2012 IEEE 53rd Annual Symposium on Foundations of Computer Science

How to Compute in the Presence of Leakage

Shafi Goldwasser*
MIT and The Weizmann Institute of Science
shafi@theory.csail.mit.edu

Guy N. Rothblum**
Microsoft Research, Silicon Valley
rothblum@alum.mit.edu

***Abstract*—We address the following problem: how to execute any algorithm P, for an unbounded number of executions, in the presence of an adversary who observes partial information on the internal state of the computation during executions. The security guarantee is that the adversary learns nothing, beyond P's input/output behavior.**

This general problem is important for running cryptographic algorithms in the presence of side-channel attacks, as well as for running non-cryptographic algorithms, such as a proprietary search algorithm or a game, on a cloud server where parts of the execution's internals might be observed.

Our main result is a compiler, which takes as input an algorithm P and a security parameter κ, and produces a functionally equivalent algorithm P'. The running time of P' is a factor of $\mathrm{poly}(\kappa)$ slower than P. P' will be composed of a series of calls to $\mathrm{poly}(\kappa)$-time computable sub-algorithms. During the executions of P', an adversary algorithm $\mathcal{A}$, which can choose the inputs of P', can learn the results of adaptively chosen leakage functions - each of bounded output size $\widetilde{\Omega}(\kappa)$ – on the sub-algorithms of P' and the randomness they use.

We prove that any *computationally unbounded* $\mathcal{A}$ observing the results of *computationally unbounded leakage functions*, will learn no more from its observations than it could given black-box access only to the input-output behavior of P. This result is unconditional and does not rely on any secure hardware components.

I. Introduction

This work addresses the question of how to compute any program P, for an unbounded number of executions, so that an adversary who can obtain partial information on the internal states of executions of P on inputs of its choice, learns nothing about P beyond its I/O behavior. Throughout the introduction, we will call such executions *leakage resilient*.

This question is of importance for non-cryptographic as well as cryptographic algorithms. In the setting of cryptographic algorithms, the program P is usually viewed as a combination of a public algorithm with a secret key, and the secret key should be protected from side channel attacks. Stepping out of the cryptographic context, P may be a proprietary search algorithm or a novel numeric computation procedure which we want to protect, say while running on an insecure environment, say a cloud server, where its internals can be partially observed. Looking ahead, our results will not rely on computational assumptions and thus will be applicable to non-cryptographic settings without adding any new conditions. They will hold even if $P = NP$ (and cryptography as we know it does not exist).

A crucial aspect of this question is how to model the partial information or *leakage* attack that an adversary can launch during executions. Proper modeling should simultaneously capture real world attacks and achieve the right level of theoretical abstraction. Furthermore, impossibility results on obfuscation [1] imply inherent limitations on the leakage attacks which can be tolerated by general programs: [2] observes that if the leakage attack model allows even a single bit of leakage to be computed by an adversarially chosen polynomial-time function applied to the entire internal state of the execution, then there exist programs P which cannot be executed in a leakage resilient manner. Thus, to enable any algorithm to run securely in the presence of continual leakage, we must put restrictions on the leakage attack model to rule out this impossibility result.

Several different leakage attack models have been considered (and meaningful results obtained) in the literature. We briefly survey these models here (and later compare known results in these models to our results).

Wire Probe (ISW-L) The pioneering work of Ishai, Sahai, and Wagner [3] first considered the question of converting general algorithms to equivalent leakage resistant algorithms. Their work views algorithms as stateful circuits (e.g. a cryptographic algorithm, whose state is the secret-key of an algorithm), and considers adversaries which can learn the value of a bounded number of wires in each execution of the circuit, whereas the values of all other wires in this execution are perfectly hidden and that all internal wire values are erased between executions.

$\mathcal{AC}^0$ Bounded Leakage(CB-L). Faust, Rabin, Reyzin, Tromer and Vaikuntanathan [4] modify the leakage model and result of [3]. They still model an algorithm as a stateful circuit, but in every execution, they let the adversary learn the result of any $\mathcal{AC}^0$ computable function f computed on the values of all the wires. Similarly to the [3] model they

* Supported in part by NSF grants CCF-0915675, CCF-1018064 and DARPA under agreement numbers FA8750-11-C-0096 and FA8750-11-2-0225. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

** Part of this research was done while the author was at Princeton University and supported by NSF Grant CCF-0832797 and by a Computing Innovation Fellowship.

0272-5428/12 $26.00 © 2012 IEEE
DOI 10.1109/FOCS.2012.34

also place a total bound on the output length of this $\mathcal{AC}^0$ function f. To obtain results in this model, [4] also augment the model to assume the existence of leak free hardware components that produce samples from a polynomial time sampleable distribution. It is assumed that there is no data leakage from the randomness generated and the computation performed inside of the device.

Only-Computation Leaks (OC-L). The Micali-Reyzin [5] only-computation axiom assumes that there is no leakage in the absence of computation, but computation always does leak. This axiom was used in the works of Goldwasser and Rothblum [6] and by Juma and Vhalis [7], who both transform an input algorithm P (expressed as a Turing Machine or a boolean circuit) into an algorithm P', which is divided into subcomputations. An adversary can learn the the value of *any* (adaptively chosen) polynomial time computable length bounded function called a *leakage function*,[1] computed on each sub-computation's input and randomness.

To obtain results in this model, both [6] and [7] augment the model to assume the existence of leak free hardware components that produce samples from a polynomial time sampleable distribution. It is assumed that there is no data leakage from the randomness generated and the computation performed inside of the device. Similarly, in independent work, Dziembowski and Faust [8] also assume leak-free components. Unlike [6], [7], they do not bound the computational power of the adversary.

RAM Cell Probe (RAM-L). The RAM model of Goldreich and Ostrovsky [9] considers an architecture that loads data from fully protected memory, and runs computations in a secure CPU. [9] allowed an adversary to view the access pattern to memory (and showed how to make this access pattern oblivious), but assumed that the CPU's internals and the contents of the memory are perfectly hidden.[2] This was recently extended by Ajtai [10]. He divides the execution into sub-computations. Within each sub-computation, the adversary is allowed to observe the *contents* of a constant fraction of the addresses read from memory (This is similar to the ISW-L model, in that a portion of memory-addresses used in a computation are either fully exposed or fully hidden, except that [10] works in the RAM model and divides the executions into sub-computations whereas [3] work in the stateful circuit model). These are called the compromised memory accesses (or times). The contents of the un-compromised addresses, and the contents of the main memory not loaded into the CPU, are assumed to be perfectly hidden.

[1] In contrast to the $\mathcal{AC}^0$ restriction on f in [4]

[2] alternatively, they assume that the memory contents are encrypted, and their decryption in the CPU is perfectly hidden.

