Electronic Colloquium on Computational Complexity, Report No. 10 (2012)

How to Compute in the Presence of Leakage

Shafi Goldwasser*
MIT and The Weizmann Institute of Science

Guy N. Rothblum†
Microsoft Research

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 $poly(\kappa)$ slower than P and is composed of a series of calls to $poly(\kappa)$ 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 $\tilde{\Omega}(\kappa)$ – on the sub-algorithms of P' and the randomness they use.

We prove that for 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.

*Research supported by DARPA Grant FA8750-11-2-0225 and NSF Grant CCF-1018064. Email: shafi@theory.csail.mit.edu.

†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. Email: rothblum@alum.mit.edu.

ISSN 1433-8092

Contents

1 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.

This question is interesting for cryptographic as well as non-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 any computational assumptions and thus will be applicable to non-cryptographic settings without adding any new conditions. They will hold even if one-way functions (and cryptography as we know it) do not exist.

The question of executing general computations for an unbounded (continual) number of executions, viewed largely within the context of cryptographic algorithms, has been addressed in the last few years with varying degrees of success in different adversarial settings. The crucial question seems to be how to model the partial information or *leakage* that an adversary can obtain during executions. The goal is to simultaneously capture real world attacks and achieve the right level of theoretical abstraction.

Impossibility results on obfuscation [BGI$^+$01] imply inherent limitations on the leakage that can be tolerated in the continual attack model for general programs P. Even if only a single bit of leakage is output in each execution, Impagliazzo [Imp10] observes that if this bit can be computed as a function of the entire internal state of the execution, then there exist polynomial time computable functions f, for which no execution can achieve leakage resilience. Thus, to rule out this impossibility, we must put additional restriction on the leakage attack model.

1.1 Continual Leakage Attack Models and Prior Work

We discuss a few leakage attack model restrictions and corresponding results which have been considered for the question of protecting *general programs under continual leakage*.

ISW-L. The pioneering work of Ishai, Sahai, and Wagner [ISW03] 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. Let L be a global bound on the number of wires that can leak. Then, they show how to convert any circuit C into a new circuit C' of size $O(|C| \cdot L^2)$ which is unconditionally resilient to leakage of up to L individual wire values. In fact, their method achieves more. The new circuit C' is composed of a sequence of sub-circuits, each of size $O(L^2)$, of which the value of L arbitrary wires can leak.

CB-L. Faust, Rabin, Reyzin, Tromer and Vaikuntanathan [FRR$^+$10] extended the leakage model and result of [ISW03]. They still model an algorithm as a stateful circuit, but in every execution, they let the adversary learn the result of any bounded length $\mathcal{AC}^0$ computable function f on the values of all the wires. Let L be a global bound on the output length of function f. under the

additional assumption that leak free hardware components exist, they show 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 f computed on the entire set of wire values. Similarly to [ISW03], their method achieves actually more. The new circuit C' is composed of a sequence of sub-circuits, each of size $O(L^2)$, and is resilient to L bits of $\mathcal{AC}^0$ leakage on each of these sub-circuits.

RAM-L. the RAM model of Goldreich and Ostrovsky [GO96] considers a CPU, which loads data from fully protected memory, and runs its computations in a secure CPU. [GO96] 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.[1] This was recently extended by Ajtai [Ajt11]. 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. 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. Taking L to be a security parameter, [Ajt11] shows how to transform a program P on input size n, to a program P' which is divided into sub-computations of size $O(L)$, and is resilient to L compromised accesses in each sub-computation.

OC-L. the Micali-Reyzin [MR04] 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 [GR10] and by Juma and Vhalis [JV10], 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 length bounded functions,[2] computed on each sub-computation's input and randomness. To obtain results in this model, both [GR10] and [JV10] needed to assume the existence of leak free hardware components that produce samples from a polynomial time sampleable distribution. Namely, it is assumed that there is no data leakage from the randomness generated and the computation performed inside of the device. Assuming the intractability of the DDH problem, [GR10] transform P to P' which is composed of $O(|P|)$ sub-computations, each of size $O(poly(L))$, that is resilient to leakage of length L on each sub-computation. [JV10] assume the existence of fully homomorphic encryption scheme, and get P' composed of $O(1)$ sub-computations, one of which has size $O(|P| \cdot \text{poly}(L))$. P' is resilient to leakage of length L on each sub-computation, assuming that the fully homomorphic encryption scheme cannot be broken in time $2^{O(L)}$.

The assumptions on the existence of leak-free secure hardware components make it possible, in the security proofs of [GR10] and [JV10], to argue that the view of the side channel attack in the real protocol is indistinguishable from the view output by a polynomial time simulator, which samples a very different, but computationally indistinguishable, distribution.

Finally, we 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. [DP08, Pie09, FKPR10, BKKV10, DHLAW10, LRW11, LLW11]), and on *storage* ([DLWW11]), has been considered under various additional leakage models in a rich body of recent works. We elaborate on a few pertinent results in Section 1.4.

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

[2] In contrast to the $\mathcal{AC}^0$ restriction on f in [FRR+10]

1.2 The New Work

In this paper, we address the question of how to transform any algorithm P into a functionally equivalent algorithm $Eval$ which can be run for an unbounded number of executions, in the presence of leakage attacks on the internal state of the executions. Before stating our exact results, let us describe the power of our leakage-adversary, and the security guarantee to be provided

Leakage Adversary. The leakage attacks we address are in the "only computation leaks information" model of [MR04]. 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. This is in contrast to previous works that only allow the adversary to obtain polynomial-time computable functions of the execution's internal state [GR10, JV10, BKKV10, DHLAW10, LRW11, LLW11, DLWW11].

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.

Informal Main Theorem. 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) $O(poly(\kappa))$.
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 $\text{poly}(\kappa)$.

Our result holds unconditionally, without the use of computational assumptions or leak-free hardware. In Section 2 we give an overview of the construction, and highlight some of the new technical ideas of our work.

OC-L and the Leaky CPU Model. An alternative model to OC-L is that of a *leaky CPU*. We proceed with an informal 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 poly(κ) 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.

Comparison to Prior Work. We now compare our main result to prior work on *protecting general programs under continual leakage*. See Section 1.4 for other related work.

Comparing to the work of Ishai, Sahai and Wagner [ISW03] in the ISW-L leakage model, they convert any circuit C into a new circuit C', which is composed of $O(|C|)$ sub-circuits each of size $O(L^2)$, and allow the leakage of L arbitrary wires from each sub-circuit. Our transformation converts C into $O(|C|)$ sub-circuits, each of size $\tilde{O}(L^\omega)$, from which L *arbitrary bits of information* can be leaked (here ω is the exponent in the best algorithm known for matrix multiplication). These leaked bits can be the output of arbitrary computations on the wire values.

Comparing to the work of Faust *et al.* [FRR+10] in the CB-L model, 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 arbitrary length bounded OC-L leakage that operates separately (if adaptively) on each sub-computation.

Comparing to the work of Ajtai [Ajt11] in the RAM-L model, he divides the computation into sub-computations of size $O(L)$, and shows resilience to an adversary who see the full contents of memory loaded into CPU for L memory accesses, whereas all the other memory accesses are perfectly hidden. Translating our result to the RAM model, we divide the computation into sub-computations of size $\tilde{O}(L^\omega)$, and show resilience against an adversary that can receive L *arbitrary bits of information* on the entire set of memory accesses and randomness. In particular, there are no protected or hidden accesses.

Comparing to the work of Goldwasser and Rothblum [GR10] and of Juma and Vhalis [JV10] in the OC-L model, the main *qualitative* difference is that both of those prior works use computational

intractability assumptions and 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, [JV10] transform a circuit of size C into a new circuit C' of size $\text{poly}(\kappa) \cdot |C|$. The new circuit C' is composed of $O(1)$ sub-circuits (one of the subcircuits is of size $\text{poly}(\kappa) \cdot |C|$). Assuming a fully-homomorphic encryption scheme that is secure against adversaries that run in time $\exp(O(L))$, their construction can withstand L bits of leakage on each sub-circuit. For example, if the FHE is secure against $\text{poly}(\kappa)$-time adversaries, then the leakage bound is $O(\log \kappa)$. In our new construction, for leakage parameter L, there are $O(|C|)$ sub-computations (i.e. more sub-computations), each of size $\tilde{O}(L^{\omega})$ (i.e. smaller), and each withstanding L bits of leakage (i.e. the amount of leakage we can tolerate, relative to the sub-computation size, is larger). The quantitative parameters of [GR10] are similar to the current work (up to polynomial factors).

Subsequent Related Work. The compiler provided in this work, and the new tools introduced in its construction, have been used in several subsequent works.

Bitansky *et al.* [BCG$^+$11] use the compiler to obfuscate programs using leaky secure hardware. In a nutshell, they run each "sub-computation" on a separate leaky secure hardware component. The new challenge in that setting is providing security even when the communication channels between the components are observed and controlled by an adversary.

Boyle *et al.* [BGJK12] use the compiler to build secure MPC protocols that are resilient to corruptions of a constant fraction of the players and to leakage on each of the players (separately). The MPC should output a function of the players' inputs computed by some circuit C. Intuitively, one can think of each player in the MPC as running one of the "sub-computations" in a compilation of C using our OC-L compiler. 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.

Using the idea of *ciphertext banks*, a technical tool introduced in this work, [Rot12] gives a compiler for $\mathcal{AC}^0$ leakage in the CB-L model. The new compiler removes the need for secure hardware components that was present in the work of [FRR$^+$10], but its security relies on an unproven computational assumption about the power (or rather, the weakness) of $\mathcal{AC}^0$ circuits with pre-processing.

1.3 Connections with Obfuscation

We remark that while protecting cryptographic algorithms from side channels is an immediate application (and motivation) for this work, the question of protecting computations is interesting for non-cryptographic computations, e.g. if one-way functions do not exist. In particular, our results do not rely on cryptographic assumptions and so they would continue to hold. As a motivating example, consider a proprietary algorithm running on a cloud server, where parts of its internals might be observed.

This motivating example brings to light the 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 [BGI$^+$01], 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 algorithm) 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.

The recent work of Bitansky *et al.* [BCG$^+$11], mentioned above, makes the connection between obfuscation and the OCL attack model even more explicit. They first strengthen the requirement of OCL attack model to allow the adversary to control the order of the execution of the sub-components (they call this DCL). They then show that any compiler that converts stateful circuits into circuits that are secure in the DML model, implies the possibility of obfuscation of any program given simple hardware components which themselves are subject to memory leakage attacks.

1.4 Other Related Work

Constructions in the OCL Leakage Model. Various constructions of *particular cryptographic primitives* [DP08, Pie09, FKPR10], such as stream ciphers and digital signatures, have been proposed in the OCL attack model and proved secure under various computational intractability assumptions. The approach in these results was to consider leakage in design time and construct new schemes which are leakage resilient, rather than a general transformation on non leakage-resilient schemes

In the context of a **bounded** number of executions, we remark that the work of Goldwasser, Kalai and Rothblum [GKR08] on one-time programs imply that any cryptographic functionality can be executed *once* in the presence of OCL attack after the initial compilation is done. There any data that is ever read or written can leak in its entirety (i.e tolerate the identity leakage function). This holds under the assumption that one-way functions exist and requires no secure hardware. The idea is that in the compilation stage, one transforms the cryptographic algorithm into a one-time program with one crucial difference. Whereas one-time programs use special hardware based memory to ensure that only certain portions of this memory cannot be read by the adversary running the one-time program, in the context of leakage the party who runs the one-time program is not an adversary but rather the honest user attempting to protect himself against OCL attacks. In the compilation stage, the honest user, stores the entire content of the special hardware based memory of [GKR08] in ordinary memory. At the execution stage, the user can be trusted to only read those memory locations necessary to run the single execution. Since an OCL attack can only view the contents of memory which are read, the execution is secure. We further observe that the follow up work of Goyal *et al.* [GIS$^+$10] on one-time programs, which removes the need for the one-way function assumption, similarly implies that any cryptographic functionality can be executed *once* in the presence of OCL attacks unconditionally.

Specific Cryptographic Primitives in the Continual Memory Leakage Model. The continual memory-leakage attack model for public key encryption and digital signatures was introduced by Brakerski et al. [BKKV10] and Dodis et al. [DHLAW10]. They consider a model where an adversary can periodically compute arbitrary polynomial time functions of bounded output length L on the entire secret memory of the device. The device has an internal notion of time periods and, at the end of each period, it updates its secret key, using some fresh local randomness, maintaining the same public key throughout. As long as the rate at which the adversary can compute its leakage functions is slower than the update rate, [BKKV10, DHLAW10, LRW11, LLW11] can

construct leakage resilient public-key primitives which are still semantically secure under various intractability assumptions on problems on bi-linear groups. The continual memory leakage model is quite strong: it does not restrict the leakage functions, as in say ISW-L, to output individual wire values, or as in CB-L, to $\mathcal{AC}^0$ bounded functions, nor does it restrict the leakage functions to compute locally on sub-computations, as in RAM-L or OC-L. However, as pointed out by the impossibility result discussed above, this model cannot offer the kind of generality or security that we are after. In particular, the results in [BKKV10, DHLAW10, LRW11, LLW11] do not guarantee that the view the attacker obtains during the execution of a decryption algorithm is "computationally equivalent" to an attacker viewing only the I/O behavior of the decryption algorithm. For example, say an adversary's goal in choosing its leakage requests is to compute a bit about the plain-text underlying ciphertext c. In the [BKKV10, DHLAW10] model, it will simply compute a leakage function that decrypts c, and output the requested bit. This could not be computed from the view of the I/O of the decryption algorithms decrypting ciphertexts which are unrelated to c.

Continual Leakage on a Stored Secret. A recent independent work of Dodis, Lewko, Waters, and Wichs [DLWW11], 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 [ISW03, FRR$^+$10, GR10, JV10] as they all must implicitly maintain the secret "state" of the input algorithm (or circuit) throughout its continual execution. The beauty of [DLWW11] is that no interaction is needed between the devices, and they can update themselves asynchronously.

We proceed to present an overview of our compiler and highlight some of our main technical contributions in Section 2 below. The full definitions, tools, and specifications of the compiler are in the subsequent sections. See the roadmap in Section 2.4.

2 Compiler Overview and Technical Contributions

The main contribution of this paper is a compiler which takes any algorithm in the form of a boolean circuit and transforms it into a functionally equivalent probabilistic stateful algorithm. A user can run this transformed secure algorithm for an unbounded (polynomial) number of executions. The security guarantee is that any computationally unbounded adversary who launches a leakage attack on the algorithm's executions, learns nothing more than the input-output behavior.

In this section, we will give an overview of the compiler, its main components, and the technical ideas introduced. The transformed secure algorithm is executed repeatedly, on a sequence of inputs chosen by an adversary. Each execution of the transformed secure algorithm proceeds by a sequence of sub-computations, and the adversary's view of each execution is through the results of a sequence of leakage functions (chosen adaptively and with bounded output length), applied to these sub-

computations.

The first component in our construction is a leakage-resilient one-time pad cryptosystem (LROTP), which we refer to as the *subsidiary* cryptosystem. See Section 2.1 for further details. We remark that it is important to distinguish between the leakage resilience of the secure transformed algorithm, and the leakage resilience of the subsidiary LROTP keys and ciphertexts. Whereas the LROTP scheme retains security even after direct applications of bounded output length leakage on the LROTP keys and ciphertexts (separately), the security guarantee for the transformed algorithm is that, even under a leakage attack on its execution, *there is will be no leakage at all on its internal state or secrets*. All that an adversary can learn is its input-output behavior.

Our compiler transforms a program by encrypting the bits of its description using the LROTP cryptosystem. In Section 2.2, we show how to use these encryptions to compute the program's output on a *single* given input. This "one-time" safe evaluation is resilient to OC leakage attacks. The main new component we use is a procedure for "safe homomorphic evaluation" of LROTP-encrypted bits.

In Section 2.3 we show how to extend the one-time safe evaluation to any polynomial number of safe evaluations. This yields a compiler that is secure against continual OC leakage attacks. Here we use a new technical tool of "ciphertext banks", which allow us to repeatedly generate secure ciphertexts even under leakage.

2.1 Leakage-Resilient One Time Pad

Our construction uses a *leakage resilient one-time pad* cryptoscheme (LROTP) as one of its main components. 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. In particular, we highlight several properties of LROTP, specified below, that are used in our construction. See Section 4 for further 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 ciphertext is of bounded length. This ensures 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 (otherwise, again, it could 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. We call such sources *independent up to orthogonality*, see Definition 3.10. Since the inner product function is a two-source extractor (see Lemma 3.7), 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. 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.

- **Key and Ciphertext Refreshing.** We give procedures for "refreshing" LROTP keys and ciphertexts, injecting new entropy while maintaining the underlying plaintexts. We overview here the case of key refresh, ciphertext refresh is similar. The key entropy generator outputs a uniformly random $\sigma \in \{0,1\}^\kappa$ s.t. $\sigma[0] = 0$. This σ is used to inject new entropy in the key by updating $key' \leftarrow (key \oplus \sigma)$, so that key' is a uniformly random key, independent of *key*. σ can also be used *on its own and without knowledge of the key*, to "correlate" $\vec{c}$ to a new ciphertext $\vec{c}'$ s.t. $Decrypt(key', \vec{c}') = Decrypt(key, \vec{c})$. The requirement that refreshing on ciphertexts must not use the key, is due to the fact that we always want to avoid loading the ciphertext and key into memory at once (otherwise a leakage attack can decrypt and learn the plaintext). It follows that *without any leakage*, the new key or ciphertext is a uniformly random one that maintains the underlying plaintext.

  In this work, key and ciphertext refreshing is used to obtain security properties *even in the presence of leakage*. One task that we will consider is permuting m key-ciphertext pairs that all have the same underlying plaintext.[3] We refresh all m pairs and then permute them using a random permutation π. If there is no leakage on this refresh-and-permute procedure, then it follows that even given the m input key-ciphertext pairs, and the m refreshed-and-permuted pairs, the permutation used looks uniformly random. Furthermore, even if there is a bounded amount of leakage on the refresh-and-permute procedure, the distribution of the permutation used, given all input and output key-ciphertext pairs, will have high entropy.

  The example above shows that a single application of key-ciphertext refresh can give security guarantees even in the presence of OC leakage. In particular, it maintains security of the underlying plaintext. It is natural to hope that a *large number of composed applications* of refresh to a key-ciphertext pair also maintains security of the underlying plaintext. However, after a large enough number of composed application, an OC leakage adversary can successfully reconstruct the underlying plaintext. This attack is described in Section 4.2. Intuitively, it "kicks in" once the length of the accumulated leakage is a large constant fraction of the key and ciphertext length. Our construction uses composed applications of refresh, but we take care that the accumulated leakage is never a large enough fraction of the key-ciphertext length. We show that the security properties we use are maintained under a bounded number of composed applications of refresh.

- **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 $\vec{c}_2$.

[3] To be precise, we will consider a related task or independently permuting m sets, each comprising 4 key-ciphertext pairs, and the ciphertexts in each set will *not* all have the same underlying plaintexts. We find the simplified question of permuting m pairs with the same underlying plaintext, as considered here, to be illuminating.

We note that the construction in [GR10] relied on several similar properties of a computationally secure public-key leakage resilient scheme: the BHHO/Naor-Segev scheme [BHHO08, NS09]. Here we achieve these properties with information theoretic security and without relying on intractability assumptions such as Decisional Diffie Hellman.

