Program Obfuscation with Leaky Hardware

Nir Bitansky[1⋆], Ran Canetti[1,2⋆], Shafi Goldwasser[3], Shai Halevi[4], Yael Tauman Kalai[5], and Guy N. Rothblum[5⋆⋆]

[1] Tel Aviv University
[2] Boston University
[3] MIT and Weizmann Institute of Science
[4] IBM T.J. Watson Research Center
[5] Microsoft Research

Abstract. We consider general program obfuscation mechanisms using "somewhat trusted" hardware devices, with the goal of minimizing the usage of the hardware, its complexity, and the required trust. Specifically, our solution has the following properties:

(i) The obfuscation remains secure even if all the hardware devices in use are *leaky*. That is, the adversary can obtain the result of evaluating any function on the local state of the device, as long as this function has short output. In addition the adversary also controls the communication between the devices.

(ii) The number of hardware devices used in an obfuscation and the amount of work they perform are polynomial in the security parameter *independently* of the obfuscated function's complexity.

(iii) A (*universal*) set of hardware components, owned by the user, is initialized only once and from that point on can be used with multiple "software-based" obfuscations sent by different vendors.

1 Introduction

Program obfuscation is the process of making a program unintelligible while preserving its functionality. (For example, we may want to publish an encryption program that allows anyone to encrypt messages without giving away the secret key.) The goal of general program obfuscation is to devise a generic transformation that can be used to obfuscate any arbitrary input program.

It is known from prior work that general program obfuscation is possible with the help of a completely trusted hardware device (e.g., [7, 28, 19]). On the other hand, Barak et al. proved that software-only general program obfuscation is impossible, even for a very weak notion of obfuscation [6]. In this work we

⋆ Supported by the Check Point Institute for Information Security, a Marie Curie reintegration grant and an ISF grant.

⋆⋆ Most of this work was done while the author was at the Department of Computer Science at Princeton University and Supported by NSF Grant CCF-0832797 and by a Computing Innovation Fellowship.

consider an intermediate setting, where we can use hardware devices but these devices are not completely trusted. Specifically, we consider using leaky hardware devices, where an adversary controlling the devices is able to learn some information about their secret state, but not all of it.

We observe that the impossibility result of Barak et al. implies that hardware-assisted obfuscation using a single leaky device is also impossible, even if the hardware device leaks only a single bit (but this bit can be an arbitrary function of the device's state). See Section 1.3. Consequently, we consider a model in which several hardware devices are used, where each device can be locally leaky but the adversary cannot obtain leakage from the global state of all the devices together. Importantly, in addition to the leakage from the separate devices, our model also gives the adversary full control over the communication between them.

The outline of our solution is as follows: Starting from any hardware-assisted obfuscation solution that uses a completely trusted device (e.g., [19, 25]), we first transform that device into a system that resists leakage in the Micali-Reyzin model of "only computation leaks" (OCL) [29] (or actually in a slightly augmented OCL model). In principle, this can be done using OCL-compilers from the literature [27, 24, 22] (but see discussion in Section 1.4 about properties of these compilers). The result is a system that emulates the functionality of the original trusted device; however, now the system is made of several components and can resists leakage from each of the components separately.

This still does not solve our problem since the system that we get from OCL-compilers only resists leakage if the different components can interact with each other over secret and authenticated channels (see discussion in Section 1.3). We therefore show how to realize secure communication channels over insecure network in a leakage-resilient manner. This construction, which uses non-committing encryption [12] and information theoretic MACs (e.g., [33, 3]), is the main technical novelty in the current work. See Section 1.4.

The transformation above provides an adequate level of security, but it is not as efficient and flexible as one would want. For one thing, the OCL-compilers in the literature [27, 24, 22] produce systems with roughly as many components as there are gates in the underlying trusted hardware device. We show that using fully homomorphic encryption [31, 18] and universal arguments [4] we can get a system where the number of components depends only on the security parameter and is (almost) independent of the complexity of the trusted hardware device that we are emulating. See Section 1.1.

Another drawback of the solution above is that it requires a new set of hardware devices for every program that we want to obfuscate. Instead, we would like to have just one set of devices, which are initialized once and thereafter can be used to obfuscate many programs. We show how to achieve such a reusable obfuscation system using a simple trick based on CCA-secure encryption, see Section 1.2.

We now proceed to provide more details on the various components of our solution.

1.1 Minimally Hardware-Assisted Obfuscation

Forgetting for the moment about leakage-resilience, we begin by describing a hardware-assisted obfuscating mechanism where the amount of work done by the trusted hardware is (almost) independent of the complexity of the program being obfuscated. The basic idea is folklore: The obfuscator encrypts the program f using a fully homomorphic encryption scheme [31, 18], gives the encrypted program to the evaluator and installs the decryption key in the trusted hardware device. Then, the evaluator can evaluate the program homomorphically on inputs of its choice and ask the device to decrypt.

Of course, the above does not quite work as is, since the hardware device can be used for unrestricted decryption (so in particular it can be used to decrypt the function f itself). To solve this, we make the evaluator prove to the device that the ciphertext to be decrypted was indeed computed by applying the homomorphic evaluation procedure on the encrypted program and some input. Note that to this end we must add the encrypted program itself or a short hash of it to the device (so as to make "the encrypted program" a well-defined quantity). To keep the device from doing a lot of work, the proof should be verifiable much more efficiently than the computation itself, e.g., using the "universal arguments" of Barak and Goldreich [4]. We formalize this idea and show that this obfuscation scheme satisfies a strong notion of simulation based obfuscation. It can even be implemented using stateless hardware with no source of internal randomness (so it is secure against concurrent executions and reset attacks). See Section 2 for more details.

1.2 Obfuscation using universal hardware devices

A side-effect of the above solution is that the trusted hardware device must be specialized for the particular program that we want to protect (e.g., by hard-wiring in it a hash of the encrypted program), so that it has a well-defined assertion to verify before decryption. Instead, we would like the end user to use a single *universal* hardware device to run all the obfuscated programs that it receives (possibly from different vendors).

We obtain this goal using a surprisingly simple mechanism: The trusted hardware device is installed with a secret decryption key of a CCA-secure cryptosystem, whose public key is known to all vendors. Obfuscation is done as before, except that the homomorphic decryption key and the hash of the encrypted program are encrypted using the CCA-secure public key and appended to the obfuscation. This results in a universal (or "sendable") obfuscation, the device is only initialized once and then everyone can use it to obfuscate their programs. See more details in Section 3.