A. The New Work

In this work, show how to transform any algorithm P into a functionally equivalent and leakage resilient algorithm $Eval$, which can be run for an unbounded number of executions, without using any secure hardware or any intractability assumptions. We work within the OC-L leakage model, but we further allow the adversary to be computationally unbounded and the leakage on sub-computations to be the result of evaluating computationally unbounded leakage functions. We proceed to precisely describe the power of our adversary, and the security guarantee to be provided:

Computationally Unbounded OC-L Leakage Adversary. The leakage attacks we address are in the "only computation leaks information" model of [5]. The algorithm $Eval$ will be composed of a sequence of calls to sub-computations. The *leakage adversary* A^λ, on input a security parameter 1^κ, can (1) specify a polynomial number of inputs to P and (2) per execution of $Eval$ on input x, request for every sub-computation of $Eval$, any λ bits of information of its choice, computed on the entire internal state of the sub-computation, including any randomness the sub-computation may generate. We stress that we did not put any restrictions on the complexity of the leakage Adversary A^λ, and that the requested λ bits of leakage may be the result of computing a computationally unbounded function of the internal state of the sub-computation.

Security Guarantee. Informally, the security guarantee that we provide will be that for any leakage adversary A^λ, whatever A^λ can compute during the execution of $Eval$, it can compute with black-box access to the algorithm P. Formally, this is proved by exhibiting a simulator which, for every leakage-adversary A^λ, given black box access to the functionality P, simulates a view which is *statistically indistinguishable* from the real view of A^λ during executions of $Eval$. The simulated view will contains the results of I/O calls to P, as well as results of applying leakage functions on the sub-computations as would be seen by A^λ. The running time of the simulator is polynomial in the running time of A^λ and the running time of the leakage functions A^λ chooses.

Main Theorem (Informal). We show a compiler that takes as input a program, in the form of a circuit family $\{C_n\}$, a secret state $y \in \{0,1\}^n$, and a security parameter κ, and produces as output a description of an uniform stateful algorithm $Eval$ such that:

1) $Eval(x) = C(y, x)$ for all inputs x.
2) The execution of $Eval(x)$ for $|x| = n$, will consist of $O(|C_n|)$ sub-computations, each of complexity (time and space) $\tilde{O}(\kappa^\omega)$, (where ω is the exponent in the best algorithm known for matrix multiplication).
3) There exists a simulator Sim, a leakage bound $\lambda(\kappa) = \tilde{\Omega}(\kappa)$, and a negligible distance bound $\delta(\kappa)$, such that for every leakage-adversary $A^{\lambda(\kappa)}$ and $\kappa \in \mathbb{N}$:

$Sim^C(1^\kappa, \mathcal{A})$ is $\delta(\kappa)$-statistically close to $view(A^\lambda)$, where $Sim^C(1^\kappa, \mathcal{A})$ denotes the output distribution of Sim, on input the description of $\mathcal{A}$, and with black-box access to C. $view(A^\lambda)$ is the view of the leakage adversary during a polynomial number of executions of $Eval$ on inputs of its choice. The running time of Sim is polynomial in that of $\mathcal{A}$ and that of the leakage functions chosen by $\mathcal{A}$. The number of oracle calls made is always $\mathrm{poly}(\kappa)$.

We emphasize that our result holds unconditionally, *without any leak-free hardware* or any computational assumptions. We stress that this is in contrast to all known works [4], [6], [7], [8], [11] on resilience of general programs against continual leakage that consider non-trivial[3] leakage functions. See Section III for a fuller comparison with prior work on leakage resilience for general programs. See Section I-B for a description of the "leaky CPU" model, an alternative to the OC leakage model.

Doing Away with Secure Hardware. The idea behind doing away with the need for secure hardware, is to first note that in previous works the use of hardware was to sample randomly from polynomial time computable distribution D_b where D_b corresponded to encryptions (or encodings) of bit b (where $b \in \{0, 1, r\}$ for r randomly chosen in $\{0, 1\}$) without leaking the coins used to compute the encryptions. The new idea will be for the compiler at the time of compiling C to prepare what we will call "ciphertexts banks", which will be a collection of samples from the relevant distributions D_b, and show we can continually "regenerate" new samples from older ones in a leakage-resilient manner by taking appropriate linear combinations of collections of ciphertexts.

Doing Away with Computational Assumptions. Previous works relied on the existence of homomorphic properties of an underlying public-key encryption scheme with good leakage resilience properties and good key-refreshing possibilities, which helped carry out the computation in a "leakage resilient" manner. We observe however that there is no need to use a public-key encryption scheme in the context of secure execution, as the scheme is not used for communication but rather as a way to carry out computation in a "secrecy-preserving" fashion. Once we make the shift to a private-key encryption scheme which offers sufficient homomorphism for our usage, we are able to inject new entropy into the key "on the fly", as the computation progresses, and to achieve unconditional security. It is crucial to use the fact that the user executing the compiled circuit in this setting is trusted rather than adversarial, and thus will choose independent randomness for this entropy boosting operation.

[3]By non-trivial, we mean that the leakage function performs some a non-trivial computation on wires or memory accesses in the execution, rather than simply releasing their values as in [3], [10].

The new private-key encryption method is simple, and uses the inner product function. The key is a string $key \in \{0,1\}^\kappa$, and the encryption of a bit $b \in \{0,1\}$ is a ciphertext $\vec{c} \in \{0,1\}^\kappa$ s.t. b is the inner product of key and $\vec{c}$. This simple scheme is resilient to separate leakage on the key and the ciphertext, is homomorphic under addition, and is refreshable. The technical challenge will be to show why these properties (and in particular this level of homomorphism) suffice.

We also mention that, whereas our focus is on enabling any algorithm to run securely in the presence of continual leakage, continual leakage on *restricted computations* (e.g. [12], [13], [14], [15], [16], [17], [18]), and on *storage* ([19]), has been considered under various additional leakage models in a rich body of recent works. See Section III for further discussions of related work.

B. Leaky CPU: An Alternative to OC-Leakage

A question that is often raised regarding the OC-L model is what constitutes a reasonable division of computation to basic sub-computations (on which leakage is computed). We suggest that to best address this question, one should think of the OC-L model in terms of an alternative model which we call a *leaky CPU*. A leaky CPU will consist of an instruction set of constant size, where instructions correspond to basic sub-computations in the OC-L model , and the instruction set is universal in the sense that every program can be written as in terms of a sequence of calls to instructions from this set. The operands to an instruction can be leaked on when the instruction is executed.

We proceed with an slightly more formal description of this model. Computations are run on a RAM with two components:

1) A CPU which executes instructions from a fixed set of special universal instructions, each of size $\mathrm{poly}(\kappa)$ for a security parameter κ.
2) A memory that stores the program, input, output, and intermediate results of the computation. The CPU fetches instructions and data and stores outputs in this memory.