2.2 Leakage-Resilient Compiler Overview: One-Time Secure Evaluation

Here we describe the high-level structure of the compilation and evaluation algorithm for a single secure execution. In Section 2.3 we will show how to extend this framework to support any polynomial number of secure executions. We note that the high-level structure of the compilation and evaluation algorithm builds on the construction of [GR10]. The building blocks, however, are very different, as the subsidiary cryptosystem is now LROTP, and we now longer use secure hardware.

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.[4] One can think of C as a universal circuit, where y describes a particular algorithm that is to be protected. Or, C can be a public cryptographic algorithm (say for producing digital signatures), and y is a secret key.

The *output* of the compiler on C and y is a probabilistic stateful evaluation algorithm *Eval* (with a *state* which 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 Section 3.4 for a formal definition of utility and security under leakage. In this section, we describe 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, $i < k$. The *Eval* algorithm keeps track of the value $v_i \in \{0,1\}$ on each wire i of the original input circuit $C(y,x)$ in a secret-shared form: $v_i = a_i \oplus b_i$, where $a_i, b_i \in \{0,1\}$. The invariant for every wire is that the a_i shares are *public* and known to all, including the leakage adversary, whereas b_i are *private*: they are kept encrypted by a LROTP ciphertext(s) encrypted under key_i. There is one key for each circuit wire i. For each input wire i, there is a single ciphertext $\vec{c}_i^{in}$. For the output wire *output*, there is a single ciphertext $\vec{c}_{output}^{out}$. For each internal wire i, an output wire for gate g and an input wire for gate h, there are *two* ciphertexts $\vec{c}_i^{out}$ and $\vec{c}_i^{in}$ (both with the same underlying plaintext b_i and the same key key_i). Intuitively, $\vec{c}_i^{out}$ is used in a computation corresponding to gate g (for which i is an output wire), and $\vec{c}_i^{in}$ is used in a computation corresponding to gate h (for which i is an input wire).

We emphasize that the adversary does not actually ever see any key or ciphertext – let alone the underlying plaintext – in their entirely. 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. To initialize *Eval* for a single secure execution, we generate keys and ciphertexts for the output wires, the internal wires, and the y-input wires (initialization is performed without leakage). This is done as follows. For each bit $y[j]$ of the y-input that is carried on a wire i, we generate a key-ciphertext pair $(key_i, \vec{c}_i^{in})$ with underlying plaintext

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

$y[j]$. The input wire's bit value v_i is thus encoded by $a_i = 0$ and $b_i = y[j]$. For each internal wire i, we choose b_i uniformly at random, and we generate a key key_i and two ciphertexts $\vec{c}_i^{out}$ and $\vec{c}_i^{in}$ that both have underlying plaintext b_i (under the key key_i). The internal wire's bit value v_i will be encoded by $b_i \in_R \{0,1\}$ and $a_i = b_i \oplus v_i$ (which we have not yet computed). For the output wire *output*, we generate $(key_{output}, \vec{c}_{output}^{out})$ with underlying plaintext 0. The output bit will be encoded by $b_{output} = 0$, and a_{output}, the public share, that will equal the output value $C(y,x)$. The output wire's public share a_{output} will be computed during evaluation once the input x is specified.

This initialization suffices for a single execution (see below). Looking ahead, the main challenge for multiple execution will be securely generating the keys ciphertexts for each wire even in the presence of OC leakage. See Section 2.3.

Eval **on input** x**.** When a (non secret) input x is selected for *Eval*, we generate ciphertexts for the x-input wires. This determines the private shares (independently of the input x), and sets the stage for computing the public shares—culminating with the computation of the output wire's public share, which equals the circuit's output.

We proceed as follows. Each bit $x[j]$ of the x-input that is carried on wire i, is encoded by $a_i = x[j]$ and $b_i = 0$, where b_i is the underlying plaintext for randomly chosen $(key_i, \vec{c}_i^{in})$. Given these keys and ciphertexts for the x input, and those generated in the initialization, we now have, for each circuit wire i, a key and (one or two) ciphertexts whose underlying plaintext(s) equal b_i. We also have, for each *circuit input* wire i, a public share a_i.

Eval proceeds to compute the public shares of the internal and output wires one by one, using a safe homomorphic computation procedure discussed below. The output is the public share $a_{output}^{out} = C(y,x)$. Throughout the computation, all the private b_i shares are protected from the leakage adversary. Each internal b_i looks "uniformly random" to the adversary, even under leakage. Thus, the public shares a_i of the internal wires reveal nothing about the actual values v_i on those wires. All the adversary "sees" are the input x and the output $a_o = C(y,x)$. The main remaining challenge is evaluating the public shares without exposing the private shares.

Challenge I: Leakage-Resilient "Safe NAND" Computation. We seek a procedure that, for a NAND gate takes as input the public shares for the gates's input wires, and the encrypted private shares for the gate's input wires and output wire. The output should be the correct public share of the gate's output wire. For security, we require 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). We also need a similar procedure for aforementioned duplication gates, but we focus here on the more challenging case of NAND. We give an overview of this procedure, which we call *SafeNAND*, in Section 2.2.1.

2.2.1 Leakage Resilient *SafeNAND*

For a NAND gate with input wires i, j and output wire k, the input to *SafeNAND* is public shares $a_i, a_j \in \{0,1\}$, and ciphertext-key pairs $(key_i, \vec{c}_i^{in}, key_j, \vec{c}_j^{in}, key_k, \vec{c}_k^{out})$. We use $b_i, b_j, b_k \in \{0,1\}$ to denote (respectively) the plaintext bits underlying these key-ciphertext pairs. The goal is to output

$$a_k = ((a_i \oplus b_i) \text{ NAND } (a_j \oplus b_j)) \oplus b_k$$

moreover, we want to do this using a procedure that, even under leakage, exposes nothing about (b_i, b_j, b_k) beyond the output a_k. We proceed with an overview, see Section 6 for details.

As a starting point, we first choose a fresh new $key \leftarrow KeyGen(1^\kappa)$, and compute c'_i, c'_j, c'_k whose underlying plaintexts under this new key remain b_i, b_j, b_k. This uses the key refresh property of the LROTP scheme. Once the ciphertexts are all encrypted under the same *key*, we can use the homomorphic addition properties of LROTP. Starting with an idea of Sanders Young and Yung [SYY99], we can compute NAND by first computing a 4-tuple of encryptions:

$$C \leftarrow (\vec{c}'_k, ((a_i, 0, \ldots, 0) \oplus \vec{c}'_i \oplus \vec{c}'_k), ((a_j, 0, \ldots, 0) \oplus \vec{c}'_j \oplus \vec{c}'_k), ((1 \oplus a_i \oplus a_j, 0, \ldots, 0) \oplus \vec{c}'_i \oplus \vec{c}'_j \oplus \vec{c}'_k))$$

Note the plaintexts underlying the 4 ciphertexts in C are:

$$(b_k, (a_i \oplus b_i \oplus b_k), (a_j \oplus b_j \oplus b_k), (1 \oplus a_i \oplus b_i \oplus a_j \oplus b_j \oplus b_k))$$

and that 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.

The first idea may be to simply decrypt C (using *key*), and compute a_k based on the number of 0's and 1's plaintext underlying C. We cannot do this, however, since the *locations* of 0's and 1's might reveal (via the adversary's leakage) information about (b_i, b_j, b_k) beyond just the value of a_k. A natural idea, then, is to *permute* the ciphertexts before decrypting. This, indeed, is what was suggested by [SYY99]. 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.

Permute: **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 iterations. The input to each iteration is two 4-tuples of keys and ciphertexts. The output from each iteration is a 4-tuple of keys and corresponding ciphertexts, whose underlying plaintexts are some permutation of those in that iteration's input. The key property is that the permutation chosen in each iteration will look "fairly random" even to a leakage adversary. As a result, the composition of these permutations over many iterations will look (statistically close to) uniformly random. The "fairly random" property of each iteration is achieved by a "duplicate and permute" step:

1. creating many copies of the input key and ciphertext 4-tuples
2. refreshing each tuple-copy using key-ciphertext refresh as in Section 2.1 (each refresh uses independent randomness)
3. permutating each tuple-copy using an independently chosen uniformly random permutation

Given (length-bounded) leakage from the above "duplicate-and-permute" step of each iteration, most of the permutations chosen will look "fairly uniform". Finally, after the leakage from each iteration's duplicate-and-permute step has occurred, one of the tuple-copies is chosen. We will show that the permutation used for this tuple-copy will (w.h.p.) look "fairly random", even given the leakage. The tuple-copy chosen in each iteration is then fed as input to the next iteration.

The *Permute* procedure does this for ℓ iterations. We show that the composition of all permutations used is $\exp(-\Omega(\ell))$-statistically close to uniformly random, even given the leakage from all ℓ iterations of *Permute*. This is the high-level intuition for the security of *Permute* and *SafeNAND* (omitting many non-trivial details).

2.3 Leakage-Resilient Compiler Overview: Multiple Secure Evaluations

In this section we modify the *Init* and *Eval* procedures described in Section 2.2 to support any polynomial number of secure evaluations. The main challenge is generating secure key-ciphertext pairs for the output and the y-input wires.

Challenge II: Ciphertext Generation under Continual Leakage. We seek a procedure for repeatedly producing $(key_i, \vec{c}_i)$ pairs. For each y-input wire i corresponding to the j-th bit of y, the underlying plaintext should be $y[j]$. For the output wire *output*, the underlying plaintext should 0. We also seek a procedure for repeatedly producing key_i and a pair of ciphertexts $(\vec{c}_i^{out}, \vec{c}_i^{in})$ that both have the same independently random underlying plaintext $b_i \in_R \{0,1\}$. For security, the underlying plaintexts of the keys and ciphertexts produced should be completely protected even under (repeated) leakage in all the generations.

In previous works such as [FRR+10, JV10, GR10], 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 for repeated secure generations with a fixed underlying plaintext bit. This is what is needed for the y-input and the output wire. We then describe how a ciphertext bank is used to generate a sequence of keys and pairs of ciphertexts (with uniformly random underlying plaintexts) for the internal wires.

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. A *BankUpdate* procedure is used between generations to inject entropy into the ciphertext bank. The intuition behind the ciphertext bank security requirement is that, even under leakage from the repeated generations, the plaintext underlying each key-ciphertext pair is protected. In particular, there are efficient simulation procedures that have arbitrary control over the plaintexts underlying the 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 in Section 2.3.1 below. See Section 5 for details.

Using ciphertext banks, we modify the initialization and evaluation outlined in Section 2.2. In initialization, for each y-input bit $y[j]$, carried on wire i, we initialize a ciphertext bank for repeatedly generating key-ciphertext pairs with underlying plaintext $y[j]$. For the output wire we initialize a ciphertext bank for repeatedly generating key-ciphertext pairs with underlying plaintext 0. In *Eval*, we add an initial step where the ciphertext banks of each y-input wire and of the output wire are used to securely generate a key-ciphertext pair for that wire. After this first step, given an input x, *Eval* proceeds as outlined in Section 2.2.

Finally, to generate a sequence of keys and pairs of ciphertexts for the internal wires, we also provide a *BankRedraw* procedure. This procedure re-draws a new, uniformly and independently random plaintext bit, that will underly the key-ciphertext pairs produced by the bank. To generate a key and a pair of ciphertext with the same underlying plaintext we simply call *BankGen* twice: the key produced in both calls will be the same, but the ciphertexts produced will be different (albeit with the same underlying plaintext). After this pair of generations, we call *BankRedraw* to re-draw the underlying plaintext bit and then *BankUpdate* to inject new entropy. We note that it is the call to *BankUpdate* that changes the key that will be produced in future *BankGen* calls. For

security, we provide efficient simulation procedures that have arbitrary control over the plaintext bits underlying the key-ciphertext pairs that are produced. As above, leakage from simulated calls is statistically close to leakage form the "real" calls. See the overview below in Section 2.3.1, and Section 5 for further details.

We now can now repeatedly generate keys and ciphertext-pairs for internal wires even under leakage. For this, we further modify the initialization and evaluation outlined in Section 2.2. In initialization, we initialize a (single) additional ciphertext bank for the internal wires. This bank is initialized to generated key-ciphertext pairs with a uniformly random underlying plaintext bit. In *Eval*, for each internal wire we use two *BankGen* calls to this bank to generate a key and two ciphertexts. After each two such calls we use *BankRedraw* and *BankUpdate* to re-draw the underlying plaintext bit for the next wire and to inject new entropy.

This completes the high-level description of our *Init* and *Eval* procedures, the full procedures are in Section 7.

2.3.1 Ciphertext Banks for Secure Generation

The ciphertext bank state 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.

The *BankUpdate* procedure injects 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 *BankRedraw* procedure chooses a uniformly random ciphertext $\vec{v} \in \{0,1\}^\kappa$, and adds 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.

For security, 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 if only "sketches" (or "pieces") of C are loaded into memory at any one time, where each "sketch" (or "piece") is a collection of

$(c \cdot \kappa)$ linear combinations of C's columns (for a small $0 < c < 1$), then it is no longer clear how a leakage adversary can compute C's rank or distinguish a real and simulated generation (even if the adversary knows the coefficients of the linear combinations of C's columns).

We show that: (i) *sketches of random matrices are leakage resilient*, and in particular leakage from sketches of C is statistically close in the real and simulated distributions, and (ii) how to implement *BankGen* and *SimBankGen* using subcomputations, where each sub-computation only loads a single "sketch" of C into memory. This implies security for a single generation (or a bounded number). We then extend our leakage-resilience results to show security for an unbounded (polynomial) number of generations. We view these roofs as our most important technical contribution.

2.4 Organization and Roadmap

Definitions, notation and preliminaries are in Section 3. This includes the definitions of secure compilers against leakage and of *independence up to orthogonality*, a central notion in many of our technical proofs. That section also includes lemmas about entropy, multi-source extractors, and leakage-resilience that will be used in the subsequent sections.

We then proceed with a full description of our construction. In Section 4 we specify the leakage-resilient one time pad scheme and its properties. We present the ciphertext bank procedures, used for secure generation of secure ciphertexts under leakage, in Section 5. The *SafeNAND* procedure for securely computing NAND gates on encrypted inputs is in Section 6. These ingredients are put together in Section 7, where we present the main construction and a proof (sketch) of its security.

3 Definitions and Preliminaries

In this section we define leakage and multi-source leakage attacks (Section 3.1) and give a brief exposition about entropy, multi-source extractors, and facts about them that will be used throughout this work (Section 3.2). We then define and discuss the notion of independence up to orthogonality (Section 3.3).

Preliminaries. For a string $x \in \Sigma^*$ (where Σ is some finite alphabet) we denote by $|x|$ the length of the string, and by x_i or $x[i]$ the i'th symbol in the string. For a finite set S we denote by $y \in_R S$ that y drawn uniformly at random from S. We use $\Delta(D, F)$ to denote the statistical (L_1) distance between distributions D and F. For a distribution D over a finite set, we use $x \sim D$ to denote the experiment of sampling x by D, and we use $D[x]$ to denote the probability of item x by distribution D. For random variables X and Y, we use $(X|Y = y)$ or $(X|y)$ to denote the distribution of X, conditioned on Y taking value y.

3.1 Leakage Model

We follow the model and notation used in [GR10].

Leakage Attack. A leakage attack is launched on an algorithm or on a data string. In the case of a data string x, an adversary can request to see any function $\ell(x)$ whose output length is bounded by λ bits. In the case of an algorithm, the algorithm is divided into ordered sub-computations. The

adversary can request to see a bounded-length (λ bit) function of each sub-computation's input and randomness. The leakage functions are computed separately on each sub-computation, in the order in which the sub-computations occur, and can be chosen adaptively by the adversary.

Remark 3.1. *Throughout this work we focus on computationally unbounded adversaries. In particular, we do not restrict the computational complexity of the leakage functions. Moreover, without loss of generality, we consider only deterministic adversaries and leakage functions.*

Definition 3.2 (Leakage Attack $\mathcal{A}^{\lambda}(x)[s]$)**.** Let s be a source: either a *data string* or a *computation*. We model a λ-bit leakage attack of adversary $\mathcal{A}$ with input x on the source s as follows.

If s is a computation (viewed as a boolean circuit with a fixed input), it is divided into m disjoint and ordered sub-computations $sub_1, \ldots, sub_m$, where the input to sub-computation sub_i should depend only on the output of earlier sub-computations. A λ-bit Leakage Attack on s is one in which $\mathcal{A}$ can adaptively choose functions $\ell_1, \ldots \ell_m$, where ℓ_i takes as input the input to sub-computation i and any randomness used in that sub-computation. Each ℓ_i has output length at most λ bits. For each ℓ_i (in order), the adversary receives the output of ℓ_i on sub-computation sub_i's input and randomness, and then chooses ℓ_{i+1}. The view of the adversary in the attack consists of the outputs to all the leakage functions.

In the case that s is a data string, we treat it as a single subcomputation.

Multi-Source Leakage Attacks. A multi-source leakage attack is one in which the adversary gets to launch concurrent leakage attacks on several sources. Each source is an algorithm or a data string. We consider both *ordered* sources, where an order is imposed on the adversary's access to the sources, and *concurrent* sources, where the leakage the leakages from each source can be interleaved arbitrarily. In both case, each leakage is computed as a function of a single source only.

Ordered Multi-Source Leakage. An *ordered* multi-source leakage attack is one in which the adversary gets to launch a leakage attack on multiple sources, where again each source is an algorithm or a data string. The attacks must occur in a specified order.

Definition 3.3 (Ordered Multi-Source Leakage Attack $\mathcal{A}(x)\{s_1^{\lambda_1}, \ldots, s_k^{\lambda_k}\}$)**.** Let $s_1, \ldots, s_k$ be leakage sources (algorithms or data strings, as in Definition 3.2). We model an *ordered* multi-source leakage attack on $\{s_1, \ldots, s_k\}$ as follows. The adversary $\mathcal{A}$ with input x runs k separate leakage attacks, one attack on each source. When attacking source s_i, the adversary can request λ_i bits of leakage. The attacks on sources $s_1, \ldots, s_k$ are run sequentially and in order, i.e. once the adversary requests leakage from s_j, it cannot get any more leakage from s_i for $i < j$.

For convenience, we drop the superscript when the source is exposed in its entirety (i.e. $\lambda_i = |s_i|$). So $\mathcal{A}(x)\{s_1^{\lambda_1}, s_2\}$ is an attack where the adversary can request λ_1 bits of leakage on s_1, and then sees s_2 in its entirety. Finally, when the leakage bound on all k sources is identical we use a "global" leakage bound λ and denote this by $\mathcal{A}^{\lambda}(x)\{s_1, \ldots, s_k\}$.

Concurrent Multi-Source Leakage. A *concurrent* leakage attack on multiple sources is one in which the adversary can interleave the leakages from each of the sources arbitrarily. Each leakage is still a function of a single source though. We allow additional flexibility by considering concurrent sources and ordered sources as above. Leakage from the ordered sources must obey the ordering, and the leakage from the concurrent sources can be arbitrarily interleaved with the leakage from the ordered sources.

Definition 3.4 (Multi-Source Leakage Attack $\mathcal{A}(x)[s_1^{\lambda_1}, \ldots, s_k^{\lambda_k}]\{r_1^{\lambda'_1}, \ldots, r_m^{\lambda'_m}\}$)**.** Let $s_1, \ldots, s_k$ and $r_1, \ldots, r_m$ be $k+m$ leakage sources (algorithms or data strings, as in Definition 3.2). We model a *concurrent* multi-source leakage attack on $[s_1, \ldots, s_k]\{r_1, \ldots, r_m\}$ as follows. The adversary runs $k+m$ leakage attacks, one on each source. The attacks on each source, s_i or r_j, for a λ_i or λ'_j-bit leakage attack as in Definition 3.2. We emphasize that each λ-bit attack on a single source consists of λ adaptive choices of 1-bit leakage functions. Between different sources, the leakages can be interleaved arbitrarily and adaptively, except for each j and j' such that $j < j'$, no leakage from r_j can occur after any leakage from $r_{j'}$. There are no restrictions on the interleaving of leakages from s_i sources.

