Program Obfuscation with Leaky Hardware[*]

Nir Bitansky[†] Ran Canetti[*] Shafi Goldwasser[‡] Shai Halevi[§]
Yael Tauman Kalai[¶] Guy N. Rothblum[||]

December 6, 2011

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.

[*]An extended abstract of this paper appears in the proceedings of ASIACRYPT '11.

[†]Tel Aviv University and Boston University. Supported by the Check Point Institute for Information Security, a Marie Curie reintegration grant and an ISF grant.

[‡]MIT and Weizmann Institute of Science.

[§]IBM T.J. Watson Research Center.

[¶]Microsoft Research.

[||]Microsoft Research. 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.

Contents

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., [9, 34, 23]). On the other hand, Barak et al. proved that software-only general program obfuscation is impossible, even for a very weak notion of obfuscation [8]. In this work we 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., [23, 29]), we first transform that device into a system that resists leakage in the Micali-Reyzin model of "only computation leaks" (OCL) [37] (or actually in a slightly augmented OCL model). In principle, this can be done using OCL-compilers from the literature [33, 28, 26] (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 [15] and information theoretic MACs (e.g., [42, 4]), 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. The OCL-compilers in the literature [28, 26] produce systems with roughly as many components as there are gates in the underlying trusted hardware device; hence, we wish to minimize the size of this device. In addition, we also wish to minimize the number of device calls during a single evaluation of the obfuscated program. We show that using fully homomorphic encryption [39, 22] and universal arguments [6] we can get a system where the number of components and amount of work per evaluation depends only on the security parameter and is (almost) independent of the complexity of the obfuscated program. 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 [39, 22], 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 [6]. 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 3 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 4.

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 [8] 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. [8]: 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.[1] 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. [2], in the context of leakage-resilient encryption.

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.

[1]This is reminiscent to the proactive security literature [38, 16].

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 [26] constructed a universal compiler, which transforms any circuit into one that is secure in the OCL^+ model. (Unlike previous compilers [21, 28, 33], the [26] 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 [15]. 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. We refer the reader to Section 2.2 for the formal definition of non-committing encryption and to Section 5 for the way we use it to implement leakage-tolerant secure channels.

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 [42, 4]. 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 5).

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 [26], 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 [23, 29]. When doing so, the end result will inherit the properties of the underlying scheme. In particular, when instantiated with [23, 29], 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. [8], who proved that software-only obfuscation is impossible in general. This was followed by other negative results [24] and some positive results for obfuscating very simple classes of functions (e.g., point functions) [41, 14, 18]. The sweeping negative results for software-only obfuscation motivated researchers to consider relaxed notions where some interesting special cases can be obfuscated (e.g., [27, 31, 10]).

In contrast, the early works of Best [9], Kent [34] and Goldreich and Ostrovsky [23] addressed the software-protection problem using a physically shielded full-blown CPU. The work of Goyal *et. al.* [29] 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.[2]

The work by Goldwasser *et. al.* [25] 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.*

[2]On the other hand, the solutions in [23, 29] can be based on one-way functions, while our solution requires stronger tools such as FHE and universal arguments.

Our focus on obfuscation with *leaky* hardware follows a large corpus of recent works addressing leakage-resilience cryptography (see, e.g., [20, 3] and references within). In particular, our construction can be based on the results of Goldwasser and Rothblum [28, 26] or Juma and Vahlis [33], which show how to convert circuits into ones that are secure in *only computation leaks* model of Micali and Reyzin [37], or even in the stronger OCL^+ model described above. ([26] is the only one though that achieves the latter without the use of any leak-free hardware.) Leakage-resilient circuits (or "private circuits") were previously considered in the works of Ishai, Sahai and Wagner [32] and of Faust et al. [21], under different restrictions than OCL. The first, only allows leakage on a bounded number of individual wires; the second, only considers AC^0 leakage (and relies on a leak-free hardware component). Another work by Ajtai[1] extends model of Goldreich and Ostrovsky [23] to also account for leakage under a restriction that is similar in spirit to that of [32] (namely that only a bounded number of individual instructions leak).

Our construction of leakage-tolerant secure channels and the relation between leakage-tolerance and adaptive security were further investigated and generalized in [11], who consider general *universally composable* leaky protocols.

2 Tools

2.1 The GR Compiler

The works of Goldwasser and Rothblum [28] and Juma and Vahlis [33] show how to convert any circuit into a circuit that is secure in the *"only computation leaks"* (OCL) model, using a very simple and stateless hardware device.[3] Very recently Goldwasser and Rothblum [26] introduced a major improvement, showing a new transformation that makes no use of secure hardware and where the security is unconditional.

In their model a circuit C is converted into a circuit C' that consists of m disjoint and ordered sub-computations (or modules) $sub_1, \ldots, sub_m$, where the input to sub-computation sub_i should depend only on the output of earlier sub-computations. Each of these sub-computations sub_i can be modeled as a non-uniform poly-size circuit, with a "secret state". They prove that no information about the circuit C is leaked even if each of these sub-computations is leaky. More specifically, the adversary can request to see a bounded-length function of each sub_i (separately) and these leakage functions may be adaptively chosen.

They also consider the continual leakage setting, where the leakage happens over and over again. In this setting, the secret state of each sub_i needs to be continually updated (or refreshed). To this end, after each computation, each of the sub_i's is updated by applying a randomized Update function to its secret state. We stress that leakage may occur during each of these update procedures, in which case the leakage may be a function of both the current secret state and the randomness used by the Update procedure.

In this work, we consider a strengthening of the OCL model, where we give the attacker more power. In the OCL model, the sub-computations are ordered $sub_1, \ldots, sub_m$ and the adversary must leak in that order (i.e., first leak from sub_1, then from sub_2, etc.). We strengthen the power of the adversary, by allowing it to leak from the sub-computations in any order it wishes. Moreover, it can leak a bit from sub_i, then leak a bit from sub_j and based on the leakage values, leak again on sub_i. So, the adversary controls which sub_i it wishes to leak from. We refer to this stronger model as the OCL^+ model. We note that the [28, 26] compilers are secure even in this stronger OCL^+ model.

For the sake of simplicity, we start by defining single-input security with bounded leakage and then we define multi-input security with continual leakage.

Throughout this paper, sub_i denotes a non-uniform poly-size circuit with a "secret state" that is hardwired into the circuit. A computation $C' = (sub_1, \ldots, sub_m)$ is a large circuit, which is the concatenation of all the sub_i's. For simplicity, we always think of sub_1 as the sub-circuit that takes the input and of sub_m and the sub-circuit that generates the output.

Definition 2.1 (Single-input leakage attack $\mathcal{A}[\lambda : sub_1, \ldots, sub_m]$)**.** *A single-input λ-bit leakage attack of adversary $\mathcal{A}$ on the computation $C' = (sub_1, \ldots, sub_m)$, is modeled as follows.*[4] *The adversary $\mathcal{A}$ can adaptively choose (at most $\lambda \cdot m$)* PPT *functions $L_1, \ldots L_{\lambda \cdot m}$, where each L_i is a Boolean circuit that chooses a sub_j, takes as input the secret state of sub_j and outputs a single bit. At any point, the adversary $\mathcal{A}$ can choose an input x for C'. From that point on, any leakage functions L_i, takes as input both the secret state of sub_j and the input to sub_j during the computation*

[3] For example, in the GR compiler, this hardware device essentially generates encryptions of 0 or encryptions of a random bit b, using the BHHO [12] encryption scheme. Thus, this hardware device does not have any secret state.

[4] The input to each sub-computation sub_i should depend only on the output of earlier sub-computations.

of C' on input x and outputs a single bit. The only restriction is that the total amount of leakage from each module sub_i is at most λ bits. The view of the adversary in the attack consists of the pair $(x, C(x))$ and the outputs of all the leakage functions.

Definition 2.2 (Continual leakage attack $\mathcal{A}[\lambda : sub_1, \dots, sub_m : \mathsf{Update}]$)**.** *A continual λ-bit leakage attack of adversary $\mathcal{A}$ on the circuit $C' = (sub_1, \dots, sub_m)$ with update procedure* Update*, is defined as follows. The adversary $\mathcal{A}$ can (adaptively) launch polynomially many single-input λ-bit leakage attacks. However, between every two consecutive attacks the secret state of each sub_i is updated, by applying a* PPT *algorithm* Update*. We allow the adversary $\mathcal{A}$ to leak during these* Update *procedures, where the leakage function takes as input both the entire (secret) state of sub_i and the randomness used by the* Update *procedure.*

For each sub_i, we let its t-th time period be the time period between the beginning of its $(t-1)$st Update *procedure and the end of its t-th* Update *procedure (note that these time periods are overlapping). We allow the adversary $\mathcal{A}$ to leak at most λ bits from each sub_i during each time period. As in the single-input case, the leakage can be adaptive and can be applied to any sub_i in an arbitrary order.*

If the leakage on sub_i is during an update procedure, the leakage function takes as input the secret state of sub_i and the randomness used by Update*. If the leakage on sub_i is during a computation of $C'(x)$ then the leakage, as in the single-input case, takes as input the (current) secret state of sub_i and the input to sub_i during the computation of C' on input x.*

The view of $\mathcal{A}$ in the attack consists of all the input-output pairs $\{x_i, C(x_i)\}$, where x_i is the input to the i-th single-input attack and the outputs of all the leakage functions.

Remark 2.1. We note that the reason that in Definition 2.2 we partitioned the time periods as we did, is so that the t time period includes the entire time in which the t-th updated secret state is in the system and thus can be leaked. Note that 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 overlap.

Definition 2.3. *We say that a* PPT *compiler $\mathcal{C}$ is secure in the single-input λ-OCL^+ model if for any* PPT *adversary $\mathcal{A}$, which executes a single-input λ-bit leakage attack, there exists a* PPT *simulator $\mathcal{S}$, such that for any ensemble of poly-size circuits $\{\mathcal{C}_n\}$:*

$$\{\mathcal{A}(z)[\lambda : sub_1, \dots, sub_m]\}_{\substack{n\in\mathbb{N}, C\in\mathcal{C}_n \\ z\in\{0,1\}^{\mathrm{poly}(n)}}} \approx_c \left\{\mathcal{S}^C(z, 1^{|C|})\right\}_{\substack{n\in\mathbb{N}, C\in\mathcal{C}_n \\ z\in\{0,1\}^{\mathrm{poly}(n)}}} ,$$

where $(sub_1, \dots, sub_m) \leftarrow \mathcal{C}(C)$ and z is an arbitrary auxiliary input.

Definition 2.4. *We say that a* PPT *compiler $\mathcal{C}$ is secure in the continual λ-OCL^+ model if for any* PPT *adversary $\mathcal{A}$, that executes a continual λ-bit leakage attack, there exists a* PPT *simulator $\mathcal{S}$, such that for any ensemble of poly-size circuits $\{\mathcal{C}_n\}$:*

$$\{\mathcal{A}(z)[\lambda : sub_1, \dots, sub_m : \mathsf{Update}]\}_{\substack{n\in\mathbb{N}, C\in\mathcal{C}_n \\ z\in\{0,1\}^{\mathrm{poly}(n)}}} \approx_c \left\{\mathcal{S}^C(z, 1^{|C|})\right\}_{\substack{n\in\mathbb{N}, C\in\mathcal{C}_n \\ z\in\{0,1\}^{\mathrm{poly}(n)}}} ,$$

where $(sub_1, \dots, sub_m : \mathsf{Update}) \leftarrow \mathcal{C}(C)$ and z is an arbitrary auxiliary input.