The adversary model is as follows:

1) For each program instruction loaded and executed in the CPU, the adversary can learn the value of an arbitrary and adaptively chosen leakage function of bounded output length (output length $\Omega(\kappa)$ in our results). The leakage function is applied to the instruction executed in the CPU – namely, it is a function of all inputs, outputs, randomness, and intermediate wires of the CPU instruction being executed.
2) Contents of memory, when not loaded into the CPU, are hidden from the adversary.

Our result, stated in this model, provides a fixed set of CPU instructions, and a compiler which can take any

polynomial time computation (say given in the form of a boolean circuit), and compile it into a program that can be run on this leaky CPU. A leakage adversary as above, who can specify inputs to the compiled program and observe its outputs, learns nothing from the execution beyond its input-out behavior. We elaborate on the set of instructions required by our compiler in Section II-D

C. Applications of the Main Theorem

Application of Our Compiler to Obfuscation with Leaky Hardware. There is a fascinating connection between the problem of code obfuscation and leakage resilience for general programs. In a nut-shell, one may think of obfuscation of an algorithm as the ultimate "leakage resilient" transformation: If successful, it implies that the resulting algorithm can be *"fully leaked"* to the adversary – it is under the adversary's complete control! Since we know that full and general obfuscation is impossible [1], we must relax the requirements on what we may hope to achieve when obfuscating a circuit. Leakage resilient versions of algorithms can be viewed as one such relaxation. In particular, one may view our result as showing that although we cannot protect general algorithms if we give the adversary complete view of code which implements the algorithm (i.e obfuscation), nevertheless we can (for any functionality) allow an adversary to have a "partial view" of the execution and only learn its black-box functionality. In our work, this "partial view" is as defined by the "only computation leaks" leakage attack model.

In a recent work, Bitansky *et al.* [20] make the connection between the OCL attack model and obfuscation explicit. They use the compiler described here to obfuscate programs using simple secure hardware components which are "leaky": they may be subject to memory leakage attacks. At a high level, they run each "sub-computation" on a separate hardware component which is subject to memory leakage. Alternatively, viewing OC leakage as an attack in the leaky CPU model, each instruction of the leaky CPU is implemented in a separate hardware component. The main challenge in their setting is providing security even when the communication channels between the hardware components are observed and controlled by an adversary, which they address using non-committing encryption [21] and MACs .

Application of Our Compiler to Leakage Resilient Multi Party Computation. In a recent work, Boyle *et al.* [22] use our compiler (and the assumption that FHE exists) to build secure MPC protocols that can compute an unbounded number of polynomial time functions C_i on an input (which has been shared among the players a one-time leak free pre-processing stage) that are resilient to corruptions of a constant fraction of the players and to leakage on all of the rest of the players (separately). Intuitively, one can think of each player in the MPC as running one of the "sub-computations" in a compilation of C_i using our OC-L compiler. Alternatively, viewing of OC leakage as an attack in the leaky CPU model, operations of the leaky CPU are implemented in by different players in the MPC protocol. The additional challenges here are both adversarial monitoring/control of the communication channels and (more significantly) that the adversary may completely corrupt many of the players/sub-computations.

II. Compiler Overview and Technical Contributions

In this section we overview the compiler and the main technical ideas introduced. The compiler takes any algorithm in the form of a (public) Boolean circuit $C(y, x)$ with a "secret" fixed input y, and transforms it into a functionally equivalent probabilistic stateful algorithm that on input x outputs $C(y, x)$ (for the fixed secret y). Each execution of the transformed algorithm consists of a sequence of sub-computations, and the adversary's view of each execution is through applying a sequence of adaptively chosen bounded-length leakage functions to these sub-computations. We overview the construction in three steps:

- In Section II-A we describe the first tool in our construction: a leakage-resilient one-time pad cryptosystem (LROTP), which is used as the subsidiary cryptosystem[4] in our construction, and is resilient to bounded leakage. In particular, we use this cryptosystem to encrypt the secret fixed input y.
- In Section II-B, we show how to use these encryptions to compute the program's output *once* on a given input. This "one-time" safe evaluation is resilient to bounded OC leakage attacks. The main challenge is to develop a "safe homomorphic evaluation" procedure for computing the NAND of LROTP encrypted bits.
- In Section II-C we show how to extend the "one-time" safe evaluation to "any polynomial number" of safe evaluations on new inputs, i.e. to resist *continual* leakage. The main new technical tool introduced here, and where the bulk of technical difficulty of our paper lies, is in using what we call "ciphertext banks": they will allow the *repeated* generation of secure ciphertexts even under leakage.

Put together, this yields a compiler that is secure against continual OC leakage attacks.

A. Leakage-Resilient One Time Pad

One of the main components of our construction is the *"leakage resilient one-time pad"* cryptoscheme (LROTP).

[4]It is important to distinguish between leakage on the secret input y taken as input by the compiler, and the leakage resilience of the subsidiary LROTP keys and ciphertexts. Whereas the LROTP keys and ciphertexts are leaked on (separately) and are designed to retain security in spite of this leakage, there is no leakage on y: all that an adversary can learn about y is the input-output behavior of $C(y, x)$.

This simple private-key encryption scheme uses a vector $key \in \{0,1\}^\kappa$ as its secret key, and each ciphertext is also a vector $\vec{c} \in \{0,1\}^\kappa$. The plaintext underlying $\vec{c}$ (under key) is the inner product: $Decrypt(key, \vec{c}) = \langle key, \vec{c} \rangle$. The scheme maintains the invariants that $key[0] = 1, \vec{c}[1] = 1$, for any key and ciphertext $\vec{c}$. We generate each key to be uniformly random under this invariant. To encrypt a bit b, we choose a uniformly random $\vec{c}$ s.t. $\vec{c}[1] = 1$ and $Decrypt(key, \vec{c}) = b$.

The LROTP scheme is remarkably well suited for our goal of transforming general computations to resist leakage attacks. We use the following properties (see the full version for details):

Semantic Security under Multi-Source Leakage. Semantic security of LROTP holds against an adversary who launches leakage attacks on both a key *and a ciphertext encrypted under that key*. This might seem impossible at first glance. The reason it is facilitated is two-fold: first due to the nature of our attack model, where the adversary can never apply a leakage function to the ciphertext and the secret-key simultaneously (otherwise it could decrypt); second, the leakage from the key and from the ciphertext is of bounded length. This ensures, for example, that the adversary cannot learn enough of the ciphertext to be useful for it at a later time—when it could apply an adaptively chosen leakage function to the secret key (and, for example, decrypt).