1.3 Dealing With Leaky Hardware

The more fundamental problem with the hardware-assisted obfuscation is that the hardware must be fully leak-free and can only provide security as long as it is

accessed as a black box. This assumption is not true in many deployments, so we replace it by the weaker assumption that our hardware components are "honest-but-leaky". Namely, in our model an obfuscated program consists of software that is entirely in the clear, combined with some *leaky hardware components*. Our goal is therefore to design an obfuscator that transforms any circuit with secrets into a system of software and hardware components that achieves strong black-box obfuscation even if the components can leak.

We remark that the impossibility of universal obfuscation [6] implies that more than one hardware component is necessary. To see this, observe that if we had a single hardware component that resists (even one-bit) arbitrary leakage then we immediately get a no-hardware obfuscation in the sense of Barak et al. [6]: The obfuscated program consists of our software and a full description of the hardware component (including all the embedded secrets). This must be a good obfuscation since any predicate that we can evaluate on this description can be seen as a one-bit leakage function evaluated on the state of the hardware component. If the device was resilient to arbitrary one-bit leakage, it would mean that any such leakage/predicate can be computed by a simulator that only has black-box access to the function; hence, we have a proper obfuscator.

The model of leaky distributed systems. Given the impossibility result for a single leaky hardware component, we concentrate on solutions that use multiple components. Namely, we have (polynomially) many hardware components, all of which are leaky. The adversary in our model can freely choose the inputs to the hardware components and obtain leakage by repeatedly choosing one component at a time and evaluating an arbitrary (polynomial-size) leakage function on the current state and randomness of that component. We place no restriction on the order or the number of times that components can be chosen to leak, so long as the total rate of leakage from each component is not too high.

In more detail, we consider continual leakage, where the lifetime of the system is partitioned into time units and within each time unit we have some bound on the number of leakage bits that the adversary can ask for. The components are running a randomized *refresh* protocol at the end of each time unit and erase their previous state.[6] A unique feature of our model is that the adversary sees and has complete control over all the communication between these components (including the communication needed for the refresh protocol). We term our leakage model the *leaky distributed system* model (LDS), indeed this is just the standard model of a distributed system with adversarially controlled communication, when we add to it the fact that the individual parties are leaky.

We stress that this model seems realistic: the different components can be implemented by physically (and even geographically) separated machines, amply justifying the assumption on separate leakage. We also note that a similar (but somewhat weaker) model was suggested recently by Akavia et al. [1], in the context of leakage-resilient encryption.

[6] This is reminiscent to the proactive security literature [30, 13].

Only-computation-leaks vs. leaky distributed systems. Our leakage model shares some similarities to the "only computation leaks" (OCL) model, in that the adversary can get leakage from different parts of the global state separately but not from the entire global state at once. These two models are nonetheless fundamentally different, for two reasons. One difference is that in the OCL the different components "interact" directly by writing to and reading from memory, and communication is neither controlled by nor visible to the adversary. In the LDS model, on the other hand, the adversary sees and controls the entire communication. Another difference is that in the OCL model, the adversary can only get leakage from the components in the order in which they perform the computation, whereas in LDS model, it can get leakage in any order.

An intermediate model, that we use as a technical tool in this work, is where the adversary can get leakage from the components in any order (as in the LDS model), but the components communicate securely as in the OCL model. For lack of a better name, we call this intermediate model the OCL^+ model. Clearly, resilience to leakage in the model of leaky distributed systems is strictly harder than in the OCL or OCL^+ models and every solution secure in our model will automatically be secure also in the two weaker models.

1.4 From OCL^+ to LDS

We present a transformation that takes any circuit secure in the OCL^+ model and converts it into a system of components that maintains the functionality and is secure in the model of leaky distributed systems. Recently, Goldwasser-Rothblum [22] constructed a universal compiler, which transforms any circuit into one that is secure in the OCL^+ model. (Unlike previous compilers [17, 24, 27], the [22] compiler does not require a leak-free hardware component.) Combining the compiler with our transformation, we obtain a compiler that takes any circuit and produces a system of components with the same functionality that is secure in the LDS model. The number of components in the resulting system is essentially the size of the original circuit, assuming we use the underlying Goldwasser-Rothblum compiler. However, as we explain in Section 1.5 below, we can reduce the number of components to be *independent* of the circuit size, by first applying the hardware-assisted obfuscator from Section 1.1.

The main gap between the OCL^+ model and our model of leaky distributed systems, is that in the former, communication between the components is completely secure, whereas in the latter it is adversarially controlled. In the heart of our transformation stands an implementation of *leakage-tolerant communication channels* that bridges the above gap, based on the following tools:

Non-Committing Encryption. Our main technical observation is that secret communication in the face of leakage can be obtained very simply using non-committing encryption [12]. Recall that non-committing encryption is a (potentially interactive) encryption scheme such that a simulator can generate a fake transcript, which can later be "opened" as either an encryption of zero or as an encryption of one. This holds even when the simulator needs to generate

the randomness of both the sender and the receiver. In our context, the distributed components use non-committing encryption to preserve the privacy of their messages. The observation is that non-committing encryption can be used to implement "leakage resilient channels", in the sense that any leakage query on the state of the communicating parties could be transformed into a leakage query on the underlying message alone (see Section 4).

Leakage-resilient MACs. In addition to secrecy, we also need to ensure authenticity of the communication between the components. We observe that this can be done easily using information-theoretic MAC schemes based on universal-hashing [33, 3]. Roughly, each pair of components will maintain rolling MAC keys that are only used $\Theta(1)$ times. To authenticate a message, they will use the MAC key sent with the prior message and will send a new MAC key to be used for the next message. (We use a short MAC key to authenticate a much longer message, so the additional bandwidth needed for sending future MAC keys is tolerable.) Since these MAC schemes offer information-theoretic security, it is very easy to prove that they can also tolerate bounded leakage. Authenticating the communication assures that secrecy is kept (e.g. the adversary cannot have a component encrypt a secret message under an unauthentic key) and also ensures that the components remain "synchronized" (see Section 4).

1.5 The End-Result: Obfuscation with Leaky Hardware

To obfuscate a program, we first apply the hardware-assisted obfuscator from Section 1.1, thus obtaining a universal hardware device, whose size and amount of computation (per input) depend only on the security parameter, and which can be used to evaluate obfuscated programs from various vendors. We next apply the Goldwasser-Rothblum compiler [22], together with our transformation from Section 1.4, to the code of the hardware device, resulting in a system of components that can still be used for obfuscation in exactly the same way (as the universal device), but is now guaranteed to remain secure even if the components are leaky and even if the communication between them is adversarially controlled.

To obfuscate a program f using this system, the obfuscator generates keys for the FHE scheme and encrypts f under these keys. In addition, it uses the public CCA2 key generated with the original universal device to encrypt the secret FHE key together with a hash of the encrypted program. The encrypted program and parameters are then sent to the user. Evaluating the obfuscated program consists of running the FHE evaluation procedure and then interacting with the system of components (in a universal argument) to decrypt the resulting ciphertext. The system verifies the proof in a leakage-resilient manner and returns the decrypted result.