Remark 2.2. One could restrict the simulator $\mathcal{S}$ in Definitions 2.3 and 2.4 to query the oracle C only on inputs x which $\mathcal{A}$ feeds to his circuit C'. We note that the GR compiler is secure also w.r.t. these more stringent definitions. For the sake of simplicity, we decided not to add this requirement to the definitions.

Theorem 2.1 (The existence of λ-OCL^+ compilers [28, 26])**.** *There (unconditionally) exists a universal compiler in the continual λ-OCL^+ model, where the size of each (leaky) sub-computation, sub_i, is $\Theta(\lambda^3)$.*[5]

2.2 Non-committing Encryption

The notion of *non-committing encryption* was introduced by Canetti *et. al.* [15]. Informally, non-committing (bit) encryption schemes are semantically secure, possibly interactive encryption schemes, with the additional property that a simulator can generate special ciphertexts that can be "opened" (i.e. demonstrated to be the encryption of) to both 0 and 1.

Definition 2.5. *[15, 19] A non-committing (bit) encryption scheme consists of a tuple* $(\mathsf{NCGen}, \mathsf{NCEnc}, \mathsf{NCDec}, \mathsf{NCS}$ *where* $(\mathsf{NCGen}, \mathsf{NCEnc}, \mathsf{NCDec})$ *is a semantically secure encryption scheme and* NCSim *is a* PPT *simulation algorithm that on input* 1^n *outputs a tuple* $(e, c, r_G^0, r_E^0, r_G^1, r_E^1)$ *such that for every* $b \in \{0,1\}$ *the following distributions are computationally indistinguishable:*

1. *The joint view of an honest sender and an honest receiver in a normal encryption of b:*

$$\{(e, c, r_G, r_E) : (e, d) = \mathsf{NCGen}(1^n; r_G), c = \mathsf{NCEnc}_e(b; r_E)\} .$$

2. *A simulated view of an encryption of b:*

$$\{(e, c, r_G^b, r_E^b) : (e, c, r_G^0, r_E^0, r_G^1, r_E^1) \leftarrow \mathsf{NCSim}(1^n)\} .$$

2.3 Leakage resilient MACs

A c-time MAC scheme, $(\mathsf{MAC}, \mathsf{Vrfy})$, assures that given authentications of c distinct messages, it is impossible to forge an authentication of a new message. The scheme is λ-leakage-resilient if it remains secure, even when the adversary obtains an arbitrary (poly-size) leakage function L of the secret key, as long as the output length of L is at most λ. The leakage function L can be chosen adaptively according to the authentication tags that the adversary had already seen.

Concretely, we shall use a scheme with the following properties:

1. It is c-time secure for some $c = \Theta(1)$.
2. The secret key is of length ℓ, where $\ell = n^{\Theta(1)}$ (polynomially related to the security parameter n).
3. It can authenticate messages of length $\mathrm{poly}(\ell) > \ell$.
4. It is λ-leakage-resilient for some $\lambda = \Theta(\ell)$.

Information-theoretic schemes with properties $(1) - (3)$ can be obtained based on universal hashing [42, 4]. We note that any information theoretic MAC also has some leakage resilience in the following sense: If the forging probability is at most $2^{-\epsilon\ell}$ for some constant ϵ, then in particular, it can withstand $\epsilon\ell - \omega(\log n)$ bits of leakage and remain secure.

[5] In [28] it is shown that allowing simple leak-free hardware, there exist under the Decision-Diffie-Hellman assumption, a universal compiler where the size of each sub-computation is $\Theta(\lambda)$ (i.e., with a constant leakage-rate.

2.4 Fully Homomorphic Encryption

A fully homomorphic public-key encryption scheme (FHE) $\mathcal{E}$ consists of algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$. The first three are the standard generation, encryption and decryption algorithms of a public key scheme. The additional algorithm Eval is a deterministic polynomial-time algorithm, that takes as input a public key pk, a ciphertext $\hat{x} = \mathsf{Enc}_{\mathsf{pk}}(x)$ and a circuit C and outputs, a new ciphertext $c = \mathsf{Eval}_{\mathsf{pk}}(\hat{x}, C)$, such that $\mathsf{Dec}_{\mathsf{sk}}(c) = C(x)$, where sk is the secret key corresponding to the public key pk. It is required that the size of c depends polynomially on the security parameter and the length of $C(x)$, but is otherwise independent of the size of the circuit C.

Such schemes were recently constructed by the breakthrough work of Gentry [22] and of van Dijk, Gentry, Halevi and Vaikuntanathan [40].

2.5 Universal Arguments

Consider the relation

$$R_{\mathcal{U}} = \{(y, w) : y = (M, x, t) \text{ and } M \text{ is a Turing machine that accepts } (x, w) \text{ after at most } t \text{ steps}\}$$

and its corresponding language

$$L_{\mathcal{U}} = \{y : \exists w \text{ such that } (y, w) \in R_{\mathcal{U}}\} \ .$$

We recall the notion of universal arguments, defined by Barak and Goldreich [6], based on the work of Kilian [35] and Micali [36].

Definition 2.6 (Universal arguments [6]). *A universal argument system is a pair of interactive Turing machines, denoted by $(\mathcal{P}, \mathcal{V})$, that satisfy the following properties.*

- **Efficient verification.** *There exists a fixed polynomial p such that for any $y = (M, x, t) \in \{0,1\}^n$ the total runtime of $\mathcal{V}$, on common input y, is at most $p(n)$. In particular, all the messages exchanged in the protocol are of length at most $p(n)$.*

- **Completeness.** *There exists a polynomial q such that for every $y = (M, x, t) \in L_{\mathcal{U}}$ the total runtime of P on input (y, w) is at most $q(|M|, t)$ and $\Pr[(\mathcal{P}(w), V)(y) = 1] = 1$.*

- **Computational soundness.** *For any polynomial-size circuit family $\mathcal{P}^* = \{\mathcal{P}^*_n\}$, any large enough n, $y \in \{0,1\}^n \setminus L_{\mathcal{U}}$:*

$$\Pr[(\mathcal{P}^*_n, V)(y) = 1] \le \mathrm{negl}(n) .^{6}$$

- **A weak proof of knowledge.** *For every polynomial p there exists a polynomial p' and a probabilistic oracle machine E satisfying the following. For any polynomial-size circuit family $\{\mathcal{P}^*_n\}_{n \in \mathbb{N}}$, every large enough n and every $y = (M, x, t) \in \{0,1\}^n$, if $\Pr[(\mathcal{P}^*_n, \mathcal{V})(y) = 1] \ge \frac{1}{p(n)}$ then*

$$\Pr_r\left[\exists w \in R_{\mathcal{U}}(y) \ s.t. \ \forall i, E_r^{\mathcal{P}^*_n}(y, i) = w_i\right] \ge \frac{1}{p'(n)} \ .^{7}$$

[6]This soundness condition might seem quite weak, as it protects only against poly-size cheating provers. However, the latter polynomial bound need not be a-priori fixed, which is sufficient for our applications.

[7]This requirement strengthens the soundness requirement and is often not required for UA's. It will be useful though for our purpose.

Theorem 2.2 ([6])**.** *Assuming the existence of collision-resistant hash functions, there exists a 4-message universal argument system with a public-coin verifier.*

3 Hardware Assisted Obfuscation

In this section we construct a hardware assisted obfuscation scheme. For this scheme, the size and amount of work performed by the device are bounded by a fixed polynomial in the security parameter that is independent of the size of the circuit being obfuscated. However, this construction will have two main drawbacks: First, the hardware is "circuit-specific" in the sense that each obfuscated circuit requires its own specific device. Second, the device is assumed to be totally opaque (i.e. it is treated as a black-box). In Sections 4 and 5 we overcome these drawbacks.

The basic model and definitions are presented in Section 3.1. The construction is presented in Section 3.2 and analyzed in Section 3.3.

3.1 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 $t = \text{poly}(n)$ 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 3.1 (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* $\text{poly}(n)$, *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* $\text{poly}(n)$, *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:*

$$\left\{\mathcal{A}^{\mathsf{HW}_1,\ldots,\mathsf{HW}_t}(z, \mathsf{obf}_1, \ldots, \mathsf{obf}_t)\right\}_{\substack{n\in\mathbb{N} \\ C_1\ldots C_t\in\mathcal{C}_n \\ z\in\{0,1\}^{\text{poly}(n)}}} \approx_c \left\{\mathcal{S}^{C_1,\ldots,C_t}(z, 1^n, |C_1|, \ldots, |C_t|)\right\}_{\substack{n\in\mathbb{N} \\ C_1\ldots C_t\in\mathcal{C}_n \\ z\in\{0,1\}^{\text{poly}(n)}}},$$

  where $\mathsf{HW}_i = \mathsf{HW}_{\mathsf{params}_i}$, $(\mathsf{obf}_i, \mathsf{params}_i) \leftarrow \mathcal{O}(C_i)$ *and z is an arbitrary auxiliary input.*

  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 [23, 29] satisfy the correctness and security requirements of Definition 3.1, 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 3.1.

Theorem 3.1. *Assuming fully homomorphic encryption, there exists a composable* CSHO *scheme for all polynomial size circuit ensembles* $\mathcal{C} = \{\mathcal{C}_n\}$.

3.2 The Construction

We now present a concrete construction satisfying Theorem 3.1. We first describe the main ideas behind the scheme, which are practically considered "folklore". Then, we present the detailed construction and its analysis.

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 [6]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 [17]). To enable this, we use similar techniques to those in [7]. 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.

3.2.1 A detailed description of our construction.

We now provide a detailed description of the scheme and prove its security. We first state some simplifying assumptions and our assumptions regarding the hardware device.

Simplifying assumptions. We assume that the input and output of the circuit C (to be obfuscated) are bounded by a fixed polynomial in the security parameter. The size of C on the other hand is not a priori bounded.

Assumptions regarding the device. As mentioned above, we assume that the device can only be accessed as a "black-box". In particular, it cannot be tampered with and does not leak any information.

The device is merely a boolean circuit which does not maintain a state between evaluator calls and does not generate randomness. The size of this circuit and the number of calls (per evaluation) should be bounded by a fixed polynomial in the security parameter n, independently of the circuit size $|C|$.

Recall that whenever the evaluator, who is given an encrypted circuit $\hat{C} = \mathsf{Enc}_{\mathsf{pk}}(C)$, wishes to evaluate the circuit on input v, it homomorphically evaluates $\hat{C} = \mathsf{Enc}_{\mathsf{pk}}(C)$ on input v, by computing $e = \mathsf{Eval}_{\mathsf{pk}}(\hat{C}, U_{v,s})$, where $s = |C|$ and $U_{s,v}$ is a universal circuit that given a circuit C of size s outputs $C(v)$. Then, he is required to prove to the device that e is indeed valid; i.e., that it is the result of applying $\mathsf{Eval}_{\mathsf{pk}}$ to $\hat{C}$ and the universal circuit $U_{s,v}$. However, the size of this computation is proportional to $|C|$, while our device is bounded by a fixed polynomial in the security parameter.

As explained above we overcome this technicality using a UA, which essentially allows one to verify the validity of a "t-long computation" in time which is polynomial in the input size and in $|t| = \log t$ (rather than in the entire computation time t).

Recall that an input $y = (M, x, t)$ for a UA consists of a "validating TM" M, input x and a time bound t. The triplet y is valid (i.e. $y \in L_{\mathcal{U}}$) if and only if there is a witness w such that M accepts (x, w) within t steps. The running time of the UA verifier is $\mathrm{poly}(|y|)$ and is independent of the witness size w, which can be "long".

