How to Compute under $\mathcal{AC}^0$ Leakage without Secure Hardware

Guy N. Rothblum*

Microsoft Research, Silicon Valley Campus

Abstract. We study the problem of computing securely in the presence of leakage on the computation's internals. Our main result is a general compiler that compiles any algorithm P, viewed as a boolean circuit, into a functionally equivalent algorithm P'. The compiled P' can then be run repeatedly on adversarially chosen inputs in the presence of leakage on its internals: In each execution of P', an $\mathcal{AC}^0$ adversary can (adaptively) choose any leakage function that can be computed in $\mathcal{AC}^0$ and has bounded output length, apply it to the values on P''s internal wires in that execution, and view its output. We show that no such leakage adversary can learn more than P's input-output behavior. In particular, the internals of P are protected.

Security does not rely on any secure hardware, and is proved under a computational intractability assumption regarding the hardness of computing inner products for $\mathcal{AC}^0$ circuits with pre-processing. This new assumption has connections to long-standing open problems in complexity theory.

1 Introduction

Modern cryptographic algorithms are often modeled as "black boxes". It is commonly assumed that their keys, internal computations, and randomness are opaque to external adversaries. In practice, however, these algorithms might be run in adversarial settings where keys are compromised and computations are not be fully opaque, e.g. because of side channel attacks.

Side channel attacks exploit the physical implementation of cryptographic algorithms. The physical implementation might enable observations and measurements on the cryptographic algorithm's internals. Attacks such as these can and have broken systems with a mathematical security proof, without violating any of the underlying mathematical principles. (see [KJJ99,RCL] for just two examples). A growing body of recent research on *cryptography resilient to leakage or side-channel attacks* aims to build robust mathematical models of realistic side channel attacks, and to develop methods grounded in modern cryptography to provably resist such attacks.

One line of research aims to construct specific primitives (e.g. encryption schemes) resilient to side-channel attacks, e.g. [AGV09,BKKV10]. A different

* Some of this research was done while the author was at the Center for Computational Intractability at Princeton University, supported by a Computing Innovation Fellowship and by NSF grant CCF-0832797.

line, which we pursue in this work, aims to construct leakage-resilience compilers for transforming general algorithms (represented as stateful circuits) into functionally equivalent stateful circuits that are resilient to side-channel attacks on any polynomial number of executions (the polynomial is not fixed in advance). Typically, these results consider a family $\mathcal{L}$ of leakage attacks, and show (via a simulation argument) that even an adversary who can adaptively choose inputs to each execution, and launch a leakage attack from the family $\mathcal{L}$ on each execution, learns no more about the underlying functionality than it would from black-box (i.e. input-output) access. It is usually assumed that the initial circuit compilation is done (once only) without any leakage. Afterwards, every execution of this stateful circuit—including any and all state updates—is subject to leakage attacks from the family $\mathcal{L}$.

A fascinating question for research on leakage-resilience compilers, from both a foundational and a practical perspective, is: *"for which leakage families $\mathcal{L}$ do there exist secure compilers?"* There are, unfortunately, inherent limitations on the leakage functions that can be handled. For example, they must be of bounded length—otherwise the entire circuit internals can leak, and we enter the more challenging domain of code obfuscation. In particular, the impossibility results of Barak *et al.* [BGI$^+$01] imply that there does not exist a leakage-resilience compiler that can protect against leakage of unbounded length. In fact, the impossibility result of [BGI$^+$01] can be used to show that there is no leakage-resilience compiler that protects against even *a single bit* of polynomial-time leakage. Given this impossibility, work on leakage-resilience compilers has considered various additional restrictions on the leakage functions (on top of bounded output length):

Wire-Probe/Bit Leakage. Ishai Sahai and Wagner [ISW03] view algorithms as boolean circuits. They considered leakage functions that expose the values on a bounded number of wires in each circuit evaluation. For this class of leakage functions they show (unconditionally) a leakage-resilience compiler for general circuits. Ajtai [Ajt11] considered the RAM model. He divided each RAM computation into sub-computations, and considered leakage functions that exposed a constant fraction of the memory words involved in each sub-computation. He obtained a leakage-resilience compiler for general RAM computations. The leakage model is qualitatively similar to that of [ISW03], in the sense that the (length bounded) leakage operates separately on each bit of the computation—either exposing it in its entirety or revealing nothing at all. We view the main (and important) improvement in [Ajt11] versus [ISW03] as the quantitative improvement to fraction of leaked bits that can be tolerated in each execution of the transformed computation.

Computationally Bounded Leakage. Faust, Rabin, Reyzin, Tromer and Vaikuntanathan [FRR$^+$10] considered leakage functions with two restrictions: (i) the functions are *computationally bounded*, e.g. capable of computing only $\mathcal{AC}^0$ functions of the values on the circuit's wires,[1] and (ii) the computation can use *perfectly secure hardware components*, whose internals never leak. We view this second assumption as a strong restriction on the leakage functions:

[1] Their full result is actually more general, and can handle $ACC^0[p]$ or noisy leakage.

they cannot even compute a single bit of information about the internals of the secure hardware components, which are used multiple times throughout the execution. For this family of leakage functions, Faust *et al.* showed (unconditionally) a general leakage-resilience compiler for transforming any circuit.

$\mathcal{AC}^0$ leakage is a qualitatively richer class than wire-probe leakage. In particular, for a fixed leakage bound λ, $\mathcal{AC}^0$ leakage can output the values of λ wires, but potentially also much more (e.g. the AND of all circuit wire values). The results of [FRR$^+$10], however, are qualitatively incomparable to [ISW03] because of the secure hardware restriction.

Only-Computation (OC) Leakage. Goldwasser and Rothblum [GR10] and Juma and Vhalis [JV10] considered leakage under the restriction that *"only computation leaks information"*, as pioneered by Micali and Reyzin [MR04]. Here, each execution of the algorithm is divided into ordered sub-computations, and the leakage function operates separately (if adaptively) on each sub-computation. Those works also assumed the existence of perfect (simple) secure hardware components, and showed general leakage-resilience compilers under different cryptographic assumptions. More recently, Goldwasser and Rothblum [GR12] showed how to remove both the use of secure hardware and the computational assumption, obtaining an unconditional compiler for protecting computations from the "only-computation leaks information" (OC) family of leakage functions.

Comparing this to the other models described above, we note that (for a fixed leakage bound) OC leakage is qualitatively richer than wire probe leakage, but incomparable to $\mathcal{AC}^0$ leakage.

