograms $\mathcal{H}$ so that $\mathcal{H}(\text{Latch})_{|m} := \text{Mask} \oplus \Delta_x$. 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 full version of the paper 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.

References

1. Adi Akavia, Shafi Goldwasser, and Vinod Vaikuntanathan. Simultaneous hardcore bits and cryptography against memory attacks. In *Theory of Cryptography, 6th Theory of Cryptography Conference, TCC 2009, San Francisco, CA, USA, March 15-17, 2009. Proceedings*, LNCS, pages 474–495, 2009.
2. Joël Alwen, Yevgeniy Dodis, and Daniel Wichs. Leakage-resilient public-key cryptography in the bounded-retrieval model. In Shai Halevi, editor, *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference. Proceedings*, volume 5677 of *LNCS*, pages 36–54. Springer, 2009.
3. Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In Joe 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.
4. Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Adaptively secure garbling with applications to one-time programs and secure outsourcing. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology – ASIACRYPT 2012*, volume 7658 of *Lecture Notes in Computer Science*, pages 134–153. Springer Berlin Heidelberg, 2012.
5. Giovanni Di Crescenzo, Richard J. Lipton, and Shabsi Walfish. Perfectly secure password protocols in the bounded retrieval model. In *Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings*, volume 3876 of *LNCS*, pages 225–244, 2006.
6. Stefan Dziembowski. Intrusion-resilience via the bounded-storage model. In *Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings*, volume 3876 of *LNCS*, pages 207–224, 2006.
7. Stefan Dziembowski, Tomasz Kazana, and Daniel Wichs. Key-evolution schemes resilient to space-bounded leakage. In Phillip Rogaway, editor, *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference. Proceedings*, volume 6841 of *LNCS*, pages 335–353. Springer, 2011.
8. Stefan Dziembowski, Tomasz Kazana, and Daniel Wichs. One-time computable self-erasing functions. In Yuval Ishai, editor, *Proceedings of the 8th conference on Theory of cryptography*, TCC'11, pages 125–143, 2011.
9. Stefan Dziembowski and Krzysztof Pietrzak. Leakage-resilient cryptography. In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008*, pages 293–302, 2008.

10. Stefan Dziembowski, Krzysztof Pietrzak, and Daniel Wichs. Non-malleable codes. In Andrew Chi-Chih Yao, editor, *Innovations in Computer Science - ICS 2010. Proceedings*, pages 434–452. Tsinghua University Press, 2010.
11. ECRYPT. Side channel cryptoanalysis lounge http://www.emsec.rub.de/research/projects/sclounge/.
12. Rosario Gennaro, Anna Lysyanskaya, Tal Malkin, Silvio Micali, and Tal Rabin. Algorithmic tamper-proof (atp) security: Theoretical foundations for security against hardware tampering. In Moni Naor, editor, *Theory of Cryptography, First Theory of Cryptography Conference, TCC 2004, Proceedings*, volume 2951 of *LNCS*, pages 258–277, 2004.
13. Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. One-time programs. In David Wagner, editor, *CRYPTO*, volume 5157 of *LNCS*, pages 39–56, 2008.
14. Kimmo Järvinen, Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. Garbled circuits for leakage-resilience: Hardware implementation and evaluation of one-time programs. In Stefan Mangard and François-Xavier Standaert, editors, *Proceedings of the 12th International Conference on Cryptographic Hardware and Embedded Systems*, CHES'10, pages 383–397, 2010.
15. Vladimir Kolesnikov and Thomas Schneider. Financial cryptography and data security, 12th international conference, fc 2008, cozumel, mexico, january 28-31, 2008, revised selected papers. In Gene Tsudik, editor, *Financial Cryptography*, volume 5143 of *Lecture Notes in Computer Science*, pages 83–97. Springer, 2008.
16. Yehuda Lindell and Benny Pinkas. A proof of security of Yao's protocol for two-party computation. *J. Cryptol.*, 22(2):161–188, April 2009.
17. Ueli M. Maurer. Conditionally-perfect secrecy and a provably-secure randomized cipher. *J. Cryptology*, 5(1):53–66, 1992.
18. Leslie G. Valiant. Universal circuits (preliminary report). In Ashok K. Chandra, Detlef Wotschke, Emily P. Friedman, and Michael A. Harrison, editors, *STOC*, pages 196–203. ACM, 1976.
19. Andrew Chi-Chih 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.
20. Yu Yu, Jussipekka Leiwo, and Benjamin Premkumar. Hiding circuit topology from unbounded reverse engineers. In Lynn Margaret Batten and Reihaneh Safavi-Naini, editors, *Proceedings of the 11th Australasian conference on Information Security and Privacy*, ACISP'06, pages 171–182, 2006.