It is important that each leakage function is computed as a function of a single sub-computation in a single source (i.e. the leakages are never a function of the internal state of multiple sources). It is also important that the attacks launched by the adversary are concurrent and adaptive, and their interleaving is controlled by the adversary. For example, $\mathcal{A}$ can request a leakage function from a sub-computation of source s_i before deciding which source to attack next, then after attacking several other sources, it can go back to source i and request a new adaptively chosen leakage attack on its next sub-computation.

As in Definition 3.3, we drop the superscript if a source s exposed in its entirety.[5] When the leakage from all sources is of the same length λ, we append the superscript to the adversary and drop it from the sources. If there are no ordered sources then we drop the curly braces.

3.2 Extractors, Entropy, and Leakage-Resilient Subspaces

In this section we define notions of min-entropy and two-source extractors that will be used in this work. We will then present the inner-product two-source extractor. Finally, we will state two lemmas that will be used in our proof of security: a lemma of [DRS04] about the connection between leakage and min-entropy, and a lemma of Brakerski *et al.* regarding leakage-resilient subspaces.

Definition 3.5 (Min-Entropy)**.** For a distribution D over a domain X, its min-entropy is:

$$H_\infty(D) \triangleq \min_{x \in X} \log \Pr_{y \sim D}[y = x]$$

Definition 3.6 ((n, m, k, ε)-two source strong extractor)**.** A function $Ext : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^m$ is a (n, m, k, ε)-2-source extractor is for every two distributions X and Y over $\{0,1\}^n$ such that $H_\infty(X), H_\infty(X) \geq k$ it is the case that:

$$\Pr_{y \sim Y}[\Delta(Ext(X, y), U_m) > \varepsilon] < \varepsilon$$

$$\Pr_{x \sim X}[\Delta(Ext(x, Y), U_m) > \varepsilon] < \varepsilon$$

Chor and Goldreich [CG88] showed that the inner-product function over any field is a two-source extractor. See also the excellent exposition of Rao [Rao07]. The claims made in those works imply the lemma below (they make more general statements).

[5] we use this only for the ordered sources, concurrent sources exposed in their entirety are w.l.o.g. given to the adversary as part of its input.

Lemma 3.7 (Inner-Product Extractor [CG88])**.** *For $\kappa \in \mathbb{N}$ and $\vec{x}, \vec{y} \in \mathbb{GF}[2]^\kappa$ define*

$$Ext(\vec{x}, \vec{y}) = \langle \vec{x}, \vec{y} \rangle$$

For any $\kappa \in \mathbb{N}$, the function $Ext(x, y)$ is a $(\kappa, 1, 0.51\kappa, 2^{-\Omega(\kappa)})$-two source strong extractor.

Finally, we will use the fact that bounded-length multi-source (or rather two-source) leakage attacks on high-entropy sources X and Y, leave an adversary with a view that is statistically close to one in which each of the sources comes from a high-entropy distribution. This follows from a result of Dodis *et al.* [DRS04].

Lemma 3.8 (Residual Entropy after Leakage [DRS04])**.** *Let X and Y be two sources with min-entropy at least k. Then for any leakage adversary $\mathcal{A}$, taking $w = \mathcal{A}^\lambda[X, Y]$, consider the conditional distributions $X' = (X|w)$ and $Y' = (Y|w)$, which are just X and Y conditioned on leakage w. For any $\delta > 0$, with probability at least $1 - \delta$ over the choice of w, $H_\infty(X'), H_\infty(Y') \geq k - \lambda - \log(1/\delta)$.*

3.3 Independence up to Orthogonality

Definition 3.9 (Independent up to Orthogonality (IuO) Distribution on Vectors)**.** Let $\mathcal{D}$ be a distribution over pairs $(\vec{x}, \vec{y}) \in \{0,1\}^\kappa \times \{0,1\}^\kappa$. We say that $\mathcal{D}$ is IuO w.r.t. $\vec{v} \in \{0,1\}^\kappa$ and $b \in \{0,1\}$, if there exist distributions $\mathcal{X}$ and $\mathcal{Y}$, both over $\{0,1\}^\kappa$, s.t. $\mathcal{D}$ is obtained by sampling $\vec{x} \sim \mathcal{X}$ and then sampling $\vec{y} \sim \mathcal{Y}$, conditioned on $\langle \vec{x} + \vec{v}, \vec{y} \rangle = b$. We call $\mathcal{X}$ and $\mathcal{Y}$ the *underlying distributions of* $\mathcal{D}$, and denote this by $\mathcal{D} = \mathcal{X} \perp_{(\vec{v},b)} \mathcal{Y}$.

When $\vec{v} = \vec{0}$ we will sometimes simply say that $\mathcal{D}$ is IuO with orthogonality b, and denote this by $\mathcal{D} = \mathcal{X} \perp_b \mathcal{Y}$.

We also consider the *independently drawn variant* of $\mathcal{D}$ which is obtained by independently sampling $\vec{x} \sim \mathcal{X}$ and $\vec{y} \sim Y$. We denote the independently drawn variant by $\mathcal{D}^\times$ or $\mathcal{X} \times \mathcal{Y}$.

Definition 3.10 (Independent up to Orthogonality (IuO) Distribution on Matrices)**.** Generalizing Definition 3.10, for an integer $m \geq 1$, let $\mathcal{D}$ be a distribution over pairs $(X, Y) \in \{0,1\}^{m \times \kappa} \times \{0,1\}^{m \times \kappa}$. We say that $\mathcal{D}$ is IuO w.r.t. $V \in \{0,1\}^{m \times \kappa}$ and $\vec{b} \in \{0,1\}^m$ if there exist distributions $\mathcal{X}$ and $\mathcal{Y}$, both over $\{0,1\}^{m \times \kappa}$, s.t. $\mathcal{D}$ is obtained by sampling $X \sim \mathcal{X}$ and then (independently) sampling $Y \sim \mathcal{Y}$ conditioned on $\forall i \in [m], \langle X[i] + V[i], Y[i] \rangle = \vec{b}[i]$. As in Definition 3.10, we call $\mathcal{X}$ and $\mathcal{Y}$ the *underlying distributions of* $\mathcal{D}$, and denote this by $\mathcal{D} = \mathcal{X} \perp_{(V,\vec{b})} \mathcal{Y}$.

When V is the all-zeros matrix, we will sometimes simply say that $\mathcal{D}$ is IuO with orthogonality $\vec{b}$, and denote this by $\mathcal{D} = \mathcal{X} \perp_{\vec{b}} \mathcal{Y}$.

We also consider the *independently drawn variant* of $\mathcal{D}$ which is obtained by independently sampling $X \sim \mathcal{X}$ and $Y \sim Y$. We denote the independently drawn variant by $\mathcal{D}^\times$ or $\mathcal{X} \times \mathcal{Y}$.

Finally, for a distribution $\mathcal{D}$ over pairs $(\vec{x}, Y) \in \{0,1\}^\kappa \times \{0,1\}^{m\kappa}$, we say that $\mathcal{D}$ is IuO (with parameters as above), if $\mathcal{D}'$, in which we replace $\vec{x}$ with a matrix X whose columns are m (identical) copies of $\vec{x}$ is IuO (as above). We emphasize that the copies of $\vec{x}$ are all identical and completely dependant.

One important property of IuO distributions, which we will use repeatedly, is that they are indistinguishable from their independently drawn variant under multi-source leakage (as long as they have sufficient entropy).

Lemma 3.11. *Let $\mathcal{D}$ be an IuO distribution over pairs $(X,Y) \in S_X \times S_Y$, with underlying distributions $\mathcal{X}$ and $\mathcal{Y}$. Suppose that $S_X = \{0,1\}^{m_X \cdot \kappa}$ and $S_Y = \{0,1\}^{m_Y \cdot \kappa}$ for m_X and m_Y s.t. $1 \leq m_X \leq m_Y \leq 10$. Suppose also that $H_\infty(\mathcal{D}) \geq (m_X + m_Y - 0.3) \cdot \kappa$. Then for any (computationally unbounded) multi-source leakage adversary $\mathcal{A}$, and leakage bound $\lambda \leq 0.1\kappa$, taking the following two distributions:*

$$\begin{aligned} \mathit{Real} &= \left(\mathcal{A}^\lambda[X,Y]\right)_{(X,Y)\sim\mathcal{D}} \\ \mathit{Simulated} &= \left(\mathcal{A}^\lambda[X,Y]\right)_{(X,Y)\sim\mathcal{D}^\times} \end{aligned}$$

it is the case that $\Delta(\mathit{Real}, \mathit{Simulated}) = \exp(-\Omega(\kappa))$.

Moreover, for any w in the support of Real: (i) we can derive from $\mathcal{X}$ a conditional underlying distribution $\mathcal{X}(w)$, and from $\mathcal{Y}$ a conditional underlying distribution $\mathcal{Y}(w)$. In particular, note that $\mathcal{D}$ is not needed for computing these conditional underlying distributions. Taking $\mathcal{D}(w) = (\mathcal{D}|w)$ to be the conditional distribution of $\mathcal{D}$, given leakage w, then $\mathcal{D}(w)$ is IuO, with underlying distributions $\mathcal{X}(w)$ and $\mathcal{Y}(w)$.

Before proving the lemma, we consider a simple application to multi-source leakage from two strings. In *Real* the strings are uniformly random with inner product 0, and in *Simulated* they are independently uniformly random. By Lemma 3.11, the leakage in both cases is statistically close. The distribution of the strings in *Real*, given the leakage, is IuO, and each of its underlying distributions can be computed (separately) given the leakage (and that the original underlying distribution were uniformly random).

Proof of Lemma 3.11. Take $w = \mathcal{A}^\lambda[X,Y]$. Since the leakage operates separately on X and on Y, there exist two sets $S_X(w) \subseteq S_X$ and $S_Y(w) \subseteq S_Y$, s.t.:

$$w = \mathcal{A}^\lambda[X,Y] \Leftrightarrow (X,Y) \in S_X(w) \times S_Y(w)$$

We take $\mathcal{X}(w)$ to be $\mathcal{X}$ conditioned on $X \in S_X(w)$, and $\mathcal{Y}(w)$ to be $\mathcal{Y}$ conditioned on $Y \in S_Y(w)$. Let $\mathcal{D}(w) = (\mathcal{D}|w)$ be the distribution $\mathcal{D}$ conditioned on leakage w. By the above, $\mathcal{D}(w)$ is $\mathcal{D}$ conditioned on $(X,Y) \in S_X(w) \times S_Y(w)$. Thus, $\mathcal{D}(w)$ is also IuO, with underlying distributions $\mathcal{X}(w)$ and $\mathcal{Y}(w)$ and the same orthogonality as $\mathcal{D}$.

Finally, to show that *Real* and *Simulated* are statistically close, let $\beta(w)$ denote the distance of the inner product $\langle X+V, Y\rangle_{X\sim\mathcal{X}(w), Y\sim\mathcal{Y}(w)}$ from uniform.

Claim 3.12. *For any $w \in \mathit{Support}(\mathit{Real})$:*

$$1 - O(\beta(w)) \leq \frac{\mathit{Simulated}[w]}{\mathit{Real}[w]} \leq 1 + O(\beta(w))$$

Proof. Observe that:

$$\begin{aligned} \mathit{Simulated}[w] &= \Pr_{X\sim\mathcal{X}, Y\sim\mathcal{Y}}[(X,Y) \in S_X(w) \times S_Y(w)] \\ \mathit{Real}[w] &= \Pr_{(X,Y)\sim\mathcal{D}}[(X,Y) \in S_X(w) \times S_Y(w)] \\ &= \Pr_{X\sim\mathcal{X}, Y\sim\mathcal{Y}'(X)}[(X,Y) \in S_X(w) \times S_Y(w)] \end{aligned}$$

where $\mathcal{Y}'(X)$ is $\mathcal{Y}$ conditioned on $\langle X+V, Y\rangle = \vec{b}$.

The claim follows because:

$$1 - O(\beta(w)\cdot 2^{m_Y}) \leq \frac{\Pr_{X\sim\mathcal{X},Y\sim\mathcal{Y}}[\langle X+V,Y\rangle = \vec{b}]}{\Pr_{X\sim\mathcal{X},Y\sim\mathcal{Y}}[\langle X+V,Y\rangle = \vec{b}|(X,Y)\in S_X(w)\times S_Y(w)]} \leq 1 + O(\beta(w)\cdot 2^{m_Y})$$

■

Claim 3.13. *With all but* $\exp(-\Omega(\kappa))$ *probability over* $w \sim$ *Real,* $\beta(w) = \exp(-\Omega(\kappa))$.

Proof. By Lemma 3.8, with all but δ probability over $w \sim$ *Real*, we have that $H_\infty(\mathcal{X}(w)) + H_\infty(\mathcal{Y}(w)) \geq (m_X + m_Y - 0.45)\cdot\kappa$. When this is the case, by Lemma 3.7 we have $\beta(w) = \exp(-\Omega(\kappa))$. ■

By Claim 3.12 and 3.13 we conclude that $\Delta(\textit{Real}, \textit{Simulated}) = \exp(-\Omega(\kappa))$. ■

3.4 Secure Compiler: Definitions

We now present formal definitions for a secure compiler against continuous and computationally unbounded leakage. We view the input to the compiler as a circuit C that is known to all parties and takes inputs x and y. The input y is fixed, whereas the input x is chosen by the user. The user can adaptively choose inputs $x_1, x_2, \ldots$ and the functionality requirement is that on each input x_i the user receives $C(y, x_i)$. The secrecy requirement is that even for a computationally unbounded adversary who chooses the inputs (say polynomially many inputs in the security parameter), even giving the adversary access(repeatedly) to a leakage attack on the secure transformed computation, the adversary learns nothing more than the circuit's outputs. In particular, the adversary should not learn y.[6]

We divide a compiler into parts: the first part, the *initialization* occurs only once at the beginning of time. This procedure depends only on the circuit C being compiled and the private input y. We assume that during this phase there is no leakage. The second part is the *evaluation*. This occurs whenever the user wants to evaluate the circuit $C(y,\cdot)$ on an input x. In this part the user specifies an input x, the corresponding output $C(y,x)$ is computed under leakage.

Definition 3.14 ($(\lambda(\cdot),\delta(\kappa))$ Continuous Leakage Secure Compiler)**.** We say that a compiler $(\textit{Init}, \textit{Eval})$ for a circuit family $\{C_n(y,x)\}_{n\in\mathbb{N}}$, where C_n operates on two n-bit inputs, is $(\lambda(\cdot),\delta(\kappa))$-*secure under continuous leakage*, if for every integer $n, \kappa \in \mathbb{N}$, and every $y \in \{0,1\}^n$, the following hold:

- Initialization: $\textit{Init}(1^\kappa, C_n, y)$ runs in time $\text{poly}(\kappa, n)$ and outputs an initial state $\textit{state}_0$
- Evaluation: for every integer $t \leq \text{poly}(\kappa)$, the evaluation procedure is run on the previous state $\textit{state}_{t-1}$ and an input $x_t \in \{0,1\}^n$. We require that for every $x_t \in \{0,1\}^n$, when we run:

  $$(\textit{out}_t, \textit{state}_t) \leftarrow \textit{Eval}(\textit{state}_{t-1}, x_t)$$

  with all but negligible probability over the coins of *Init* and the t invocations of *Eval*, $\textit{out}_t = C_n(y, x_t)$.

[6] Unless, of course, y can be computed from the outputs of the circuit on the inputs the adversary chose.

- $(\lambda(\kappa), \delta(\kappa))$-Continuous Leakage Security: There exists a simulator Sim, s.t. for every (computationally unbounded) leakage adversary $\mathcal{A}$, the view $Real_{\mathcal{A}}$ of $\mathcal{A}$ when adaptively choosing $T = \text{poly}(\kappa)$ inputs $(x_1, x_2, \ldots x_T)$ while running a continuous leakage attack on the sequence $(Eval(state_0, x_1), \ldots, Eval(state_{T-1}, x_T))$, with adaptively and adversarially chosen x_t's, is $(\delta(\kappa))$-statistically close to the view $Simulated_{\mathcal{A}}$ generated by Sim, which only gets the description of the adversary and the input-output pairs $((x_1, C(y, x_1)), \ldots, (x_T, C(y, x_T)))$.

  Formally, the adversary repeatedly and adaptively, in iterations $t \leftarrow 1, \ldots, T$, chooses an input x_t and launches a $\lambda(\kappa)$-bit leakage attack on $Eval(state_{t-1}, x_t)$ (see Definition 3.2). $Real_{\mathcal{A},t}$ is the view of the adversary in iteration t, including the input x_t, the output o_t, and the (aggregated) leakage w_t from the t-th iteration. The complete view of the adversary is

  $$Real_{\mathcal{A}} = (Real_{\mathcal{A},1}, \ldots, Real_{\mathcal{A},T})$$

  a random variable over the coins of the adversary, of $Init$ and of $Eval$ (in all of its iterations).

  The simulator's view is generated by running the adversary with *simulated* leakage attacks. The simulator includes $SimInit$ and $SimEval$ procedures. The initial state is generated using $SimInit$. Then, in each iteration t the simulator gets the input x_t chosen by the adversary and the circuit output $C(y, x_t)$. It generates simulated leakage w_t. It is important that the simulator sees nothing of the internal workings of the evaluation procedure. We compute:

  $$state_0 \leftarrow SimInit(1^{\kappa}, C_n)$$

  $$x_t \leftarrow \mathcal{A}(Simulated_{\mathcal{A},1}, \ldots, Simulated_{\mathcal{A},t-1})$$

  $$(state_t, Simulated_{\mathcal{A},t}) \leftarrow SimEval(state_{t-1}, x_t, , C(y, x_t), \mathcal{A}, Simulated_{\mathcal{A},1}, \ldots, Simulated_{\mathcal{A},t-1})$$

  where $Sim_{\mathcal{A},t}$ is a random variable over the coins of the adversary when choosing the next input and of the simulator. The complete view of the simulator is

  $$Simulated_{\mathcal{A}} = (Simulated_{\mathcal{A},1}, \ldots, Simulated_{\mathcal{A},T})$$

  We require that the two views $Real_{\mathcal{A}}$ and $Simulated_{\mathcal{A}}$ are $(\exp(-\Omega(\kappa)))$-statistically close.

We note that modeling the leakage attacks requires dividing the $Eval$ procedure into sub-computations. In our constructions, the size of these sub-computations will always be $O(\kappa^{\omega})$, where ω is the exponent in the running time of an algorithm for matrix multiplication.

4 Leakage-Resilient One-Time Pad (LROTP)

In this section we present the *leakage resilient one-time pad* cryptoscheme, a main component of our construction. See the overview in Section 2.1. Here we specify the scheme and its properties that will be used in the main construction.