Remark 1 (A Foundational Perspective.). While the study of leakage-resilient cryptography is motivated by real-world attacks and security considerations, it explores a foundational question: The issue at its heart is the difference between giving an adversary *black-box access to a program* and *access to the program's code or internals.* This question is central to the foundations and the practice of cryptography. [GR12] note that the connection between obfuscation and leakage-resilience hinted at above is no accident: Obfuscation considers the task of compiling a program to make it completely unintelligible, or "impervious to all leakage" (i.e. even to an adversary with complete access to the program's code). Unfortunately, full-blown obfuscation is provably impossible in many settings [BGI$^+$01,GK05], and is considered intractable in practice. Perhaps as a result of this impossibility, much of cryptography only considers adversaries that have (at best) "black box" access to the programs under attack. Leakage-resilience compilation can be viewed as exploring the middle ground between full access to the code and black-box access: Giving the adversary *limited access* to the program's internals and its code. Our primary motivation in this work is understanding which kinds of restricted access to code permit secure compilation (i.e. leakage resilience). On this note, an interpretation of this paper's main result for the setting of obfuscation is provided below, following Theorem 1.

From a real-world security perspective, we note that we do not view $\mathcal{AC}^0$ leakage as a realistic model for *all* side-channel attacks. In fact, some common real-world side channel attacks are not covered by $\mathcal{AC}^0$ leakage (or any of the leakage families considered in the leakage-resilience literature). See e.g. the work of Renauld *et al.* [RSVC+11]). We view this as motivation for further study of leakage-resilience compilation against more and richer classes of attacks.

$\lambda(\cdot)$-IPPP Assumption (informal). The Inner-Product with Pre-Processing (IPPP) Problem considers predicting the inner product of two uniformly random vectors $x, y \in \{0,1\}^\kappa$ using an $\mathcal{AC}^0$ circuit and an arbitrary polynomial-time pre-processing step that is *run separately* on x and on y (with polynomial output length). Without pre-processing, it is known that predicting the inner product is hard for $\mathcal{AC}^0$ [Raz87,Smo87]. With *joint* pre-processing on x and y, one can simply compute the inner product, and so the problem becomes easy. With (polynomial time) pre-processing that is run separately on x and on y, there is no known $\mathcal{AC}^0$ predictor with non-negligible advantage (we emphasize that the pre-processing step's output length can be polynomial). This question has been explicitly considered in the literature for some time (e.g. [BFS86]). We introduce the 1-IPPP Assumption, which says that even given polynomial-time pre-processing (separately on x and on y), no $\mathcal{AC}^0$ circuit ensemble can predict the inner product with non-negligible advantage.

More generally, we consider also the problem of *compressing* the instance size of inner product from κ to $\lambda(\kappa) < \kappa$ bits using an $\mathcal{AC}^0$ circuit and arbitrary polynomial-time pre-processing on x and on y separately. By "compressing", we mean that the instance size is reduced while still maintaining noticeable statistical difference between the distribution of YES and NO instances (inner product 1 and 0 respectively), see [HN10,DI06]. Without pre-processing, Dubrov and Ishai [DI06] showed that it is hard for $\mathcal{AC}^0$ to compress parity instances to sub-linear length $\kappa^{1-\delta}$ (in fact this was later used in the result of [FRR+10]). We introduce the $\lambda(\kappa)$-IPPP Assumption, which says that even given polynomial-time pre-processing (on x and on y, each separately), no $\mathcal{AC}^0$ circuit ensemble can compress the instance size of inner product to length $\lambda(\kappa)$.

See Section 2.2 and the full version for formal definitions and a further study of IPPP. We note here that (even for $\lambda(\kappa) = \kappa^{1-\delta}$), the $\lambda(\kappa)$-IPPP Assumption might be viewed as relatively mild compared to many standard cryptographic assumptions. In particular, it may hold even if $P = NP$.

Main result. Our main result is a leakage resilience compiler for $\mathcal{AC}^0$ leakage. Security relies on the $\lambda(\cdot)$-IPPP assumption. For security parameter κ, the transformed circuit is resilient to $\lambda(\kappa)$ bits of $\mathcal{AC}^0$ leakage from each execution.

Theorem 1. *For any function $\lambda(\cdot) : \mathbb{N} \to \mathbb{N}$, under the $\lambda(\cdot)$-IPPP assumption there exists a leakage resilience compiler for $\mathcal{AC}^0$ leakage. For security parameter $\kappa \in \mathbb{N}$, and for a* $\mathrm{poly}(\kappa)$*-size input circuit C to be transformed, the adversary's runtime can be polynomial in κ, and the leakage from each execution can be any (adaptively chosen) $\mathcal{AC}^0$ function of length $\lambda(\kappa)$. The size of the transformed circuit is $O(|C| \cdot \kappa^3)$.*

See Definition 2 for the formal definition of a secure compiler for $\mathcal{AC}^0$ leakage.

We emphasize that the leakage bound in this result is equal to the "compression" factor in the IPPP assumption. In particular, assuming the $\lambda(\kappa)$-IPPP Assumption for all sub-linear $\lambda(\kappa) = \kappa^{1-\delta}$ (we find this to be a plausible assumption), we get resilience to any sub-linear leakage amount of leakage per execution (similar to the leakage bound in [FRR$^+$10]). We remark that even a compiler that handles only a single bit of leakage from each execution is already non-trivial—in particular, since the number of executions is an unbounded polynomial, the total combined leakage from the repeated executions can be much larger than the size of the transformed circuit. We also recall that for polynomial-time leakage (i.e. leakage not restricted to $\mathcal{AC}^0$), it is impossible to build a leakage-resilience compiler that handles even a single bit of leakage (see above).

$\mathcal{AC}^0$-Leakage Resilience and Obfuscation. Theorem 1 can also be interpreted as providing an obfuscator that is secure against $\mathcal{AC}^0$-adversaries. Obfuscation is the task of "garbling" programs to make them unintelligible. Barak *et al.* [BGI$^+$01] define an obfuscator as a compiler that takes an input program P and outputs a secure obfuscation P': a different program with the same functionality as P. The security requirement is that access to (the code of) P' "leaks no more" than black-box access to P. More formally, they require that for any efficient adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$, s.t. any *predicate* that $\mathcal{A}$ can compute from the obfuscated P', can also be computed by $\mathcal{S}$ from black-box access to P. [BGI$^+$01] show that this strong notion of obfuscation is impossible to achieve in general.

We note that one can view the predicate computed by the obfuscation adversary $\mathcal{A}$ as *a single bit of leakage* on the internals of P'. Taking this view, Theorem 1 shows that (under the 1-IPPP Assumption) *obfuscation against bounded $\mathcal{AC}^0$ adversaries is possible.* In fact, obfuscation is possible even when the $\mathcal{AC}^0$ adversary can run the program on inputs of its choice, and observe these executions in their entirety. We note that previous works, such as [FRR$^+$10], that rely on secure hardware do not provide this strong obfuscation guarantee: There is no analogue to secure hardware in the standard obfuscation setting (their work can be interpreted as providing obfuscation against an $\mathcal{AC}^0$ adversary who can only observe an a-priori bounded number of executions).