We remark that our transformation from any circuit/device to a leaky system of components, as well as our transformation from circuit-specific obfuscation schemes to general-purpose ones, are generic and can be applied to any device-assisted obfuscation scheme, such as the schemes of [19, 25]. When doing so, the

end result will inherit the properties of the underlying scheme. In particular, when instantiated with [19, 25], the amount of work performed by the devices is proportional to the size of the entire computation (the hardware used for each gate in the obfuscated circuit).

1.6 Related Work

Research on formal notions of obfuscation essentially started with the work of Barak et. al. [6], who proved that software-only obfuscation is impossible in general. This was followed by other negative results [20] and some positive results for obfuscating very simple classes of functions (e.g., point functions) [32, 11, 15]. The sweeping negative results for software-only obfuscation motivated researchers to consider relaxed notions where some interesting special cases can be obfuscated (e.g., [23, 26, 8]).

In contrast, the early works of Best [7], Kent [28] and Goldreich and Ostrovsky [19] addressed the software-protection problem using a physically shielded full-blown CPU. The work of Goyal *et. al.* [25] showed that the same can be achieved also with small stateless hardware tokens. These solutions only consider perfectly opaque hardware. Furthermore, in these works the amount of work performed by the secure hardware device during the evaluation of one input is proportional to the size of the entire computation.[7]

The work by Goldwasser *et. al.* [21] on one-time programs shows that programs can be obfuscated using very simple hardware devices that do very little work. However, their resulting obfuscated program can be run *only once.*

Our focus on obfuscation with *leaky* hardware follows a large corpus of recent works addressing leakage-resilience cryptography (see, e.g., [16, 2] and references within). In particular, our construction uses results of Goldwasser and Rothblum [24, 22], which show how to convert circuits into ones that are secure in *only computation leaks* model of Micali and Reyzin [29] (or even in the stronger OCL^+ model described above).

Our construction of leakage-tolerant secure channels and the relation between leakage-tolerance and adaptive security were further investigated and generalized in [10], who consider general *universally composable* leaky protocols.

Organization In Section 2 we construct a hardware-assisted obfuscation scheme where the amount of work done by the hardware is minimal (polynomial in the security parameter). In Section 3 we show how to transform any "circuit-specific" scheme, such as the one constructed in Section 2, to a "general-purpose" scheme where the same hardware device can be used for multiple obfuscated programs. In Section 4 we show how to transform any hardware-assisted obfuscation, such as the above, to a leakage-resilient scheme. The full details and proofs as well as some of the secondary results can be found in the full version of this paper [9].

[7] On the other hand, the solutions in [19, 25] can be based on one-way functions, while our solution requires stronger tools such as FHE and universal arguments.

2 Hardware Assisted Obfuscation

In this section we construct a hardware assisted obfuscation scheme. The basic model and definitions are presented in Section 2.1. An overview of the construction is presented in Section 2.2. The detailed construction and its analysis can be found in the full version of this paper[9].

2.1 The Model

In the setting of hardware assisted obfuscation, a circuit C (taken from a family $\mathcal{C}_n$ of poly-size circuits) is obfuscated in two stages. First, the PPT obfuscation algorithm $\mathcal{O}$ is applied to C, producing the "software part" of the obfuscation obf, together with (secret) parameters params for device initialization. At the second stage, the hardware device HW is initialized with params. The evaluator is given obf and black-box access to the initialized device $\mathsf{HW}_{\mathsf{params}}$. In our security definition, we consider a setting in which the adversary is given $t = \mathrm{poly}(n)$ independent obfuscations of t circuits, where obfuscation i consists of a corresponding device $\mathsf{HW}_{\mathsf{params}_i}$ and obfuscated data obf_i. In this model each obfuscated circuit may have its own specialized device.

Definition 2.1 (Circuit-specific hardware-assisted obfuscation (CSHO)). $(\mathcal{O}, \mathsf{HW}, \mathsf{Eval})$ *is a* CSHO *scheme for a circuit ensemble* $\mathcal{C} = \{\mathcal{C}_n\}$*, if it satisfies:*

- **Functional Correctness.** Eval *is a poly-time oracle aided TM , such that for any* $n \in \mathbb{N}$*,* $C \in \mathcal{C}_n$ *and input* v *for* C*:* $\mathsf{Eval}^{\mathsf{HW}_{\mathsf{params}}}\left(1^{|C|}, \mathsf{obf}, v\right) = C(v)$*, where* $(\mathsf{obf}, \mathsf{params}) \leftarrow \mathcal{O}(C)$*.*
- **Circuit-Independent Efficiency.** *The size of* $\mathsf{HW}_{\mathsf{params}}$ *is* $\mathrm{poly}(n)$*, independently of* $|C|$*, where* $(\mathsf{params}, \mathsf{obf}) \leftarrow \mathcal{O}(C)$*. Also, during each run of* $\mathsf{Eval}^{\mathsf{HW}_{\mathsf{params}}}\left(1^{|C|}, \mathsf{obf}, v\right)$ *on any input* v*, the total amount of work performed by* $\mathsf{HW}_{\mathsf{params}}$ *is* $\mathrm{poly}(n)$*, independently of* $|C|$*.*
- **Polynomial Slowdown.** $\mathcal{O}$ *is a* PPT *algorithm. In particular, there is a polynomial* q*, such that for any* $n \in \mathbb{N}$ *and* $C \in \mathcal{C}_n$*,* $|\mathsf{obf}| \leq q(|C|)$*.*
- t**-Composable Virtual Black Box (VBB)**. *Any adversary, given* t *obfuscations, can be simulated, given oracle access to the corresponding circuits. That is, for any* PPT $\mathcal{A}$ *(with arbitrary output) there is a* PPT $\mathcal{S}$ *such that:*

$$\left\{\mathcal{A}^{\mathsf{HW}_1,\ldots,\mathsf{HW}_t}(z, \mathsf{obf}_1, \ldots, \mathsf{obf}_t)\right\} \approx_c \left\{\mathcal{S}^{C_1,\ldots,C_t}(z, 1^n, |C_1|, \ldots, |C_t|)\right\} \ ,$$

where $C_1 \ldots C_t \in \mathcal{C}_n$*,* $z \in \{0,1\}^{\mathrm{poly}(n)}$ *is an arbitrary auxiliary input,* $\mathsf{HW}_i = \mathsf{HW}_{\mathsf{params}_i}$ *and* $(\mathsf{obf}_i, \mathsf{params}_i) \leftarrow \mathcal{O}(C_i)$*.*
We say that the scheme is **stand-alone VBB** *if it is* 1*-composable. We say that the scheme is* **composable** *if its* t*-composable for any polynomial* t*.*

While previous solutions [19, 25] satisfy the correctness and security requirements of Definition 2.1, they require that the total amount of work performed by the device for a single evaluation is proportional to $|C|$, the size of the entire circuit. Namely, they do not achieve circuit-independent efficiency. In this section we show that how to construct schemes which do achieve this feature, based on a different approach. The main result is given by Theorem 2.1.