Leakage-Resilient One-Time Pad (LROTP) Cryptosystem $(KeyGen, Encrypt, Decrypt)$

- $KeyGen(1^\kappa)$: output a uniformly random $key \in \{0,1\}^\kappa$ s.t. $key[0] = 1$
- $CipherGen(1^\kappa)$: output a uniformly random $\vec{c} \in \{0,1\}^\kappa$ s.t. $\vec{c}[1] = 1$.
- $Encrypt(key, b \in \{0,1\})$: output a uniformly random $\vec{c} \in \{0,1\}^\kappa$ s.t. $\vec{c}[1] = 1$ and $\langle key, \vec{c} \rangle = b$
- $Decrypt(key, \vec{c})$: output $\langle key, \vec{c} \rangle$

Figure 1: Leakage-Resilient One-Time Pad (LROTP) Cryptosystem

4.1 Semantic Security under Multi-Source Leakage

Definition 4.1 (Semantic Security Under $\lambda(\cdot)$-Multi-Source Leakage)**.** An encryption scheme $(KeyGen, Encrypt, Decrypt)$ is semantically secure under computationally unbounded multi-source leakage attacks if for every (unbounded) adversary $\mathcal{A}$, when we run the game below, the adversary's advantage in winning (over 1/2) is $\exp(-\Omega(\kappa))$:

1. The game chooses key $key \leftarrow KeyGen(1^\kappa)$, chooses uniformly at random a bit $b \in_R \{0,1\}$, and generates a ciphertext $\vec{c} \leftarrow Encrypt(key, b)$.
2. The adversary launches a leakage attack on key and $\vec{c}$, and outputs a "guess" b':

$$b' \leftarrow \mathcal{A}^{\lambda(\kappa)}(1^\kappa)[key, \vec{c}]$$

the adversary wins if $b' = b$.

Lemma 4.2. *The LROTP cryptosystem, as defined in Figure 1, is semantically secure in the presence of multi-source leakage with leakage bound $\lambda(\kappa) = \kappa/3$.*

Proof. The proof follows directly from Lemma3.11. ■

4.2 Key and Ciphertext Refreshing

As discussed in the introduction, the LROTP scheme supports procedures for injecting new entropy into a key or a ciphertext. This is done using *entropy generators* $KeyEntGen$ and $CipherEntGen$. The values these procedures produce can be used to refresh a key or ciphertext using $KeyRefresh$ or $CipherRefresh$ (respectively). Key entropy σ can also be used, *without knowledge of key*, to correlate a ciphertext $\vec{c}$ so that the plaintext underlying the correlated ciphertext $\vec{c}'$ under $key' \leftarrow KeyRefresh(key, \sigma)$, is equal to the plaintext underlying $\vec{c}$ under key. This is done using the $CipherCorrelate$ procedure.A similar $KeyCorrelate$ procedure for correlating keys using ciphertext entropy. These procedures are all in Figure 2 below.

We proceed with a discussion of the security properties of the refreshing procedures, and their limitation. For a key-ciphertext pair $(key, \vec{c})$, a *refresh operation* on the pair injects new entropy into the key and the ciphertext, while maintaining the underlying plaintext, as follows:

1. $\sigma \leftarrow KeyEntGen(1^\kappa)$
2. $key' \leftarrow KeyRefresh(key, \sigma)$

LROTP key and ciphertext refresh

- $KeyEntGen(1^\kappa)$: output a uniformly random $\sigma \in \{0,1\}^\kappa$ s.t. $\sigma[0] = 0$
- $KeyRefresh(key, \sigma)$: output $key \oplus \sigma$
- $CipherCorrelate(\vec{c}, \sigma)$: modify $\vec{c}[0] \leftarrow \vec{c}[0] \oplus \langle \vec{c}, \sigma \rangle$, and then output $\vec{c}$
- $CipherEntGen(1^\kappa)$: output a uniformly random $\tau \in \{0,1\}^\kappa$ s.t. $\tau[1] = 0$
- $CipherRefresh(\vec{c}, \tau)$: output $\vec{c} \oplus \tau$
- $KeyCorrelate(key, \tau)$: modify $key[1] \leftarrow key[1] \oplus \langle key, \tau \rangle$, and then output key

Figure 2: LROTP key and ciphertext refresh Cryptosystem

3. $\vec{c}' \leftarrow CipherCorrelate(\vec{c}, \sigma)$
4. $\pi \leftarrow CipherEntGen(1^\kappa)$
5. $\vec{c}'' \leftarrow CipherRefresh(\vec{c}', \pi)$
6. $key'' \leftarrow KeyCorrelate(key', \pi)$

The output of the refresh operation is $(key'', \vec{c}'')$. We treat each step of the key-refresh as a sub-computation, and so the leakage operates separately on the keys and on the ciphertexts.

Security Properties. The security properties of the refreshing procedures are, first, that a key-ciphertext pair can be refreshed without ever loading the key and ciphertext into memory at the same time, i.e. while operating separately on the key and on the ciphertext. We will use this to argue that an OC leakage adversary learns nothing about the plaintext bit underlying a pair that is being refreshed (as long as the total amount of leakage is bounded). The second property we use is that *without any leakage*, a the refreshed pair is a uniformly random key-ciphertext pair with the same underlying plaintext bit.

We use these properties to prove security of the *Permute* procedure which is used in *SafeNAND* (see Sections 2.2.1 and 6.2). *Permute* proceeds in iterations. In each iteration, we refresh a tuple of key-ciphertext pairs and then permute them using a random permutation. The property of the refresh procedure that we will use is that *without any leakage*, even given both the input and the output of a single iteration of *Permute*, *nothing is leaked about the permutation chosen* (beyond what can be gleaned from the underlying plaintexts). This will then be used to argue that, even under a bounded amount of leakage from each iteration, the permutation chosen in each iteration of *Permute* has (w.h.p.) high entropy. This is later used to prove the security of *SafeNAND*.

Refresh Forever? It is natural to ask whether key-ciphertext refreshing maintains security of the underlying plaintext under OC leakage for an unbounded polynomial number of refreshings. If so, we could hope to do away with the (significantly more complicated) ciphertext banks, replacing the ciphertexts generated by each bank with a sequence of ciphertexts generated using repeated refresh calls. Unfortunately, there is an OC attack that exposes the plaintext underlying a key-ciphertext pair that is refreshed too many times. The attack is outlined below.

We consider a sequence of refresh operations, where the output of the i-th refresh is used as input for the $(i+1)$-th refresh. During the first refresh, an OC adversary leaks the inner product (i.e. the product) of the first bit of the output key and the first bit of the output ciphertext. This requires only one bit of leakage from each. In the second refresh, the adversary will learn the inner product of the first *two* bits of the output key and the output ciphertext. To do so, let $(key_1, \vec{c}_1)$ be the inputs to the second refresh. The adversary leaks the second bits of key_2 during *KeyRefresh*, and of $\vec{c}_2$ during *CipherRefresh*. It also keeps track of the *change* in inner product of the *first* bit of $key'_1 = (key_1 + \sigma)$ and of $\vec{c}'_1 = \mathit{CipherCorrelate}(\vec{c}_1, \sigma)$ using a single bit of leakage: The change (w.r.t. the inner product of key_1 and $\vec{c}_1$) is just a function of σ and $\vec{c}_1$, which are loaded into memory during *CipherCorrelate*. Similarly, the adversary can keep track of the subsequent change to the inner product of the first bits of $key_2 = \mathit{KeyCorrelate}(key'_1, \pi)$ and $\vec{c}_2 = \vec{c}'_1 \oplus \pi$, using a single bit of leakage from *KeyCorrelate*. Putting these pieces together, the adversary learns the inner product of the first two bits of key_2 and $\vec{c}_2$. More generally, after the i-th refresh call, the key point is that if the adversary knows the inner product of the first i bits of the input key and ciphertext, it can track the change in this inner product for the output key and cipher. Tracking the change requires only two bits of OC leakage. The adversary uses two additional bits of OC leakage to expand its knowledge to the inner product of the first $(i+1)$ bits.

Continuing the above attack for κ refresh calls, the adversary learns the inner product of the key and ciphertext obtained, i.e. the underlying plaintext is exposed. Note that this used only $O(1)$ bits of leakage from each sub-computation. If ℓ bits of leakage from each sub-computation were allowed, then the underlying plaintext would be exposed after $O(\kappa/\ell)$ refresh calls. When using refresh, we will take care that the total leakage accumulated from a sequence of refresh calls to a key-ciphertext pair will be well under κ bits. Since refresh operates separately on keys and ciphertexts, the semantic security of LROTP in the presence of multi-source leakage will guarantee that the underlying plaintext is hidden.

4.3 "Safe" Homomorphic Computations

The LROTP cryptoscheme supports homomorphic computation on ciphertexts as follows:

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 $\vec{c}_2$.

Homomorphic NAND. LROTP supports safe computation of a masked NAND functionality. This functionality takes three input key-ciphertext pairs, and outputs the NAND of the first two underlying plaintexts, XORed with the third underlying plaintext. Moreover, this can be performed via the *SafeNAND* procedure, which guarantees that even an OC leakage attacker who gets leakage on the computation, learns nothing about the input plaintexts beyond the procedure's output. See Sections 2.2.1 and 6 for details.

We note that this can be extended to "standard" homomorphic computation of NAND, where the input is two key-ciphertext pairs, and the output is a "blinded" key-ciphertext pair whose underlying plaintext is the NAND of the plaintexts underlying the inputs. The details are omitted (this second property follows from the security of *SafeNAND*, but is not used in our construction).

5 Ciphertext Banks

In this section we present the procedures for maintaining, utilizing, and simulating banks of secure ciphertexts. We use these to create fresh secure ciphertexts under leakage attacks. The security property we want is that, even though the generation of new ciphertexts is done under leakage, a simulator can create an indistinguishable simulated view with complete and arbitrary control over these ciphertexts' underlying plaintexts. See Section 2.3.1 for an overview.

This section is organized as follows. In Section 5.1 we describe the ciphertext bank procedures, and those of the simulator, and state the security properties that will be used in the main construction (the profs follow in subsequent sections). These procedures make use of secure procedures for piecemeal matrix multiplication and for refreshing collections of ciphertexts, which are in section Section 5.2. In Section 5.3 we define piecemeal attacks on matrices and prove that random matrices are resilient to piecemeal leakage. In Section 5.4 we state and prove security properties of piecemeal matrix multiplication. Finally, we use these claims to prove the ciphertext bank's security. We conclude with proofs of the ciphertext bank's security in Section 5.5.

5.1 Ciphertext Bank: Interface and Security

We present a full description of the ciphertext bank procedures and simulator. Recall that (as in Section 4), keys and ciphertexts are vectors in $\{0,1\}^\kappa$, and the decryption of ciphertext $\vec{c}$ under key is the inner product $b = \langle key, \vec{c} \rangle$. We call b the plaintext underlying ciphertext $\vec{c}$.

Ciphertext Bank Procedures. The ciphertext bank is used to generate fresh ciphertext-key pairs. The bank is initialized (without leakage) using a *BankInit* procedure that takes as input a bit $b \in \{0,1\}$. It can then be accessed (repeatedly) using a *BankGen* procedure, which produces a key-ciphertext pair whose underlying plaintext is b. Between generations, the bank's internal state is updated using a *BankUpdate* Procedure. Leakage from a sequence of *BankGen* and *BankUpdate* calls can be simulated. The simulator has arbitrary control over the plaintext bits underlying the generated ciphertexts. Simulated leakage is statistically close to leakage from the real calls.

In addition, we provide a *BankRedraw* procedure.The *BankRedraw* procedure re-draws a uniformly random plaintext bit that will underly ciphertexts produced by the bank. The redrawn plaintext bit looks uniformly random even in the presence of leakage on the *BankRedraw* procedure (and on all ciphertext generations).

These functionalities are implemented as follows. The ciphertext bank consists of 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, the keys is drawn uniformly at random, and the columns of C are drawn uniformly at random s.t their inner product with key is b. This invariant will be maintained throughout the ciphertext bank's operation. We sometimes refer to b as the ciphertext bank's *underlying plaintext bit*.

The *BankGen* procedure outputs 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. The linear combination is taken using a secure "piecemeal" matrix-vector multiplication procedure *PiecemealMM*.

The *BankUpdate* procedure injects new entropy into key and into C. We refresh the key using a ("piecemeal") key refresh procedure *PiecemealRefresh*. We refresh C by multiplying it with a

random matrix whose columns all have parity 1. Matrix multiplication is again performed securely using *PiecemealMM*.

The *BankRedraw* procedure adds a uniformly random vector in $\{0,1\}^\kappa$ to each column of C (here *key* is left unchanged). With probability 1/2, the vector has inner product 1 with *key*, and the underlying plaintext bit is flipped. Otherwise, the underlying plaintext bit is unchanged. Adding the vector to each column of the matrix is performed using a secure *PiecemealAdd* procedure.

The full ciphertext bank procedures are in Figure 3. The piecemeal matrix multiplication, addition, and key refresh procedures are below in Section 5.2.

BankInit$(1^\kappa, b)$: initializes a ciphertext bank; No leakage

1. pick $key \leftarrow KeyGen(1^\kappa)$
2. for $i \leftarrow 1, \ldots 2\kappa$: $C[i] \leftarrow Encrypt(key, b)$
3. output $Bank \leftarrow (key, C)$

BankGen$(Bank)$**:** generates a new ciphertext; Under leakage

1. pick $\vec{r} \in_R \{0,1\}^{2\kappa}$ with parity 1
2. $\vec{c} \leftarrow PiecemealMM(C, \vec{r})$
3. output $(key, \vec{c})$

BankUpdate$(Bank)$**:** updates the bank between generations; Under leakage

1. refresh the key:
   $(key', D) \leftarrow PiecemealRefresh(key, C)$
2. refresh the ciphertexts:
   pick $R \in_R \{0,1\}^{2\kappa \times 2\kappa}$ s.t. its columns all have parity 1,
   $C' \leftarrow PiecemealMM(D, R)$
3. $Bank \leftarrow (key', C')$

BankRedraw$(Bank)$**:** re-draws the bank's underlying plaintext bit; Under leakage

1. pick $\vec{v} \in_R \{0,1\}^\kappa$, compute $C' \leftarrow PiecemealAdd(C, \vec{v})$
2. $Bank \leftarrow (key, C')$

Figure 3: Ciphertext Bank

Simulated Ciphertext Bank. Next, we provide a simulator for simulating the ciphertext bank procedure, while arbitrarily controlling the plaintext bits underlying the ciphertexts that are produced. Towards this end, we maintain a simulated ciphertext bank, consisting of a key and a matrix, similarly to the real ciphertext bank. These are initialized, without leakage, in 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 independent and uni-

formly random bits (rather than all 0 or all 1). The simulator also keeps track of the plaintexts bits underlying the columns of C, storing them in a vector $\vec{x} \in \{0,1\}^{2\kappa}$.

Calls to *BankGen* are simulated using *SimBankGen*. This procedure operates similarly to *BankGen*, except that it uses a biased linear combination of C's columns to control the plaintext underlying its output ciphertext. We also provide *SimBankUpdate* and *SimBankRedraw* procedures. These operate similarly to *BankUpdate* and *BankRedraw*, except that they keep track of changes to the vector $\vec{x}$ of plaintext bits underlying C. The simulation procedures are in Figure 4.

SimBankInit(1^κ); No leakage

1. pick $key \leftarrow KeyGen(1^\kappa), \vec{x} \in_R \{0,1\}^{2\kappa}$
2. for $i \leftarrow 1, \ldots m$: $C[i] \leftarrow Encrypt(key, \vec{x}[i])$
3. output $Bank \leftarrow (key, C)$; Save also $\vec{x}$

SimBankGen$(Bank, b)$

1. pick $\vec{r} \in_R \{0,1\}^{2\kappa}$ with parity 1, and s.t. $\langle \vec{x}, \vec{r} \rangle = b$
2. run exactly as in *BankGen*, except in Step 1 use the above "biased" $\vec{r}$
   leakage is (only) on this operation of BankGen (with the biased $\vec{r}$)

SimBankUpdate$(Bank)$

1. run exactly as in *BankUpdate*
   leakage is (only) on this operation of BankUpdate
2. update $\vec{x}$ to contain the new bits underlying the updated C

SimBankRedraw$(Bank)$

1. run exactly as in *BankRedraw*
   leakage is (only) on this operation of BankRedraw
2. update $\vec{x}$ to contain the new bits underlying the updated C

Figure 4: Simulated Ciphertext Bank

Ciphertext Bank Security. We show several security properties of the ciphertext bank. In all of these security properties, we consider sequences of ciphertext bank generations, real or simulated. A *sequence of real generations* starts with a call to *BankInit* to initialize the ciphertext bank. This is followed by a sequence of ciphertext generations, each performed via a call to *BankGen*, and followed by an update call to *BankUpdate*. A *sequence of simulated generations* is similar, except that initialization is performed using *SimBankInit*, each generation is performed by specifying an underlying plaintext bit b and then calling *SimBankGen*, and each update is performed using *SimBankUpdate*.

We also consider sequences of random generations of ciphertext-pairs. A *sequence of real random generations* begins with an initialization call to *BankInit* with a uniformly random bit value.

This is followed by a sequence of generations as follows. For each item in the sequence, we begin by generating a key and *two* ciphertexts, $\vec{c}^{\alpha}$ and $\vec{c}^{\beta}$ (both with the same underlying plaintext bit). Next, we call *BankRedraw* to redraw the bank's underlying plaintext bit. Lastly, we update the bank using *BankUpdate*. This is done repeatedly, yielding a sequence of keys and pairs of ciphertexts, where the plaintext bit underlying each ciphertext pair is independent and uniformly random. A *sequence of simulated random generations* is performed similarly, except that *BankInit*, *BankRedraw*, *BankUpdate* are replaced by *SimBankInit*, *SimBankRedraw*, *SimBankUpdate*, and each pair of calls to *BankGen* is replaced by a pair of calls to *SimBankGen* with some specified plaintext bit b (we will always use the same plaintext bit b in both generations).

We now describe several security properties for sequences of real and simulated generations and random generations of pairs. Intuitive description are listed below, and the formal lemma statements follow.

Real and simulated sequences, identical underlying plaintexts. Consider an OC leakage attacker's "real" view, given leakage from a real sequence of generations using a bank initialized with bit b. Consider also a "simulated" view for the same attacker, given leakage from a simulated sequence of calls, where all calls to *SimBankGen* specify the same underlying plaintext bit b. I.e., the plaintexts underlying the ciphertexts generated in these real and simulated views are all identical. We show that the distributions of the leakage obtained in these two views, *in conjunction with the explicit list of key-ciphertext pairs produced*, are statistically close.

This is stated formally in Lemma 5.1 below. The proof is in Section 5.5.

Two simulated sequences, different underlying plaintexts. Consider an OC leakage attacker's view, given *two simulated* sequences of generations. The two sequences each produce the same number of ciphertexts, but *differ in the underlying plaintext bits that are specified*.

We show that the distributions of leakage obtained in these two views are statistically close. Note that, unlike the previous property, here statistical closeness does not hold in conjunction with the explicit keys and ciphertexts produced (since the underlying plaintexts differ). We note also that, combining this with the previous property, we conclude statistical closeness of leakage distributions produced by an OC attack on a real sequence and on a simulated sequence with *different underlying plaintexts* (leakage only - without the explicit plaintext and ciphertext produced).

This is stated formally in Lemma 5.2 below. The proof is in Section 5.5.

Single simulated sequence, independence up to orthogonality. Consider an OC leakage attack on a (single) sequence of *simulated* generations. We show that, given the leakage in the attack, the (joint) distribution of keys and ciphertexts produced, is *independent up to orthogonality* (see Definition 3.10). Moreover, the underlying distributions on keys and ciphertexts depend only on the leakage (and the adversary), but not on the sequence of bits given as input to the simulated generations. Finally, these underlying (conditional) distributions have high entropy on each key and each ciphertext produced.