Translating this reasoning into a proof, we show that semantic security is retained under concurrent attacks of bounded leakage $O(\kappa)$ length on key and $\vec{c}$. As long as leakage is of bounded length and operates separately on key and on $\vec{c}$, they remain (w.h.p.) high entropy sources, and are independent up to their inner product equaling the underlying plaintext. Since the inner product function is a two-source extractor, the underlying plaintext is statistically close to uniformly random even given the leakage. Moreover, this is true even for computationally unbounded adversaries and leakage functions. To ensure that the leakage operates separately on key and $\vec{c}$, we take care in our construction not to load ciphertexts and keys into working memory simultaneously.[5] We note that two-source extractors were used for enabling leakage-resilient cryptography in [6], [23], [8].

Key and Ciphertext Refreshing. We give procedures for "refreshing" an LROTP key and ciphertexts: the output key and ciphertext will be a "fresh" uniformly random encryption of the same underlying plaintext bit. Moreover, the refreshing procedure operates separately on the key and on the ciphertext, and so an OC leakage attack will not be able to determine the underlying plaintext. In fact, security of the underlying plaintext is maintained even under OC leakage from multiple composed applications of the refreshing procedure. Security is maintained as long as the accumulated leakage is a small constant fraction of the key and ciphertext length. After a large enough number of composed applications, however, security is lost: An OC leakage adversary can successfully reconstruct the underlying plaintext. This attack is described in the full version. Intuitively, it "kicks in" once the length of the accumulated leakage is a large constant fraction of the key and ciphertext length.

Homomorphic Addition. For key and two ciphertexts $\vec{c}_1, \vec{c}_2$, we can homomorphically add by computing $\vec{c}' \leftarrow (\vec{c}_1 \oplus \vec{c}_2)$. By linearity, the plaintext underlying $\vec{c}'$ is the XOR of the plaintexts underlying $\vec{c}_1$ and $\vec{c}_2$. For a key-ciphertext pair $(key, \vec{c})$ and a plaintext bit b, we can homomorphically add plaintext to the ciphertext by computing $\vec{c}' \leftarrow (\vec{c} \oplus (b, 0, \ldots, 0))$. Since $key[0] = 1$ we get that the plaintext underlying $\vec{c}'$ is the XOR of b and the plaintext underlying $\vec{c}$.

We note that the construction in [6] relied on several similar properties of a computationally secure public-key leakage resilient scheme: the BHHO/Naor-Segev scheme [24], [25]. Here we achieve these properties with information theoretic security and without relying on intractability assumptions such as Decisional Diffie Hellman.

B. One-Time Secure Evaluation

Next, we describe the high-level structure of the compilation and evaluation algorithm for a *single* secure execution. In Section II-C we will show how to extend this framework to support any polynomial number of secure executions. The *input* to the compiler is a *secret* input $y \in \{0,1\}^n$, and a *public circuit* C of size $poly(n)$ that is known the adversary. The circuit takes as inputs the secret y, and also public input $x \in \{0,1\}^n$ (which may be chosen by the adversary), and produces a single bit output.[6] For example, C can be a public cryptographic algorithm (say for producing digital signatures), y a secret key, and x a public message to sign. More generally, to compile general algorithms, C can be the universal circuit, y the description of any particular algorithm that is to be protected, and x a public input to the protected algorithm.

The *output* of the compiler on C and y is a probabilistic stateful evaluation algorithm $Eval$ (with a *state* that will be updated during each run of $Eval$), such that for all $x \in \{0,1\}^n$, $C(y,x) = Eval(y,x)$. The compiler is run exactly once at the beginning of time and is not subject to leakage. See the full version for a formal definition of utility and security under leakage. In this section, we describe

[5] There will be one exception to this rule (see below), where a key and ciphertext will be loaded into working memory simultaneously, but this will be done only after ensuring that the ciphertext are "blinded" and contain no sensitive information.

[6] We restrict our attention to single bit output, the case of multi-bit outputs also follows using the same ideas.

an initialization of $Eval$ that suffices for a *single* secure execution on any adversarially chosen input.

Without loss of generality, the circuit C is composed of NAND gates with fan-in 2 and fan-out 1, and *duplication* gates with fan-in 1 and fan-out 2. We assume a lexicographic ordering on the circuit wires, s.t. if wire k is the output wire of gate g then for any input wire i of the same gate, $i < k$. We use $v_i \in \{0,1\}$ to denote the bit value on wire i of the original circuit $C(y,x)$. $Eval$ does not compute or load into memory the explicit v_i values for internal wires (or y-input wires) into memory: any such value loaded into memory might leak and exposes non black-box information about the circuit C! Instead, $Eval$ keeps track of each v_i value in LROTP encrypted form $(key_i, \vec{c}_i)$. I.e., there is a key and a ciphertext (with underlying plaintext v_i) for each circuit wire, and v_i is protected because the key and ciphertext are not be loaded into memory at once.

We emphasize that the adversary does not actually ever see any key or ciphertext in its entirety, nor does he see any underlying plaintext. Rather, the adversary only sees the result of bounded-length leakage functions that operate separately on these keys and ciphertexts.

Initialization for One-Time Evaluation. For each y-input wire i carrying bit $y[j]$ of the y-input, generate an LROTP encryption of $y[j]$: $(key_i, \vec{c}_i)$. For each x-input wire i, generate an LROTP encryption of 0: $(\vec{c}_i, key_i)$. For the output wire $output$, generate an LROTP encryption of 0: $(\vec{\ell}_{output}, \vec{d}_{output})$. For each internal wire i, choose a bit $r_i \in_R \{0,1\}$ uniformly at random. Generate two independent LROTP encryptions of r_i: $(\vec{\ell}_i, \vec{d}_i)$ and $(\vec{\ell}'_i, \vec{d}'_i)$. Finally, for each internal wire i (and for the output wire too), generate an LROTP encryption of 1: $(\vec{o}_i, \vec{e}_i)$

Recall that initialization is performed without any leakage. Looking ahead, the main challenge for multiple execution will be securely generating the keys and ciphertexts for each wire even in the presence of OC leakage. See Section II-C.

$Eval$ **on input** x**.** Once a (non secret) input x is selected for $Eval$, for each wire i carrying bit $x[j]$, "toggle" $\vec{c}_i$ so that the underlying plaintext is $x[j]$. This is done using homomorphic ciphertext-plaintext addition, taking $\vec{c} \leftarrow \vec{c} \oplus (x[j], 0, \ldots, 0)$. Taking these encryptions together with those generated in initialization, we get that for each input wire i of the original circuit C (carrying a bit of y or a bit of x), we now have an LROTP encryption $(key_i, \vec{c}_i)$ of v_i.