Since the encryption $\hat{C}$ itself is too large for our device to process, it cannot appear explicitly in the input y. To solve this, we first use a *collision resistant hash function* (CRH) $h \leftarrow \mathcal{H}$ and let the input include $h, h(\hat{C})$ instead of $\hat{C}$. The hash function h and the value $h(\hat{C})$ are generated by $\mathcal{O}$ during the obfuscation phase and are hardwired into the device (as part of the specifying parameters params).

We then use the *proof of knowledge* property of the UA. That is, to validate an evaluation e, the evaluator proves in a UA that "it knows a witness" $\hat{C}_w$ with the same hash $h(\hat{C}_w) = h(\hat{C})$ such that $e = \mathsf{Eval}_{\mathsf{pk}}(\hat{C}_w, U_{s,v})$. We stress that while the mere existence of a witness $\hat{C}_w$ is trivially true (due to hash collisions), "proving knowledge" of $\hat{C}_w$ essentially implies that $\hat{C}_w = \hat{C}$ as required. Otherwise, the prover can find collisions in h.

Remark 3.1. We make the following remarks regarding the above hashing approach:

1. The above strategy is similar to the one used by [5] in the context of non-black-box ZK for non-uniform verifiers.

2. The hashing can be avoided if instead of obfuscating circuits of arbitrary size, we would restrict ourselves to obfuscating TM's whose running time is not a-priori bounded by a fixed poly (n), but their description is.

3. For our purpose it is sufficient to assume *universal one way functions* rather than the stronger notion of CRH's. However, the strong tools used in this section (FHE and UA's) already imply (or require) the use of CRH's. Hence, we allow ourselves to stick to this more familiar notion.

In our case, the input $y = (\mathbf{M}, x, t)$ to the UA consists of a fixed prescribed machine $\mathbf{M}$ and the extra input (x, t), which is jointly constructed by the evaluator and the device. We next specify $\mathbf{M}$ and the construction process of (x, t).

The evaluation validating machine M. Let $\mathcal{E} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ be a FHE scheme and let $\mathcal{H} = \{H_n : \{0,1\}^* \to \{0,1\}^n\}_{n\in\mathbb{N}}$ be a collision resistance hash ensemble (CRH). $\mathbf{M}$ is a (deterministic) TM which accepts $(x, w) = \left(\left(v, e, \mathsf{pk}, h, h(\hat{C}), s\right), \hat{C}_w\right)$ if and only if

$$h(\hat{C}_w) = h(\hat{C}) \ \wedge \ e = \mathsf{Eval}_{\mathsf{pk}}(\hat{C}_w, U_{s,v}) \ ,$$

where $U_{s,v}$ is a universal circuit which given a circuit C of size s returns $C\left(v\right)$ and h is interpreted as a hash function taken from H_n.

The running time of M and the associated UA time bound T. The machine $\mathbf{M}$ runs in time $p\left(|x|, |\hat{C}_w|\right)$ for some fixed polynomial p (assuming $|\hat{C}_w|$ is polynomially related to s). Indeed, the hash function h and the evaluation algorithm Eval are polynomial-time computable and the circuit $U_{s,v}$ is computable in time polynomial in s. In particular, there exists a polynomial p' such that $p\left(|x|, |\hat{C}_w|\right) \leq p'\left(n, s\right)$, where n is the security parameter and $s = |C|$ is the circuit size. We accordingly set the running time bound $T = T\left(n, s\right)$ for the UA to be $p'\left(n, s\right)$.

Constructing the input for the UA. The input $(\mathbf{M}, x, T)$ for the UA is constructed jointly by the evaluator and the device. The evaluator should supply an input v (for C) and an evaluation e (to be validated). The device supplies the code of $\mathbf{M}$, the time bound T, the public key pk, the hash function and the hash value $\left(h, h(\hat{C})\right)$ and the circuit size $s = |C|$.

Handling multiple UA sessions without state or randomness. Our device is eventually used to evaluate the obfuscated circuit on multiple inputs. This is done by engaging a UA session, as the one described above, for each input v. Since we require the device to manage without state or (fresh) randomness, we need to take care of two issues: (a) We need to answer the need of the UA verifier to generate randomness; (b) We need to handle attackers carrying out "reset attacks" [17]. Specifically, the standard UA verifier maintains a state along its interaction with the prover, while our device lacks such an inner state. As a result the state of the device (or the underlying UA verifier) is re-specified as part of the input with each device call. Thus, the attacker can essentially run the device from any state at any point in time; in particular, it can interleave sessions of different inputs. [10]

[10]For a more elaborate discussion on the resettable model see [17, 7].

To deal with the above issues we use the same ideas used in [7] to achieve "resettable soundness". [11] This is done based on the following two properties of the UA system given in Theorem 2.2: (a) It is constant round; (b) It is public-coins. (Concretely, the UA verifier sends a total of two messages, where each of them is just a fresh random string.)

When validating an input-evaluation pair (v, e) the device will essentially need to deal with two types of messages: (a) A message specifying the input-evaluation pair (v, e); (b) A UA prover message in a UA interaction for validating the pair (v, e).

Since the device does not maintain a state between calls, for each pair (v, e) we require that the evaluator specifies in each message the transcript of all the previous messages. To answer the evaluator's messages we equip the device with a PRF $f : \{0,1\}^* \to \{0,1\}^{\text{poly}(n)}$ (chosen at random from a PRF family). The random messages of the UA verifier are replaced with pseudo-random messages, which are computed by applying f to the evaluator's messages. The last (third) evaluator message, in an interaction on a pair (v, e), is validated by applying the UA verifier to the entire transcript (with the corresponding coins generated by applying f to this transcript). In case the proof is accepted, the device decrypts e for the evaluator.

The full description of the obfuscation scheme and the device is given in Figure 1.

[11] We remark that Goyal and Sahai [30] show how to implement any two-party functionality in a setting where one of the parties is resettable. Their result, however, is less sensitive of the resettable party's running time and hence can not be applied as is in our case. In addition, we are only interested in the security of the resettable party (the device); hence, we find that it is more natural to adopt the relevant techniques of [7] rather than adjust the general compiler of [30] to our setting.

Ingredients.

- Let $\mathcal{E} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ be a FHE scheme.
- Let $(\mathcal{P}, \mathcal{V})$ be the prover and verifier of the UA system.
- Let $\mathcal{H} = \{H_n : \{0,1\}^* \to \{0,1\}^n\}$ be a CRH, let $\mathcal{F} = \left\{F_n : \{0,1\}^* \to \{0,1\}^{\mathrm{poly}(n)}\right\}$ be a PRF, where $\mathrm{poly}\,(n)$ is a bound on the randomness required for $\mathcal{V}$.
- Let $\mathbf{M}$ be the evaluation validating machine, as defined above. Let $T = p'(n, s)$ be a bound on the running time on $\mathbf{M}$ and thus a time bound for the UA.

Obfuscator $\mathcal{O}$.

- **Input.** Security parameter n. Circuit C of size s.
- **Encryption.** Sample $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}\,(1^n)$. Encrypt $\hat{C} = \mathsf{Enc}_{\mathsf{pk}}\,(C)$.
- **Device initialization.** Sample a CRH $h \leftarrow H_n$ and a PRF $f \leftarrow F_n$. Initialize the device HW with $\mathsf{params} = \left(\mathsf{sk}, \mathsf{pk}, f, h, h(\hat{C}), s\right)$.
- **Output.** $\mathsf{obf} = \left(\mathsf{pk}, \hat{C}, h, s\right)$ and the initialized device $\mathsf{HW}_{\mathsf{params}}$.

Device $\mathsf{HW}_{\mathsf{params}}$

- **Input specification.** Given a message $X = (v, e)$ return $f\,(X)$ (corresponds to the first message of $\mathcal{V}$ in a UA).
- **First UA prover message.** Given a message $Y = ((v, e), M_{\mathcal{P},1})$ return $f\,(Y)$ (corresponds to the second message of $\mathcal{V}$ in a UA).
- **Second UA prover message.** Given a message $Z = ((v, e), M_{\mathcal{P},1}, M_{\mathcal{P},2})$, compute $r_1 = f\,(v, e)$, $r_2 = f\,((v, e), M_{\mathcal{P},1})$. Apply the UA verifier $\mathcal{V}$ with randomness r_1, r_2, input $\left(\mathbf{M}, \left(v, e, \mathsf{pk}, h, h(\hat{C}), s\right), T\,(n, s)\right)$ and prover responses $M_{\mathcal{P},1}, M_{\mathcal{P},2}$. In case the verifier is convinced, return $\mathsf{Dec}_{\mathsf{sk}}\,(e)$. Otherwise, return $\perp$.
- **Faults.** Given a message of invalid form, i.e. inconsistent with any of the above, return $\perp$.

Evaluation. To evaluate the obfuscated circuit C on input v, compute $e = \mathsf{Eval}_{\mathsf{pk}}(\hat{C}, U_{s,v})$ and apply $\mathcal{P}$ to prove to HW that e is a valid evaluation, according to the rules prescribed above.

Figure 1: Obfuscation with a proof checking decryption device.

Device size and work load. The device is implemented as a deterministic circuit with hardwired params. The total size of the hardwired parameters is bounded by a fixed polynomial in the security parameter n; i.e., $|\mathsf{params}| = \left|\left(\mathsf{sk}, \mathsf{pk}, f, h, h(\hat{C}), s\right)\right| \leq \text{poly}(n)$.[12] Since $\mathcal{V}, \mathsf{Dec}, h, f$ can all be implemented by fixed polynomial size circuits, the circuit size implementing $\mathsf{HW}_{\mathsf{params}}$ is also of fixed polynomial size in n.

The number of rounds of communications between the evaluator and the device for each evaluation is a constant (three).

3.3 Proof of Security

In this section we analyze the security of the scheme, proving Theorem 3.1.

The correctness of the scheme follows directly from the correctness of the FHE $\mathcal{E}$ and the completeness of the UA system. The polynomial slowdown property is also evident. We concentrate on the VBB security guarantee. We first show that the scheme is stand-alone VBB and then explain how to deduce composability.

Proof outline. Recall that to show stand-alone VBB, we need to show that for any PPT adversary $\mathcal{A}$ there exists a PPT simulator $\mathcal{S}$, such that for every $C \in \mathcal{C}_n$ and for every auxiliary input z, the simulator S simulates the output of $\mathcal{A}$ given z and black-box access to C, whereas $\mathcal{A}$ gets z and an obfuscation of C (software, obf and device, HW).

The simulator $\mathcal{S}$ will feed the adversary with an encryption of $0^{|C|}$ (instead of $\hat{C}$)and will simulate the proof checking part of the device. Specifically, $\mathcal{S}$ verifies the UA argument as prescribed in the scheme. Whenever the simulated device accepts, i.e. gets convinced of the validity of an input-evaluation pair (v, e), the simulator will simulate the decryption by querying the oracle C with the input v and will return the result to the evaluator. The formal description of $\mathcal{S}$ is given in Figure 2.