Intuitively, this means that the keys and ciphertexts produced will be resilient to subsequent multi-source leakage. I.e., bounded leakage that operates separately on keys and on ciphertexts will not be able to distinguish the underlying plaintexts. We note that independence up to orthogonality holds even given the list of ciphertexts in the bank in all generations and all randomness used by the ciphertext except the randomness for generating the "target" key and ciphertext.

This is stated formally in Lemma 5.3 below. The proof is in Section 5.5.

Real and simulated sequences *of random generations*. Consider an OC leakage attacker's "real" view, given leakage from a real sequence of random generations of ciphertext pairs. Consider also a "simulated" view for the same attacker, given leakage from a simulated sequence of calls, where each pair of calls to *SimBankGen* specify a uniformly random bit (independent of all other pairs). In particular, the plaintexts underlying the ciphertexts generated in these real and simulated views are identically distributed (uniformly random for each pair independently). We show that the distributions of the leakage obtained in these two views, *in conjunction with the explicit list of keys and ciphertext pairs produced*, are statistically close.

This is stated formally in Lemma 5.4 below. This is similar to the guarantee of Lemma 5.1 and we omit the proof.

Single simulated sequence *of random generations*, independence up to orthogonality. Consider an OC leakage attack on a (single) sequence of *simulated* random generations of pairs of ciphertexts. We show that, given the leakage in the attack, the (joint) distribution of keys and ciphertexts produced, is *independent up to orthogonality* (see Definition 3.10). Moreover, the underlying distributions on keys and ciphertexts depend only on the leakage (and the adversary), but not on the sequence of underlying plaintext bits. Finally, these underlying (conditional) distributions have high entropy on each key and each ciphertext produced.

Intuitively, this means that the keys and ciphertexts produced will be resilient to subsequent multi-source leakage. I.e., bounded leakage that operates separately on keys and on ciphertexts will not be able to distinguish the underlying plaintexts. Moreover, within each pair of ciphertexts, independence up to orthogonality for the key and each ciphertext separately continues to hold even if the other ciphertext in the pair is released in its entirety. We note that, as was the case above, independence up to orthogonality holds even given the list of ciphertexts in the bank in all generations and all randomness used by the ciphertext except the randomness for generating the "target" key and ciphertext.

This is stated formally in Lemma 5.5 below. The guarantee is quite similar to that of Lemma 5.3 and we omit the proof.

Lemma 5.1. *There exists a leakage bound $\lambda(\kappa) = \Omega(\kappa)$, and a distance bound $\delta(\kappa) = \exp(-\Omega(\kappa))$, s.t. for any bit $b \in \{0,1\}$, security parameter $\kappa \in \mathbb{N}$, execution bound $T = \mathrm{poly}(\kappa)$, and (computationally unbounded) leakage adversary $\mathcal{A}$:*

Let Real and Simulated be as follows, where in Real we begin by running $Bank_0 \leftarrow BankInit(b)$,

and in Simulated we begin by running $Bank_0 \leftarrow SimBankInit$ *(both without leakage):*

$$\begin{aligned}
Real \;=\; & \mathcal{A}\big\{((key_0, \vec{c}_0) \leftarrow BankGen(Bank_0))^{\lambda(\kappa)}, \\
& (Bank_1 \leftarrow BankUpdate(Bank_0))^{\lambda(\kappa)}, key_0, \vec{c}_0, \\
& ((key_1, \vec{c}_1) \leftarrow BankGen(Bank_1))^{\lambda(\kappa)}, \\
& (Bank_2 \leftarrow BankUpdate(Bank_1))^{\lambda(\kappa)}, key_1, \vec{c}_1, \\
& \ldots \\
& ((key_{T-1}, \vec{c}_{T-1}) \leftarrow BankGen(Bank_{T-1}))^{\lambda(\kappa)}, \\
& (Bank_T \leftarrow BankUpdate(Bank_{T-1}))^{\lambda(\kappa)}, key_{T-1}, \vec{c}_{T-1}\big\}
\end{aligned}$$

$$\begin{aligned}
Simulated \;=\; & \mathcal{A}\big\{((key_0, \vec{c}_0) \leftarrow SimBankGen(Bank_0, b))^{\lambda(\kappa)}, \\
& (Bank_1 \leftarrow SimBankUpdate(Bank_0))^{\lambda(\kappa)}, key_0, \vec{c}_0, \\
& ((key_1, \vec{c}_1) \leftarrow SimBankGen(Bank_1, b))^{\lambda(\kappa)}, \\
& (Bank_2 \leftarrow SimBankUpdate(Bank_1))^{\lambda(\kappa)}, key_1, \vec{c}_1, \\
& \ldots \\
& ((key_{T-1}, \vec{c}_{T-1}) \leftarrow SimBankGen(Bank_{T-1}, b))^{\lambda(\kappa)}, \\
& (Bank_T \leftarrow SimBankUpdate(Bank_{T-1}))^{\lambda(\kappa)}, key_{T-1}, \vec{c}_{T-1}\big\}
\end{aligned}$$

then $\Delta(Real, Simulated) = \delta(\kappa)$.

Lemma 5.2. *There exists a leakage bound* $\lambda(\kappa) = \Omega(\kappa)$*, and a distance bound* $\delta(\kappa) = \exp(-\Omega(\kappa))$*, s.t. for any security parameter* $\kappa \in \mathbb{N}$*, any execution bound* $T = \mathrm{poly}(\kappa)$*, any vectors* $\vec{b}', \vec{b}'' \in \{0,1\}^T$*, and any (computationally unbounded) leakage adversary* $\mathcal{A}$*:*

Let Simulated$'$ *and Simulated*$''$ *be the following two distributions, where in both distributions we*

begin by running $Bank_0 \leftarrow SimBankInit$ *(without leakage):*

$$\begin{aligned}
Simulated' \quad = \quad & \mathcal{A}^{\lambda(\kappa)}\big\{[(key_0, \vec{c}_0) \leftarrow SimBankGen(Bank_0, \vec{b}'[0])], \\
& [Bank_1 \leftarrow SimBankUpdate(Bank_0)], \\
& [(key_1, \vec{c}_1) \leftarrow SimBankGen(Bank_1, \vec{b}'[1])], \\
& [Bank_2 \leftarrow SimBankUpdate(Bank_1)], \\
& \ldots \\
& [(key_{T-1}, \vec{c}_{T-1}) \leftarrow SimBankGen(Bank_{T-1}, \vec{b}'[T-1])] \\
& [Bank_T \leftarrow SimBankUpdate(Bank_{T-1})]\big\} \\
Simulated'' \quad = \quad & \mathcal{A}^{\lambda(\kappa)}\big\{[(key_0, \vec{c}_0, Bank_1) \leftarrow SimBankGen(Bank_0, \vec{b}''[0])], \\
& [Bank_1 \leftarrow SimBankUpdate(Bank_0)], \\
& [(key_1, \vec{c}_1, Bank_2) \leftarrow SimBankGen(Bank_1, \vec{b}''[1])], \\
& [Bank_2 \leftarrow SimBankUpdate(Bank_1)], \\
& \ldots \\
& [(key_{T-1}, \vec{c}_{T-1}, Bank_T) \leftarrow SimBankGen(Bank_{T-1}, \vec{b}''[T-1])] \\
& [Bank_T \leftarrow SimBankUpdate(Bank_{T-1})]\big\}
\end{aligned}$$

then $\Delta(Simulated', Simulated'') = \delta(\kappa)$.

Lemma 5.3. *There exists a leakage bound* $\lambda(\kappa) = \Omega(\kappa)$*, and a probability bound* $\delta(\kappa) = \exp(-\Omega(\kappa))$*, s.t. for any* $\kappa \in \mathbb{N}$*, any execution bound* $T = \mathrm{poly}(\kappa)$*, any vector* $\vec{b} \in \{0,1\}^T$*, and any (computationally unbounded) leakage adversary* $\mathcal{A}$*, the following holds:*

Let Simulated be the following distribution, where we begin by running $Bank_0 \leftarrow SimBankInit$ *(without leakage):*

$$\begin{aligned}
Simulated \quad = \quad & \mathcal{A}^{\lambda(\kappa)}\big\{[(key_0, \vec{c}_0) \leftarrow SimBankGen(Bank_0, \vec{b}[0])], \\
& [Bank_1 \leftarrow SimBankUpdate(Bank_0)], \\
& [(key_1, \vec{c}_1) \leftarrow SimBankGen(Bank_1, \vec{b}[1])], \\
& [Bank_2 \leftarrow SimBankUpdate(Bank_1)], \\
& \ldots, \\
& [(key_{T-1}, \vec{c}_{T-1}) \leftarrow SimBankGen(Bank_{T-1}, \vec{b}[T-1])] \\
& [Bank_T \leftarrow SimBankUpdate(Bank_{T-1})]\big\}
\end{aligned}$$

For any w *in the support of Simulated, and for any* $i \in [T]$*, fixing all ciphertexts except the* i*-th pair produced, let* $D_i(w)$ *be the joint distribution of* $(key_i, \vec{c}_i)$ *given* w *and the remaining* $T-1$ *ciphertexts. There exist distributions* $\mathcal{K}_i(w)$ *and* $\mathcal{C}_i(w)$ *s.t. the following holds:*[7]

The distribution $\mathcal{D}_i(w)$ *is IuO with orthogonality* $\vec{b}[i]$ *and underlying distributions* $\mathcal{K}_i(w)$ *and* $\mathcal{C}_i(w)$*. With all but* $\delta(\kappa)$ *probability over the choice (by Simulated) of* w *and of all ciphertexts except the* i*-th, the min-entropy of* $\mathcal{K}_i(w)$ *and of* $\mathcal{C}_i(w)$ *is at least* $\kappa - O(\lambda(\kappa))$*.*

[7]Note that these distributions *do not depend on* $\vec{b}_i$ (they depend only on w, on $\mathcal{A}$ and on the $T-1$ remaining ciphertexts).

Lemma 5.4. *There exists a leakage bound $\lambda(\kappa) = \Omega(\kappa)$, and a distance bound $\delta(\kappa) = \exp(-\Omega(\kappa))$, s.t. for any security parameter $\kappa \in \mathbb{N}$, execution bound $T = \mathrm{poly}(\kappa)$, and (computationally unbounded) leakage adversary $\mathcal{A}$:*

Let Real and Simulated be as follows. Choose $\vec{b} \in_R \{0,1\}^T$. In Real, we begin by running $Bank_0 \leftarrow BankInit(\vec{b}[0])$. In Simulated we begin by running $Bank_0 \leftarrow SimBankInit$:

$$
\begin{aligned}
Real \;=\; & \mathcal{A}\big\{((key_0, \vec{c}_0^{\alpha}) \leftarrow BankGen(Bank_0))^{\lambda(\kappa)}, (key_0, \vec{c}_0^{\beta} \leftarrow BankGen(Bank_0))^{\lambda(\kappa)}, \\
& (Bank'_0 \leftarrow BankRedraw(Bank_0))^{\lambda(\kappa)}, (Bank_1 \leftarrow BankUpdate(Bank'_0))^{\lambda(\kappa)}, \\
& \qquad key_0, \vec{c}_0^{\alpha}, \vec{c}_0^{\beta}, \\
& ((key_1, \vec{c}_1^{\alpha}) \leftarrow BankGen(Bank_1))^{\lambda(\kappa)}, ((key_1, \vec{c}_1^{\beta}) \leftarrow BankGen(Bank_1))^{\lambda(\kappa)} \\
& (Bank'_1 \leftarrow BankRedraw(Bank_1))^{\lambda(\kappa)}, (^{\lambda(\kappa)} Bank_2 \leftarrow BankUpdate(Bank'_1))^{\lambda(\kappa)}, \\
& \qquad key_1, \vec{c}_1^{\alpha}, \vec{c}_1^{\beta}, \\
& \ldots \\
& ((key_{T-1}, \vec{c}_{T-1}^{\alpha}) \leftarrow BankGen(Bank_{T-1}))^{\lambda(\kappa)}, ((key_{T-1}, \vec{c}_{T-1}^{\beta}) \leftarrow BankGen(Bank_{T-1}))^{\lambda(\kappa)}, \\
& (Bank'_{T-1} \leftarrow BankRedraw(Bank_{T-1}))^{\lambda(\kappa)}, (Bank_T \leftarrow BankUpdate(Bank'_{T-1}))^{\lambda(\kappa)}, \\
& \qquad key_{T-1}, \vec{c}_{T-1}^{\alpha}, \vec{c}_{T-1}^{\beta}\big\}
\end{aligned}
$$

$$
\begin{aligned}
Simulated \;=\; & \mathcal{A}\big\{((key_0, \vec{c}_0^{\alpha}) \leftarrow SimBankGen(Bank_0, \vec{b}[0]))^{\lambda(\kappa)}, (key_0, \vec{c}_0^{\beta} \leftarrow SimBankGen(Bank_0, \vec{b}[0]))^{\lambda(\kappa)}, \\
& (Bank'_0 \leftarrow SimBankRedraw(Bank_0))^{\lambda(\kappa)}, (Bank_1 \leftarrow SimBankUpdate(Bank'_0))^{\lambda(\kappa)}, \\
& \qquad key_0, \vec{c}_0^{\alpha}, \vec{c}_0^{\beta}, \\
& ((key_1, \vec{c}_1^{\alpha}) \leftarrow SimBankGen(Bank_1, \vec{b}[1]))^{\lambda(\kappa)}, ((key_1, \vec{c}_1^{\beta}) \leftarrow SimBankGen(Bank_1), \vec{b}[1])^{\lambda(\kappa)} \\
& (Bank'_1 \leftarrow SimBankRedraw(Bank_1))^{\lambda(\kappa)}, (Bank_2 \leftarrow SimBankUpdate(Bank'_1))^{\lambda(\kappa)}, \\
& \qquad key_1, \vec{c}_1^{\alpha}, \vec{c}_1^{\beta}, \\
& \ldots \\
& ((key_{T-1}, \vec{c}_{T-1}^{\alpha}) \leftarrow SimBankGen(Bank_{T-1}, \vec{b}[T-1]))^{\lambda(\kappa)}, \\
& ((key_{T-1}, \vec{c}_{T-1}^{\beta}) \leftarrow SimBankGen(Bank_{T-1}, \vec{b}[T-1]))^{\lambda(\kappa)}, \\
& (Bank'_{T-1} \leftarrow SimBankRedraw(Bank_{T-1}))^{\lambda(\kappa)}, (Bank_T \leftarrow SimBankUpdate(Bank'_{T-1}))^{\lambda(\kappa)}, \\
& \qquad key_{T-1}, \vec{c}_{T-1}^{\alpha}, \vec{c}_{T-1}^{\beta}\big\}
\end{aligned}
$$

then $\Delta(Real, Simulated) = \delta(\kappa)$.

Lemma 5.5. *There exists a leakage bound $\lambda(\kappa) = \Omega(\kappa)$, and a probability bound $\delta(\kappa) = \exp(-\Omega(\kappa))$, s.t. for any $\kappa \in \mathbb{N}$, any execution bound $T = \mathrm{poly}(\kappa)$, any vector $\vec{b} \in \{0,1\}^T$, and any (computationally unbounded) leakage adversary $\mathcal{A}$, the following holds:*

Let Simulated be the following distribution, where we begin by running $Bank_0 \leftarrow SimBankInit$*:*

$$
\begin{aligned}
Simulated \quad = \quad & \mathcal{A}\big\{((key_0, \vec{c}_0^{\alpha}) \leftarrow SimBankGen(Bank_0, \vec{b}[0]))^{\lambda(\kappa)}, (key_0, \vec{c}_0^{\beta} \leftarrow SimBankGen(Bank_0, \vec{b}[0]))^{\lambda(\kappa)}, \\
& (Bank'_0 \leftarrow SimBankRedraw(Bank_0))^{\lambda(\kappa)}, (Bank_1 \leftarrow SimBankUpdate(Bank'_0))^{\lambda(\kappa)}, \\
& \qquad key_0, \vec{c}_0^{\alpha}, \vec{c}_0^{\beta}, \\
& ((key_1, \vec{c}_1^{\alpha}) \leftarrow SimBankGen(Bank_1, \vec{b}[1]))^{\lambda(\kappa)}, ((key_1, \vec{c}_1^{\beta}) \leftarrow SimBankGen(Bank_1), \vec{b}[1])^{\lambda(\kappa)} \\
& (Bank'_1 \leftarrow SimBankRedraw(Bank_1))^{\lambda(\kappa)}, (Bank_2 \leftarrow SimBankUpdate(Bank'_1))^{\lambda(\kappa)}, \\
& \qquad key_1, \vec{c}_1^{\alpha}, \vec{c}_1^{\beta}, \\
& \ldots \\
& ((key_{T-1}, \vec{c}_{T-1}^{\alpha}) \leftarrow SimBankGen(Bank_{T-1}, \vec{b}[T-1]))^{\lambda(\kappa)}, \\
& ((key_{T-1}, \vec{c}_{T-1}^{\beta}) \leftarrow SimBankGen(Bank_{T-1}, \vec{b}[T-1]))^{\lambda(\kappa)}, \\
& (Bank'_{T-1} \leftarrow SimBankRedraw(Bank_{T-1}))^{\lambda(\kappa)}, (Bank_T \leftarrow SimBankUpdate(Bank'_{T-1}))^{\lambda(\kappa)}, \\
& \qquad key_{T-1}, \vec{c}_{T-1}^{\alpha}, \vec{c}_{T-1}^{\beta}\big\}
\end{aligned}
$$

For any w *in the support of Simulated, and for any* $i \in [T]$*, fixing all ciphertexts except the* i*-th pair, let* $D_i^{\alpha}(w)$ *and* $D_i^{\beta}(w)$ *be the joint distribution of* $(key_i, \vec{c}_i^{\alpha})$ *and* $(key_i, \vec{c}_i^{\beta})$ *(respectively) given:* w*, the remaining* $T-1$ *keys and ciphertext pairs, and (respectively)* $\vec{c}_i^{\beta}$ *and* $\langle key_i, \vec{c}_i^{\beta} \rangle$*, or* $\vec{c}_i^{\alpha}$ *and* $\langle key_i, \vec{c}_i^{\alpha} \rangle$*. Then there exist distributions* $\mathcal{K}_i^{\alpha}(w)$*,* $\mathcal{C}_i^{\alpha}(w)$ *and* $\mathcal{K}_i^{\beta}(w)$*,* $\mathcal{C}_i^{\beta}(w)$ *s.t. the following holds:*[8]

The distributions $\mathcal{D}_i^{\alpha}(w)$ *and* $\mathcal{D}_i^{\beta}(w)$ *are both IuO with orthogonality* $\vec{b}[i]$ *and underlying distributions* $\mathcal{K}_i^{\alpha}(w)$ *and* $\mathcal{C}_i^{\alpha}(w)$ *or* $\mathcal{K}_i^{\beta}(w)$ *and* $\mathcal{C}_i^{\beta}(w)$ *(respectively). With all but* $\delta(\kappa)$ *probability over the choice (by Simulated) of the fixed values, the min-entropies of all these underlying distributions are at least* $\kappa - O(\lambda(\kappa))$*.*

5.2 Piecemeal Matrix Computations

Recall that we treat collections of ciphertexts as matrices, where each column of the matrix is a ciphertext. We refer to the procedures in this section as "piecemeal", because they access the matrices by dividing them into *"pieces" or "sketches"*, and loading each piece (or sketch) into memory separately. Each piece/sketch is a collection of linear combinations of the matrix's columns. We refer to these as pieces (rather than sketches) throughout this section.