$Eval$ proceeds to compute, for each internal wire i of the circuit (and for the output wire $output$), a secure LROTP encryption $(key_i, \vec{c}_i)$ of v_i. This is accomplished using a safe homomorphic evaluation procedure discussed below. The homomorphic evaluation follows the computation of the original circuit C gate by gate in lexicographic order (from the input wires to the output wire). The adversary learns nothing about the v_i values, even under leakage. All the adversary "sees" are the input x and the output $v_{output} = C(y,x)$. The challenge is homomorphic evaluation of the internal NAND gates.

Leakage-Resilient "Safe NAND" Computation. We provide a procedure that, for a NAND gate, takes as input LROTP encryptions of the bits on the gate's input wires, and outputs an LROTP encryption of the bit on the gate's output wire. We prove that even under leakage, this procedure exposes nothing about the private shares of the gate's input wires and output wire (beyond the value of the output wire's public share). This "Safe NAND" procedure uses a secure LROTP encryption of 1 and two secure LROTP encryptions of a random bit (which were generated in the initialization phase above). We also need a similar procedure for aforementioned duplication gates, but we focus here on the more challenging case of NAND.

For a NAND gate with input wires i, j and output wire k, the input to the $SafeNAND$ procedure is ciphertext-key pairs: $(key_i, \vec{c}_i)$, $(key_j, \vec{c}_j)$ (underlying plaintexts v_i, v_j), $(\vec{\ell}_k, \vec{d}_k)$ (random underlying plaintext r_k), and $(\vec{o}_k, \vec{e}_k)$ (underlying plaintext 1). The goal is to compute the bit $a_k = (v_i \text{ NAND } v_j) \oplus r_k$, without leaking anything more about the underlying plaintexts (v_i, v_j, r_k).

Note that, in its own right, the bit a_k exposes nothing about v_i or v_j. This is because the random bit r_k masks $(v_i \text{ NAND } v_j)$. Once we have securely computed the bit a_k, we use the pair $(\vec{\ell}'_k, \vec{d}'_k)$ (with underlying plaintext r_k) to obtain an LROTP encryption $(key_k, \vec{c}_k)$ of $v_k = (v_i \text{ NAND } v_j)$. This is done using homomorphic ciphertext-plaintext addition, by setting $key_k \leftarrow \vec{\ell}'_k$ and $\vec{c}_k \leftarrow (\vec{d}'_k \oplus (a_k, 0, 0, \ldots, 0))$.[7]

We proceed with an overview of $SafeNAND$, see the full version for details. As a starting point, we first choose a single key, and compute from $(key_i, \vec{c}_i)$, $(key_j, \vec{c}_j)$, $(\vec{\ell}_k, \vec{d}_k)$, $(\vec{o}_k, \vec{e}_k)$, new ciphertexts $(\vec{c}^*_i, \vec{c}^*_j, \vec{d}^*_k, \vec{e}^*_k)$ whose underlying plaintexts under this single key remain $(v_i, v_j, r_k, 1)$ (respectively). This uses homomorphic properties of the LROTP cryptosysm keys. Once the ciphertexts are all encrypted under the same key, our goal is to compute the bit $a_k = (v_i \text{ NAND } v_j) \oplus r_k$.

To compute a_k, we start with an idea of Sanders Young and Yung [26] for homomorphically computing the NAND of two ciphertexts with underlying plaintexts v_i, v_j. They

[7] It is natural to ask why we needed two different LROTP encryptions $(\vec{\ell}_k, \vec{d}_k)$ and $(\vec{\ell}'_k, \vec{d}'_k)$ of the same random bit r_k. Why not simply use $(\vec{\ell}_k, \vec{d}_k)$ twice? The reason is that, during the execution of $SafeNAND$, $\vec{\ell}_k$ and $\vec{d}_k$ are used to determine LROTP keys and ciphertexts that are eventually loaded into memory together and decrypted. While we will argue that this exposes nothing about the bit r_k, the leakage might create statistical dependencies between the strings $\vec{\ell}_k$ and $\vec{d}_k$. If we then re-used $\vec{\ell}_k$ and $\vec{d}_k$ to compute the output $(key_k, \vec{c}_k)$ of $SafeNAND$, then they might later be involved in another $SafeNAND$ computation (as inputs). The statistical dependencies might accumulate and break security. Using a fresh pair of ciphertexts $(\vec{\ell}'_k, \vec{d}'_k)$ (encrypting the same bit) that have never been loaded into memory together avoids the accumulation of any statistical dependencies and allows us to prove security.

used homomorphic addition to create a 3-tuple of ciphertexts s.t. the number of ciphertexts with underlying plaintext 0 in this 3-tuple specifies whether $(v_i \text{ NAND } v_j)$ is 0 or 1. The locations of 0's and 1's in the 3-tuple expose information about v_i and v_j beyond their NAND , but [26] permute the 3-tuple of ciphertexts using a random permutation (and also refresh each ciphertext). They showed that the resulting 3-tuple of permuted ciphertexts exposes only $(v_i \text{ NAND } v_j)$ and nothing more. They use this idea to build secure function evaluation protocols for NC^1 circuits.

We translate this idea to our setting. We use the homomorphic addition properties of LROTP to compute a 4-tuple of encryptions (all under the same key):

$$C \leftarrow (\vec{d}_k^*, (\vec{c}_i^* \oplus \vec{d}_k^*), (\vec{c}_j^* \oplus \vec{d}_k^*), (\vec{c}_i^* \oplus \vec{c}_j^* \oplus \vec{d}_k^* \oplus \vec{e}_k^*))$$

the plaintexts underlying the 4 ciphertexts in C are:

$$(r_k, (v_i \oplus r_k), (v_j \oplus r_k), (v_i \oplus v_j \oplus r_k \oplus 1))$$

now if $a_k = 0$, then 3 of these plaintexts will be 1, and one will be 0, whereas if $a_k = 1$, then 3 of the plaintexts will be 0 and one will be 1.

Now, as was the case for [26], the *locations* of 0's and 1's might reveal (via the adversary's leakage) information about (v_i, v_j, r_k) beyond just the value of a_k. Trying to follow [26], we might try to *permute* the ciphertexts before decrypting. Our problem, however, is that *any permutation we use might leak*. What we seek, then, is a method for randomly permuting the ciphertexts even under leakage.

Securely Permuting under Leakage. The leakage-resilient permutation procedure $Permute$ that takes as input key and a 4-tuple C, consisting of 4 ciphertexts. $Permute$ makes 4 copies of key, and then proceeds in ℓ iterations $i \leftarrow 1, \ldots, \ell$. The input to each iteration i is a 4-tuple of keys and a 4-tuple of corresponding ciphertexts. The output from each iteration is a 4-tuple of keys and a 4-tuple of corresponding ciphertexts, whose underlying plaintexts are some permutation $\pi_i \in S_4$ of those in that iteration's input. The output of iteration i is fed as input to iteration $(i+1)$, and so after ℓ iterations the plaintexts underlying the output keys and ciphertexts of iteration ℓ will be a "composed" permutation $\pi = \pi_1 \circ \ldots \circ \pi_\ell$ of the plaintexts underlying the first iteration's input keys and ciphertexts.