Theorem 2.1. *Assuming fully homomorphic encryption, there exists a composable* CSHO *scheme for all polynomial size circuit ensembles* $\mathcal{C} = \{\mathcal{C}_n\}$.

2.2 The Construction

We next overview the main aspects of the constructions.

The main ideas. Informally, given a FHE scheme $\mathcal{E}$, we obfuscate a circuit C by sampling $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}\,(1^n)$, encrypting $\hat{C} = \mathsf{Enc}_{\mathsf{pk}}\,(C)$ and creating a "proof-checking decryption device" $\mathsf{HW} = \mathsf{HW}_{\mathsf{sk}}$ which is meant to decrypt "proper evaluations". The obfuscation consists of $\mathsf{obf} = (\hat{C}, \mathsf{pk})$ and oracle access to HW. To evaluate the obfuscation on input v, compute $e = \mathsf{Eval}_{\mathsf{pk}}(\hat{C}, U_{s,v})$, where $U_{s,v}$ is a universal circuit that given a circuit C of size s outputs $C\,(v)$.[8] Then, "prove" to HW that indeed $e = \mathsf{Eval}_{\mathsf{pk}}(\hat{C}, U_{s,v})$. In case HW is "convinced", it decrypts $C\,(v) = \mathsf{Dec}_{\mathsf{sk}}\,(e)$ and returns the result to the evaluator. Intuitively, the semantic security of $\mathcal{E}$ and the soundness of the proof system in use should prevent the evaluator from learning anything about the original circuit C other than its input-output behavior.

We briefly point out the main technical issues that arise when applying the above approach and the way we deal with these issues.

- **Minimizing the device's workload.** Proving the validity of an evaluated ciphertext e w.r.t. an encrypted circuit $\hat{C}$ amounts to proving that a poly($|C|$)-long computation was performed correctly. However, the running time of our device should be independent of $|C|$ and hence cannot process such a computation. In fact, it cannot even process the assertion itself as it includes the poly($|C|$)-long encryption $\hat{C}$. To overcome this, we use *universal arguments* (UA's) that also have a *proof of knowledge* property [4]and *collision resistant hashing*. Specifically, the device only stores a (short) hash $h(\hat{C})$ and the evaluator proves it "knows" an encrypted circuit $\hat{C}'$ with the same hash and that the evaluated ciphertext is the result of applying $\mathsf{Eval}_{\mathsf{pk}}$ to $\hat{C}'$ and the universal circuit $U_{s,v}$ (corresponding to some input v).
- **Using a stateless device with no fresh randomness.** Our device can be implemented as a boolean circuit that need not maintain a state between evaluator calls nor generate fresh randomness; in particular, it should withstand concurrent proof attempts and "reset attacks" (as termed by [14]). To enable this, we use similar techniques to those in [5]. Informally, these techniques allow transforming the UA protocol we use to a "resettable" protocol, where the verifier's randomness is fixed to some *pseudo random function.* [9]

[8] Abusing notation, we denote by Eval both evaluation algorithms $\mathsf{Eval}^{\mathsf{HW}_{\mathsf{params}}}(\mathsf{obf}, v)$ and $\mathsf{Eval}_{\mathsf{pk}}$. To distinguish between the two, we always denote the evaluation algorithm of the FHE scheme by $\mathsf{Eval}_{\mathsf{pk}}$.

[9] The mentioned techniques essentially transform any public-coin constant-round protocol to a "resettable" one.

3 General-Purpose (Sendable) Obfuscation

In this section we show how to convert any *circuit-specific* obfuscation scheme, such as the one in Section 2, to a scheme which uses a single *universal* (general-purpose) hardware device. The basic model and definitions are presented in Section 3.1, the transformation is presented in Section 3.2 and analyzed in the full version of this paper [9].

3.1 The Model

In circuit-specific obfuscation, the obfuscator gives the user a device that depends on the obfuscated circuit C. More precisely, the "specifying parameters" params, produced by $\mathcal{O}(C)$, depend on C and are hardwired into the device before it is sent to the user. Thus, each device supports only a single obfuscated circuit.

We consider a more natural setting in which different parties can send obfuscations to each other online, without the need of exchanging devices per each obfuscation. Informally, in this setting we assume that a trusted manufacturer creates devices, where each device is associated with private and public parameters $(\mathsf{prv}, \mathsf{pub})$. The private parameters are hardwired into the device and are never revealed (they can be destroyed), while the public ones are published together with the "identity" of the device (e.g., on the manufacturer's web page www.obfuscationdevices.com). Any user, who wishes to send an obfuscation of a circuit C to another user who holds such a device, retrieves the corresponding public parameters and sends the required obfuscation.

Concretely, a general-purpose obfuscation scheme consists of two randomized algorithms $(\mathsf{Gen}, \mathcal{O})$ and a device HW. First, $\mathsf{Gen}(1^n)$ generates private and public parameters $(\mathsf{prv}, \mathsf{pub})$ (independently of any circuit). Then, HW is initialized with prv and the initialized device $\mathsf{HW}_{\mathsf{prv}}$ is given to the user. The corresponding pub are published. Anyone in hold of pub can obfuscate a circuit C by computing $\mathsf{obf} \leftarrow \mathcal{O}(C, \mathsf{pub})$ and sending obf to the user holding the device.