Simulator $\mathcal{S}$

- **Oracle Access.** Circuit C. Adversary $\mathcal{A}$.
- **Input.** Security parameter n. Circuit size $s = |C|$. Auxiliary input z (to be used by $\mathcal{A}$).
- **Simulation** Sample a public key, a hash function and a PRF: $\mathsf{pk} \leftarrow \mathsf{Gen}(1^n), h \leftarrow \mathcal{H}_n, f \leftarrow \mathcal{F}_n$. Create a dummy encryption $\hat{C}_d = \mathsf{Enc}_{\mathsf{pk}}(0^s)$. Emulate an execution of $\mathcal{A}$ on input $\mathsf{obf} = \left(\mathsf{pk}, \hat{C}_d, h, s\right)$, a dummy advice HW_d defined below and auxiliary input z.
- **Simulating a dummy device.** Emulate a dummy device HW_d with parameters $\left(\mathsf{pk}, f, h, h(\hat{C}_d), s\right)$. When required to give a decryption answer for an input-evaluation pair (v, e), after the simulated device accepts a validity proof, use oracle C to return $C(v)$.
- **Output.** The output of the emulated $\mathcal{A}$.

Figure 2: The Simulator

We next prove the validity of the simulator $\mathcal{S}$. To this end, for every security parameter n, fix a circuit $C \in \mathcal{C}_n$ and auxiliary input $z \in \{0,1\}^{\text{poly}(n)}$. We first show that in a real execution, $\mathcal{A}$ cannot

[12]Since the circuit size s is not a priori bounded by a specific polynomial, we assume that it is always represented by, say, $\log^2(n)$ bits.

fool the device to decrypt invalid evaluations, except with negligible probability. That is, the device does not accept pairs (v, e) such that $e \neq \mathsf{Eval}_{\mathsf{pk}}(\hat{C}, U_{s,v})$, except with negligible probability.

Lemma 3.1. *Let F denote the event that $\mathcal{A}$ convinces $\mathsf{HW} = \mathsf{HW}_{\mathsf{params}}$ to decrypt an invalid evaluation (at some point during the entire execution). Then:*

$$\Pr[F] = \text{negl}\,(n) \ ,$$

where the probability is over the coins of the obfuscator $\mathcal{O}$, including the choice of parameters $\mathsf{pk} \leftarrow \mathsf{Gen}\,(1^n)\,, h \leftarrow H_n, f \leftarrow F_n$ and the coins used for encrypting C.

Proof of Lemma 3.1. Assume toward contradiction that F occurs with non-negligible probability $\epsilon = \epsilon\,(n)$ (for infinitely many n's). First, we show how to use $\mathcal{A}$ to construct a circuit prover strategy $\mathcal{P}^*$ for a single stand-alone UA. $\mathcal{P}^*$ will be given as input a hash function $h \leftarrow \mathcal{H}$ and will generate an instance $y \notin L_{\mathcal{U}}$ and convince the UA verifier $\mathcal{V}$ to accept y with non-negligible probability. We will then use the proof-of-knowledge property of the UA to show that $\mathcal{P}^*$ can in fact be used to find collisions in h.

As a first step, we assume that the hardware device HW, given to $\mathcal{A}$, has access to a truly random function $R : \{0,1\}^* \to \{0,1\}^{\text{poly}(n)}$ instead of a PRF. We denote this device by HW_R. Indeed, using HW_R the event F still occurs with probability at least $\epsilon\,(n) - \text{negl}\,(n)$, since otherwise, we could violate the security of the PRF $\mathcal{F}$.

In order to simplify the analysis, we adjust $\mathcal{A}$ to behave as follows: (a) It does not repeat the same device call twice; (b) it is monotone; i.e., before making a device call $((v, e), M_{\mathcal{P},1}, M_{\mathcal{P},2})$, it already made the two previous device calls $X = (v, e)$ and $Y = ((v, e), M_{\mathcal{P},1})$;[13] (c) Once $\mathcal{A}$ manages to convince the device to accept an invalid pair (v, e) it halts (this event can indeed be verified by $\mathcal{A}$ by applying $\mathcal{V}$ on the session's transcript). We note that the above adjustments do not reduce the occurrence of F nor add a significant overhead to the size of $\mathcal{A}$.

Let $d = d\,(n)$ be the polynomial bounding the number of device calls that $\mathcal{A}$ makes. We note that in any execution where $\mathcal{A}$ succeeds in proving an invalid statement, there are three calls $1 \le i_1 < i_2 < i_3 \le d$ which consist the "fooling session". That is, the UA session where the device accepts a proof for an invalid pair (v, e).

The cheating prover $\mathcal{P}^*$ simulates an entire execution of $\mathcal{A}$ and tries to guess the three "fooling locations" in order to use them in its interaction with the external UA verifier $\mathcal{V}$. Concretely, given a hash function h as external input, $\mathcal{P}^*$ picks at random $1 \le i_1 < i_2 < i_3 \le d$ and emulates $\mathcal{A}$ on input $\left(\mathsf{pk}, \hat{C}, h, s\right)$, where $\mathsf{pk} \leftarrow \mathsf{Gen}(1^n)$, $\hat{C} \leftarrow \mathsf{Enc}_{\mathsf{pk}}(C)$ and $s = |C|$.

For each device call $j \notin \{i_1, i_2, i_3\}$, $\mathcal{P}^*$ simulates the device as follows. A call of the form (v, e) or of the form $((v, e), M_{\mathcal{P},1})$ is answered with a random string. $\mathcal{P}^*$ stores these random answers in a table. For a call of the form $((v, e), M_{\mathcal{P},1}, M_{\mathcal{P},2})$, $\mathcal{P}^*$ checks whether $\mathcal{V}$ accepts the proof, with the corresponding randomness taken from the table. If the proof is not accepted, $\mathcal{P}^*$ answers $\mathcal{A}$ with $\perp$. Otherwise, $\mathcal{P}^*$ aborts. As for the calls $i_1 < i_2 < i_3$. If call i_1 is of the form (v, e) then it is forwarded to the external verifier $\mathcal{V}$ and its answer is returned to $\mathcal{A}$. Otherwise, $\mathcal{P}^*$ aborts. If call i_2 is of the form $((v, e), M_{\mathcal{P},1})$, then $M_{\mathcal{P},1}$ is forwarded to the external verifier $\mathcal{V}$. Otherwise, $\mathcal{P}^*$ aborts.

We note that due to the adjustments we made in $\mathcal{A}$ and the fact that the UA protocol is public coin (i.e., each verifier message is merely a fresh random string), the view of the simulated $\mathcal{A}$ is

[13] $\mathcal{A}$ may interleave calls for different proof sessions but in a monotone manner.

distributed identically to the view of $\mathcal{A}$ in a true interaction with HW^R. It follows that:

$$\begin{aligned}\Pr[\mathcal{P}^* \text{ fools } \mathcal{V}] &\geq \Pr[\mathcal{A} \text{ fools } \mathsf{HW}^R \ \wedge (i_1, i_2, i_3) \in_R [d]^3 \text{ hit the fooling session}] \\ &\geq \frac{\Pr[F]}{d^3} \geq \frac{\epsilon(n) - \mathrm{negl}(n)}{d^3} \ .\end{aligned}$$

We now show how $\mathcal{P}^*$ can be used to break the collision resistance of $\mathcal{H}$. To this end, fix the coins of $\mathcal{P}^*$ so that:

$$\Pr_{\mathcal{V},h}[\mathcal{P}^* \text{ fools } \mathcal{V}] \geq \frac{\epsilon}{d^3} - \mathrm{negl}(n) \ .$$

This, in particular, fixes the public key pk and encryption $\hat{C} = \mathsf{Enc}_{\mathsf{pk}}(C)$. This, together with the input h, determines the pair (v, e), which in turn determines the instance $y = (\mathbf{M}, (v, e, \mathsf{pk}, h, h(\hat{C}), s), T(n, s))$, for the UA.

We now treat $\mathcal{P}^*$ as a two stage prover strategy $\mathcal{P}_1^*, \mathcal{P}_2^*$, where $\mathcal{P}_1^*$ takes as input $h \leftarrow \mathcal{H}_n$ and generates an instance $y = (\mathbf{M}, (v, e, \mathsf{pk}, h, h(\hat{C}), s), T(n, s))$ and a state (state). Then, $\mathcal{P}_2^*(\mathsf{state})$ tries to prove to $\mathcal{V}$ that $y \in L_{\mathcal{U}}$. Our assumption on $\mathcal{P}^*$, together with a standard Markovian argument, implies that for a $\left(\frac{\epsilon}{2d^3} - \mathrm{negl}(n)\right)$-fraction of the functions in $h \in \mathcal{H}_n$,

$$\Pr\left[y \notin L_{\mathcal{U}} \ \wedge \ (\mathcal{P}_2^*(\mathsf{state}), \mathcal{V})(y) = 1\right] \geq \frac{\epsilon}{2d^3} \ , \tag{1}$$

where $(y, \mathsf{state}) = \mathcal{P}^*(h)$ and where the probability is over the random coin tosses of $\mathcal{V}$. We denote the set of $h \in \mathcal{H}$ that satisfy Equation (1) by GOOD. Our assumption that $\mathcal{P}^*$ is deterministic implies that for every $h \in \text{GOOD}$, it holds that $y \notin L_{\mathcal{U}}$, where $(y, \mathsf{state}) = \mathcal{P}^*(h)$.

We next define an algorithm $\mathcal{B}$ that finds collisions in $\mathcal{H}$: Given $h \leftarrow \mathcal{H}_n$, the algorithm $\mathcal{B}(h)$ computes $(y, \mathsf{state}) \leftarrow \mathcal{P}_1^*(h)$, where $y = (\mathbf{M}, (v, e, \mathsf{pk}, h, h(\hat{C}), s), T(n, s))$. Then, it does the following at most $M = \mathrm{poly}\left(n, \frac{2d^3}{\epsilon}\right)$ times:

1. Choose randomness r for the UA extractor E.
2. For every $i \in [s]$, compute $w_i = E_r^{\mathcal{P}_2^*(\mathsf{state})}(y, i)$. Denote the witness obtained by $\hat{C}_w$.
3. If $\hat{C}_w$ satisfies $h(\hat{C}_w) = h(\hat{C})$ and $e = \mathsf{Eval}_{\mathsf{pk}}(\hat{C}_w, U_{s,v})$, then it must be that $\hat{C}_w \neq \hat{C}$ and output the pair $(\hat{C}, \hat{C}_w)$ as a collision to h. Otherwise, go back to 1. If the number of repetitions exceeds M then abort.

We note the weak proof-of-knowledge property of the UA system (see Definition 2.6), together with a Chernoff bound, implies that for every $h \in \text{GOOD}$,

$$\Pr[\mathcal{B}(h) \text{ finds a collision}] = 1 - \mathrm{negl}(n) \ ,$$

where the probability is over the randomness of $\mathcal{B}$. This in turn implies that

$$\Pr[\mathcal{B}(h) \text{ finds a collision}] \geq \Pr[h \in \text{GOOD}] - \mathrm{negl}(n) \geq \frac{\epsilon}{2d^3} - \mathrm{negl}(n) \ ,$$

where the probability is over $h \leftarrow \mathcal{H}_n$ and over the randomness of $\mathcal{B}$.