The goal is that π_i used in each iteration will look "fairly random" even under leakage. This will imply that the composed permutation π will be statistically close to uniformly random even under leakage. To this end, each iteration i operates as follows:

Sub-Computation 1: Duplicate-Refresh-Permute. Create κ copies of the input key and ciphertext 4-tuples. Refresh each tuple-copy using key-ciphertext refresh as in Section II-A (each refresh uses independent randomness). Finally, permute each tuple-copy using an independent uniformly random permutation $\pi_i^j \in_R S_4$ (π_i^j is used in iteration i on the j-th refreshed tuple-copy).

Sub-Computation 2: Choose. Choose, uniformly and at random, one of the tuple copies as this iteration's output

We first observe that *without leakage* from sub-computation 1, all κ permutations $(\pi_{i,1}, \ldots, \pi_{i,\kappa})$ look independently and uniformly random.[8] Thus, given λ bits of leakage from sub-computation 1, where $\lambda < 0.1\kappa$, most permutations still look "fairly random": by a counting argument, even given the λ bits of leakage, the entropy in many of the permutations $(\pi_{i,1}, \ldots, \pi_{i,\kappa})$ will remain high. In other words, while significant leakage can occur on *some* of the permutations, it cannot occur on *all* of them. After this leakage occurs, in sub-computation 2 we choose one of the tuple-copies $j^* \in \{1, \ldots, \kappa\}$ (and its permutation) uniformly at random and set $\pi_i \leftarrow \pi_i^{j^*}$. By the above, with constant probability we get that π_i has high entropy even given the leakage. Composing the permutations chosen in many iterations, with overwhelming probability in a constant fraction of the iterations the permutation chosen has high entropy. When this is the case, the composed permutation is statistically close to uniform. See the full version for further details on $Permute$ and a formal statement and proof of its security properties.

C. Multiple Secure Evaluations

In this section we modify the $Init$ and $Eval$ procedures described in Section II-B to support any polynomial number of secure evaluations. The main challenge is generating secure key-ciphertext pairs for the various circuit wires.

Ciphertext Generation under Continual Leakage. We seek a procedure for repeatedly generate secure LROTP key-ciphertext pairs with a fixed underlying plaintext bit. The underlying plaintexts will be as before in the construction of Section II-B: for each y-input wire i corresponding to the j-th bit of y, the underlying plaintext should be $y[j]$. For each x-input wire and for the output wire $output$, the underlying plaintext should 0. For each internal wire (and for the output wire), we will generate a key-ciphertext pair with underlying plaintext 1. Finally, we also seek a procedure for repeatedly generating a two LROTP key-ciphertext pairs $(\vec{\ell}_i, \vec{c}_i)$ and $(\vec{\ell}_i', \vec{c}_i')$ whose underlying plaintexts are a uniformly random bit $r_i \in \{0, 1\}$ (the same bit in both pairs).

[8] In slightly more detail, we consider the case where the underlying plaintexts are all 0, and show that **without leakage**, *even given all the refreshed and permuted tuple-copies*, the permutation chosen for each copy looks uniformly random. This is because the refreshing procedure outputs a uniformly random key-ciphertext pair encrypting the same underlying plaintext. We will then show that when the underlying plaintexts are all 0, the *composed* permutation looks uniformly random **even under leakage**. Finally, we will claim that when the underlying plaintexts are not all 0 the composed permutation also looks uniformly random under leakage. This is because, by LROTP security of the underlying plaintext bits, a leakage adversary cannot distinguish whether the underlying plaintexts are all 0 or have some other values.

For security, the underlying plaintexts of the keys and ciphertexts produced should be completely protected even under continual leakage on the repeated generations. In previous works such as [4], [7], [6], similar challenges were (roughly speaking) overcome using secure hardware to generate "fresh" encodings of leakage-resilient plaintexts from scratch in each execution.

We generate key-ciphertext pairs using *ciphertext banks*. We begin by describing this new tool and how it is used for repeated secure generations with a fixed underlying plaintext bit. This is what is needed for input wires and for the output wire. We then describe how to "randomize" the fixed underlying plaintext bit to be uniformly random (which is used to repeatedly generate two key-ciphertext pairs with a uniformly and independently random underlying plaintext).

A ciphertext bank is initialized once using a $BankInit(b)$ procedure, where b is either 0 or 1 (there is no leakage during initialization). It can then be used, via a $BankGen$ procedure, to repeatedly generate key-ciphertext pairs with underlying plaintext bit b, for an unbounded polynomial number of generations. We refer to b as the Ciphertext Bank's underlying plaintext bit. We also provide a $BankGenRand$ procedure for generating pairs of key-ciphertext pairs with a uniformly random underlying plaintext bit. Informally, the ciphertext bank security property is that, even under leakage from the repeated generations, the plaintext underlying each key-ciphertext pair is protected. More formally, there are efficient simulation procedures that have arbitrary control over the plaintexts underlying all key-ciphertext pairs that the bank produces. Leakage from the simulated calls is statistically close to leakage from the "real" ciphertext bank calls. We outline these procedures below, see the full version for further details.

Using ciphertext banks, we modify the initialization and evaluation outlined in Section II-B. In initialization, we initialize a ciphertext bank with a fixed underlying plaintext bit for each input wire, for each internal wire (with underlying plaintext 1), and two banks for the output wire (see Section II-B for all the fixed underlying plaintexts). We also initialize a ciphertexts bank with a random underlying plaintext bit, which will be used for generating pairs of key-ciphertext pairs with a random underlying plaintext for the internal wires ((see Section II-B. Before each evaluation, we use these ciphertext banks to generate all of the key-ciphertext pairs that are needed for each circuit wire. After this first step, evaluation proceeds as outlined in Section II-B.

Ciphertext Bank Implementation. A ciphertext bank consists of an LROTP key, and a collection C of 2κ ciphertexts. We view C as a $\kappa \times 2\kappa$ matrix, whose columns are the ciphertexts. In the $BankInit$ procedure, on input b, key is drawn uniformly at random, and the columns of C are drawn uniformly at random s.t the plaintext underlying each column equals b. This invariant will be maintained throughout the ciphertext bank's operation, and we call b the bank's underlying plaintext bit.

The $BankGen$ procedure outputs key and a linear combination of C's columns. The linear combination is chosen uniformly at random s.t. it has parity 1. This guarantees that it will yield a ciphertext whose underlying plaintext is b. We then inject new entropy into key and into C: we refresh the key using the LROTP key refresh property, and we refresh C by multiplying it with a random $2\kappa \times 2\kappa$ matrix whose columns all have parity 1. These refresh operations are performed under leakage.