Definition 3.1 (General-purpose hardware-assisted obfuscation (GPHO)). *$(\mathcal{O}, \mathsf{Gen}, \mathsf{HW}, \mathsf{Eval})$ is a GPHO scheme for $\mathcal{C} = \{\mathcal{C}_n\}$ if it satisfies:*

- **Functional Correctness.** Eval *is a polynomial-time oracle aided TM, such that for any $n \in \mathbb{N}$, $C \in \mathcal{C}_n$ and input v for C: $\mathsf{Eval}^{\mathsf{HW}_{\mathsf{prv}}}\left(1^{|C|}, \mathsf{obf}, v\right) = C(v)$, where $(\mathsf{prv}, \mathsf{pub}) \leftarrow \mathsf{Gen}(1^n)$ and $\mathsf{obf} \leftarrow \mathcal{O}(C, \mathsf{pub})$.*
- **Circuit-Independent Efficiency.** *The size of $\mathsf{HW}_{\mathsf{prv}}$ is polynomial in n, independent of $|C|$, where $(\mathsf{prv}, \mathsf{pub}) \leftarrow \mathsf{Gen}(1^n)$. Moreover, during each run of $\mathsf{Eval}^{\mathsf{HW}_{\mathsf{prv}}}\left(1^{|C|}, \mathsf{obf}, v\right)$ on any input v, the total amount of work performed by $\mathsf{HW}_{\mathsf{prv}}$ is polynomial in n, independent of $|C|$.*
- **Polynomial Slowdown.** *$\mathcal{O}$ and Gen are PPT algorithms. In particular, there is a polynomial q such that for any $n \in \mathbb{N}$, $C \in \mathcal{C}_n$, $|\mathsf{pub}, \mathsf{prv}| \leq q(n)$ and $|\mathsf{obf}| \leq q(|C|)$.*
- **Virtual Black Box (VBB).** *For any PPT adversary $\mathcal{A}$ and polynomial t there is a PPT simulator $\mathcal{S}$ such that:*

$$\left\{\mathcal{A}^{\mathsf{HW}_{\mathsf{prv}}}(z, \mathsf{obf}_1, \ldots, \mathsf{obf}_t)\right\} \approx_c \left\{\mathcal{S}^{C_1,\ldots,C_t}(z, 1^n, |C_1|, \ldots |C_t|)\right\} ,$$

where $C_1 \ldots C_t \in \mathcal{C}_n$, $z \in \{0,1\}^{\mathrm{poly}(n)}$ *is an arbitrary auxiliary input* $(\mathsf{prv}, \mathsf{pub}) \leftarrow \mathsf{Gen}\,(1^n)$ *and* $\mathsf{obf}_i \leftarrow \mathcal{O}\,(C_i, \mathsf{pub})$.

3.2 The Transformation

Essentially, we wish to avoid restricting the device to a specific circuit C (like hard-wiring $h(\hat{C})$ into the device as done in our circuit-specific scheme). Instead, we would like to have the user "initialize" his device with the required parameters params for each obfuscation he wishes to evaluate. However, params cannot be explicitly given to the evaluator as they contain sensitive information.

For this purpose, we simply use a CCA2 public key encryption scheme. That is, the obfuscator will generate params, but instead of hard-wiring them into the hardware device (which will make the device circuit-specific), he will encrypt params and send the resulting ciphertext to the user. The fact that the underlying encryption scheme is CCA2 secure implies that the user can neither gain any information about params nor change it to related parameters params'.

More formally, the new general-purpose device HW' is manufactured together with a pair of CCA2 keys $(\mathsf{prv}, \mathsf{pub}) = (\mathsf{sk}, \mathsf{pk})$. The secret key sk is hardwired into the device (and destroyed), while pk is published. Each device call is appended with the CCA2 encryption of params. The device HW' answers its calls by first decrypting the encrypted parameters params and then applying the device $\mathsf{HW}_{\mathsf{params}}$ of the underlying circuit-specific scheme (e.g. the scheme in Section 2). In the full version [9] we present the detailed construction and show:

Theorem 3.1. *Given a CCA2 encryption scheme, any circuit-specific obfuscation scheme as in Definition 2.1 can be transformed to a general-purpose one as in Definition 3.1.*

Corollary 3.1 (of Theorems 2.1,3.1). *Assume that there exists a fully homomorphic encryption scheme and a CCA2 encryption scheme, then there exists a general-purpose obfuscation scheme.*

Remark 3.1. The above transformation would also work (as is) for schemes with no circuit-independent efficiency. The amount of work performed by the general-purpose device is essentially inherited from the underlying scheme (with the fixed overhead of CCA2 decryption). In particular, we can apply it to the scheme of [25] and get a general-purpose solution that is based solely on the existence of CCA2 schemes, but which makes $\mathrm{poly}(|C|)$ device calls.

4 Obfuscation with Leaky Hardware

We now turn to the task of dealing with leaky hardware. As we explained in the introduction, if we allow arbitrary leakage functions (even with small output) then it is impossible to obfuscate using a *single* leaky hardware device. Hence, our goal is to show how to use many leaky hardware devices to achieve obfuscation.

We first show how to obfuscate any function f using leaky hardware devices, where the number of devices is proportional to the size of the circuit computing f. Then, when we apply this obfuscator to the function computed by the hardware device from Section 2 (or Section 3, respectively), to get circuit-specific (or general-purpose, respectively) obfuscation with leaky hardware devices, where the number of devices is polynomial in the security parameter, *independent* of the function being obfuscated.

4.1 An Overview

In what follows, we give an informal definition of obfuscation with leaky hardware and a high-level overview of our construction. The formal definitions and detailed construction are given in Sections 4.2 and 4.3. The security analysis can be found in the full version of this paper [9].

The leaky distributed system (LDS) model. In the LDS model a functionality f (with secrets) is implemented by a system of multiple hardware components $(\mathsf{HW}_1, \mathsf{HW}_2, \ldots, \mathsf{HW}_m)$. The components can maintain a state and generate fresh randomness. To evaluate the functionality f, an input v is given to HW_1 and the components communicate to jointly compute $f(v)$, which is eventually outputted by HW_m. The adversary (evaluator) in our model can freely choose the inputs to the computation and is given full control over the communication between the components. In addition, the adversary can choose one component at a time and evaluate a leakage function on its inner state and randomness.

We consider a continual leakage model, where the lifetime of each component HW_i is partitioned into time periods (that are set according to the inputs that HW_i receives). At the end of each time period, HW_i "refreshes" its inner state by applying an Update procedure (that erases the previous state). The Update procedures performed at different components are coordinated by exchange of messages. As the rest of the computation, the Update procedure is also exposed to leakage and the adversary controls the exchange of messages during the update.

We place no restriction on the order and timing of the adversary's interaction with the system. In particular, it can pass messages to any component at any time and get leakage on any component at any time (which can depend on previous leakage and messages).

Constructing secure leaky distributed systems (LDS). Our goal is to compile (or "obfuscate") any functionality, given by some circuit C (with hardwired secrets), into an LDS that *perfectly protects* C, as long as **the leakage from each HW_i in each time period is bounded**. In the terminology of obfuscation, the LDS should perform as a *virtual black-box*: The view of any adversary $\mathcal{A}$ attacking the LDS can be simulated by a simulator $\mathcal{S}$ which can only access C as a black-box. In particular, $\mathcal{S}$ should simulate on its own the communication between the components and all the leakage. We achieve this goal in two main steps:

1. We apply the Goldwasser-Rothblum compiler to the circuit C to get a circuit that is secure in the (augmented) *only computation leaks* (OCL^+) model.