This completes the proof of Lemma 3.1

□

Proving the validity of $\mathcal{S}$. Let $\mathcal{A}$ be a PPT adversary and let

$$\mathsf{View}_{\mathcal{A}} = \left\{ \mathcal{A}^{\mathsf{HW}_{\mathsf{params}}}(z, \mathsf{obf}) \right\}_{\substack{n \in \mathbb{N}, C \in \mathcal{C}_n \\ z \in \{0,1\}^{\mathrm{poly}(n)}}}$$

denote the output distribution of $\mathcal{A}$, where $\mathsf{obf} = \left(\mathsf{pk}, \hat{C}, h, s\right)$ and $\mathsf{params} = \left(\mathsf{sk}, \mathsf{pk}, f, h, h(\hat{C}), s\right)$. Denote by

$$\mathsf{View}_{\mathcal{S}} = \left\{ \mathcal{S}^{C}\left(1^{|C|}, z\right) \right\}_{\substack{n \in \mathbb{N}, C \in \mathcal{C}_n \\ z \in \{0,1\}^{\mathrm{poly}(n)}}}$$

the output distribution of the simulator $\mathcal{S}$. We show that

$$\mathsf{View}_{\mathcal{A}} \approx_c \mathsf{View}_{\mathcal{S}}.$$

To this end, we first consider an alternative simulation process $\mathsf{View}'_{\mathcal{S}}$, where $\mathcal{S}$ is given as extra auxiliary input a public key pk and an encryption of the circuit C, $\hat{C} = \mathsf{Enc}_{\mathsf{pk}}(C)$. It then uses pk and $\hat{C}$ for the emulation of $\mathcal{A}$ instead of using the dummy encryption. The semantic security of the encryption scheme (against polynomial size circuits) implies that $\mathsf{View}_{\mathcal{S}} \approx_c \mathsf{View}'_{\mathcal{S}}$.

Next, we note that the behavior of the emulated device in $\mathsf{View}'_{\mathcal{S}}$ differs from the behavior of the real device in $\mathsf{View}_{\mathcal{A}}$ only when the adversary $\mathcal{A}$ convinces the device HW that an invalid evaluation pair (v, e) is a valid one. In this case, the real device will decrypt the evaluation, while the simulated device will turn to the oracle C, which might produce a different answer. However, by Lemma 3.1 this event occurs only with negligible probability. The validity of $\mathcal{S}$ follows.

$\square$

Deducing composability. The proof above holds for the stand-alone case where $\mathcal{A}$ gets a single obfuscation. However, it can be easily extended to the case of multiple obfuscations, yielding a composable scheme. For this purpose, we note that $\mathcal{S}$ is a *straight-line black box simulator scheme.* That is, there is a single simulator $\mathcal{S}$ which simulates **all** PPT adversaries $\mathcal{A}$ as follows. Given input $(z, 1^n, |C|)$, $\mathcal{S}$ simulates an input obf for $\mathcal{A}$ using $(1^n, |C|)$ only (i.e. independently of z). $\mathcal{S}$ then runs $\mathcal{A}$ on (z, obf), simulating the answers for all the device calls made by $\mathcal{A}$. Composability now follows by the following general Proposition 3.1.

Proposition 3.1. *If* $(\mathcal{O}, \mathsf{HW})$ *has a straight-line black box simulator, then it is composable.*

The proof for Proposition 3.1 naturally follows the same ideas used in the *Universal Composition Theorem* [13].

4 General-Purpose (Sendable) Obfuscation

In this section we show how to convert any *circuit-specific* obfuscation scheme, such as the one in Section 3, to a scheme which uses a single *universal* (general-purpose) hardware device. The basic model and definitions are presented in Section 4.1, the transformation is presented in Section 4.2 and analyzed in Section 4.3.

4.1 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 4.1 (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:*

$$\left\{\mathcal{A}^{\mathsf{HW}_{\mathsf{prv}}}(z, \mathsf{obf}_1, \ldots, \mathsf{obf}_t)\right\}_{\substack{n \in \mathbb{N} \\ C_1 \ldots C_t \in \mathcal{C}_n \\ z \in \{0,1\}^{\mathrm{poly}(n)}}} \approx_c \left\{\mathcal{S}^{C_1, \ldots, C_t}(z, 1^n, |C_1|, \ldots |C_t|)\right\}_{\substack{n \in \mathbb{N} \\ C_1 \ldots C_t \in \mathcal{C}_n \\ z \in \{0,1\}^{\mathrm{poly}(n)}}},$$

where $(\mathsf{prv}, \mathsf{pub}) \leftarrow \mathsf{Gen}(1^n)$ *and* $\mathsf{obf}_i \leftarrow \mathcal{O}(C_i, \mathsf{pub})$ *and* z *is an arbitrary auxiliary input.*

4.2 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 3). The full description of the transformation is given in Figure 3.

- **Ingredients.** A circuit-specific scheme $(\mathcal{O}, \mathsf{HW})$ and a CCA2 scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$.
- **Parameter generation and device initialization.** Sample $(\mathsf{prv}, \mathsf{pub}) = (\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^n)$.
- **Obfuscator $\mathcal{O}'$.** Given a circuit C and a public key pk (of to the underlying CCA2 encryption scheme), sample $(\mathsf{params}, \mathsf{obf}) \leftarrow \mathcal{O}(C)$, compute $\mathsf{bind} = \mathsf{Enc}_{\mathsf{pk}}(\mathsf{params})$ and output $\mathsf{obf}' = (\mathsf{obf}, \mathsf{bind})$.
- **Device $\mathsf{HW}' = \mathsf{HW}'_{\mathsf{sk}}$.** On input $M' = (M, \mathsf{bind})$. Compute $\mathsf{params} = \mathsf{Dec}_{\mathsf{sk}}(\mathsf{bind})$. If params is not of a valid form return $\perp$; otherwise return $\mathsf{HW}_{\mathsf{params}}(M)$.
- **Evaluation.** Given $\mathsf{obf}' = (\mathsf{obf}, \mathsf{bind})$ and input v, apply the evaluation procedure of the underlying scheme using (obf, v). Each time the evaluation procedure queries its hardware device with input M, query HW' with input $M' = (M, \mathsf{bind})$.

Figure 3: From circuit-specific obfuscation to general-purpose obfuscation.

Theorem 4.1. *Assume* $(\mathcal{O}, \mathsf{HW})$ *is a circuit-specific obfuscation scheme as in Definition 3.1 and assume that* $\mathcal{E} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is a CCA2 secure encryption scheme. Then the scheme given in Figure 3 is a general-purpose obfuscation scheme as in Definition 4.1.*

Corollary 4.1 (of Theorems 3.1,4.1)**.** *Assume that there exists a fully homomorphic encryption scheme and a CCA2 encryption scheme, then there exists a general-purpose obfuscation scheme.*

Remark 4.1. 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 [29] and get a general-purpose solution that is based solely on the existence of CCA2 schemes, but which makes poly($|C|$) device calls.

4.3 Proof of Security

In this section we prove Theorem 4.1.

Proof outline. The functional correctness of the scheme follows directly from the correctness of the underlying circuit-specific obfuscation scheme and from the completeness of the underlying encryption scheme. Similarly, the polynomial slowdown property and the circuit-independent efficiency property follow immediately from the fact that the underlying circuit-specific obfuscation scheme has these properties. We thus focus on proving the virtual black-box property.

To this end, we show that the view of any adversary $\mathcal{A}'$ for the general-purpose scheme (who gets t obfuscations) can be simulated by an adversary $\mathcal{A}$ for the underlying circuit-specific scheme $(\mathcal{O}, \mathsf{HW})$, who gets t obfuscations according to $(\mathcal{O}, \mathsf{HW})$ (including access to t devices).

Informally, the adversary $\mathcal{A}^{\mathsf{HW}_1,\dots,\mathsf{HW}_t}(z, \mathsf{obf}_1, \dots, \mathsf{obf}_t)$ (who is given input as in $(\mathcal{O}, \mathsf{HW})$) generates a pair of keys $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^n)$ for the CCA2 secure encryption scheme. Then for every $i \in [t]$, it creates a dummy binding encryption $\mathsf{bind}_i = \mathsf{Enc}_{\mathsf{pk}}(\bar{0})$. It then emulates $(\mathcal{A}')^{\mathsf{HW}'_{\mathsf{sk}}}(z, (\mathsf{obf}_1, \mathsf{bind}_1), \dots, (\mathsf{obf}_t, \mathsf{bind}_t))$. This is done by separating its oracle queries into two types: (a) Honest calls, in which $\mathcal{A}'$ passes a correct value bind_i where $i \in [t]$; (b) Malicious calls, in which $\mathcal{A}'$ changes the binding ciphertexts and gives some other value $\mathsf{bind}' \notin \{\mathsf{bind}_i\}$. The calls of type (a) can be simulated by the corresponding device HW_i. For calls of type (b), $\mathcal{A}$ will decrypt the bind' value on its own and answer accordingly. The fact that $\mathcal{A}$ succeeds in emulating the output of $\mathcal{A}$ follows from the fact that if $\mathcal{A}$ distinguishes this emulated interaction from a real one, it would also be able to break the security of the CCA2 encryption scheme. Details follow.

- **Oracle Access.** Adversary $\mathcal{A}'$ (for the general-purpose scheme). Multiple devices $\left\{\mathsf{HW}_i = \mathsf{HW}_{\mathsf{params}_i}\right\}_{i \in [t]}$.
- **Input.** $\{\mathsf{obf}_i\}_{i \in [t]}$. Auxiliary input z (used by $\mathcal{A}'$).
- **Simulation.** Sample a pair of keys $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(1^n)$ for the underlying CCA2 secure encryption scheme. Create dummy binding encryptions $\mathsf{bind}_i = \mathsf{Enc}_{\mathsf{pk}}\left(0^{\ell_i}\right)$, where $\ell_i = |\mathsf{params}_i|$. Emulate an execution of $(\mathcal{A}')^{\mathsf{HW}'_{\mathsf{sk}}}$ on input $\Big(z, \{\mathsf{obf}_i, \mathsf{bind}_i\}_{i \in [t]}\Big)$.
- **Simulating the device $\mathsf{HW}'_{\mathsf{sk}}$.** On input (M, bind_i) feed the input M to HW_i. On input $\left(M, \mathsf{bind}'\right)$ where $\mathsf{bind}' \notin \{\mathsf{bind}_i\}_{i \in [t]}$, compute $\mathsf{params}' = \mathsf{Dec}'_{\mathsf{sk}}\left(\mathsf{bind}'\right)$. If params' is of a valid form return $\mathsf{HW}_{\mathsf{params}'}(M)$. Otherwise return $\perp$.
- **Output.** The output of the emulated $\mathcal{A}'$.

Figure 4: The Simulator $\mathcal{A}$.

Proof of Theorem 4.1. Let $\mathcal{A}'$ be a PPT adversary and denote the output distribution of $\mathcal{A}'$ by

$$\mathsf{View}_{\mathcal{A}'} = \left\{ (\mathcal{A}')^{\mathsf{HW}'_{\mathsf{sk}}} \left(z, \left\{ \mathsf{obf}'_i \right\}_{i \in [t]} \right) \right\}_{\substack{n \in \mathbb{N} \\ C_1, \dots, C_t \in \mathcal{C}_n \\ z \in \{0,1\}^{\mathrm{poly}(n)}}},$$

where $\mathsf{obf}'_i = (\mathsf{obf}_i, \mathsf{bind}_i)$, $(\mathsf{params}_i, \mathsf{obf}_i) \leftarrow \mathcal{O}(C_i)$ and $\mathsf{bind}_i = \mathsf{Enc}_{\mathsf{pk}}(\mathsf{params}_i)$. Denote the output distribution of the corresponding simulator by

$$\mathsf{View}_{\mathcal{A}} = \left\{ \mathcal{A}^{\{\mathsf{HW}_i\}} \left(z, \{\mathsf{obf}_i\}_{i \in [t]} \right) \right\}_{\substack{n \in \mathbb{N} \\ C_1, \dots, C_t \in \mathcal{C}_n \\ z \in \{0,1\}^{\mathrm{poly}(n)}}},$$

where $\mathsf{HW}_i = \mathsf{HW}_{\mathsf{params}_i}$ and $(\mathsf{params}_i, \mathsf{obf}_i) \leftarrow \mathcal{O}(C_i)$. We show that $\mathsf{View}_{\mathcal{A}'} \approx_c \mathsf{View}_{\mathcal{A}}$.