Theorem 1 side-steps the impossibility result of [BGI$^+$01], because there the adversary needs to run computations that are as complex as the obfuscated circuit. In our construction and setting, on the other hand, the compiled circuits run computations (such as parities) that provably cannot be done in $\mathcal{AC}^0$, and the adversary is bounded to $\mathcal{AC}^0$ computations. We find the general question of obfuscation against bounded adversaries to be a fascinating one, and note that it is closely related to leakage-resilience compilation.

Comparison to prior work. We now compare the result of Theorem 1 to prior works on leakage-resilience compilers (see also the discussion above on these prior works).

Wire Probe Leakage. Comparing to the work of [ISW03] on wire probe leakage and to the more recent work of Ajtai [Ajt11], the main novelty of our result is in handling the richer class of $\mathcal{AC}^0$ leakage. On the other hand, those results did not rely on unproven assumptions and the quantitative leakage bounds (as a fraction of the transformed computation's size) were better.

$\mathcal{AC}^0$ Leakage. The most closely related work is that of Faust *et al.* [FRR$^+$10], and their construction is a starting point for ours (see below). The main added benefit of our work is in removing the secure hardware assumption (again, this can be viewed as handling a larger class of leakage functions). The main qualitative disadvantage is in the introduction of the unproved IPPP assumption. Quantitatively, the amount of leakage we can handle depends on the function $\lambda(\cdot)$ for which IPPP is hard. If we assume hardness for any sub-linear $\lambda(\cdot)$ function, we get similar leakage bounds to [FRR^{+}10]. Finally, the circuit blowup of the [FRR^{+}10] compiler is $O(\kappa^2)$, whereas ours is $O(\kappa^3)$.

OC Leakage. Our end result is qualitatively incomparable to Goldwasser and Rothblum [GR12], because only-computation leakage and $\mathcal{AC}^0$ leakage are incomparable. We do note, however, that their result is unconditional and the circuit blowup is smaller (κ^ω, where ω is the exponent for matrix multiplication, versus κ^3 in our work). Both our work and theirs tackle the challenge of leakage-resilience compilation for a rich class of leakage functions without using secure hardware. In fact, we use the "ciphertext bank" machinery introduced in [GR12] for handling this challenge.

Our high-level approach is to build on the construction of [FRR^{+}10] and remove the secure hardware using the techniques of [GR10]. Unfortunately, this is far from straightforward. In a nutshell, the main technical challenge is constructing an $\mathcal{AC}^0$ and length-preserving security reduction from the problem of compressing inner product instances (or rather from the IPPP problem), to distinguishing real and simulated leakage on repeated executions of the transformed circuit. The problem is that the [GR10] machinery relies (extensively) on computations that cannot be performed in $\mathcal{AC}^0$ (e.g. matrix multiplication). We elaborate in Section 1.1.

1.1 Overview of the Construction and Security Reduction

At a high level, one central difficulty in leakage-resilience compilation without secure hardware is that it requires simulating a *complete view of multiple executions in their entirety.* The simulated view needs to be indistinguishable from the real execution under a wide family of leakage functions. Intuitively, secure hardware makes the simulation task considerably easier because some regions of the computation are opaque to the leakage, and their internals need not be simulated. Work on specific leakage-resilient cryptographic primitives (e.g. [AGV09,DP08,BKKV10]) avoids this difficulty because the security definitions do not require simulation. We follow [GR12] in tackling on this simulation challenge.

The simulator has no knowledge of the computation's internals (beyond its input and output). Moreover, the *entire computation* in the simulated view should be consistent with the initial state and the choice of random coins; otherwise, an $\mathcal{AC}^0$ leakage function that checks consistency of the internal computations will distinguish the real and simulated views. This means that *the simulator has to generate a self-consistent view of the computation, and cannot make any "illegal" computational steps.* Presumably, however, the simulator is still acting very differently from the real execution. Note that this is a crucial difference from the setting where secure hardware is used: the simulated outputs of the secure hardware can essentially be an "illegal" output that would never be generated in a real execution (but the leakage functions cannot tell the difference).[2] Indeed, this is what the [FRR$^+$10] simulator does (see more below).

In our setting, without secure hardware, the simulator cannot make any "illegal" steps. Its only freedom to diverge from a "real" execution is in generating the initial state (where there is no leakage), and in choosing the random coins. Our simulator generates (only once) a "trapdoor" initial state, and this gives it extensive freedom in shaping the simulated view, even under repeated leakage from multiple executions and state updates.

Against this backdrop, we recall the approach of [FRR$^+$10]. They proved security by showing a reduction from compressing an instance of the inner product problem (or rather the problem of computing a vector's parity) to distinguishing the real and simulated views (or rather various hybrids of these views). They showed that the security reduction can be computed using length-bounded $\mathcal{AC}^0$ access to the inner product instance, and so the real and simulated views are statistically close. Our high-level approach is to build on the construction of [FRR$^+$10], and remove the secure hardware using the techniques of [GR10]. As hinted above, this is far from straightforward, mainly because running the [GR10] machinery requires computations, such as matrix multiplication, that cannot be implemented in $\mathcal{AC}^0$. This creates (multiple) difficulties in implementing a reduction from compressing inner product instances to distinguishing the real and simulated views (or hybrids thereof) that only uses length-bounded $\mathcal{AC}^0$ access to the inner product instance. We relax the requirement from the security reduction: we reduce from the IPPP problem, rather than the full-blown inner product problem without pre-processing. We use the additional power of pre-processing to implement a reduction from the IPPP problem to distinguishing (hybrids of) the real and simulated views. This is our main technical contribution.

We proceed with an overview of our construction and security proof. In what follows we restrict our attention to transforming *stateless* computations to be leakage resilient, but the results all hold also for stateful circuits (as considered say in [FRR$^+$10]). We note that the transformed computations will be stateful circuits (even if the original computation is stateless), and their state is updated (under leakage) between executions.

Overview of [FRR$^+$10] In the Faust *et al.* construction, every wire i in the original circuit, say carrying value a_i (on a certain input), was replaced by a *bundle* of κ wires carrying a uniformly random vector of bits whose parity/XOR is a_i. Intuitively, an adversary with bounded-length $\mathcal{AC}^0$ leakage access to all of an execution's wire-bundles cannot distinguish the true value on any wire (i.e. the parity of any wire-bundle), and so the adversary learns nothing about the internals of the original computation. The main challenge is for the transformed circuit to emulate the computation of each gate in the original circuit, while maintaining the invariant that the bundle corresponding to each wire is a uniformly random vector with the correct parity, and without leaking anything about the parity. For example, to emulate an AND gate, the transformed circuit

[2] This difficulty is avoided in [ISW03] as their leakage functions are too weak to check consistency of the computation.

needs to take two wire bundles and output a wire bundle carrying a uniformly random vector whose parity is the AND of the parities of the input wire bundles.