2. Then, we provide a general transformation that takes any OCL^+-secure circuit and transforms it to a secure LDS.

Hence, our main goal is to show that an adversary in the LDS model can be simulated by an adversary in the OCL^+ model (that does not witness the communication between the modules). Then, by the OCL^+-security (implied by the GR compiler), we can deduce that simulation can be done only with black-box access to the underlying functionality.

In the heart of our transformation stands an implementation of *leakage tolerant communication channels*. We first explain the main ideas required to achieve secrecy and then explain how to get authenticity.

Leaky secret channels from non-committing encryption. In the OCL^+ model, the components can securely exchange messages. Still, the adversary might get some leakage on the contents of these messages as the (leaky) state of the components includes the messages at some point. The OCL^+ security guarantee implies, however, that a bounded amount of leakage does not compromise the security of the entire system.

To enhance OCL^+-security to LDS-security we implement the secure communication channels. As explained above, we assume for now that the adversary delivers all messages intact and deal only with secrecy. The standard solution for secret channels would be to encrypt all communication between the components; however, in the face of leakage this approach encounters the following difficulty: Consider a sender component HW_S in the LDS model that wishes to communicate a message M to a receiver component HW_R (using some encryption scheme). Note that the adversary can obtain arbitrary (bounded) leakage on the state of both $\mathsf{HW}_S, \mathsf{HW}_R$, including leakage on both the plaintext M and the randomness r_S, r_R used to encrypt/decrypt. Moreover, the leakage function can depend on the corresponding ciphers which were already sent. This implies that naively simulating the communication (by say encryptions of 0) won't work.

Our main technical observation is that the above obstacle can be overcome using non-committing encryption (NCE) [12]. NCE schemes (which can potentially be interactive) allow simulating a fake cipher (or transcript) c together with two optional random strings $(r_S^0, r_S^1), (r_R^0, r_R^1)$ for both the sender S and the receiver R. The simulated cipher can later be "opened" as an encryption of either 1 or 0 (using the suitable randomness).[10] This tool allows us to show that the view of an attacker $\mathcal{A}$ in the LDS model can be simulated by an attacker $\mathcal{A}'$ in the OCL^+ model, provided that the components communicate using NCE.

[10] NCE was so far mainly used in the setting of multi-party-computation as a tool for dealing with adaptive corruptions. Indeed, leakage can be viewed as a restricted form of "honest but curious" corruption, where the adversary learns part of the state, whereas in full corruption, it learns the entire state. In both cases, the choice of leakage/corruption is done adaptively according to the view of the adversary so far. The relation between leakage-tolerant protocols and adaptively secure protocols is further generalized in [10].

Specifically, for any single bit message, the OCL^+ adversary $\mathcal{A}'$ (which does not see any communication) will use the NCE to generate fake communication with corresponding randomness $\bar{r} = (r_S^0, r_S^1), (r_R^0, r_R^1)$. Then, when the simulated $\mathcal{A}$ performs a leakage query L to be evaluated on both the plaintext b and the encryption's randomness, $\mathcal{A}'$ can translate it to a new leakage query L' which **will only be evaluated on the plaintext message**. The leakage function L' will have the simulated randomness $\bar{r}$ hardwired into it and will choose which randomness to use according to the plaintext b.

Leakage resilient MACs. To deal with adversaries that interfere with message delivery we use leakage-resilient c-time MAC schemes. Informally, each two components maintain rolling MAC keys that are used at most $c = O(1)$ times. After $c - 1$ times the components run the Update protocol to regain fresh MAC keys. The communication during the update is done using NCE as described above, while authentication is done using the c-th application of the previous key.

4.2 The LDS Model

Our leakage model postulates an adversary $\mathcal{A}$ that interacts with a system of distributed leaky hardware components. Each component maintains a state and is capable of producing fresh randomness. At the onset of the interaction, the components are pre-loaded with some secret state and thereafter they can receive messages, send messages and leak information to the attacker. In our model all the I/O of the components and their communication is done via the attacker $\mathcal{A}$.

Definition 4.1 (Single-input leakage). *In a distributed single-input λ-leakage attack a PPT adversary $\mathcal{A}$ interacts with hardware components $(\mathsf{HW}_1, \ldots, \mathsf{HW}_m)$ and can do the following (in any order, possibly in an interleaving manner):*

1. *Feed $\mathcal{O}(C)$ a single input of his choice.*
2. *Interact with each component, sending it messages and receiving the resulting outputs and replies. These devices are message-driven, so they are activated by receiving messages from the attacker, then they compute and send the result, then wait for more messages.*
3. *Adaptively send up to λ 1-bit leakage queries to each of the hardware components. Each leakage query is modeled as a poly-size Boolean circuit and is applied to the entire state of a single hardware device. Without loss of generality, we can think of the state of the device as it was in the last time that the device was activated, including all the randomness that the device generated in order to deal with the last activation.*

We denote the output of $\mathcal{A}$ in such attack by $\mathcal{A}[\lambda : \mathsf{HW}_1, \ldots, \mathsf{HW}_m]$.

Definition 4.2 (Continual leakage). *A continual λ-leakage attack is an attack where a PPT adversary $\mathcal{A}$ repeats a single-input λ-leakage attack poly many times, where between any two consecutive attacks the devices' secret state is updated by applying a PPT algorithm Update to the state of each HW_i separately. $\mathcal{A}$*

obtains leakage during the Update *procedure, where the leakage function takes as input both the current secret state of* HW_i *and the randomness used by* Update.

We denote by time period t *at device* HW_i *the time period between the beginning of the* $(t-1)$*st* Update *procedure and the end of the* t*-th* Update *procedure (note that these time periods are overlapping).*[11] *We allow the adversary* $\mathcal{A}$ *to leak at most* λ *bits from each* HW_i *during each (local) time period.*

We denote the output of $\mathcal{A}$ *in such attack by* $\mathcal{A}[\lambda : \mathsf{HW}_1, \ldots, \mathsf{HW}_m : \mathsf{Update}]$.

Below we consider an obfuscator $\mathcal{O}$ that takes as input a circuit C and outputs an "obfuscated" version of C that uses leaky hardware devices as above. Namely, we have $(\mathsf{HW}_1, \ldots, \mathsf{HW}_m) \leftarrow \mathcal{O}(C)$, where the HW_i's are the leaky hardware devices, initialized with the appropriate circuits.

Remark 4.1. In Definitions 2.1 and 3.1, the obfuscator $\mathcal{O}$ outputs a "software part" obf and parameters params for initializing the hardware. In the current setting, the obfuscation does not contain a software part. The simplified notation $(\mathsf{HW}_1, \ldots, \mathsf{HW}_m) \leftarrow \mathcal{O}(C)$, should be interpreted as sampling $\{\mathsf{params}_i\} \leftarrow \mathcal{O}(C)$ (where params_i corresponds to the i-th sub-computation)and initializing the hardware devices $\{\mathsf{HW}_i\}$ accordingly.