We present piecemeal procedures for matrix multiplication, for refreshing the key under which the ciphertexts in a matrix's columns are encrypted, and for adding a vector to the columns of a matrix (we refer to this as matrix-vector addition). We show that these procedures have several security properties under leakage attacks. In all these procedures, no matrix is ever loaded into memory in its entirety. Rather, the matrices are only accessed in a piecemeal manner.

As an (important) example for why this facilitates security, consider the rank of a matrix on which we are computing. If this matrix is loaded into memory in its entirety, then a leakage adversary can compute its rank. If, however, only "pieces" of the matrix are loaded into memory

[8] Note that these distributions *do not depend on* $\vec{b}_i$ (they depend only on w, on $\mathcal{A}$, on the $T-1$ remaining key-ciphertext pairs, and on the additional i-th ciphertext ($\vec{c}_i^{\beta}$ or $\vec{c}_i^{\alpha}$ respectively).

at any once time, then it is no longer clear how a leakage adversary can compute the rank. In fact, we will show that (under the appropriate matrix distribution), as long as the matrix is accessed in a piecemeal fashion, its rank is completely hidden, even from a computationally unbounded leakage adversary. This fact will be used extensively in our security proofs. See the subsequent sections for security properties and proofs.

PiecemealMM(A, B): multiplies matrices $A \in \{0,1\}^{\kappa \times m}$ and $B \in \{0,1\}^{m \times n}$; Under leakage

Parse: $A = [A_1, \ldots, A_a]$, where each A_i is a $\kappa \times \ell$ matrix, and $B^T = [B_1^T, \ldots, B_b^T]$, where each B_j is an $m \times \ell$ matrix. Further parse each $B_i^T = [B_{i,1}^T, \ldots, B_{i,a}^T]$, where each $B_{i,j}$ is an $\ell \times \ell$ matrix.

1. For $i \leftarrow 1, \ldots, b$:
   (a) Set $D_0 = \bar{0}$
   (b) For $j \leftarrow 1, \ldots, a$: $D_j \leftarrow D_{j-1} + (\mathcal{A}_j \times B_{i,j})$; *leakage on each tuple* $(D_{j-1}, A_j, B_{i,j})$ *separately*
   (c) $C_i \leftarrow D_a$
2. Output the product matrix $C = [C_1, \ldots, C_b]$

Figure 5: Piecemeal Matrix Multiplication for $\kappa, \ell \in \mathbb{N}$

PiecemealRefresh(key, A): refreshes the key for matrix $A \in \{0,1\}^{\kappa \times m}$

Parse: $A = [A_1, \ldots, A_a]$, where each A_i is a $\kappa \times \ell$ matrix.

1. $\sigma \leftarrow KeyEntGen(1^\kappa)$
2. for $i \leftarrow 1 \ldots a$: $A_i' \leftarrow CipherCorrelate(A_i, \sigma)$; *leakage on* (A_i, σ) *for each* i *separately*
3. $key' \leftarrow KeyRefresh(key, \sigma)$; *leakage on* (key, σ)
4. Output key and the refreshed matrix $A' = [A_1', \ldots, A_a']$

Figure 6: Piecemeal Matrix Refresh for $\kappa, \ell \in \mathbb{N}$

PiecemealAdd$(A, \vec{v})$: adds $\vec{v} \in \{0,1\}^\kappa$ to each column of $A \in \{0,1\}^{\kappa \times m}$

Parse: $A = [A_1, \ldots, A_a]$, where each A_i is a $\kappa \times \ell$ matrix.

1. for $i \leftarrow 1 \ldots a, j \leftarrow 1 \ldots \ell$: $A_i'[\ell] \leftarrow A_i[\ell] + \vec{v}$; *leakage on* $(A_i, \vec{v})$ *for each* i *separately*
2. $A' = [A_1', \ldots, A_a']$

Figure 7: Piecemeal Matrix Addition for $\kappa, \ell \in \mathbb{N}$

5.3 Piecemeal Leakage Attacks on Matrices and Vectors

In this section, we define "piecemeal leakage attacks" on matrices. In particular, these attacks capture the leakage that can be computed via a leakage attack on the piecemeal matrix procedures

(multiplication, refresh, and matrix-vector addition). We prove then that random matrices are resilient to several flavors of such piecemeal attacks.

Attack on a Matrix. A *piecemeal leakage attack* on a matrix, is a multi-source leakage attack, where the sources are *key* and (one or many) "pieces" of the matrix. Recall that each "piece" here is a collection of linear combinations of the matrix columns. See Definition 5.6 below. We focus here on the case where the matrix is either independent of *key*, or has columns orthogonal to *key* (as is the case for a ciphertext bank corresponding to underlying plaintext bit 0). The case where the columns have inner product 1 with *key* is handled similarly.

We will show that a random matrix M is resilient to piecemeal leakage: the leakage computed in such an attack is statistically close when (i) the columns of M are all in the kernel of *key*, (ii) M is a uniformly random matrix, and (iii) M is a uniformly random matrix of rank $\kappa-1$ (independent of *key*). Moreover, this statistical closeness holds even if *key* is later exposed in it's entirety. We begin in Section 5.3.1 with a warmup for the case of an attack on a single piece (Lemma 5.8). We then show security for large number of pieces in Section 5.3.2 (Lemma 5.10).

Definition 5.6 (Piecemeal Leakage Attack on (key, M))**.** Take $a, \kappa, \lambda, \ell, m \in \mathbb{N}$. Let $\vec{Lin} = (Lin_1, \ldots, Lin_a)$ be a sequence of (one or more) matrices, where for each Lin_i, its columns each specify the coefficients of a linear combination of the rows of M. Thus, for $M \in \{0,1\}^{\kappa \times m}$ and $Lin_i \in \{0,1\}^{m \times \ell}$, the matrix piece $M \times Lin_i$ is a collection of ℓ linear combinations of M's columns.

Let $\mathcal{A}$ be a leakage adversary, operating separately on $key \in \{0,1\}^\kappa$ and on several matrices in $\{0,1\}^{\kappa \times \ell}$ (each matrix is $M \times Lin_i$ for some i). We denote $\mathcal{A}$'s output by:

$$\mathcal{A}^\lambda_{\kappa,\ell,m,\vec{Lin}}(key, M) \triangleq \mathcal{A}^\lambda(1^\kappa)[key]\{(M \times Lin_1), \ldots, (M \times Lin_a)\}$$

we refer to $\mathcal{A}$ as a "piecemeal adversary" operating on (key, M). We omit κ, λ, ℓ, m and $\vec{Lin}$ when they are clear from the context.

Attack on a Matrix and Vector. We extend these results further, considering piecemeal leakage that operates separately on *key*, and on pieces of a matrix M (as before), each piece jointly with a vector $\vec{v}$. See Definition 5.7 below.

We show that, for a matrix M with columns in the kernel of *key*, the leakage computed in such an attack is statistically close when (i) the vector $\vec{v}$ is in the kernel of *key*, and (ii) the vector $\vec{v}$ is *not* in the kernel of *key*. Moreover, this statistical closeness holds even if *key* is later exposed in its entirety (as above) *and also M is later exposed in its entirety.* See Section 5.3.3 and Lemma 5.15.

Definition 5.7 (Piecemeal Leakage Attack on $(key, (M, \vec{v}))$)**.** Take $a, \kappa, \lambda, \ell, m \in \mathbb{N}$. Let $\vec{Lin} = (Lin_1, \ldots, Lin_a)$ be a sequence of matrices, where for each Lin_i, its columns each specify the coefficients of a linear combination of the rows of M as in Definition 5.6.

Let $\mathcal{A}$ be a leakage adversary, operating separately on $key \in \{0,1\}^\kappa$ and on several matrices in $\{0,1\}^{\kappa \times \ell}$ (as in Definition 5.6), each matrix jointly with a vector $\vec{v} \in \{0,1\}^\kappa$. We denote $\mathcal{A}$'s output by:

$$\mathcal{A}^\lambda_{\kappa,\ell,m,\vec{Lin}}(key, (M, \vec{v})) \triangleq \mathcal{A}^\lambda(1^\kappa)[key]\{((M \times Lin_1) \circ \vec{v}), \ldots, ((M \times Lin_a) \circ \vec{v})\}$$

we refer to $\mathcal{A}$ as a "piecemeal adversary" operating on $(key, (M, \vec{v}))$. We omit κ, λ, ℓ, m and $\vec{Lin}$ when they are clear from the context.

5.3.1 Piecemeal Leakage Resilience: One Piece

We begin by showing that, for a uniformly random $key \in \{0,1\}^\kappa$, and a matrix M, given separate leakage from *key* and from a *single piece* of the matrix, the following two cases induce statistically close distributions. In the first case, the matrix M is uniformly random with columns in the kernel of *key*. In the second case, M is a uniformly random matrix of rank $\kappa - 1$ (independent of *key*). By a "single piece" of M we mean any (adversarially chosen) collection of ℓ linear combinations of vectors from M, where here we take $\ell = 0.1\kappa$. This result, stated in Lemma 5.8, is a warm-up for the results in later sections.

Lemma 5.8 (Matrices are Resilient to Piecemeal Leakage with One Piece)**.** *Take $\kappa, m \in \mathbb{N}$ where $m \geq \kappa$. Fix $\ell = 0.1\kappa$ and $\lambda = 0.05\kappa$. Let $Lin \in \{0,1\}^{m \times \ell}$ be any collection of coefficients for ℓ linear combinations, and $\mathcal{A}$ be any piecemeal leakage adversary. Take Real and Simulated to be the following two distributions:*

$$\begin{aligned} Real &= \left(key, \mathcal{A}^{\lambda}_{\kappa,\ell,m,Lin}(key, \mathbf{M})\right)_{key \in_R \{0,1\}^\kappa, \mathbf{M} \in_R \{0,1\}^{\kappa \times m} : \forall i, M[i] \in kernel(key)} \\ Simulated &= \left(key, \mathcal{A}^{\lambda}_{\kappa,\ell,m,Lin}(key, \mathbf{M})\right)_{key \in_R \{0,1\}^\kappa, \mathbf{M} \in_R \{0,1\}^{\kappa \times m} : rank(M) = \kappa - 1} \end{aligned}$$

then $\Delta(Real, Simulated) \leq 2m \cdot 2^{-0.2\kappa}$.

Remark 5.9. *We note that, without* any *leakage access to key (i.e. given only leakage from the chosen piece of M), a qualitatively similar result to Lemma 5.8 can be derived from a Lemma of Brakerski* et al. *[BKKV10] on the leakage resilience of random linear subspaces. Their work focused on the more challenging setting where the leakage operates on vectors that are drawn from a low-dimensional subspace (e.g. constant dimension). .*

Proof of Lemma 5.8. The proof is by a hybrid argument over the matrix columns. For $i \in \{0, \ldots, m\}$, let $\mathcal{H}_i$ be the i-th hybrid, where the view is as above but using a matrix M drawn s.t. the first i columns of M_i are uniformly random in the kernel of *key*, and the last $m - i$ columns are uniformly random s.t. $rank(M) = \kappa - 1$. We show that for all i, $\Delta(\mathcal{H}_i, \mathcal{H}_{i+1}) \leq 2 \cdot 2^{-0.2\kappa}$. The lemma follows because $\mathcal{H}_0 = Simulated$ and $\mathcal{H}_m = Real$.

We show that the hybrids are close by giving a reduction from the task of predicting the inner product of two vectors under multi-source leakage, to the task of distinguishing $\mathcal{H}_i$ and $\mathcal{H}_{i+1}$. Since the inner product cannot be predicted under multi-source leakage (by Lemma 3.7), we conclude that the hybrids are statistically close.

To set up the reduction, first fix i. Draw a uniformly random matrix $M \in \{0,1\}^{\kappa \times m}$ of rank $\kappa - 1$. Let $\vec{v}$ be the $(i+1)$-th column of M.Let $M_{-(i+1)}$ be the matrix M with the $(i+1)$-th column set to 0. Now draw $key \in \{0,1\}^\kappa$ s.t *key* is orthogonal to the first i columns in $M_{-(i+1)}$.

We show a reduction from predicting the inner product $\langle key, \vec{v} \rangle$ given multi-source leakage and $(M_{-(i+1)} \times Lin)$, to distinguishing $\mathcal{H}_i$ and $\mathcal{H}_{i+1}$. This is done by running $\mathcal{A}(key, M)$ on *key* and on the matrix M drawn above. The reduction computes $\mathcal{A}$'s (multi-source) leakage on *key* using multi-source leakage from *key*. $\mathcal{A}$'s (multi-source) leakage from $M \times Lin$ is computed using leakage from $\vec{v}$ (since *Lin* and $M_{-(i+1)} \times Lin$ are "public"). Note now that the joint distribution of (key, M) is exactly as in $\mathcal{H}_i$. If, however, we condition on the inner product of *key* and $\vec{v}$ being 0, we get that the joint distribution of (key, M) is exactly as in $\mathcal{H}_{i+1}$. Thus, if $\mathcal{A}$ has advantage δ in distinguishing $\mathcal{H}_i$ and $\mathcal{H}_{i+1}$, then the reduction has advantage δ in distinguishing the case that the inner product of *key* and $\vec{v}$ is 0 from the case that there is no restriction on the inner product.

Now observe that, given $(M_{-(i+1)} \times Lin)$, the vector *key* is a random variable with min-entropy at least $\kappa - \ell \geq 0.9\kappa$. This is because *key* is uniformly random under the restriction that it is in the kernel of the first i columns of M. The matrix piece $(M_{-(i+1)} \times Lin)$ contains only $\ell = 0.1\kappa$ vectors, and so it cannot give more than ℓ bits of information on *key*. Note also that, given $(M_{-(i+1)} \times Lin)$, the $(i+1)$-th column $\vec{v}$ is independent of *key*, and also $\vec{v}$ has min entropy at least $(\kappa - 1)$ (in fact $\vec{v}$ has high min entropy even given all of $M_{-(i+1)}$.

The reduction uses $\lambda = 0.05\kappa$ bits of multi-source leakage, and so by lemma 3.8 with all but $2^{-0.2\kappa}$ probability, even given the leakage *key* and $\vec{v}$ are still independent random sources, both with min entropy at least 0.7κ. When this is the case, by lemma 3.7 we know that, even given *key*, the inner product of *key* and $\vec{v}$ is $2^{-0.2\kappa}$-close to uniform. We conclude that $\delta \leq 2 \cdot 2^{-0.2\kappa}$. The lemma follows. ■

5.3.2 Piecemeal Leakage Resilience: Many Pieces

In this section, we show our main technical result regarding piecemeal matrix leakage. We show that random matrices are resilient to piecemeal leakage on *multiple pieces of the matrix* (operating separately on each piece). In particular, the leakage is statistically close in the case where the matrix is one whose columns are all orthogonal to *key* and in the case where the matrix is uniformly random. Moreover, this remains true even if *key* is later exposed in its entirety.

Lemma 5.10 (Matrices are Resilient to Piecemeal Leakage with Many Pieces)**.** *Take $a, \kappa, m \in \mathbb{N}$, where $m \geq \kappa$. Fix $\ell = 0.1\kappa$, and $\lambda = 0.05\kappa/a$. Let $\vec{Lin} = (Lin_1, \ldots, Lin_a)$ be any sequence of collections of coefficients for linear combinations, where for each i, $Lin_i \in \{0,1\}^{m \times \ell}$ has full rank ℓ. Let $\mathcal{A}$ be any piecemeal leakage adversary. Take Real and Simulated to be the following two distributions:*

$$\begin{aligned} Real &= \left(key, \mathcal{A}^{\lambda}_{\kappa,\ell,m,\vec{Lin}}(key, \mathbf{M})\right)_{key \in_R \{0,1\}^\kappa, \mathbf{M} \in_R \{0,1\}^{\kappa \times m} : \forall i, M[i] \in kernel(key)} \\ Simulated &= \left(key, \mathcal{A}^{\lambda}_{\kappa,\ell,m,\vec{Lin}}(key, \mathbf{M})\right)_{key \in_R \{0,1\}^\kappa, \mathbf{M} \in_R \{0,1\}^{\kappa \times m} : rank(M) = \kappa - 1} \end{aligned}$$

then $\Delta(Real, Simulated) \leq 5a^2 \cdot 2^{-0.04\kappa/a}$.

Proof. For $i \in \{0, \ldots, a\}$, we denote $P_i = M \times Lin_i$ the matrix "piece" being leaked on/attacked in the i-th part of the attack. We use w_i to denote the leakage accumulated by $\mathcal{A}$ up to and including the i-th attack. We will consider $\mathcal{V}_i$, the conditional distribution on (key, M), drawn as in *Real*, given the leakage w_i. Namely, in $\mathcal{V}_0$ we have *key* drawn uniformly at random and M is random with columns in $kernel(key)$. Note that the random variables *key* and M, when drawn by $\mathcal{V}_i$, are not independent. In particular, *key* and the columns of M are orthogonal. Let $\mathcal{K}_i$ and $\mathcal{M}_i$ be the marginal distributions of $\mathcal{V}_i$ on *key* and on M.

Hybrids. We will prove Lemma 5.10 using a hybrid argument. For $i \in \{0, \ldots, a\}$, we define a hybrid distribution $\mathcal{H}_i$. Each hybrid's output domain will be $key \in \{0,1\}^\kappa$ and leakage values computed by $\mathcal{A}(key, M)$.

For each i, we define $\mathcal{H}_i$ by drawing $(key, M) \sim \mathcal{V}_0$, and simulating the piecemeal leakage attack $\mathcal{A}(key, M)$. We always use *key* for computing the key leakage in the attack. For leakage on the j-th matrix piece, however, we use P_j's drawn differently for each $\mathcal{H}_i$:

- For $j \in \{1, \ldots, i\}$, we define $P_j = (\mathbf{M} \times \mathit{Lin}_j)$.
- For $j \in \{i+1, \ldots a\}$, re-draw $M_j \sim \mathcal{M}_{j-1}$. I.e., we re-draw the matrix from the current marginal distribution of $\mathcal{V}_{j-1}$ on M, independently of *key*. Define $P_j = (M_j \times \mathit{Lin}_j)$.

Clearly, $\mathcal{H}_a = \mathit{Real}$, because in $\mathcal{H}_a$ we never compute leakage on a re-drawn matrix M_j. We will show that $\mathcal{H}_0 = \mathit{Simulated}$, see Claim 5.11. Note that this is non-trivial because in $\mathcal{H}_0$ the matrix M is continually re-drawn from M_j (independently of *key*), whereas in *Simulated* the matrix M is never redrawn. Nonetheless, Claim 5.11 below shows that, because the leakage operates separately on *key* and on M, these two distributions are identical.

Claim 5.11. $\mathcal{H}_0 = \mathit{Simulated}$

Proof of Claim 5.11. Fix leakage w_j for the first j attacks on pieces of M. In the distribution $\mathcal{H}_0$, for the $(j+1)$-th matrix piece, we use $P_{j+1} = M_{j+1} \times \mathit{Lin}_{j+1}$, where M_{j+1} is re-drawn from the marginal distribution $\mathcal{M}_j$.

In the distribution *Simulated*, on the other hand, we use $P_{j+1} = M \times \mathit{Lin}_{j+1}$, where M is drawn from $\mathcal{M}'_j$, the distribution of uniformly random M's of rank $\kappa - 1$ (independent of *key*), given that the multi-source leakage so far was w_j.

Other than this difference, the distributions are identical. Thus, it suffices to show that, for every j and every fixed leakage w_j in the first j attacks, we have that $\mathcal{M}_j = \mathcal{M}'_j$.