Indeed, assume there exists a PPT distinguisher $\mathcal{D}$ that distinguishes between these two distribution ensembles with non-negligible probability $\epsilon(n)$, for (infinitely many) $C_1, \ldots, C_t \in \mathcal{C}_n$ and $z \in \{0,1\}^{\mathrm{poly}(n)}$. We show how to construct a polynomial size breaker $\mathcal{B}$ that uses $\mathcal{D}$ to break the security of the CCA2 scheme $\mathcal{E}$ w.r.t t messages.

Breaker $\mathcal{B}$

- **Input.** pk (corresponding to a secret key sk).
- **Message generation.** Sample $\{(\mathsf{params}_i, \mathsf{obf}_i) \leftarrow \mathcal{O}(C_i)\}_{i\in[t]}$ as specified in the underlying circuit-specific obfuscation scheme. Send the CCA2 challenger $\vec{M}_0 = \{\mathsf{params}_i\}_{i\in[t]}, \vec{M}_1 = \left\{0^{|\mathsf{params}_i|}\right\}_{i\in[t]}$.
- **Challenge.** $\{\mathsf{bind}_i = \mathsf{Enc}_{\mathsf{pk}}(M_{b,i})\}_{i\in[t]}$, where $b \overset{U}{\leftarrow} \{0,1\}$.
- **Guessing b with access to a decryption oracle.** Given access to a decryption oracle $\mathsf{Dec}_{\mathsf{sk}}|_{\notin\{\mathsf{bind}_i\}}$ (which decrypts all ciphers but ciphers in $\{\mathsf{bind}_i\}$), emulate $\mathcal{A}'$ on input $\left(z, \{\mathsf{obf}_i, \mathsf{bind}_i\}_{i\in[t]}\right)$. When $\mathcal{A}'$ makes an oracle query (M, bind_i) to its hardware device answer with $\mathsf{HW}_{\mathsf{params}_i}(M)$. Given a call (M, bind'), where $\mathsf{bind}' \notin \{\mathsf{bind}_i\}$, use the decryption oracle to compute $\mathsf{params}' = \mathsf{Dec}_{\mathsf{sk}}(\mathsf{bind}')$. If params' is of a valid form, answer with $\mathsf{HW}'_{\mathsf{params}}(M)$. Otherwise, answer $\mathcal{A}'$ with $\perp$. Eventually, when $\mathcal{A}'$ outputs View, guess $b = \mathcal{D}(\mathsf{View})$.

Figure 5: CCA2 Breaker.

One can verify that when $b = 0$, View is distributed identically to the output of the real interaction $\mathsf{View}_{\mathcal{A}'}$, whereas when $b = 1$, View is distributed identically to output of the simulated interaction $\mathsf{View}_{\mathcal{A}}$. The result follows. $\square$

5 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 3 (or Section 4, 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.

5.1 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 5.2 and 5.3. The security analysis can be found in Section 5.4.

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) [15]. 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).[14] 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.

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 (as described in Section 2.3). 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(see details in Section 5.3).

5.2 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 ran-

[14]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 [11].

domness. 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 5.1 (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 5.2 (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).*[15] *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 5.1. In Definitions 3.1 and 4.1, 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 5.3. *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*

[15]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.

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.

Remark 5.2. As in Definitions 2.3 and 2.4, one could restrict the simulator $\mathcal{S}$ in Definition 5.3, to query the oracle C only on inputs x which $\mathcal{A}$ feeds to his circuit C'. We note that our construction guarantees these more stringent definitions (assuming the security of our underlying OCL^+ compiler satisfies these more stringent definitions). We chose not to add this restriction to the definitions for the sake of simplicity.

We now present the detailed construction, followed by its analysis.

5.3 The Construction

In this section we show how to construct obfuscation with leaky hardware.

We build our solution using a compiler $\mathcal{C}$ that is secure in the continual λ-OCL^+ model. (Recall that the OCL^+ model is similar to our distributed-system model, except that the different components can interact freely, not under the control of the adversary.) 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'), which are secure long as the adversary can only get λ leakage from each in each time unit and cannot see or influence the communication between them. (See Section 2.1 for definitions of the OCL^+ model.)

In our model, however, we need to worry about the communication between our components, so the system $(sub_1, \ldots, sub_m)$ cannot be used as such. To secure the communication, we use non-committing encryption (as defined in Section 2.2) for secrecy and use c-time leakage resilient MACs (as defined in Section 2.3) for authentication.

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

1. Apply the λ-OCL^+ compiler $\mathcal{C}$ to the circuit 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 $t = \Theta(\lambda) + n^{\Omega(1)}$.[16] For

[16]We usually consider λ that polynomially related to the security parameter n, so one can think of $t = \Theta(\lambda)$.

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.[17]

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 the non-committing encryption scheme NCE. 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 string $r_{G,k} \in \{0,1\}^{\mathrm{poly}(n)}$ 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 string $r_{E,k} \in \{0,1\}^{\mathrm{poly}(n)}$, 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 $K_{i,j}$.

[17]For simplicity, one can think of $K_{i,j} = K_{j,i}$.

Remark 5.3. For the sake of simplicity, we assume that at the end of each computation, all the hardware devices HW_i apply the Update procedure to their state and we think of these updates as starting at the same time and ending at the same time. This assumption is only for the sake of simplicity, to avoid the need to consider both the case where HW_i updates its own state and the case where it interacts with another HW_j during the Update of HW_j. Instead, we consider all these communications as part of the Update of each HW_i.

We denote $\mathcal{O}(C) = (\mathsf{HW}_1, \ldots, \mathsf{HW}_m : \mathsf{Update})$. The user (evaluator) should hand the input to HW_1 and get the output from HW_m. We next claim that the circuit $\mathcal{O}(C)$ is secure against continual λ-leakage attacks.

Theorem 5.1. *$\mathcal{O}$ is an obfuscator with continual λ-leaky hardware. Namely, for every* PPT *adversary $\mathcal{A}$, executing a continual λ-leakage attack, there exists a* PPT *simulator $\mathcal{S}$ such that for every ensemble of poly-size circuits $\mathcal{C} = \mathcal{C}_n$,*

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

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

5.4 Obfuscation with Leaky Hardware - Proof of Security

In this section we prove Theorem 5.1, i.e., we prove that the obfuscator above is secure against continual λ-leakage attacks. First, as a warm-up, we prove that the obfuscator is secure against single-input λ-leakage attacks. Namely, we change the construction so that the component HW_1 refuses to accept more than one input and then prove security against bounded leakage without an update. Then we prove that it is secure against continual λ-leakage attacks (with multiple inputs and corresponding updates).

Proof Sketch of Theorem 5.1 (for single-input λ-leakage attackers). Fix any PPT adversary $\mathcal{A}(z)[\lambda : \mathsf{HW}_1, \ldots, \mathsf{HW}_m]$ that executes a λ-leakage attack. We need to construct a PPT simulator $\mathcal{S}$, such that

$$\mathcal{A}(z)[\lambda : \mathsf{HW}_1, \ldots, \mathsf{HW}_m] \approx_c \mathcal{S}^C(z, 1^{|C|}) \ . \tag{2}$$

To this end, we construct a (OCL^+) PPT adversary $\mathcal{A}'(z)[\lambda : sub_1, \ldots, sub_m]$ that launches a single-input λ-leakage attack on $C' = (sub_1, \ldots, sub_m)$, such that

$$\mathcal{A}(z)[\lambda : \mathsf{HW}_1, \ldots, \mathsf{HW}_m] \approx_c \mathcal{A}'(z)[\lambda : sub_1, \ldots, sub_m] \ . \tag{3}$$

Then, we use the security of C' in the λ-OCL^+ model to claim that there exists a PPT simulator $\mathcal{S}$, such that

$$\mathcal{A}'(z)[\lambda : sub_1, \ldots, sub_m] \approx_c \mathcal{S}^C(z, 1^{|C|}) \ .$$

This, together with Equation (3), implies Equation (2), as desired.

Thus, it remains to construct a (OCL^+) PPT adversary $\mathcal{A}'(z)[\lambda : sub_1, \ldots, sub_m]$ that satisfies Equation (3). In what follows we define such a PPT adversary $\mathcal{A}'(z)[\lambda : sub_1, \ldots, sub_m]$. The adversary $\mathcal{A}'$ simulates $\mathcal{A}$ as follows.

1. Choose a random MAC key $K_{i,j}$ for every two communicating hardware devices HW_i and HW_j.

2. When the adversary $\mathcal{A}$ feeds HW_1 with some input $x \in \{0,1\}^n$, the adversary $\mathcal{A}'$ simulates the entire message exchange between all the hardware devices using the simulator NCSim of the NCE scheme, as follows: Each time HW_i is supposed to send HW_j a message (of length ℓ) via the NCE scheme, the adversary $\mathcal{A}'$ does the following:

   (a) For $k \in [\ell]$, sample an equivocal tuple, $(e_k, c_k, r^0_{G,k}, r^0_{E,k}, r^1_{G,k}, r^1_{E,k}) \leftarrow \mathsf{NCSim}(1^n)$.
   (b) Compute $\sigma_{\mathbf{e}} = \mathsf{MAC}(\mathbf{e}; K_{i,j})$
   (c) Simulate HW_j sending the pair $(\mathbf{e}, \sigma_{\mathbf{e}})$ to HW_i.
   (d) Compute $\sigma_{\mathbf{c}} = \mathsf{MAC}(\mathbf{c}; K_{i,j})$ and simulate HW_i sending the pair $(\mathbf{c}, \sigma_{\mathbf{c}})$ to HW_j.

   In addition, $\mathcal{A}'$ will start the computation of $C'(x)$ and obtain $y = C(x)$. Finally, it simulates HW_m outputting y.

   The adversary $\mathcal{A}$ sees all the communication $\mathsf{View} \triangleq (\{(\mathbf{e}, \sigma_{\mathbf{e}}), (\mathbf{c}, \sigma_{\mathbf{c}})\}, y)$ and based on this information, may choose to adaptively leak from each hardware device.

3. Each time $\mathcal{A}$ sends a leakage query L to HW_i, the adversary $\mathcal{A}'$ will send a leakage query L' to sub_i. Given the state of sub_i, L' will simulate the corresponding state of HW_i and apply L on the simulated state. The simulated state is computed as follows.

   (a) If $\mathcal{A}$ determines the leakage function L before it determines the input x, then the (secret) state of HW_i contains only the state of sub_i and the set of MAC keys $\{j, K_{i,j}\}$ for every sub_j that communicates with sub_i. In this case,
   $$L'(sub_i) \triangleq L(sub_i, \{j, K_{i,j}\}) \ ,$$
   simulates the leakage correctly, since
   $$(sub_i, \{j, K_{i,j}\}) \equiv \mathsf{state}(\mathsf{HW}_i) \ ,$$
   where $\mathsf{state}(\mathsf{HW}_i)$ is the internal state of HW_i, as desired and sub_i should be interpreted as the entire state of sub_i, including inputs and randomness.
   (b) If L is determined after the adversary $\mathcal{A}$ specifies the input x, then L' will also need to simulate all the encryption randomness within the state of HW_i. This includes both the randomness $\{r_G\}$, used in key generation for incoming messages and randomness $\{r_E\}$, used for encryption of outgoing messages.
   The input sub_i for L' determines all incoming and outgoing (plaintext) messages, which will be used to simulate the input for L as follows.
      i. For each incoming message $\mathbf{M} = (M_1, \dots, M_\ell)$ from some sub_j, use the equivocal (key generation) randomness, computed in Step $2(a)$ above, choosing the randomness corresponding to the plaintext $\mathbf{M}$. Denote this simulated randomness by $R_G(j, \mathbf{M}) = (r^{M_1}_{G,1} \dots, r^{M_\ell}_{G,\ell})$.
      ii. For each incoming message $\mathbf{M}' = (M'_1, \dots, M'_\ell)$ from some sub_j, use the equivocal (encryption) randomness according to the plaintext $\mathbf{M}'$, to compute
      $R_E(j, \mathbf{M}') = (r^{M'_1}_{E,1} \dots, r^{M'_\ell}_{E,\ell})$.