Definition 4.3. *We say that* $\mathcal{O}$ *is an LDS-obfuscator with continual* λ*-leaky hardware if for any circuit* C *and* $(\mathsf{HW}_1, \ldots, \mathsf{HW}_m) \leftarrow \mathcal{O}(C)$*, the distributed system* $(\mathsf{HW}_1, \ldots, \mathsf{HW}_m)$ *maintains the functionality of* C *when all the messages between them are delivered intact and in addition we have the following:*

For any PPT *attacker* $\mathcal{A}$*, executing a continual* λ*-bit leakage attack, there exists a* PPT *simulator* $\mathcal{S}$*, such that for any ensemble of poly-size circuits* $\{\mathcal{C}_n\}$*:*

$$\{\mathcal{A}(z)[\lambda : \mathsf{HW}_1, \ldots, \mathsf{HW}_m : \mathsf{Update}]\}_{\substack{n\in\mathbb{N}, C\in\mathcal{C}_n \\ z\in\{0,1\}^{\mathrm{poly}(n)}}} \approx_c \left\{\mathcal{S}^C(z, 1^{|C|})\right\}_{\substack{n\in\mathbb{N}, C\in\mathcal{C}_n \\ z\in\{0,1\}^{\mathrm{poly}(n)}}},$$

where $(\mathsf{HW}_1, \ldots, \mathsf{HW}_m) \leftarrow \mathcal{O}(C)$ *and* z *is an arbitrary auxiliary input.*

4.3 The Construction

We build our solution using a compiler $\mathcal{C}$ that is secure in the continual λ-OCL^+ model. Namely, $\mathcal{C}$ converts any circuit C into a collection of leaky sub-components $(sub_1, \ldots, sub_m)$ (that also have an update procedure, Update') that is secure long as the adversary can only get λ leakage from each component in each time unit and cannot see or influence the communication between them. In our model, however, the communication is under the control of the adversary. To secure the communication, we use non-committing encryption and c-time leakage resilient MACs (as described in the overview).

[11] Intuitively, time period t is the entire period where the t-th updated secret states can be leaked. During the t-th Update procedure, both the $(t-1)$st secret state and the t-th secret state may leak, which is why the time periods are overlapping.

The construction. Given a circuit C, the obfuscator $\mathcal{O}$ does the following:

1. Apply the λ-OCL^+ compiler $\mathcal{C}$ to C and obtain a circuit $C' = (sub_1, \ldots, sub_m)$ and an Update' procedure, such that (C', Update') is secure in the continual λ-OCL^+ model.
   We assume for simplicity that: (a) sub_1 is the input module, that takes as input the "original" input $x \in \{0,1\}^n$ and passes it to the relevant sub_j's. (b) sub_m generates the final output. (c) The exchanged messages between the modules are all of the same size $\ell = \ell(n)$.
2. Put each module sub_i in a separate hardware component HW_i.
3. For every two communicating modules $i, j \in [m]$, generate a random key $K_{i,j} \leftarrow \{0,1\}^t$ for a λ-leakage-resilient MAC scheme $(\mathsf{MAC}, \mathsf{Vrfy})$, with keys of length $t = \Theta(\lambda)$. For every $i \in [m]$, hard-wire in HW_i the set of keys $\{(j, K_{i,j})\}$, for every j such that sub_j and sub_i communicate.
4. For every $i \in \{1, \ldots, m-1\}$ and every $j \in \{2, \ldots, m\}$, whenever sub_i is supposed to send a message $\mathbf{M} = (M_1, \ldots, M_\ell)$ to sub_j, the corresponding hardware HW_i sends $\mathbf{M}$ to HW_j using a non-committing encryption scheme $(\mathsf{NCGen}, \mathsf{NCEnc}, \mathsf{NCDec})$. Moreover, all the communication in this process is authenticated using the MAC scheme $(\mathsf{MAC}, \mathsf{Vrfy})$. More specifically, the hardware devices HW_i and HW_j communicate as follows:
   (a) Hardware HW_j does the following:
      i. For each $k \in [\ell]$, sample a random $r_{G,k} \in \{0,1\}^{\mathrm{poly}(n)}$ and compute $(e_k, d_k) = \mathsf{NCGen}(1^n; r_{G,k})$. Henceforth, let $\mathbf{e} = (e_1, \ldots, e_\ell), \mathbf{d} = (d_1, \ldots, d_\ell)$.
      ii. Compute $\sigma_{\mathbf{e}} = \mathsf{MAC}(\mathbf{e}; K_{i,j})$.
      iii. Send $(\mathbf{e}, \sigma_{\mathbf{e}})$ to HW_i and keep $\mathbf{d}$ as part of the secret state.
   (b) Hardware HW_i does the following:
      i. Verify that $\mathsf{Vrfy}(\mathbf{e}, \sigma_{\mathbf{e}}; K_{i,j}) = 1$ and verify that $(\mathbf{e}, \sigma_{\mathbf{e}})$ was not already sent by HW_j during this time period. If this check fails then discard the message $\mathbf{e}$.
      ii. If the check passes, for each $k \in [\ell]$ choose a random $r_{E,k} \in \{0,1\}^{\mathrm{poly}(n)}$, compute $c_k = \mathsf{NCEnc}(M_k, e_k; r_{E,k})$. Henceforth, let $\mathbf{c} = (c_1, \ldots, c_\ell)$.
      iii. Compute $\sigma_{\mathbf{c}} = \mathsf{MAC}(\mathbf{c}; K_{i,j})$.
      iv. Send $(\mathbf{c}, \sigma_{\mathbf{c}})$ to HW_j.
   (c) Hardware HW_j does the following:
      i. Verify that $\mathsf{Vrfy}(\mathbf{c}, \sigma_{\mathbf{c}}; K_{i,j}) = 1$ and verify that $(\mathbf{c}, \sigma_{\mathbf{c}})$ wasn't already sent by HW_i. If this check fails then discard the message $\mathbf{c}$.
      ii. If the check passes, compute for each $k \in [\ell]$, $M_i = \mathsf{NCDec}(c_i, d_i)$.

   Once HW_j gets $\mathbf{M}$, it runs sub_j on input $\mathbf{M}$ (unless sub_j is waiting for additional inputs).
5. Finally, HW_m sends an output message (assuming sub_m is the sub-computation that generates the outputs).
6. For each HW_i, after each "valid" activation (i.e., after it did its share in a computation), HW_i erases all its computations and updates its secret state, using an update procedure Update, defined as follows.
   (a) Apply the Update' procedure to update the state of sub_i.