The gate computations of the original circuits are emulated using "gate gadgets", one for every gate in the original circuit. In their elegant security proof, [FRR$^+$10] separate the wires of the transformed circuit into the "external wires" described above, where each *wire* in the original circuit corresponds to the κ *"external wires"* in the transformed circuit. In each execution (on a certain input), these κ wires carry a bundle whose parity equals the wire's value in the original circuit (on that input). Each *gate* in the original circuit is replaced by a *"gate gadget"* in the transformed circuit. We call the wires in these gate gadgets "internal wires". In their security proof, [FRR$^+$10] show that: (i) the *external wire distributions* (the distributions of values on the external wires) in the real and simulated executions are indistinguishable using bounded length $\mathcal{AC}^0$ leakage. As sketched above, this follows from the hardness of compressing parity instances in $\mathcal{AC}^0$. Then (ii) they show that the values on the internal wires of each gate gadget can be simulated *in* $\mathcal{AC}^0$ given only the gate gadget's input and output external wires. In fact, the simulation for gate gadgets is not perfect, but rather indistinguishable under bounded length $\mathcal{AC}^0$ leakage. The $\mathcal{AC}^0$ simulators for gate gadgets are called *reconstruction procedures*, and their existence guarantees that indistinguishability of the external wire distributions in the real and simulated executions implies indistinguishability of the complete view (including the internal wires of the gate gadgets).

Given this framework, the main challenge is implementing the gate gadgets and their $\mathcal{AC}^0$ reconstruction procedures. This is where the secure hardware devices come into play. The secure hardware outputs a uniformly random bundle of κ values with parity 0. The simulator can simulate the secure hardware's output to be a vector with any desired parity, and the leakage cannot tell the difference. As an example of how such a device is used, consider the (relatively simple) case of addition: the gadget gets two input wire bundles, $\boldsymbol{d}_i$ and $\boldsymbol{d}_j$ and computes the pairwise XORs of their bits (call this bundle $\boldsymbol{q}$). Rather than simply output this $\boldsymbol{q}$ (which already has the correct XOR), the gadget calls the secure hardware to compute a bundle $\boldsymbol{o}$ whose XOR is 0, and finally outputs the pairwise XOR of $\boldsymbol{q}$ and this "masking" bundle $\boldsymbol{o}$. This is not the only way in which the secure hardware is used (e.g. it is called more extensively for multiplication gates), but it is instructive. Intuitively, masking the gadget's output with the output $\boldsymbol{o}$ of the secure hardware helps secure simulation: the gate gadget's reconstruction procedure has some freedom in choosing a bundle (with arbitrary XOR) as the output of the secure hardware. Another important point here is that using the secure hardware "erases" any accumulated leakage on the input bundles: the gadget's output bundle is statistically independent (given its XOR) from the input bundles. In particular, for any given values for the gate gadget's input and output bundles, we can (in $\mathcal{AC}^0$) compute an output of the secure hardware (i.e. a "secure hardware" bundle) for which the gate gadget, on the given input bundles, outputs the given output bundle.

Construction. The secure hardware in [FRR$^+$10] generates a random bundle whose parity is 0, but can be simulated as having generated bundles whose parity is 0 or 1 (as needed by the simulator). An initial idea is simply to pre-compute all the needed 0-bundles as part of the initial state (i.e. without leakage). The simulator can then choose whatever bundles it wants (0 or 1) to put in the initial state. The main drawback to this approach, of course, is that we can only precompute a finite and bounded number of these bundles, and so this construction cannot support repeated executions (alternatively, the initial state grows linearly with the number of executions).

This recalls the situation in [GR12]. They suggested using a "ciphertext bank". Translating that idea to our setting, our construction can use a small number of pre-computed 0-bundles, a *"bundle bank"*, to generate an essentially unbounded (polynomial) number of new 0-bundles. The simulator can generate an (illegally formed) initial bundle bank, and then control each subsequent generation arbitrarily to create a 0-bundle or a 1-bundle. All of this is done in a way that is indistinguishable, even under repeated $\mathcal{AC}^0$ leakage, from the real execution.

We implement the bundle banks as follows. Recall that each bundle encodes a bit $b \in \{0, 1\}$ as a vector in $\{0, 1\}^\kappa$ with parity b. The initial bundle bank includes 2κ such bundles, whose parities (in the real execution) are all 0. Within each execution of the circuit we generate 0-bundles as needed by taking random linear combinations of the bundle bank (a random linear combination of vectors whose parities are all 0 also has parity 0). Between executions we "regenerate" or update the entire bundle bank by taking 2κ new random linear combinations, and then we erase the old bundle bank. In the simulation, the initial bundle bank is generated so that each bundle is a uniformly random vector encoding a uniformly random bit. Now when we generate a new bundle, some linear combinations of the bundles in the bank give a new bundle encoding 0, and some give a new bundle encoding 1. The simulator chooses a uniformly random linear combination yielding an encoding of whatever value it wants. Between executions, as in the real view, the bundle bank is refreshed by taking 2κ new random linear combination, giving a new bank of uniformly random bit encodings (independent of the old bank). The full construction is in Section 3.

We note that our implementation of the bundle bank is considerably simpler than the ciphertext banks of [GR12]. In particular, there is no need for their "piecemeal matrix multiplication" procedure, and we compute matrix multiplication using the straightforward naive procedure (in time κ^3). We also remark that proving the security properties of the bundle bank in our setting requires completely different arguments (due to the completely different nature of the leakage attack).

Security. The intuition for indistinguishability of the real and simulated views is that an adversary cannot distinguish (under bounded length $\mathcal{AC}^0$ leakage) whether the bundles in the bank encode 0's (real execution) or are uniformly random (simulated execution). Nor can the adversary distinguish whether the linear combinations are uniformly random (real execution) or chosen from a $(\kappa - 1)$-dimensional subspace so as to fix the value of the resulting bundle to 0 or 1 (simulation).

Transforming this intuition into a reduction from compressing parity/inner-product instances to distinguishing the real and simulated views, however, is far from straightforward. The major source of difficulty is that neither the real construction nor the simulator are actually $\mathcal{AC}^0$ algorithms, and neither are the computations in the "gate gadgets" used to emulate each gate in the original circuit.[3] In particular, if we want to use the bundle bank machinery, the gate gadgets need to compute linear combinations of bundles in the bank. Moreover, this difficulty is compounded by the fact that, unlike [FRR$^+$10], we cannot use leakage-free hardware to make the bundles used between gates and across executions statistically independent of each other. Even within a single circuit execution, this is already a serious concern. Each bundle is dependant on the bank, and through it on all other bundles. If (as seems natural) we want to use hybrid arguments to focus on differences in the distribution of a single bundle or gate emulation, we need to generate the rest of the view (on which both hybrids agree). This generation, however, is both not in $\mathcal{AC}^0$ and is not independent of the hybrid bundle. We now give intuition for how these two difficulties are overcome, further details are in Section 3.