The $BankGenRand$ procedure re-draws the bank's underlying plaintext bit by choosing a uniformly random ciphertext $\vec{v} \in \{0,1\}^\kappa$, and adding it to all the columns of C. If the inner product of key and $\vec{v}$ is 0 (happens w.p. 1/2), then the bank's underlying plaintext bit is unchanged. If the inner product is 1 (also w.p. 1/2), then the bank's underlying plaintext bit is flipped.

Int the security proof, we provide a simulation procedure $SimBankGen$ that can arbitrarily control the value of the plaintext bit underlying the key-ciphertext pair it generates. Here we maintain a simulated ciphertext bank, consisting of a key and a matrix, similarly to the real ciphertext bank. These are initialized, without leakage, using a $SimBankInit$ procedure that draws key and the columns of C uniformly at random from $\{0,1\}^\kappa$. Note that here, unlike in the real ciphertext bank, the plaintexts underlying C's columns are uniformly random bits (rather than a single plaintext bit b). The operation of $SimBankGen$ is similar to $BankGen$, except that it uses a *biased linear combination* of C's columns to control the underlying plaintext it produces.

The main technical challenge and contribution here is showing that leakage from the real and simulated calls is statistically close. Note that, even for a single generation, this is non-obvious. As an (important) example, consider the rank of the matrix C: in the real view (say for $b = 0$), C's columns are all orthogonal to key, and the rank is at most $\kappa - 1$. In the simulated view, however, the rank will be κ (w.h.p). If the matrix C was loaded into memory in its entirety, then the real and simulated views would be distinguishable!

Observe, however, that computing an linear combination of C's columns does not require loading C into memory in its entirety. Instead, we can compute the linear combination in a "piecemeal" manner: first, load only $(c \cdot \kappa)$ columns of C into memory (for a small $0 < c < 0.5$). Compute their contribution $\vec{x}_1$ to the linear combination. Then, load $\vec{x}_1$ into memory together with the next $(c \cdot \kappa)$ columns of C, and add $\vec{x}_1$ to these columns' contribution to the linear combination. This gives $\vec{x}_2$, which is the contribution of the first $(2c \cdot \kappa)$ columns to the linear combination. We can continue this process for $(2/c)$ sub-steps, eventually computing the linear combination of C's columns without ever loading C into memory in its entirety. All we need to

load into memory at one time is a collection of $((c \cdot \kappa) + 1)$ linear combinations of columns of C. We call each such collection a "sketch" (or a "piece") of C. We prove that *sketches of random matrices are leakage resilient*, and in particular leakage from sketches of C is statistically close in the real and simulated distributions (i.e. when C is of rank $\kappa - 1$ or uniformly random). Thus, the above procedure for computing a linear combination of C's rows is leakage resilient. Similarly, we show how to implement $BankGen$ and $SimBankGen$ using sub-computations, where each sub-computation only loads a single "sketch" of C into memory. We use this to show security of the ciphertext bank for any unbounded (polynomial) number of generations. We view these proofs as our most important technical contribution.

D. Leaky CPU: What are The Universal Instructions?

Recall that in the leaky CPU model (which is equivalent to the OC-leakage model), a leaky CPU executes atomic operations from a fixed set of universal instructions. Leakage operates separately on each atomic operation. The atomic operations are equivalent to the sub-computations performed by our compiler.

We elaborate here on the set of universal CPU instructions required. These are fairly simple and straightforward. They include instructions for generating a random matrix/vector of 0/1 bits, for vector-vector addition and multiplication (i.e. inner product), for matrix-matrix addition, and for matrix-vector and matrix-matrix multiplication. Beyond these, the only additional functionality used is permuting a sequence of vectors. This, in a nutshell, is a complete (high-level) list of the required instructions. This set of instructions suffices for LROTP operations such as decryption, key and ciphertext refresh, homomorphic operations (implemented using vector operations), as well as for the ciphertext banks outlined in Section II-C (implemented using matrix-matrix and matrix-vector multiplication). The $SafeNAND$ and $Permute$ procedures outlined above use these procedures, as well as a duplicate-refresh-and-permute operation. This operation can be implemented as a single atomic instruction (as described above), or as a sequence of instructions for duplication, refreshing and permuting. Both implementations are leakage resilient.

For security parameter κ, the instructions used all have input and output size $O(\kappa^2)$, and can be implemented by circuits of size $O(\kappa^\omega)$, where ω is the exponent of the circuit-size required for matrix multiplication.

III. Further Comparison to Related Work

We provide a more detailed comparison prior work *leakage-resilience compilers for general programs* in various continual leakage attack models. Comparing to the work of Goldwasser and Rothblum [6] and of Juma and Vhalis [7] in the OC-L model, the main *qualitative* difference is that both of those prior works use computational intractability assumptions (DDH in [6] and the existence of fully homomorphic encryption scheme (FHE) in [7]) as well as secure hardware. Our result, on the other hand, is unconditional and uses no secure hardware components.

In terms of *quantitative* bounds, for security parameter κ, [7] transform a circuit of size C into a new circuit C' of size $\mathrm{poly}(\kappa) \cdot |C|$. The new circuit C' is composed of $O(1)$ sub-circuits (one of the sub-circuits is of size $\mathrm{poly}(\kappa) \cdot |C|$). Assuming a fully-homomorphic encryption scheme that (for the security parameter κ) is secure against adversaries that run in time T, their construction can withstand $O(\log T)$ bits of leakage on each sub-circuit. For example, if the FHE is secure against $\mathrm{poly}(L)$-time adversaries, then the leakage bound is $O(\log L)$. In our new construction, for any leakage parameter L, there are $O(|C|)$ sub-computations (i.e. more sub-computations), each of size $\tilde{O}(L^\omega)$, where ω is the exponent is the exponent in the best algorithm known for matrix multiplication (i.e. smaller). The new construction can withstand L bits of leakage from each sub-computation (i.e. the amount of leakage we can tolerate, relative to the sub-computation size, is larger). The quantitative parameters of [6] are similar to ours (up to polynomial factors).

The work of Ishai, Sahai and Wagner [3] in the ISW-L leakage model, may be viewed as converting any circuit C into $O(|C|)$ sub-circuits each of size $O(L^2)$, and allow the leakage of L wire values from each sub-circuit. Our transformation converts C into $O(|C|)$ sub-circuits, each of size $\tilde{O}(L^\omega)$, and allow the leakage of L bits from each sub-circuit where these bits can be the output of arbitrary computations on the wire values (rather than the wire values themselves as in [3]).