The leakage in the first j attacks operates separately on *key* and on M. Thus, we know that conditioning the joint distribution $\mathcal{V}_0$ on w_j, is equivalent to conditioning $\mathcal{V}_0$ on (key, M) falling in a product set. Let $S_{\mathit{key}} \subseteq \{0,1\}^\kappa$ and $S_M \subseteq \{0,1\}^{\kappa \times 2\kappa}$ be the sets s.t. for all $(\mathit{key}, M) \in S_{\mathit{key}} \times S_M$, the leakage on the first j pieces in a piecemeal attack on (key, M) equals w_j. Now we know that $\mathcal{M}_j$ is exactly equal to $\mathcal{M}_0$, conditioned on M falling in the set S_M.

Similarly, in *Simulated* the distribution $\mathcal{M}'_j$ is the uniform distribution on rank $\kappa - 1$ matrices, conditioned on the leakage w_j, i.e. on M falling in the set S_M. Since $\mathcal{M}_0$ is uniform on rank $\kappa - 1$ matrices, for any w_j we get that $\mathcal{M}_j = \mathcal{M}'_j$. The claim follows. ■

To complete the proof of Lemma 5.10, we will show that $\Delta(\mathcal{H}_i, \mathcal{H}_{i+1}) \leq 4m \cdot 2^{-0.04\kappa/a}$. The lemma follows by a hybrid argument. For this, consider the joint distribution of *key*, and of the leakage w_{i+1} computed on the first $(i+1)$ pieces. We will show that the joint distribution is statistically close in both hybrids. This suffices to show that the hybrids themselves are statistically close, because, for both hybrids, the leakage on pieces $((i+2), \ldots, a)$, and the remaining leakage on *key*, can be computed as a function of (key, w_{i+1}) (the same function for both hybrids).

In both $\mathcal{H}_i, \mathcal{H}_{i+1}$, leakage on the first i pieces is computed in exactly the same way. The difference is in leakage on the $(i+1)$-th piece. Fixing the leakage w_i on the first i pieces, in $\mathcal{H}_{i+1}$ we have P_{i+1} computed using dependent $(\mathit{key}, M) \sim \mathcal{V}_i$. In $\mathcal{H}_i$ we use independent $\mathit{key} \sim \mathcal{K}_i, M \sim \mathcal{M}_i$. These two different distributions yield different leakage w on the $(i+1)$-th piece.

Piecemeal Leakage from IuO Distributions. *key* and M drawn (jointly) by $\mathcal{V}_i$ are not independent. In general, for a dependant distribution $\mathcal{V}_i$ on *key* and M with marginal distributions $\mathcal{K}_i$ and $\mathcal{M}_i$, leakage on $(\mathit{key}, M) \sim \mathcal{V}_i$ could looks very different from leakage on $(\mathit{key} \sim \mathcal{K}_i, M \sim \mathcal{M}_i)$. We will show, however, that piecemeal leakage resilience *does hold* in a special case where the joint distribution $\mathcal{V}_i$ is independent up to orthogonality (IuO, see Definition 3.10). We will also show it holds when $\mathcal{V}_i$ is statistically close to IuO, as defined below.

Definition 5.12 (Key-Matrix α-Independence up to Orthogonality)**.** Let $\mathcal{V}$ be a distribution on pairs (key, M), where $key \in \{0,1\}^\kappa, M \in \{0,1\}^{\kappa \times 2\kappa}$ and M is always of rank $\kappa - 1$. We say that $\mathcal{V}$ is *α-independent up to orthogonality*, if there exists distribution $\mathcal{V}'$ that is independent up to orthogonality and $\Delta(\mathcal{V}, \mathcal{V}') \leq \alpha$.

We will show that piecemeal leakage on an IuO distribution is statistically close to piecemeal leakage when *key* and M are sampled from the independently drawn variant, see Claim 5.13 below. We also show that $\mathcal{V}_i$ is (w.h.p over w_i) an IoU distribution, see Claim 5.14. Statistical closeness of the hybrids $\mathcal{H}_i$ and $\mathcal{H}_{i+1}$ follows.

Claim 5.13. *Take $a, \kappa, m, \ell, \lambda$ as in Lemma 5.10. Let $\mathcal{V}$ be any distribution over pairs (key, M), where $key \in \{0,1\}^\kappa, M \in \{0,1\}^{\kappa \times m}$ and M has rank $\kappa - 1$. Suppose that $\mathcal{V}$ is IuO, with underlying distributions $\mathcal{K}$ and $\mathcal{M}$. Suppose further that $\mathcal{V}$ has min-entropy at least $(\kappa + (\kappa - 1) \cdot 2\kappa - 0.15\kappa)$.*

Let $Lin \in \{0,1\}^{m \times \ell}$ be a collection of coefficients for linear combinations, specified by a matrix of rank ℓ. Let $\mathcal{A}$ be any piecemeal leakage adversary. Take $\mathcal{D}$ and $\mathcal{F}$ to be the following distributions:

$$\begin{aligned} \mathcal{D} &= (key, w)_{(key,\mathbf{M}) \sim \mathcal{V}, w \leftarrow \mathcal{A}(key, M)} \\ \mathcal{F} &= (key, w)_{key \sim \mathcal{K}, \mathbf{M} \sim \mathcal{M}, w \leftarrow \mathcal{A}(key, M)} \end{aligned}$$

Take $\delta = (4\ell \cdot 2^{-0.05\kappa})$. Then $\Delta(D, F) \leq 2\delta$. Moreover, with all but δ probability over $w \sim D$, we have that $\Delta((D|\mathcal{A}(key, M) = w), (F|\mathcal{A}(key, M) = w)) \leq \delta$.

The proof of Claim 5.13 is below.

Claim 5.14. *Take $a, \kappa, \ell, \lambda, \mathcal{V}, L, \mathcal{A}$ as in Claim 5.13. Suppose here that $\mathcal{V}$: (i) has min-entropy at least $(\kappa + (\kappa - 1) \cdot 2\kappa - 0.15\kappa)$ (as in Claim 5.13), and (ii) is α-close to independence up to orthogonality (see Definition 5.12). Define the distribution:*

$$\mathcal{V}(w) = (key, M)_{(key,M) \sim \mathcal{V} : \mathcal{A}(key,M) = w}$$

and take $\delta = (4\ell \cdot 2^{-0.05\kappa})$. For any $0 < \beta < 1$, with all but $(\beta + \delta)$ probability over $w \leftarrow \mathcal{A}(key, M)_{(key,M) \sim \mathcal{V}}$ it is the case that $\mathcal{V}(w)$ is $((\alpha/\beta) + \delta)$-close to independence up to orthogonality.

The proof of Claim 5.14 is below. We now complete the proof of Lemma 5.10:

1. With all but $2^{-0.05\kappa}$ probability over w_i, for all $j \leq i$ simultaneously, the min-entropy of $\mathcal{V}_j$ is at least $\kappa + (\kappa - 1) \cdot 2\kappa - 0.15\kappa$. This is by Lemma 3.8, because the min-entropy of $\mathcal{V}_0$ is $\kappa + (\kappa - 1) \cdot 2\kappa$, and the amount of leakage in the first $i \leq a$ attacks (leakage from both *key* and M) is less than 0.1κ.

2. Take $\delta = (4\ell \cdot 2^{-0.05\kappa}), \beta = 2^{-0.04\kappa/a}$. We show the following by induction for $j \leq i$:

   with all but $(2^{-0.05\kappa} + j \cdot (\delta + \beta))$ probability over w_i, we have that $\mathcal{V}_j$ is $(2\delta/\beta^j)$-close to independence up to orthogonality (and also the min entropy bound of Item 1 holds). The induction basis follows because $\mathcal{V}_0$ is perfectly independent up to orthogonality. The induction step follows from Claim 5.14 (and the min-entropy bound in Item 1).

Finally, we use Claim 5.13 to conclude that with all but $(2^{-0.05\kappa} + i \cdot (\delta + \beta))$ probability over w_i, the hybrids $\mathcal{H}_i$ and $\mathcal{H}_{i+1}$ are $(2\delta/\beta^i + 2\delta)$-statistically close. In particular, this implies that

$$\Delta(\mathcal{H}_i, \mathcal{H}_{i+1}) \leq (2^{-0.05\kappa} + i \cdot (\delta + \beta)) + (2\delta/\beta^i) + 2\delta) \leq 5a \cdot 2^{-0.04\kappa/a}$$

where the second inequality assumes $i \cdot \beta$ is the largest term in the sum (and using $i \leq a$). ■

Proof of Claim 5.13. The proof is by a hybrid argument. We denote $P = M \times L$. For $i \in [a+1]$, take the i-th hybrid $\mathcal{H}_i$ to be:

$$\mathcal{H}_i = (key, w)_{M \sim \mathcal{M}, P \leftarrow M \times L, key \sim (\mathcal{K}|P[1],\ldots,P[i]), w \leftarrow \mathcal{A}(key,P)}$$

i.e. the key is drawn from a conditional distribution on $\mathcal{K}$, conditioning on the first i columns of P. We get that $\mathcal{H}_0 = \mathcal{F}$, because *key* is drawn without conditioning on any columns (i.e. independently of M). Also $\mathcal{H}_\ell = \mathcal{D}$, because *key* is re-drawn conditioned on all of P, which is the same as just drawing $(key, M) \sim \mathcal{V}$ and taking $P = M \times L$.

For each pair of hybrids, we bound $\Delta(\mathcal{H}_i, \mathcal{H}_{i+1})$. To do so, consider the following experiment: draw $(P[1], \ldots, P[i]) \sim M$ (as in both $\mathcal{H}_i$ and $\mathcal{H}_{i+1}$). Fixing these draws, in $\mathcal{H}_i$ the distribution of $P[i+1]$ is an random sample from $\mathcal{P}_i = (P[i+1]_{M \sim \mathcal{M}|P[1],\ldots,P[i]})$. Similarly, in $\mathcal{H}_i$ we have that *key* is a random sample from $\mathcal{K}_i = (\mathcal{K}|P[1], \ldots, P[i])$. In particular, note that *key* is independent of $P[i+1]$.

We now examine $\mathcal{H}_i^+$, obtained from $\mathcal{H}_i$ by including also the inner product of *key* and $P[i+1]$. We can also consider $\mathcal{H}_i^R$, obtained from $\mathcal{H}_i$ by adding a uniformly random bit:

$$\begin{aligned}
\mathcal{H}_i^+ &= (key, \langle key, P[i+1] \rangle, w)_{key \sim \mathcal{K}_i, P[i+1] \sim \mathcal{P}_i, (P[i+2],\ldots,P[\ell]) \sim (\mathcal{M}|P[1],\ldots,P[i+1])), w \leftarrow \mathcal{A}(key,P)} \\
\mathcal{H}_i^R &= (key, \mathbf{r} \qquad\qquad\qquad\quad , w)_{key \sim \mathcal{K}_i, P[i+1] \sim \mathcal{P}_i, (P[i+2],\ldots,P[\ell]) \sim (\mathcal{M}|P[1],\ldots,P[i+1])), w \leftarrow \mathcal{A}(key,P), \mathbf{r} \in_{\mathbf{R}} \{\mathbf{0},\mathbf{1}\}}
\end{aligned}$$

We will show that $\Delta(\mathcal{H}_i, \mathcal{H}_{i+1}) \leq 2\Delta(\mathcal{H}_i^+, \mathcal{H}_i^R)$. To show this, consider now $\mathcal{H}_{i+1}$. Again, $P[i+1]$ is an independent sample from $\mathcal{P}_i$ (as in $\mathcal{H}_i$). Here, however, we have that *key* depends on $P[i+1]$ and is a sample from $\mathcal{K}_{i+1} = (\mathcal{K}|w, P[1], \ldots, P[i], \mathbf{P[i+1]})$. Since $\mathcal{V}$ is independent up to orthogonality, we have:

$$\begin{aligned}
\mathcal{K}_{i+1} &= (key, P[1], \ldots, P[i], P[i+1])_{(key,M) \sim \mathcal{V}, P \leftarrow M \times L} \\
&= (key, P[1], \ldots, P[i], \langle key, \mathbf{P[i+1]} \rangle = \mathbf{0})_{(key,M) \sim \mathcal{V}, P \leftarrow M \times L}
\end{aligned}$$

given $(key, P[1], \ldots, P[i+1])$, the marginal distributions of $(P[i+2], \ldots, P[\ell])$ and of w in $\mathcal{H}_{i+1}$ are identical to $\mathcal{H}_i$. Thus, the only difference between $\mathcal{H}_i$ and $\mathcal{H}_{i+1}$ is that in $\mathcal{H}_{i+1}$ we add an extra condition on *key* to be in the kernel of $P[i+1]$.

Re-examining $\mathcal{H}_i^+$, by definition $\mathcal{H}_i$ is the marginal distribution of $\mathcal{H}_i^+$ on (key, w). We now conclude also that $\mathcal{H}_{i+1}$ is the marginal distribution on (key, w) in $\mathcal{H}_i^+$ conditioned on $\langle key, P[i+1] \rangle = 0$. Thus $\Delta(\mathcal{H}_i, \mathcal{H}_{i+1}) \leq 2\Delta(\mathcal{H}_i^+, \mathcal{H}_i^R)$.

It remains to bound $\Delta(\mathcal{H}_i^+, \mathcal{H}_i^R)$. We know that in both these distributions, given $(P[1], \ldots, P[i])$ (without w), we have that *key* and $P[i+1]$ are drawn independently and the joint distribution of $(key, P[i+1])$ has entropy at least $(1.85\kappa - i) \geq 1.75\kappa$. This is simply by the min-entropy of $\mathcal{V}$. By Lemma 3.8, with all but $2^{-0.05\kappa}$ probability over the choice of w, the min-entropy of $(key, P[i+1])$ given also w (of length at most 0.1κ) is at least 1.6κ.

We conclude, by Lemma 3.7, that with all but $2^{-0.05\kappa}$ probability over $w \sim \mathcal{H}_i$, it is the case that with all but $2^{-0.05\kappa}$ probability over *key* conditioned on w, the inner product of *key* and $P[i+1]$ (given (key, w)) is $2^{-0.05\kappa}$-close to uniform. In particular, when this is the case, with all but $2 \cdot 2^{-0.05\kappa}$ probability over $(key, w) \sim \mathcal{H}_i$, we have that the probabilities of (key, w) by $\mathcal{H}_i$ and by $\mathcal{H}_{i+1}$ differ by at most a $\exp(1.5 \cdot 2^{-0.05\kappa})$ multiplicative factor. The claim follows. ∎

Proof of Claim 5.14. $\mathcal{V}$ is α-close to IuO. Let $\mathcal{V}'$ be an IuO distribution s.t. $\Delta(\mathcal{V},\mathcal{V}') \leq \alpha$. Let $\mathcal{K}'$ and $\mathcal{M}'$ be the marginal distributions of $\mathcal{V}'$ on *key* and M (respectively). Now take:

$$\begin{aligned}
\mathcal{Z}' &\triangleq (key, M, w)_{(key,M)\sim\mathcal{V}', \quad w\leftarrow\mathcal{A}(key,M),} \\
&= (key, \mathbf{M}, w)_{(key,\mathbf{M}')\sim\mathcal{V}', \quad w\leftarrow\mathcal{A}(key,M'),\mathbf{M}\sim(\mathcal{M}'|key,\mathcal{A}(key,M)=w)} \\
\mathcal{Z}'' &\triangleq (key, \mathbf{M}, w)_{key\sim\mathcal{K}',\mathbf{M}'\sim\mathcal{M}',w\leftarrow\mathcal{A}(key,M'),\mathbf{M}\sim(\mathcal{M}'|key,\mathcal{A}(key,M)=w)}
\end{aligned}$$

Let $\mathcal{Z}'(w)$ and $\mathcal{Z}''(w)$ be the marginal distributions of $\mathcal{Z}'$ and $\mathcal{Z}''$ (respectively) on (key, M), conditioned on $\mathcal{A}(key, M) = w$. Note that $\mathcal{Z}'(w)$ is also the conditional distribution of $\mathcal{V}'$ (conditioned on w). By Claim 5.13, we know that with all but δ probability over $w \sim \mathcal{Z}'$ we have that $\Delta(\mathcal{Z}'(w), \mathcal{Z}''(w)) \leq \delta$. Claim 5.13 shows this is true for the marginal distributions on (key, w), but in $\mathcal{Z}'$ and $\mathcal{Z}''$, the matrix M is just a probabilistic function of (key, w), and so the bound on the statistical distance holds also when M is added to the output.

We claim that (for any w), the distribution $\mathcal{Z}''(w)$ is (perfectly) independent up to orthogonality. This is because in $\mathcal{Z}''$, the leakage w is computed as multi-source leakage on independently drawn *key* and M. Thus, conditioning $\mathcal{Z}''$ on w is conditioning $\mathcal{Z}''$ on (key, M) falling in a product set $S_{key} \times S_M$. We know that $\mathcal{Z}''$ is (perfectly) independent up to orthogonality, and so conditioning $\mathcal{Z}''$ on a product set $S_{key} \times S_M$ will also yield a distribution that is independent up to orthogonality.

We conclude that, with all but δ probability over $w \sim \mathcal{Z}'$, we have that $\Delta(\mathcal{Z}'(w), \mathcal{Z}''(w)) \leq \delta$ and $\mathcal{Z}''(w)$ is independent up to orthogonality. Let W_{bad} be the set of "bad" w's for which $\Delta(\mathcal{Z}'(w), \mathcal{Z}''(w)) > \delta$. Since $\Delta(\mathcal{V}, \mathcal{V}') \leq \alpha$, we know that:

$$\begin{aligned}
&\Pr_{w\sim\mathcal{V}} \quad [w \in W_{bad}] \leq \alpha + \delta \\
&\Pr_{w\sim\mathcal{V}} \quad [\Delta(\mathcal{V}(w), \mathcal{V}'(w)) \geq (\alpha/\beta)] \leq \beta
\end{aligned}$$

where the second equation follows by Markov's inequality. We conclude (by a union bound, and since $\mathcal{V}'(w) = \mathcal{Z}'(w)$), that with all but $(\alpha + \beta + \delta)$ probability over $w \sim \mathcal{V}$, we have that $\mathcal{V}(w)$ is $((\alpha/\beta) + \delta)$-close to $Z''(w)$ and to independence up to orthogonality. ■

5.3.3 Piecemeal Leakage Resilience: Jointly with a Vector

In this section, we show further security properties of random matrices under piecemeal leakage. We focus on piecemeal leakage that operates jointly on (each piece of) a matrix and a vector (and separately on *key*). The matrix will always have columns that are (random) in the kernel of *key*. We show that the leakage is statistically close in the cases where the vector is and is not in the kernel. Moreover, this statistical closeness is *strong* and holds even if the matrix is later released *in its entirety*. The proof is based on Lemma 5.10 (piecemeal leakage resilience of random matrices) and on a "pairwise independence" property under piecemeal leakage, stated separately in Claim 5.16 below.