Step *I*: $\mathcal{AC}^0$ reconstruction via "beefed up" external wire distributions. Our first step towards resolving these difficulties is to add more information to the *external wire distribution* so that we can reconstruct the internal view of each gate-gadget in $\mathcal{AC}^0$. For example, adapting the addition gate gadget of [FRR$^+$10] to our setting, we take the bundle bank to be a matrix $G \in \{0,1\}^{\kappa \times 2\kappa}$. Our addition gate gadget takes as input two bundles $\boldsymbol{d}_i$ and $\boldsymbol{d}_j$, generates a "masking" 0-bundle $\boldsymbol{o} = G \times \boldsymbol{r}$ using the bundle bank and a random linear combination $\boldsymbol{r} \in_R \{0,1\}^{2\kappa}$. It then computes an output bundle $\boldsymbol{d}_k = ((\boldsymbol{d}_i + \boldsymbol{d}_j) + \boldsymbol{o})$. The reconstruction procedure gets the input and output bundles $\boldsymbol{d}_i, \boldsymbol{d}_j, \boldsymbol{d}_k$, as well as the bundle bank G (a $\kappa \times 2\kappa$ matrix), and needs to generate the internal view of the computation. This requires finding a vector $\boldsymbol{r}$ s.t. $G \times \boldsymbol{r} = (\boldsymbol{d}_k - (\boldsymbol{d}_i + \boldsymbol{d}_k))$ and generating the wire values of the matrix-vector multiplication $G \times \boldsymbol{r}$. For arbitrary $\boldsymbol{d}_i, \boldsymbol{d}_j, \boldsymbol{d}_k$ and G this cannot be done in $\mathcal{AC}^0$. To resolve this, we give the reconstruction some additional "advice"; We "beef up" the external wire distribution with vectors $\boldsymbol{r}_i, \boldsymbol{r}_j, \boldsymbol{r}_k$ s.t. $\boldsymbol{d}_i = G \times \boldsymbol{r}_i$, $\boldsymbol{d}_j = G \times \boldsymbol{r}_j$, and $\boldsymbol{d}_k = G \times \boldsymbol{r}_k$, and with all the wire values for computing these matrix-vector multiplications. The reconstruction procedure (for addition gates) can now compute in $\mathcal{AC}^0$ the vector $\boldsymbol{r}_o = (\boldsymbol{r}_k - (\boldsymbol{r}_i - \boldsymbol{r}_j))$ and (by linearity of matrix multiplication) the wire values for the matrix-vector multiplication $\boldsymbol{o} = G \times \boldsymbol{r}_o$ to output a consistent view of the addition gate gadget's internal wires. We show that when the external wire distribution is distributed as in the real or simulated execution, our reconstruction procedures generate an indistinguishable view for the internal wires of each gate gadget. We note that reconstructing the multiplication gadget is significantly more complicated, and requires further "beefing up" of the external wire distribution.

[3] We note that a similar difficulty also arose in [FRR$^+$10], where in the real view (but not the simulated one) the emulation of multiplication gates required computing parities. In their work this difficulty was solved using the secure hardware (essentially replacing each real emulation of a multiplication gate with an indistinguishable emulation that could be computed in $\mathcal{AC}^0$). We, on the other hand, want to avoid the use of secure hardware.

Step II: indistinguishability of "beefed up" external wire distributions. Given the $\mathcal{AC}^0$ reconstruction procedures, it remains to argue that the "beefed up" external wire distributions of the real and simulated executions are indistinguishable under bounded-length $\mathcal{AC}^0$ leakage. The external wire distribution now includes a lot of information about the bundles, and in particular the bundles are no longer independent of each other because they are tied together via the bundle bank G and, for each bundle $\boldsymbol{d}$, the vector $\boldsymbol{r}$ for which $\boldsymbol{d} = G \times \boldsymbol{r}$. We will argue indistinguishability using a hybrid argument, replacing the bundle bank from real (i.e. 0 parities) to simulated (i.e. uniform), and replacing the parities of bundles on the external wires one-by-one from real to simulated values. As noted above, using a hybrid argument over the bundles one-by-one creates a challenge because the bundles (which all depend on the same bundle bank) are not independent of each other.

To use a hybrid argument over bundles, we *decompose* the execution's view into two parts: the first part depends on the bundle bank, *but not on the hybrid bundle*, and contains essentially all the information needed to generate the parts of the computation that do not involve the hybrid bundle. The second part depends only on the randomness used to form the hybrid bundle. I.e., if $\boldsymbol{d}$ is the hybrid bundle then we take $\boldsymbol{r}$ to be the linear combination of the bundle bank that yields $\boldsymbol{d}$. We use $\boldsymbol{r}$ to pre-compute non-$\mathcal{AC}^0$ information that helps in generating the part of the view involving the hybrid bundle. The key point is that we give *an $\mathcal{AC}^0$ procedure* for combining these two separate parts and generating the entire view of one of the hybrid distributions. Of the two hybrids, which one is generated is determined by the inner product of $\boldsymbol{r}$ with a vector $\boldsymbol{x}$ that depends only on the bundle bank. Now, by the IPPP assumption (assuming we can use $\boldsymbol{x}$ to generate the bundle bank), we know that even given these two "pre-processed" parts of the view (computed from $\boldsymbol{x}$ and $\boldsymbol{r}$ separately), we can combine them in $\mathcal{AC}^0$ to get one of the hybrid external wire distributions, and thus no $\mathcal{AC}^0$ leakage on the hybrid can be statistically correlated with the inner product of $\boldsymbol{x}$ and $\boldsymbol{r}$. Thus, under the IPPP Assumption, $\mathcal{AC}^0$ leakage should not be able to distinguish the hybrids.

In summary, we construct (hybrids of) the real and simulated external wire distributions using "pre-processing", where two pre-computed pieces are combined in $\mathcal{AC}^0$ to give the appropriate external wire distribution. We get a reduction from the IPPP Problem to distinguishing the real and simulated external wire distributions.