Finally, let

$$L'(sub_i) \triangleq L(sub_i, \{j, K_{i,j}\}, \{R_G(j, \mathbf{M})\}, \{R_E(j, \mathbf{M}')\}) \ .$$

Let $\mathsf{COM}_i^{\mathsf{SIM}}$ denote the incoming/outgoing communication of HW_i in a simulated execution and let $\mathsf{COM}_i^{\mathsf{REAL}}$ denote the communication in a real execution, with the additional assumption that the adversary does not interfere with the communication. (This assumption can be removed as explained later, in the Item 4). We note that by the definition of a non-committing encryption,

$$\mathsf{COM}_i^{\mathsf{SIM}}, (sub_i, \{j, K_{i,j}\}, \{R_G(j, \mathbf{M})\}, \{R_E(j, \mathbf{M}')\}) \approx_c \mathsf{COM}_i^{\mathsf{REAL}}, \mathsf{state}(\mathsf{HW}_i) \ .$$

This follows from the fact that if there exists a PPT adversary that can distinguish between the two, where in the latter the public keys and the ciphertext are generated using the NCGen and NCEnc (as opposed to using the simulator NCSim), then one can use this adversary to break the computational indistinguishability between the real view and the simulated view of the non-committing encryption (see Definition 2.5).

4. If at any time $\mathcal{A}$ tries to interfere the message exchange, by trying to feed HW_i an input (m, σ), which supposedly came from HW_j, the adversary $\mathcal{A}'$ discards this message.

   Note that if HW_j hasn't previously sent HW_i the message (m, σ) then the probability that σ is a valid MAC for m is negligible. This follows from the fact that the underlying MAC scheme is robust to λ-bits of leakage (i.e., it remains secure even if λ bits of its secret keys are leaked). Thus, with overwhelming probability HW_i will discard m. On the other hand, if (m, σ) is a message that was already sent from HW_j to HW_i, then HW_i also discards m.

Denote all the simulated communication between the hardware devices by

$$\mathsf{View} \triangleq \left\{\mathsf{COM}_i^{\mathsf{SIM}}\right\} \triangleq \{(\mathbf{e}, \sigma_{\mathbf{e}}), (\mathbf{c}, \sigma_{\mathbf{c}})\}$$

and by o_i the output of the i-th leakage query. The adversary $\mathcal{A}'$ outputs whatever $\mathcal{A}$ would had it received o_i from its i-th leakage query. Namely, $\mathcal{A}'$ outputs $\mathcal{A}(\mathsf{View}, o_1, \ldots, o_t)$.

All in all, it follows that the simulation of $\mathcal{A}$ is computationally indistinguishable from a real leakage attack and thus

$$\mathcal{A}(z)[\lambda; \mathsf{HW}_1, \ldots, \mathsf{HW}_m] \approx_c \mathcal{A}'(z)[\lambda; sub_1, \ldots, sub_m],$$

as desired. (Formally, the above is shown via a standard hybrid argument, where in the i'th hybrid, $\mathsf{HW}_1, \ldots, \mathsf{HW}_{i-1}$ communicate as in a simulated interaction, $\mathsf{HW}_{i+1}, \ldots, \mathsf{HW}_m$ communicate as in the real interaction and HW_i communicates with the first components as in a simulated interaction and with latter components as in a real interaction.)

□

Proof Sketch of Theorem 5.1 (for continual λ-leakage attacks). Fix any PPT adversary $\mathcal{A}(z)[\lambda; \mathsf{HW}_1, \ldots, \mathsf{HW}_m : \mathsf{Update}]$ that executes a continual λ-leakage attack. We need to construct a PPT simulator $\mathcal{S}$, such that

$$\mathcal{A}(z)[\lambda : \mathsf{HW}_1, \ldots, \mathsf{HW}_m : \mathsf{Update}] \approx_c \mathcal{S}^C(z, 1^{|C|}) \ , \tag{4}$$

To this end, as in the single-input case, we construct a (OCL^+) PPT adversary

$$\mathcal{A}'(z)[\lambda : sub_1, \ldots, sub_m : \mathsf{Update}']$$

that launches a continual λ-leakage attack on $\big(C' = (sub_1, \ldots, sub_m), \mathsf{Update}'\big)$, such that

$$\mathcal{A}(z)[\lambda : \mathsf{HW}_1, \ldots, \mathsf{HW}_m : \mathsf{Update}] \approx_c \mathcal{A}'(z)[\lambda : sub_1, \ldots, sub_m : \mathsf{Update}'] \ . \tag{5}$$

Then, we use the security of (C', Update') in the continual λ-OCL^+ to claim that there exists a PPT simulator $\mathcal{S}$ such that

$$\mathcal{A}'(z)[\lambda : sub_1, \ldots, sub_m : \mathsf{Update}'] \approx_c \mathcal{S}^C(z, 1^{|C|}) \ .$$

This, together with Equation (5), implies Equation (4), as desired. It remains to construct a (OCL^+) PPT adversary $\mathcal{A}'(z)[\lambda : sub_1, \ldots, sub_m : \mathsf{Update}']$ that satisfies Equation (5). In what follows we define such a PPT adversary $\mathcal{A}'(z)[\lambda : sub_1, \ldots, sub_m : \mathsf{Update}']$.

The adversary $\mathcal{A}'$ simulates $\mathcal{A}$ exactly as in the single-input case. The only difference is that in this setting the secret states are being updated and there may be leakage during the update procedures.

- The adversary $\mathcal{A}'$ simulates the Update procedure of each HW_i as follows:

  1. The secret state of sub_i is updated by the Update' procedure of $C' = (sub_1, \ldots, sub_m)$.
  2. To simulate the update of the secret keys $\{K_{i,j}\}$ for the MACs of HW_i, the adversary $\mathcal{A}'$ chooses a fresh key $K'_{i,j}$ for every $j > i$ for which HW_i and HW_j communicate. Then, it simulates HW_i sending $K'_{i,j}$ to HW_j encrypted using the NCE scheme and adds a MAC using the previous MAC key $K_{i,j}$. However, $\mathcal{A}'$ will use the simulator NCSim to simulate the ciphertext (rather than using the real encryption algorithm NCEnc). Namely, denote $K'_{i,j} = (b_1, \ldots, b_\ell)$. The adversary $\mathcal{A}'$ does the following.
     (a) For $k \in [\ell]$, compute $(e_k, c_k, r^0_{G,k}, r^0_{E,k}, r^1_{G,k}, r^1_{E,k}) \leftarrow \mathsf{NCSim}(1^n)$.
     Let $\mathbf{e} = (e_1, \ldots, e_\ell)$,$\mathbf{c} = (c_1, \ldots, c_\ell)$
     (b) Compute $\sigma_{\mathbf{e}} = \mathsf{MAC}(\mathbf{e}; K_{i,j})$ and simulate HW_j sending the pair $(\mathbf{e}, \sigma_{\mathbf{e}})$ to HW_i.
     (c) Compute $\sigma_{\mathbf{c}} = \mathsf{MAC}(\mathbf{c}; K_{i,j})$ and simulate HW_i sending the pair $(\mathbf{c}, \sigma_{\mathbf{c}})$ to HW_j.

  The simulation property of the NCE scheme (see Definition 2.5) implies that the simulated communication during the simulated update is computationally indistinguishable from the communication in the real Update.

- The adversary $\mathcal{A}'$ simulates any leakage query L that $\mathcal{A}$ sends to HW_i during the above Update procedure, by sending a leakage function L' to sub_i during its own Update procedure. L' applies L to the "simulated state" of HW_i, which contains the following values (in addition to the state of sub_i).

  1. The MAC keys $\{j, K_{i,j}\}$ for every $j \in [m]$ for which HW_i and HW_j interact.
  2. The new MAC keys $\{j, K'_{i,j}\}$ for every $j \in [m]$ for which HW_i and HW_j interact.

3. Simulated randomness for the transmission of the new keys $\{R_G(j, K'_{i,j})\}$ and $\{R_E(j, K'_{j,i})\}$, which is taken from the equivocal tuples generated in Step 2(a) above, according to each one of the plaintexts $K'_{i,j}$.

We remark that the incoming and outgoing messages (as well as the randomness for their transmission) are not a part of the state of HW_i during the update phase.

$\mathcal{A}'$ then creates the leakage function

$$L'(sub_i, r') \triangleq L\left(sub_i, r'; \{K_{i,j}\}; \{K'_{i,j}\}, \{R_G(j, K'_{j,i})\}, \{R_E(j, K'_{j,i})\}\right) ,$$

where r' denotes the randomness used by Update' and the tuple $(r', \{K'_{i,j}\}, \{R_G(j, K'_{j,i})\}, \{R_E(j, K'_{j,i})\})$ should be thought of as the randomness used by the Update procedure of HW_i. (Recall that as mentioned in Remark 5.3, we consider all the computation done by HW_i in between two consecutive computations, as part of the Update procedure of HW_i).

As was argued in the single-input case, the simulation property of the non-committing encryption scheme implies:

$$\left(sub_i, r'; \{K_{i,j}\}; \{K'_{i,j}\}, \{R_G(j, K'_{j,i})\}, \{R_E(j, K'_{j,i})\}\right), \mathsf{COM}_i^{\mathsf{SIM}} \approx_c \mathsf{state}(\mathsf{HW}_i), r, \mathsf{COM}_i^{\mathsf{REAL}} ,$$

where r denotes the real randomness used by Update and where $\mathsf{COM}_i^{\mathsf{SIM}}, \mathsf{COM}_i^{\mathsf{REAL}}$ denote the simulated and real communication during an update.

Therefore, $\mathcal{A}'$ simulates both the Update procedures and leakage during the Update procedures in a computationally indistinguishable manner. In addition, as argued in the single-input case, $\mathcal{A}'$ simulates the communication and the leakage which is not during the Update phase also in a computationally indistinguishable manner. Thus, we conclude that

$$\mathcal{A}(z)[\lambda; \mathsf{HW}_1, \ldots, \mathsf{HW}_m : \mathsf{Update}] \approx_c \mathcal{A}'(z)[\lambda; sub_1, \ldots, sub_m : \mathsf{Update}],$$

as desired. (As before, to show the above formally, one should use an hybrid argument, where in each hybrid the first invocations are performed as in a simulated execution and the last as in a real execution.)

□

5.5 The Leakage-Rate

Theorem 5.1 states that the obfuscator $\mathcal{O}$, constructed in Section 5.3, is secure with continual λ-leaky hardware. Here, λ refers to the maximal number of leakage bits that the underlying OCL^+ compiler can take, when the secret state of each sub_i is of size $\Sigma = \Sigma(\lambda, n)$ (where n is the security parameter). In what follows, we address the relation between λ and the size of the secret state of each HW_i in our construction, which we will denote by $\Sigma' = \Sigma'(\lambda, n)$. We show that (with a slight augmentation) our construction can tolerate the same leakage-rate as the underlying OCL^+ compiler, up to constant factors; namely, we show that $\Sigma' = O(\Sigma)$.