The work of Faust *et al.* [4] in the CB-L model, under the additional assumption that leak free hardware components exist, shows how to convert any circuit C into a new circuit C' of size $O(|C| \cdot L^2)$, which is resilient to leakage of the result of any $\mathcal{AC}^0$ function f of output length L computed on the entire set of wire values. Qualitatively, the main differences are (i) that construction used secure hardware, whereas we do not use secure hardware, and (ii) in terms of the class of leakage tolerated, they can handle bounded-length $\mathcal{AC}^0$ leakage *on the entire computation* of each execution. We, on the other hand, can handle length bounded OC-L leakage *of arbitrary complexity* that operates separately (if adaptively) on each sub-computation. A more recent result of [11] removes the need for hardware component and shows how to convert C into C' of size $O(|C| \cdot poly(L)$ which is resilient against $\mathcal{AC}^0$ leakage functions of length L, under the computational assumption that computing inner-product cannot be done in $\mathcal{AC}^0$, even if polynomial time pre-processing (of the inputs to the inner product) is allowed. [11] uses ciphertext banks, a tool introduced in this work.

Comparing to the work of Ajtai [10] in the RAM-L model, he divides the computation of program P into sub-computations, each utilizing $O(L)$ memory accesses,

and shows resilience to an adversary who, for each sub-computation, sees the contents of L memory accesses out of the $O(L)$ accesses in that sub-computation. I.e a constant fraction of all memory accesses in each sub-computation are exposed, whereas all the other memory accesses are perfectly hidden. Translating our result to the RAM model, we divide the computation into sub-computations of $\tilde{O}(L^{\omega})$ accesses, and show resilience against an adversary that can receive L *arbitrary bits of information* computed on the entire set of memory accesses and randomness. In particular, there are no protected or hidden accesses.

Continual Leakage on a Stored Secret. A recent independent work of Dodis, Lewko, Waters, and Wichs [19], addresses the problem of how to store a value S secretly on devices that continually leak information about their internal state to an external attacker. They design a leakage resilient distributed storage method: essentially storing an encryption of S denoted $E_{sk}(S)$ on one device and storing sk on another device, for a semantically secure encryption method E which: (i) is leakage resilient under the linear assumption in prime order groups, and (ii) is "refreshable" in that the secret key sk and $E_{sk}(S)$ can be updated periodically. Their attack model is that an adversary can only leak on each device separately, and that the leakage will not "keep up" with the update of sk and $E_{sk}(S)$. One may view the assumption of leaking separately on each device as essentially a weak version of the only computation leak axiom, where locality of leakage is assumed per "device" rather than per "computation step". We point out that storing a secret on continually leaky devices is a special case of the general results described above [3], [4], [6], [7] as they all must implicitly maintain the secret "state" of the input algorithm (or circuit) throughout its continual execution. The beauty of [19] is that no interaction is needed between the devices, and they can update themselves asynchronously.

REFERENCES

[1] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang, "On the (im)possibility of obfuscating programs," in *CRYPTO*, 2001, pp. 1–18.

[2] R. Impagliazzo, "Personal communication," 2010.

[3] Y. Ishai, A. Sahai, and D. Wagner, "Private circuits: Securing hardware against probing attacks," in *CRYPTO*, 2003, pp. 463–481.

[4] S. Faust, T. Rabin, L. Reyzin, E. Tromer, and V. Vaikuntanathan, "Protecting circuits from leakage: the computationally-bounded and noisy cases," in *EUROCRYPT*, 2010, pp. 135–156.

[5] S. Micali and L. Reyzin, "Physically observable cryptography (extended abstract)," in *TCC*, 2004, pp. 278–296.

[6] S. Goldwasser and G. N. Rothblum, "Securing computation against continuous leakage," in *CRYPTO*, 2010, pp. 59–79.

[7] A. Juma and Y. Vahlis, "Protecting cryptographic keys against continual leakage," in *CRYPTO*, 2010, pp. 41–58.

[8] S. Dziembowski and S. Faust, "Leakage-resilient circuits without computational assumptions," in *TCC*, 2012, pp. 230–247.

[9] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," *J. ACM*, vol. 43, no. 3, pp. 431–473, 1996.

[10] M. Ajtai, "Secure computation with information leaking to an adversary," in *STOC*, 2011.

[11] G. N. Rothblum, "How to compute under $\mathcal{AC}^0$ leakage without secure hardware," in *CRYPTO*, 2012.

[12] S. Dziembowski and K. Pietrzak, "Leakage-resilient cryptography," in *FOCS*, 2008, pp. 293–302.

[13] K. Pietrzak, "A leakage-resilient mode of operation," in *EUROCRYPT*, 2009, pp. 462–482.

[14] S. Faust, E. Kiltz, K. Pietrzak, and G. N. Rothblum, "Leakage-resilient signatures," in *TCC*, 2010, pp. 343–360.

[15] Z. Brakerski, Y. T. Kalai, J. Katz, and V. Vaikuntanathan, "Overcoming the hole in the bucket: Public-key cryptography resilient to continual memory leakage," in *FOCS*, 2010, pp. 501–510.

[16] Y. Dodis, K. Haralambiev, A. López-Alt, and D. Wichs, "Cryptography against continuous memory attacks," in *FOCS*, 2010, pp. 511–520.

[17] A. Lewko, Y. Rouselakis, and B. Waters, "Achieving leakage resilience through dual system encryption," in *TCC*, 2011.

[18] A. Lewko, M. Lewko, and B. Waters, "How to leak on key updates," in *STOC*, 2011.

[19] Y. Dodis, A. B. Lewko, B. Waters, and D. Wichs, "Storing secrets on continually leaky devices," in *FOCS*, 2011, pp. 688–697.

[20] N. Bitansky, R. Canetti, S. Goldwasser, S. Halevi, Y. T. Kalai, and G. N. Rothblum, "Program obfuscation with leaky hardware," in *ASIACRYPT*, 2011, pp. 722–739.

[21] R. Canetti, U. Feige, O. Goldreich, and M. Naor, "Adaptively secure multi-party computation," in *STOC*, 1996, pp. 639–648.

[22] E. Boyle, S. Goldwasser, A. Jain, and Y. T. Kalai, "Multiparty computation secure against continual leakage," in *STOC*, 2012.

[23] S. Halevi and H. Lin, "After-the-fact leakage in public-key encryption," in *TCC*, 2011, pp. 107–124.

[24] D. Boneh, S. Halevi, M. Hamburg, and R. Ostrovsky, "Circular-secure encryption from decision diffie-hellman," in *CRYPTO*, 2008, pp. 108–125.

[25] M. Naor and G. Segev, "Public-key cryptosystems resilient to key leakage," in *CRYPTO*, 2009, pp. 18–35.

[26] T. Sander, A. Young, and M. Yung, "Non-interactive cryptocomputing for $\mathcal{NC}^1$," in *FOCS*, 1999, pp. 554–567.