2 Model and Definitions

Preliminaries. For a vector or string x we denote by $|x|$ the length of the vector, and by x_i or $x[i]$ the i'th item in the vector. We denote by $x|_i$ the restriction of a vector x to its first i items. For a vector x in $\{0,1\}^n$ we say that x is an *encoding* of $b \in \{0,1\}$ if the XOR (or sum over $\mathbb{GF}[2]$) of x's entries equals b. For vectors $\boldsymbol{x}, \boldsymbol{y} \in \{0,1\}^\kappa$, we use $\boldsymbol{x} + \boldsymbol{y}$ to denote bitwise addition over $\mathbb{GF}[2]$ (i.e. XOR) of the vectors' entries. We use $\boldsymbol{e}_1, \boldsymbol{e}_2, \ldots, \boldsymbol{e}_n$ to denote the unit vectors over $\{0,1\}^n$ (n will be clear from the context), and we use U_n to denote the uniform distribution over $\{0,1\}^n$. For a finite set S we denote by $y \in_R S$

that y is a uniformly distributed sample from S. For a distribution D we use $y \sim D$ to denote the experiment of sampling y by D. We use $\Delta(D, F)$ to denote the statistical (L_1) distance between distributions D and F. For a circuit (or function) C and an oracle PPTM M we use $M^C(x)$ to denote M run on input x with oracle access to C.

2.1 Leakage Attacks and Security

Definition 1 ($\mathcal{AC}^0$-Leakage Attack $\mathcal{A}^\lambda[C, \mathit{Update}](1^\kappa)$). *A* continual *$\lambda$-bit $\mathcal{AC}^0$-leakage attack of adversary $\mathcal{A}$ on C with update procedure Update proceeds in rounds. In each round $t = 1, 2, \ldots$, there is a circuit C_t computed in the previous round (where $C_1 = C$). The $\mathcal{AC}^0$ adversary $\mathcal{A}$ chooses an input x_t for C_t and an $\mathcal{AC}^0$ and λ-bit output leakage function $\ell_t(\cdot)$, which takes as input: (i) the entire computation of C_t on input x_t (including all circuit wire values and all random coins), (ii) the entire computation of the update procedure Update run on C_t and outputting C_{t+1} (including all circuit wire values and all random coins of Update). The adversary's view in round t is $\mathit{view}_t = (x_t, C_t(x_t), \ell_t(\text{entire computation of } C(x) \text{ and } \mathit{Update}(C_t)))$. The adversary's choices of the input x_t and the leakage ℓ_t are adaptive can depend on the views in all previous rounds.*

The attack proceeds for T rounds (where T is chosen by the adversary). The attack's output (or adversary view in the attack) is $\mathit{view} = (\mathit{view}_1, \mathit{view}_2, \ldots, \mathit{view}_T)$. The running time of $\mathcal{A}$ is its total running time in the attack, i.e. a polynomial-time adversary can only run for $\mathrm{poly}(\kappa)$ *rounds.*

Remark 2. Throughout this work when any of our algorithms compute matrix multiplication (or matrix-vector/vector-vector multiplication), this is done in the straightforward (if inefficient) way. The *partial sums* are the sub-computations in the multiplication, e.g. in computing the inner product $\langle x, y\rangle$, the partial sums are $(\langle x|_1, y|_1\rangle, \langle x|_2, y|_2\rangle, \ldots, \langle x, y\rangle)$. The leakage on the computation takes all of these partial sums as a part of its input.

Definition 2 ($\lambda(\cdot)$-Secure Compiler for $\mathcal{AC}^0$ leakage). *Let Init and Update be two PPTMs that take as input a circuit C and security parameter κ. We say that $(\mathit{Init}, \mathit{Update})$ is a $\lambda(\cdot)$-secure compiler for $\mathcal{AC}^0$ leakage, for any circuit C the following two requirements hold:*

- *Functionality: the circuits $C_1 = \mathit{Init}(C, \kappa), C_2 = \mathit{Update}(C_1, \kappa), C_3 = \mathit{Update}(C_2, \kappa), \ldots$ are each with all but negligible probability functionally equivalent to the original circuit C. For all $i \geq 1$ the circuits C_i are of the same size.*
- *Security: for any PPTM adversary $\mathcal{A}$ there exists a PPTM simulator $\mathcal{S}$ such that the view $\mathcal{A}^{\lambda(\kappa)}[\mathit{Init}(C, \kappa), \mathit{Update}(\cdot, \kappa)](1^\kappa)$ of the adversary in an $\mathcal{AC}^0$ leakage attack is indistinguishable from the view $\mathcal{S}^C(1^\kappa)$ generated by the simulator from black-box access to C. Note that there is no leakage on the Init procedure.*

Remark 3. Note we may consider many relaxations and strengthenings of this definition. For example, we could relax functionality to require only that each C_i

is functionally equivalent to C w.h.p. on each input over the coins of the *Init* and *Update* procedures and/or the coins of C_i. We could also strengthen security to restrict the simulator $\mathcal{S}$ to only have oracle access to the inputs queried by the adversary (and in fact our construction meets this more stringent requirement). The choices in the definition above were made mainly for the sake of simplicity.

2.2 The IPPP Problem and Assumption

We give formal definitions of the IPPP Problem and Assumption, see also the discussion in the introduction. See the full version for further details and discussion, including connections to problems in complexity theory, as well as further properties such as a worst-case to average-case hardness reduction.

Problem 1 ($(\lambda(\cdot), s(\cdot), \varepsilon(\cdot))$-Inner Product with Pre-Processing (IPPP)). A triplet $(\mathcal{C}, \mathcal{C}^1, \mathcal{C}^2)$ of circuits ensembles solves the $(\lambda(\cdot), s(\cdot), \varepsilon(\cdot))$-Inner Product with Pre-Processing Problem (IPPP) if:

1. there exists a polynomial $p(\cdot) : \mathbb{N} \to \mathbb{N}$, such that $\forall \kappa \in \mathbb{N}$:
   (a) $\mathcal{C}^1_\kappa$ and $\mathcal{C}^2_\kappa$ are of size at most $p(\kappa)$, with input length κ (and output length at most $p(\kappa)$). These are both randomized circuits with a common random input. We call $\mathcal{C}^1$ and $\mathcal{C}^2$ the *pre-processing circuits ensembles* (or circuits for short)
   (b) $\mathcal{C}_\kappa$ is of size at most $s(\kappa)$, with input length at most $p(\kappa)$ and output length $\lambda(\kappa)$.
   We call $\mathcal{C}$ the *output circuit ensemble* (or circuit for short).
   (c) the combined output lengths of $\mathcal{C}^1_\kappa$ and $\mathcal{C}^2_\kappa$ equal the input length of $\mathcal{C}_\kappa$
2. for infinitely many $\kappa \in \mathbb{N}$:

$$\begin{aligned}\Delta(\ & \{C_\kappa(C^1_\kappa(x), C^2_\kappa(y)) : x, y \in_R \{0,1\}^\kappa \text{ s.t. } \langle x, y\rangle = 0\},\\ & \{C_\kappa(C^1_\kappa(x), C^2_\kappa(y)) : x, y \in_R \{0,1\}^\kappa \text{ s.t. } \langle x, y\rangle = 1\}) \geq \varepsilon(\kappa)\end{aligned}$$

   randomness in both distributions is over the choice of x, y and the (shared) coins of $\mathcal{C}^1, \mathcal{C}^2$.