(b) Refresh the MAC keys by choosing new random MAC keys $K'_{i,j}$ for every $j > i$ such that HW_i and HW_j communicate. Then send $K'_{i,j}$ to HW_j.
(c) Erase the previous MAC keys $K_{i,j}$.
(d) **Communication:** All the communication within the update procedure is done as in step 4. Namely, for each message, repeat steps $4(a) - 4(c)$, where the MACs are w.r.t. the previous MAC key $K_{i,j}$.

Theorem 4.1. *Assuming the compiler $\mathcal{C}$ used in the above construction is secure in the λ-OCL^+ model. Then the above construction yields an LDS-obfuscator with continual λ-leaky hardware $\mathsf{HW}_1, \ldots, \mathsf{HW}_m$.*

The proof of Theorem 4.1 is given in the full version of this paper [9].

References

[1] Adi Akavia, Shafi Goldwasser, and Carmit Hazay. Distributed Public Key Encryption Schemes. manuscript, 2010.
[2] Adi Akavia, Shafi Goldwasser, and Vinod Vaikuntanathan. Simultaneous hardcore bits and cryptography against memory attacks. In Omer Reingold, editor, *Theory of Cryptography - TCC 2009*, volume 5444 of *Lecture Notes in Computer Science*, pages 474–495. Springer, 2009.
[3] Mustafa Atici and Douglas R. Stinson. Universal hashing and multiple authentication. In *CRYPTO*, pages 16–30, 1996.
[4] Boaz Barak and Oded Goldreich. Universal arguments and their applications. *SIAM J. Comput.*, 38(5):1661–1694, 2008.
[5] Boaz Barak, Oded Goldreich, Shafi Goldwasser, and Yehuda Lindell. Resettably-sound zero-knowledge and its applications. In *FOCS*, pages 116–125, 2001.
[6] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *CRYPTO*, pages 1–18, 2001.
[7] Robert M. Best. Microprocessor for executing enciphered programs. US Patent 4168396, 1979.
[8] Nir Bitansky and Ran Canetti. On strong simulation and composable point obfuscation. In *Advances in Cryptology - CRYPTO 2010*, pages 520–537, 2010.
[9] Nir Bitansky, Ran Canetti, Shafi Goldwasser, Shai Halevi, and Guy Rothblum. Obfuscation with leaky hardware, 2011. Long Version on http://eprint.iacr.org.
[10] Nir Bitansky, Ran Canetti, and Shai Halevi. Leakage tolerant interactive protocols. Manuscript, 2011. `http://eprint.iacr.org/2011/204`.
[11] Ran Canetti and Ronny Ramzi Dakdouk. Obfuscating point functions with multibit output. In *EUROCRYPT'08*, pages 489–508, 2008.
[12] Ran Canetti, Uri Feige, Oded Goldreich, and Moni Naor. Adaptively Secure Multi-party Computation. In *28th Annual ACM Symposium on the Theory of Computing - STOC'96*, pages 639–648, Philadelphia, PA, May 1996. ACM.
[13] Ran Canetti, Rosario Gennaro, Amir Herzberg, and Dalit Naor. Proactive security: Long-term Protection against break-ins. *CryptoBytes*, 3(1), 1997.
[14] Ran Canetti, Oded Goldreich, Shafi Goldwasser, and Silvio Micali. Resettable zero-knowledge (extended abstract). In *STOC*, pages 235–244, 2000.
[15] Ran Canetti, Guy N. Rothblum, and Mayank Varia. Obfuscation of hyperplane membership. In *TCC*, pages 72–89, 2010.

[16] Stefan Dziembowski and Krzysztof Pietrzak. Leakage-resilient cryptography. In *49th FOCS - 2008*, pages 293–302. IEEE Computer Society, 2008.

[17] Sebastian Faust, Tal Rabin, Leonid Reyzin, Eran Tromer, and Vinod Vaikuntanathan. Protecting circuits from leakage: the computationally-bounded and noisy cases. In *EUROCRYPT*, pages 135–156, 2010.

[18] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st ACM Symposium on Theory of Computing – STOC 2009*, pages 169–178. ACM, 2009.

[19] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

[20] Shafi Goldwasser and Yael Tauman Kalai. On the impossibility of obfuscation with auxiliary input. In *46th FOCS*, pages 553–562. IEEE Computer Society, 2005.

[21] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. One-time programs. In David Wagner, editor, *Advances in Cryptology - CRYPTO 2008*, volume 5157 of *Lecture Notes in Computer Science*, pages 39–56. Springer, 2008.

[22] Shafi Goldwasser and Guy Rothblum. Unconditionally securing general computation against continuous only-computation leakage. Manuscript, 2011.

[23] Shafi Goldwasser and Guy N. Rothblum. On best-possible obfuscation. In *TCC'07*, pages 194–213, 2007.

[24] Shafi Goldwasser and Guy N Rothblum. Securing computation against continuous leakage. In Tal Rabin, editor, *Advances in Cryptology - CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 59–79. Springer, 2010.

[25] Vipul Goyal, Yuval Ishai, Amit Sahai, Ramarathnam Venkatesan, and Akshay Wadia. Founding cryptography on tamper-proof hardware tokens. In *TCC*, pages 308–326, 2010.

[26] Dennis Hofheinz, John Malone-Lee, and Martijn Stam. Obfuscation for cryptographic purposes. In *TCC'07*, pages 214–232, 2007.

[27] Ali Juma and Yevgeniy Vahlis. Protecting Cryptographic Keys against Continual Leakage. In Tal Rabin, editor, *Advances in Cryptology - CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 41–58. Springer, 2010.

[28] Stephen Thomas Kent. *Protecting externally supplied software in small computers*. PhD thesis, Massachusetts Institute of Technology, 1981.

[29] Silvio Micali and Leonid Reyzin. Physically observable cryptography. In *TCC'04*, volume 2951 of *Lecture Notes in Computer Science*, pages 278–296. Springer, 2004.

[30] Rafail Ostrovsky and Moti Yung. How to withstand mobile virus attacks. In *10th Annual ACM Symposium on Principles of Distributed Computing, PODC'91*, pages 51–59. "ACM", 1991.

[31] R. Rivest, L. Adleman, and M. Dertouzos. On data banks and privacy homomorphisms. In *Foundations of Secure Computation*, pages 169–177. Academic Press, 1978.

[32] Hoeteck Wee. On obfuscating point functions. In *STOC'05*, pages 523–532, 2005.

[33] M. Wegman and L. Carter. New hash functions and their use in authentication and set equality. In *J. of Computer and System Sciences*, volume 22, pages 265–279, 1981.