We first note the following facts regarding the construction presented in Section 5.3: During each activation, each component HW_i interacts with a constant number of components and this interaction includes exchanging a constant number of messages. Hence, the **secret** state of HW_i includes:

1. The secret state of sub_i, which is of size Σ.

2. A constant number of MAC keys $K_{i,j}$, each of size $\Theta(\lambda) + n^{\Omega(1)}$.

3. A constant number of decryption keys $\mathbf{d}$. Each $\mathbf{d}$ corresponds to a single incoming message $\mathbf{M}$ and is of size at least $|\mathbf{M}| \cdot n^{\Omega(1)}$. Indeed, $\mathbf{d}$ includes an NCE decryption key of size $n^{\Omega(1)}$ for each message bit. In particular, since the messages during updates include the MAC keys, the size of the corresponding $\mathbf{d}$'s is at least $\lambda \cdot n^{\Omega(1)}$.

4. When HW_i transmits a message, its secret state also includes the randomness r_E for the encryption, which is also done bit-by-bit. Here however, we can assume that after creating each encryption, r_E is deleted. Hence, the size of this part only is dominated by the other parts.

Therefore, the total size of the secret state is

$$\Sigma' = \Sigma + \Theta(\lambda) + \lambda \cdot n^{\Omega(1)} \ .$$

It follows that if $\Sigma = \lambda \cdot n^{\Omega(1)}$, then indeed $\Sigma' = O(\Sigma)$. For the [26] compiler (see Theorem 2.1) it indeed holds that $\Sigma = \Theta(\lambda^3)$, where λ is polynomially related to the security parameter n.

We next explain how to augment our scheme so that $\Sigma' = O(\Sigma)$, even if $\Sigma/\lambda = n^{o(1)}$. (Indeed, [28] achieve $\Sigma = O(\lambda)$, albeit with the assumption of a leak-free hardware component.) For this purpose, for each transmitted message $\mathbf{M}$ with ℓ bits, instead of storing all the keys in the corresponding tuple $\mathbf{d} = (d_1, \ldots, d_\ell)$, we would like to transmit $\mathbf{M}$ bit-by-bit and in between delete the keys d_i (as we do with the randomness r_E). However, this also implies that we would have to authenticate each corresponding public key e_i separately, instead of authenticating the entire $\mathbf{e}$ all at once, as we do now. This, however, can not be done, as long as we are using information theoretic MAC schemes. Indeed, the number of messages that can be authenticated by a single key can not exceed the key's length (while the messages do exceed the key length, as they include the MAC keys for the next activation). The solution for this is rather simple. Instead of transmitting the message itself bit-by-bit, we first establish one-time-pad random bits, using bit-by-bit NCE transmissions (with secret key erasures), but without authentication. Only once we finished transmitting, we authenticate all the communication at once. If the authentication went through, we can then use the one-time pads to transmit the message. Storing the random bits blows up the size only by an additive factor of $O(\Sigma)$, as required.

6 Putting it All Together.

To combine the results of Sections 3,4 and 5. We start with a "circuit-specific" scheme as the one constructed in Section 3. We then use the transformation from Section 4 to get a "general-purpose" scheme. Finally, we augment the scheme as follows. Instead of supplying the adversary with the single (general-purpose) device, we use the procedure in Section 5 to re-compile the device into multiple devices, which communicate via the adversary and can withstand leakage.

Specifically, our scheme consists of algorithms $(\mathsf{Gen}, \mathcal{O}, \mathsf{Eval})$. $\mathsf{Gen}(1^n, \overrightarrow{\mathsf{HW}})$ outputs public parameters pub and initializes the system of hardware devices $\overrightarrow{\mathsf{HW}} = \{\mathsf{HW}_i\}_{i\in[m]}$ with corresponding private parameters. The system $\overrightarrow{\mathsf{HW}}$ is then given to the evaluator and the parameters pub are published. Any party that wishes to send an obfuscation of a circuit C to the evaluator, applies $\mathsf{obf} \leftarrow \mathcal{O}(\mathsf{pub}, C)$ and sends the resulting blob obf. To evaluate C on input x, the evaluator applies $\mathsf{Eval}^{\overrightarrow{\mathsf{HW}}}(x, \mathsf{obf})$ (including interaction with the system $\overrightarrow{\mathsf{HW}}$).

The algorithms $(\mathsf{Gen}, \mathcal{O}, \mathsf{Eval})$ are instantiated according to the general-purpose scheme $(\mathsf{Gen}_{GP}, \mathcal{O}_{GP}, \mathsf{Eval}_{GP})$ (that is not leakage-resilient) and the LDS-obfuscator (compiler) $\mathcal{O}_{LDS}$. That is, Gen applies Gen_{GP} to obtain the public parameters pub and a secret circuit C_{prv} implementing the general-purpose device. It then applies $\mathcal{O}_{LDS}(C_{\mathsf{prv}})$ to get the initialized hardware system $\overrightarrow{\mathsf{HW}}$. The obfuscation $\mathcal{O}$ is done by applying $\mathcal{O}_{GP}$. The evaluation is done by applying Eval_{GP} as follows. Whenever, Eval_{GP} performs a query q meant for C_{prv}, we interact with the system $\overrightarrow{\mathsf{HW}}$ to compute $C_{\mathsf{prv}}(q)$.

Theorem 6.1. *Let* $(\mathsf{Gen}_{GP}, \mathcal{O}_{GP}, \mathsf{Eval}_{GP})$ *be a general-purpose obfuscation scheme (as in Definition 4.1) and let* $\mathcal{O}_{LDS}$ *be an LDS-obfuscator against* λ*-continual leakage (as in Definition 5.3). Then the scheme* $(\mathsf{Gen}, \mathcal{O}, \mathsf{Eval})$ *described above is a general-purpose obfuscation scheme which is secure in the leaky distributed system model against* λ*-continual leakage.*

Proof sketch. The functional correctness of the scheme is straight forward. We focus on the security requirement. We would like to show that any attacker $\mathcal{A}$ has a simulator $\mathcal{S}$ so that for any poly set of circuits $C_1, \ldots, C_t$ and auxiliary input $z \in \{0,1\}^{\mathrm{poly}(n)}$ it holds that:

$$\mathcal{A}(z, \mathsf{obf}_1, \ldots, \mathsf{obf}_t)\left[\lambda : \overrightarrow{\mathsf{HW}}\right] \approx_c \mathcal{S}(z)^{C_1,\ldots,C_t}(z, |C_1|, \ldots, |C_t|)$$

Where $\mathsf{obf}_i \leftarrow \mathcal{O}_{GP}(\mathsf{pub}, C_i)$, $(\mathsf{pub}, C_{\mathsf{prv}}) \leftarrow \mathsf{Gen}_{GP}(1^n)$ and $\overrightarrow{\mathsf{HW}} \leftarrow \mathcal{O}_{LDS}(C_{\mathsf{prv}})$.

Indeed, since $\mathcal{O}_{LDS}$ is secure against λ-continual leakage, $\mathcal{A}$ can be simulated by a PPT simulator $\mathcal{S}_{LDS}$ that only gets oracle access to C_{prv}, i.e.:

$$\mathcal{A}(z, \mathsf{obf}_1, \ldots, \mathsf{obf}_t)\left[\lambda : \overrightarrow{\mathsf{HW}}\right] \approx_c \mathcal{S}_{LDS}^{C_{\mathsf{prv}}}(z, \mathsf{obf}_1, \ldots, \mathsf{obf}_t)$$

Moreover, since $(\mathsf{Gen}_{GP}, \mathcal{O}_{GP})$ is a general-purpose scheme $\mathcal{S}_{LDS}$ has a simulator $\mathcal{S}$ such that:

$$\mathcal{S}_{LDS}^{C_{\mathsf{prv}}}(z, \mathsf{obf}_1, \ldots, \mathsf{obf}_t) \approx_c \mathcal{S}(z)^{C_1,\ldots,C_t}(z, |C_1|, \ldots, |C_t|)$$

The result follows. □

References

[1] Miklós Ajtai. Secure computation with information leaking to an adversary. In *STOC*, pages 715–724, 2011.

[2] Adi Akavia, Shafi Goldwasser, and Carmit Hazay. Distributed Public Key Encryption Schemes. manuscript, 2010.

[3] 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.

[4] Mustafa Atici and Douglas R. Stinson. Universal hashing and multiple authentication. In *CRYPTO*, pages 16–30, 1996.

[5] Boaz Barak. How to go beyond the black-box simulation barrier. In *FOCS*, pages 106–115, 2001.

[6] Boaz Barak and Oded Goldreich. Universal arguments and their applications. *SIAM J. Comput.*, 38(5):1661–1694, 2008.

[7] Boaz Barak, Oded Goldreich, Shafi Goldwasser, and Yehuda Lindell. Resettably-sound zero-knowledge and its applications. In *FOCS*, pages 116–125, 2001.

[8] 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.

[9] Robert M. Best. Microprocessor for executing enciphered programs. US Patent 4168396, 1979.

[10] Nir Bitansky and Ran Canetti. On strong simulation and composable point obfuscation. In *Advances in Cryptology - CRYPTO 2010*, pages 520–537, 2010.

[11] Nir Bitansky, Ran Canetti, and Shai Halevi. Leakage tolerant interactive protocols. Manuscript, 2011. `http://eprint.iacr.org/2011/204`.

[12] Dan Boneh, Shai Halevi, Michael Hamburg, and Rafail Ostrovsky. Circular-Secure Encryption from Decision Diffie-Hellman. In *Advances in Cryptology - CRYPTO 2008*, volume 5157 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2008.

[13] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145, 2001.

[14] Ran Canetti and Ronny Ramzi Dakdouk. Obfuscating point functions with multibit output. In *EUROCRYPT'08*, pages 489–508, 2008.

[15] 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.

[16] Ran Canetti, Rosario Gennaro, Amir Herzberg, and Dalit Naor. Proactive security: Long-term Protection against break-ins. *CryptoBytes*, 3(1), 1997.

[17] Ran Canetti, Oded Goldreich, Shafi Goldwasser, and Silvio Micali. Resettable zero-knowledge (extended abstract). In *STOC*, pages 235–244, 2000.

[18] Ran Canetti, Guy N. Rothblum, and Mayank Varia. Obfuscation of hyperplane membership. In *TCC*, pages 72–89, 2010.

[19] Seung Geol Choi, Dana Dachman-Soled, Tal Malkin, and Hoeteck Wee. Improved non-committing encryption with applications to adaptively secure protocols. In *ASIACRYPT*, pages 287–302, 2009.

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

[21] 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.

[22] 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.

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

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

[25] 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.

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

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

[28] 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.

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

[30] Vipul Goyal and Amit Sahai. Resettably secure computation. In *EUROCRYPT*, pages 54–71, 2009.

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

[32] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In *CRYPTO*, pages 463–481, 2003.

[33] 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.

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

[35] Joe Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *STOC*, pages 723–732, 1992.

[36] Silvio Micali. Computationally sound proofs. *SIAM J. Comput.*, 30(4):1253–1298, 2000.

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

[38] 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.

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

[40] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In *Advances in Cryptology - EUROCRYPT'10*, volume 6110 of *Lecture Notes in Computer Science*, pages 24–43. Springer, 2010. `http://eprint.iacr.org/2009/616`.

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

[42] 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.