Assumption 2 ($\lambda(\cdot)$-IPPP Assumption) *For any polynomial $s(\cdot)$ and inverse polynomial $\varepsilon(\cdot)$, no triplet of circuit ensembles $(\mathcal{C}, \mathcal{C}^1, \mathcal{C}^2)$ with $\mathcal{C}$ in $\mathcal{AC}^0$ can solve the $(\lambda(\cdot), s(\cdot), \varepsilon(\cdot))$-IPPP problem.*

3 Transformation Against $\mathcal{AC}^0$ Leakage

In this section we detail a secure compiler for $\mathcal{AC}^0$ leakage and give more details on the proof of Theorem 1. For ease of exposition we consider throughout this section a circuit $C(\cdot, \cdot)$ that is known to the adversary and takes two inputs x and y. The input x is the one chosen adaptively by the adversary in each round, whereas the input y is *secret* and protected by the compiler. In particular this gives a compiler a la Definition 2 by viewing C as a universal circuit and y as the description of a particular circuit to be compiler. This is similar to what is done in the secure multi-party computation literature.

The compiler transforms y into a secret state for an emulation of the universal computation. This secret state includes a *wire bundle* for each y-input wire of the circuit. The XOR of this bundle is the value of its y-input wire. The secret state also includes a "bundle bank" of bundles encoding 0 (see the overview in the introduction). Given bundles for the input wires, the transformed circuit's computation then proceeds gate by gate, using the bundles computed for that gate's input wires and the bundles in the bank to compute an output bundle. In the construction below, we present *Init* and *Update* as procedures for initializing and updating the secret state — a collection Y (viewed as a matrix whose columns are the bundles) of bundles for the y input wires, and the bundle bank G (also a matrix). We call the "bundle bank" in round t the *"generating matrix"* for that round (as it is used to generate fresh 0-bundles).

In Figure 3 we present the procedures for emulating the computation of each gate in the original circuit. Given this view of the compiler's operation (slightly modified from the one in Definition2), we view the *Init* and *Update* procedure's outputs as secret states that are later plugged into the gate-by-gate emulator of (public) circuit's computation. They are in Figures 1 and 2. The secret states can be transformed into a full-blown circuit a la Definition 2 by augmenting them with the (publicly and known to all parties) emulation of the circuit C.

```
Initialization  $Init(1^\kappa, y)$

1. for every input wire $i$, corresponding to bit $j$ of the input $y$, generate a new
   encoding: $\boldsymbol{d}_i \in \{0,1\}^\kappa$, a uniformly random vector whose XOR is $y_j$.
   Let $Y$ be the matrix whose columns are these encodings.
2. generate a new uniformly random $\kappa \times 2\kappa$ matrix $G$ whose columns are all encodings of 0.
3. output $state_1 \leftarrow (Y, G)$.
```

Fig. 1. *Init* procedure, to be run in an offline stage on circuit C and secret y.

```
State Update  $Update(1^\kappa, state_t = (Y, G))$

1. for each column $Y_i$ of $Y$: $Y'_i \leftarrow Y_i + G \times \boldsymbol{r}$, where $\boldsymbol{r} \sim U_{2\kappa}$
2. pick a uniformly random $2\kappa \times 2\kappa$ matrix $R$. $G' \leftarrow G \times R$.
3. output $state_{t+1} \leftarrow (Y', G')$.
```

Fig. 2. *Update* procedure, to be run under leakage between evaluations.

Security Proof: Further Details and Organization. The simulator is specified in Figure 4. We want to prove that the view it generates is indistinguishable from the real view, under bounded length $\mathcal{AC}^0$ leakage in each round. See the

```
Gate Computations (with bundle-bank/generating matrix $G$)

Addition $(\boldsymbol{d}_i, \boldsymbol{d}_j)$:
  1. $\boldsymbol{q} \leftarrow \boldsymbol{d}_i + \boldsymbol{d}_j$
  2. $\boldsymbol{o} \leftarrow G \times \boldsymbol{r}$, where $\boldsymbol{r} \sim U_{2\kappa}$
  3. output $\boldsymbol{d}_k \leftarrow \boldsymbol{q} + \boldsymbol{o}$

Multiplication $(\boldsymbol{d}_i, \boldsymbol{d}_j)$:
  1. $B \leftarrow \boldsymbol{d}_i \times \boldsymbol{d}_j^T$,
     i.e. the $\kappa \times \kappa$ matrix where entry $[\ell, m]$ equals $\boldsymbol{d}_i[\ell] \cdot \boldsymbol{d}_j[m]$
  2. $S \leftarrow G \times R$,
     where $R$ is a $\{0,1\}$ uniformly random $2\kappa \times \kappa$ matrix
  3. $U \leftarrow B + S$
  4. for each row $\ell$ of $U$, compute $\boldsymbol{q}[\ell] = \oplus_{m \in [\kappa]} U[\ell, m]$
  5. output $\boldsymbol{d}_k \leftarrow \boldsymbol{q}$

Constant Gate $(b_i)$:
  1. $\boldsymbol{q} \leftarrow [b_i, 0, 0, \ldots, 0]$
  2. $\boldsymbol{o} \leftarrow G \times \boldsymbol{r}$, where $\boldsymbol{r} \sim U_{2\kappa}$
  3. output $\boldsymbol{d}_k \leftarrow \boldsymbol{q} + \boldsymbol{o}$

Duplication $(\boldsymbol{d}_i)$:
  1. $\boldsymbol{o} \leftarrow G \times \boldsymbol{r}$, where $\boldsymbol{r} \sim U_{2\kappa}$
  2. $\boldsymbol{o}' \leftarrow G \times \boldsymbol{r}'$, where $\boldsymbol{r}' \sim U_{2\kappa}$
  3. output $\boldsymbol{d}_j \leftarrow \boldsymbol{d}_i + \boldsymbol{o}$ and $\boldsymbol{d}_k \leftarrow \boldsymbol{d}_i + \boldsymbol{o}'$

Output Gate $(\boldsymbol{d}_{output})$:
  1. $\boldsymbol{o} \leftarrow G \times \boldsymbol{r}$, where $\boldsymbol{r} \sim U_{2\kappa}$
  2. $\boldsymbol{d} \leftarrow \boldsymbol{d}_{output} + \boldsymbol{o}$
  3. output $\oplus_{m \in [\kappa]} \boldsymbol{d}[m]$
```

Fig. 3. Gate computations during evaluation, all run under leakage.

introduction for the high-level intuition behind the security proof. The formal argument used in the reduction is more involved. We consider the real and simulated views, *Real* and *Simulated* (respectively). We want to use a distinguisher between the real and simulated view to build an $\mathcal{AC}^0$ circuit for solving the IPPP problem, leading to a contradiction. This requires care, and in particular we will use several hybrid views and seek methods for generating them in $\mathcal{AC}^0$ (with some pre-processing, see below).

One important difference between the *Real* and *Simulated* views is in the generating matrices they use (whose columns all encode 0 in *Real*, but are uniformly random in *Simulated*). In both views we want to use each generating matrix G to generate encodings of 0's (or 1's in *Simulated*) upon request. The immediate way of doing this is choosing randomness $\boldsymbol{r}$ from the proper distribution and computing $G \times \boldsymbol{r}$ together with all of the partial sums (the partial

Simulator

Initialize G_0 as a uniformly random $\kappa \times 2\kappa$ matrix, and Y_0 as a uniformly random $\kappa \times n$ matrix (where n is the bit length of y).
For rounds $t \leftarrow 1, 2, \ldots$, on input x_t, generate the view for that round gate by gate:

1. For addition, multiplication, constant and duplication gates, operate exactly as the real gate computations in Figure 3 (albeit here G is uniformly random).
2. For the output gate, on encoding $\boldsymbol{d}_{output}$, retrieve the output bit $C(x_t, y)$.
   Generate $\boldsymbol{o}$ as a uniform linear combination of the columns of G s.t. $\oplus_{m \in [\kappa]} (\boldsymbol{d}_{output} + \boldsymbol{o})[m] = C(x_t, y)$.
3. For the state update, operate as in the real state update in Figure 2. I.e. generate Y_{t+1} by adding random linear combinations of G_t to the columns of Y_t.
   To generate G_{t+1}, multiply G_t by a random invertible $2\kappa \times 2\kappa$ matrix.

Fig. 4. Simulator $\mathcal{S}$.

sums are needed for generating the entire view of each wire in the matrix-vector multiplication). Unfortunately, this is not an $\mathcal{AC}^0$ computation. See the discussion in Section 1.1 for intuition regarding this obstacle and the high-level ideas for overcoming it.

We will use several hybrid views, and set up the views (real, simulated or hybrid) as follows. For every round t we will have a generating matrix G_t (either uniformly random, or random s.t. all columns encode 0). We generate an *external wire distribution* which specifies the encoding/bundle on each of the circuit wires together with additional information about it and global information about this round. This recalls the high-level structure of the security proof in [FRR$^+$10], though here even the *simulated* view cannot be generated in $\mathcal{AC}^0$ and so we need a "'beefed up" external wire distribution (as discussed in Section 1.1). In particular, for each wire encoding we also include its decomposition into a linear combination of G's columns (the bundle bank) XORed with a 0 or a 1 bit. See the full version for a full specification of the external wire distribution.

We argue that distinguishing the external wire distributions for any pair of adjacent views (simulated, hybrid, real) is hard given only bounded length $\mathcal{AC}^0$ leakage from each round (this usually involves another hybrid argument over the rounds and/or wires).

To finish the argument and show the complete views in their entirety are also indistinguishable we need to generate the actual views, including also the internal gate wires. As discussed in Section 1.1, we give *reconstruction procedures* for completing the view and setting values for all of the internal gate wires in both the *Real* and *Simulated* views. The reconstruction procedures are in $\mathcal{AC}^0$, and so indistinguishability of the views follows immediately from the indistinguishability of their external wire distributions.

The full security proof has been omitted for lack of space. See the full version for details.

4 Acknowledgements

Much of this research is joint work with Shafi Goldwasser. I am grateful for her countless insights, suggestions and contributions. Thanks also to Vinod Vaikuntanathan for suggesting that the main result implies obfuscation secure against $\mathcal{AC}^0$ adversaries.

References

[AGV09] Adi Akavia, Shafi Goldwasser, and Vinod Vaikuntanathan. Simultaneous hardcore bits and cryptography against memory attacks. In *TCC*, pages 474–495, 2009.

[Ajt11] Miklós Ajtai. Secure computation with information leaking to an adversary. In *STOC*, pages 715–724, 2011.

[BFS86] László Babai, Peter Frankl, and Janos Simon. Complexity classes in communication complexity theory (preliminary version). In *FOCS*, pages 337–347, 1986.

[BGI$^+$01] 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.

[BKKV10] Zvika Brakerski, Yael Tauman Kalai, Jonathan Katz, and Vinod Vaikuntanathan. Overcoming the hole in the bucket: Public-key cryptography resilient to continual memory leakage. In *FOCS*, pages 501–510, 2010.

[DI06] Bella Dubrov and Yuval Ishai. On the randomness complexity of efficient sampling. In *STOC*, pages 711–720, 2006.

[DP08] Stefan Dziembowski and Krzysztof Pietrzak. Leakage-resilient cryptography. In *FOCS*, pages 293–302, 2008.

[FRR$^+$10] 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.

[GK05] Shafi Goldwasser and Yael Tauman Kalai. On the impossibility of obfuscation with auxiliary input. In *FOCS*, pages 553–562, 2005.

[GR10] Shafi Goldwasser and Guy N. Rothblum. Securing computation against continuous leakage. In *CRYPTO*, pages 59–79, 2010.

[GR12] Shafi Goldwasser and Guy N. Rothblum. How to compute in the presence of leakage. *Electronic Colloquium on Computational Complexity (ECCC)*, (010), 2012.

[HN10] Danny Harnik and Moni Naor. On the compressibility of *np* instances and cryptographic applications. *SIAM J. Comput.*, 39(5):1667–1713, 2010.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In *CRYPTO*, pages 463–481, 2003.

[JV10] Ali Juma and Yevgeniy Vahlis. Protecting cryptographic keys against continual leakage. In *CRYPTO*, pages 41–58, 2010.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Wiener, editor, *CRYPTO99*, pages 388–397, 1999.

[MR04] Silvio Micali and Leonid Reyzin. Physically observable cryptography (extended abstract). In *TCC*, pages 278–296, 2004.

[Raz87] Alexander Razborov. Lower bounds for the size of circuits of bounded depth with basis and, xor. Math. Notes of the Academy of Science of the USSR 41, 1987.

[RCL] Boston University Reliable Computing Laboratory. Side channel attacks database. http://www.sidechannelattacks.com.

[RSVC$^+$11] Mathieu Renauld, François-Xavier Standaert, Nicolas Veyrat-Charvillon, Dina Kamel, and Denis Flandre. A formal study of power variability issues and side-channel attacks for nanoscale devices. In *EUROCRYPT*, pages 109–128, 2011.

[Smo87] Roman Smolensky. Algebraic methods in the theory of lower bounds for boolean circuit complexity. In *STOC*, pages 77–82, 1987.