Lemma 5.15 (Strong Resilience to Matrix-Vector Piecemeal Leakage)**.** *Take $a, \kappa, m \in \mathbb{N}$, where $m \geq \kappa$. Fix $\ell = 0.1\kappa$, and $\lambda = 0.01\kappa/a^2$. Let $\vec{Lin} = (Lin_1, \ldots, Lin_a)$ be any sequence of collections of coefficients for linear combinations, where for each i, $Lin_i \in \{0,1\}^{m\times\ell}$ has full rank ℓ. Let $\mathcal{A}$ be any piecemeal leakage adversary. Take Real and Simulated to be the following two distributions:*

$$\begin{aligned}
Real &= \left(key, M, \mathcal{A}^{\lambda}_{\kappa,\ell,m,\vec{Lin}}(key, (M, \vec{v}))\right)_{key\in_R\{0,1\}^\kappa, M\in_R\{0,1\}^{\kappa\times m}:\forall i, M[i]\in kernel(key), \vec{v}\in_R kernel(key)} \\
Simulated &= \left(key, M, \mathcal{A}^{\lambda}_{\kappa,\ell,m,\vec{Lin}}(key, (M, \vec{v}))\right)_{key\in_R\{0,1\}^\kappa, M\in_R\{0,1\}^{\kappa\times m}:\forall i, M[i]\in kernel(key), \vec{v}\in_R \overline{kernel(key)}}
\end{aligned}$$

then $\Delta(Real, Simulated) \leq 3a \cdot 2^{-0.01\kappa/a}$.

Proof. We define the "midpoint" distribution:

$$\mathcal{D} = 1/2 \cdot Real + 1/2 \cdot Simulated = (key, M, w = \mathcal{A}(key, (M, \vec{v})))_{key, M, \vec{v} \in_R \{0,1\}^\kappa}$$

For fixed (key, M, w), we consider their *bias*:

$$bias(key, M, w) \triangleq \frac{Real[key, M, w] - Simulated[key, M, w]}{\mathcal{D}[key, M, w]}$$

And note that (by definition):

$$\Delta(Real, Simulated) = \mathbb{E}_{(key,M,w)\sim\mathcal{D}}[|bias(key, M, w)|]/2 \tag{1}$$

Thus we focus on bounding $\mathbb{E}_{(M,w)\sim H}[|bias(key, M, w)|]$. We will use a "pairwise independence" property of matrices under piecemeal leakage.

Claim 5.16 (Pairwise Independence under Piecemeal Leakage)**.** *Take* $a, \kappa, m, \ell, \lambda, \vec{Lin}, \mathcal{A}$ *as in Lemma 5.16. Let* $\mathcal{F}$ *and* $\mathcal{F}'$ *be the following distributions. In both* $\mathcal{F}$ *and* $\mathcal{F}'$*, take* $key \in_R \{0,1\}^\kappa$*, a matrix* $M \in_R \{0,1\}^{\kappa \times m}$ *s.t. all of* M*'s columns are in the kernel of* key*. Choose* $\vec{v}_1, \vec{v}_2 \in_r \{0,1\}^\kappa$ *s.t.* $\mathcal{A}(key, (M, \vec{v}_1) = \mathcal{A}(key, (M, \vec{v}_2))$.

$$\begin{aligned} \mathcal{F} &= (\vec{v}_1, \vec{v}_2, b_1, b_2, \mathcal{A}(key, (M, \vec{v}_1)))_{key, M, \vec{v}_1, \vec{v}_2, \mathbf{b_1} = \langle key, \vec{\mathbf{v}_1} \rangle, \mathbf{b_2} = \langle key, \vec{\mathbf{v}_2} \rangle} \\ \mathcal{F}' &= (\vec{v}_1, \vec{v}_2, b_1, b_2, \mathcal{A}(key, (M, \vec{v}_1)))_{key, M, \vec{v}_1, \vec{v}_2, \mathbf{b_1}, \mathbf{b_2} \in_R \{0,1\}} \end{aligned}$$

then $\Delta(\mathcal{F}, \mathcal{F}') \leq \delta = 5a^2 \cdot 2^{-0.03\kappa/a}$.

The proof of Claim 5.16 is below.

We will show that if $\mathbb{E}_{(M,w)\sim H}[|bias(key, M, w)|]$ is too high, then we can predict the inner products of $\vec{v}_1, \vec{v}_2$ as above with key and distinguish $\mathcal{F}$ and $\mathcal{F}'$ (a contradiction to Claim 5.16). We do this by considering a distinguisher $\mathcal{DIS}$ that gets $(\vec{v}_1, \vec{v}_2, b_1, b_2, w)$ (where $(\vec{v}_1, \vec{v}_2, w)$ are distributed as in both $\mathcal{F}$ and $\mathcal{F}'$), and attempts to distinguish whether $b_1, b_2 \in \{0,1\}$ are uniformly random (distribution $\mathcal{F}'$), or are the inner products of $\vec{v}_1, \vec{v}_2$ with key (distribution $\mathcal{F}$). The distinguisher $\mathcal{DIS}$ outputs 1 if $b_1 = b_2$ and outputs 0 otherwise. By Claim 5.16, the advantage of (any distinguisher, and in particular also of) $\mathcal{DIS}$ is bounded by $\delta = 6a^2 \cdot 2^{-0.03\kappa}$.

For distribution $\mathcal{F}'$, the bits b_1, b_2 are independent uniform bits, and so the probability that $\mathcal{DIS}$ outputs 1 is exactly 1/2. In distribution $\mathcal{F}$, however, if $\mathbb{E}_{(M,w)\sim\mathcal{D}}[|bias(key, M, w)|]$ is high then $\mathcal{DIS}$ will output 1 with significantly higher probability (this gives a bound on the expected magnitude of the bias).

To see this, fix (key, M). For a possible leakage value $w \in \{0,1\}^{a \cdot \lambda}$, denote by $p_{key,M,w}$ the probability of leakage w given key and M (for $(key, M, \vec{v}) \sim \mathcal{D}$). Conditioning $\mathcal{D}$ on (key, M), the probability of identical leakage from uniformly random $\vec{v}_1$ and $\vec{v}_2$ is the "collision probability" $cp(key, M) \triangleq \sum_{w \in \{0,1\}^{a \cdot \lambda}} p^2_{key,M,w}$. Conditioning $\mathcal{D}$ on (key, M) and identical leakage from $\vec{v}_1$ and $\vec{v}_2$, the probability that the leakage is some specific value w is exactly $p^2_{key,M,w}/cp(key, M)$. Conditioning $\mathcal{D}$ on (key, M) and identical leakage w from $\vec{v}_1, \vec{v}_2$, the probability that the inner products of $\vec{v}_1$ and $\vec{v}_2$ with key are equal and $\mathcal{DIS}$ outputs 1 is exactly $1/2 + 2|bias(key, M, w)|^2$

(notice that the advantage over 1/2 is always "in the same direction"). Since (by Claim 5.16) the advantage of $\mathcal{DIS}$ is at most δ, we get that:

$$\begin{aligned}\delta &\geq E_{key,M}\left[\mathcal{DIS}\text{'s advantage in outputting 1 given } (key, M)\right] \\ &= E_{key,M}\left[\sum_{w\in\{0,1\}^{a\cdot\lambda}} (p^2_{key,M,w}/cp(key,M)) \cdot 2|bias(key,M,w)|^2\right]\end{aligned}$$

Now because $cp(key, M) \geq 2^{-a\cdot\lambda}$, we get that:

$$E_{key,M}\left[\sum_{w\in\{0,1\}^{a\cdot\lambda}} p^2_{key,M,w} \cdot 2|bias(key,M,w)|^2\right] \leq 2^{a\cdot\lambda}\cdot\delta \tag{2}$$

We also have that:

$$\begin{aligned}2\Delta(Real, Simulated) &= E_{(key,M,w)\sim\mathcal{H}}\left[|bias(key,M,w)|\right] \\ &= E_{key,M}\left[\sum_{w\in\{0,1\}^{a\cdot\lambda}} p_{key,M,w}\cdot|bias(key,M,w)|\right] \\ &\leq \sqrt{2^{a\cdot\lambda}\cdot E_{key,M}\left[\sum_{w\in\{0,1\}^{a\cdot\lambda}} p^2_{key,M,w}\cdot|bias(key,M,w)|^2\right]}\end{aligned}$$

where the last inequality is by Cauchy-Schwartz. Putting this together with Equation 2, we get:

$$\Delta(Real, Simulated) \leq 2^{a\cdot\lambda}\cdot\sqrt{\delta} < 3a\cdot 2^{-0.01\kappa/a}$$

which completes the proof. ■

Proof of Claim 5.16. Consider the following distribution $\mathcal{E}$, where *key* is uniformly random, M is a uniformly random matrix with columns in *key*'s kernel, and $\vec{v}_1, \vec{v}_2$ are uniformly random pair s.t. $\mathcal{A}(key, (M, \vec{v}_1)) = \mathcal{A}(key, (M, \vec{v}_2))$:

$$\mathcal{E} = (key, \vec{v}_1, \vec{v}_2, \mathcal{A}(key, (\mathbf{M}, \vec{v}_1)))_{key,\mathbf{M}\in_R\{0,1\}^{\kappa\times m}:\forall i, M[i]\in kernel(key),\vec{v}_1,\vec{v}_2}$$

Consider also the distribution $\mathcal{H}$ that uses a uniformly random matrix M of rank $\kappa - 1$:

$$\mathcal{H} = (key, \vec{v}_1, \vec{v}_2, \mathcal{A}(key, (\mathbf{M}, \vec{v}_1)))_{key,\mathbf{M}\in_R\{0,1\}^{\kappa\times m}:rank(M)=\kappa-1,\vec{v}_1,\vec{v}_2}$$

We will show that:

1. $\Delta(\mathcal{E}, \mathcal{H}) < 5a^2\cdot 2^{-0.03\kappa/a}$, this will follow by piecemeal leakage resilience (Lemma 5.10).
2. In $\mathcal{H}$, the advantage in distinguishing $(\langle key, \vec{v}_1\rangle, \langle key, \vec{v}_2\rangle)$ from uniformly random unbiased bits is bounded by $2^{-0.1\kappa+3}$. I.e., in $\mathcal{H}$ the inner products of $\vec{v}_1$ and $\vec{v}_2$ with *key* are (close to) pairwise independent.

The claim will follow from the two items above (we assume $2^{-0.1\kappa+3} \leq a^2\cdot 2^{-0.03\kappa/a}$).

Item 1, $\mathcal{E}$ and $\mathcal{H}$ are close. Let $\mathcal{A}$ be an adversary for which we get $\varepsilon = \Delta(\mathcal{E}, \mathcal{H})$. Given $\mathcal{A}$, we show a piecemeal leakage attack $\mathcal{A}'$ on (key, M) a la Lemma 5.10. We show that if $\mathcal{A}$ has advantage ε in distinguishing $\mathcal{E}$ and $\mathcal{H}$, then $\mathcal{A}'$ has advantage ε' (where $\varepsilon' \geq \varepsilon \cdot 2^{-a \cdot \lambda}$) in distinguishing whether M is in key's kernel or M is independent of key. By Lemma 5.10, we conclude a bound on ε' and (through it) on ε.

The piecemeal leakage attack $\mathcal{A}'$ proceeds as follows. The adversary chooses two uniformly random vectors $\vec{v}_1, \vec{v}_2 \in_R \{0,1\}^\kappa$. It then computes piecemeal leakage $\mathcal{A}(key, (M, \vec{v}_1))$, and also computes whether $\mathcal{A}(key, (M, \vec{v}_1)) = \mathcal{A}(key, (M, \vec{v}_2))$ (for the randomly chosen $\vec{v}_1, \vec{v}_2$). This requires $(\lambda + 1)$ bits of piecemeal leakage from key and (each piece of) M (it takes λ bits to determine the leakage from each piece $\vec{v}_1$ and an extra bit to tell whether the leakage on $\vec{v}_2$ is identical). If the leakage from $\vec{v}_1$ and $\vec{v}_2$ is identical, we output

$$\mathcal{A}'(key, M) = (\vec{v}_1, \vec{v}_2, \mathcal{A}(key, (M, \vec{v}_1))$$

Otherwise, we output $\mathcal{A}'(key, M) = \perp$. Observe now that, conditioning on $\mathcal{A}(key, (M, \vec{v}_1)) = \mathcal{A}(key, (M, \vec{v}_2))$, we have that the output of $\mathcal{A}'$ on M with columns in key's kernel (together with key) is exactly the distribution $\mathcal{E}$. The output of $\mathcal{A}'$ on M that is independent of key (conditioned on identical leakage from $\vec{v}_1, \vec{v}_2$, and together with key) is distributed exactly as $\mathcal{H}$. In both cases, when the leakage from $\vec{v}_1, \vec{v}_2$ is *not* identical, the output is simply $\perp$. We conclude that the statistical distance ε' between the output of $\mathcal{A}'$ in both cases (M in the kernel and independent M) is at least ε multiplied by the probability that the leakage on $\vec{v}_1$ and $\vec{v}_2$ is identical (say w.l.o.g. we refer to the "leakage collision" probability for M in the kernel).

For any fixed (key, M), the probability that we get identical leakage on $\vec{v}_1$ and $\vec{v}_2$ chosen uniformly at random is at least the inverse of the total amount of possible leakage values. I.e. at least $2^{-a \cdot \lambda}$. This gives a lower bound on ε' as a function of ε. By Lemma 5.10 we also have an upper bound on ε'. Putting these together:

$$\varepsilon \cdot 2^{-a \cdot \lambda} \leq \varepsilon' \leq 5a^2 \cdot 2^{-0.04\kappa}$$

we conclude that:

$$\Delta(\mathcal{E}, \mathcal{H}) \leq 5a^2 \cdot 2^{-0.04\kappa} \cdot 2^{a \cdot \lambda} = 5a^2 \cdot 2^{-0.03\kappa}$$

Item 2, $\mathcal{H}$ is pairwise independent. Consider the piecemeal leakage in $\mathcal{H}$ as a multi-source leakage attack on key and on $(\vec{v}_1, \vec{v}_2)$ (chosen conditioned on $\vec{v}_1$ and $\vec{v}_2$ yielding the same leakage). For any fixed M, the amount of leakage from key in the attack is bounded by $0.01\kappa/a$. In particular, by Lemma 3.8 we have that, given the leakage, with all but $2^{-0.1\kappa}$ probability, key is an independent sample in a source with min-entropy at least 0.85κ.

We now consider $(\vec{v}_1, \vec{v}_2)$. We claim that (for any fixed (key, M)) with all but $2^{-0.1\kappa}$ probability over the choice of $\vec{v}_1, \vec{v}_2$ yielding the same leakage, the set of vectors yielding the same leakage as $\vec{v}_1$ and $\vec{v}_2$ is of size at least $2^{0.85\kappa}$. To see this, for a vector $\vec{v}$, let $S(\vec{v})$ be the set of vectors that give the same leakage as $\vec{v}$. Let S_{bad} be the set of all vectors $\vec{v}$ for which $S(\vec{v})$ is of size less than $2^{-0.85\kappa}$. By Lemma 3.8 we get that:

$$\alpha = \Pr_{\vec{v} \in_R \{0,1\}^\kappa}[\vec{v} \in S_{bad}] \leq 2^{-0.1\kappa}$$

The probability that $\vec{v}_1, \vec{v}_2$ drawn s.t. their leakage is identical both land in S_{bad} is at most α^2 divided by the total probability that the leakage from uniformly random $\vec{v}_1, \vec{v}_2$ is identical (the

"collision probability"). The total leakage is of bounded length $a \cdot \lambda$, so the collision probability is at least $2^{-a \cdot \lambda}$. We conclude that:

$$\Pr_{\vec{v}_1, \vec{v}_2 \in_R \{0,1\}^\kappa : \mathcal{A}(key,(M,\vec{v}_1)) = \mathcal{A}(key,(M,\vec{v}_2))} [\vec{v}_1, \vec{v}_2 \in S_{bad}] \leq \alpha^2 \cdot 2^{a \cdot \lambda} < 2^{-0.1\kappa}$$

We conclude that with all but $2 \cdot 2^{-0.1\kappa}$ probability, given the leakage, the random variables $key, \vec{v}_1, \vec{v}_2$ are independent and each of min entropy at least 0.85κ. By Lemma 3.7, we conclude that the joint distribution of inner products of $\vec{v}_1$ and $\vec{v}_2$ with key is at statical distance $2^{-0.1\kappa+3}$ from uniformly random (or pairwise independent). ■

5.4 Piecemeal Matrix Multiplication: Security

In this section we use security of random matrices under piecemeal leakage to prove several security properties for piecemeal matrix multiplication. These will serve as building blocks for proving the security of the ciphertext bank as a whole (see the lemmas in Section 5.1). The proofs follow from the lemmas above, and are omitted.

Lemma 5.17. *Take $\kappa, m, n \in \mathbb{N}$ s.t. $m, n \geq \kappa$. Set $\ell = 0.1\kappa$ and leakage bound $\lambda = 0.01\kappa \cdot (\ell/m)^2$. Let $\mathcal{A}$ be any piecemeal adversary and $\mathcal{A}'$ any leakage adversary. Let $\mathcal{D}$ and $\mathcal{F}$ be the following two distributions, where in both cases we draw $key \in_R \{0,1\}^\kappa$, $\vec{x} \in_R \{0,1\}^m$ and $B \in_R \{0,1\}^{m \times n}$ s.t. the columns of B are all in the kernel of $\vec{x}$ and with parity 1.*

$$\begin{aligned}
\mathcal{D} &= (key, C, w \leftarrow \mathcal{A}^\lambda_{\kappa,\ell,m,Lin}(key, \mathbf{A}), \\
&\qquad \mathcal{A}'^\lambda(w, \vec{x}, B)[key, C \leftarrow PiecemealMM(\mathbf{A}, B)])_{\mathbf{A} \in_R \{0,1\}^{\kappa \times m} : \forall i, \langle key, A[i] \rangle = 0} \\
\mathcal{F} &= (key, C, w \leftarrow \mathcal{A}^\lambda_{\kappa,\ell,m,Lin}(key, \mathbf{A}), \\
&\qquad \mathcal{A}'^\lambda(w, \vec{x}, B)[key, C \leftarrow PiecemealMM(\mathbf{A}, B)])_{\mathbf{A} \in_R \{0,1\}^{\kappa \times n} : \forall i, \langle key, A[i] \rangle = \vec{x}[i]}
\end{aligned}$$

then $\Delta(\mathcal{D}, \mathcal{F}) = \exp(-\Omega(\kappa))$.

Lemma 5.18. *Take $\kappa, m, n \in \mathbb{N}$ s.t. $m, n \geq \kappa$. Set $\ell = 0.1\kappa$ and leakage bound $\lambda = 0.01\kappa \cdot (\ell/m)^2$. Let $\mathcal{A}$ be any (computationally unbounded) leakage adversary. Let $\mathcal{D}$ and $\mathcal{F}$ be the following two distributions, where in both distributions $key \in_r \{0,1\}^\kappa$, $\vec{x} \in_R \{0,1\}^{2\kappa}$, $A \in_R \{0,1\}^{\kappa \times m}$ s.t. the i-th column of A has inner product $\vec{x}[i]$ with key:*

$$\begin{aligned}
\mathcal{D} &= (key, A, w \leftarrow \mathcal{A}^\lambda_{\kappa,\ell,m,Lin}(key, A), \\
&\qquad \mathcal{A}'^\lambda(w)[key, \vec{c} \leftarrow PiecemealMM(A, \vec{r})])_{\vec{r} \in_R \{0,1\}^{m \times 1} : (\oplus_i \vec{r}[i]) = 1, \langle \vec{x}, \vec{r} \rangle = 0} \\
\mathcal{F} &= (key, A, w \leftarrow \mathcal{A}^\lambda_{\kappa,\ell,m,Lin}(key, A), \\
&\qquad \mathcal{A}'^\lambda(w)[key, \vec{c} \leftarrow PiecemealMM(A, \vec{r})])_{\vec{r} \in_R \{0,1\}^{m \times 1} : (\oplus_i \vec{r}[i]) = 1, \langle \vec{x}, \vec{r} \rangle = 1}
\end{aligned}$$

then $\Delta(\mathcal{D}, \mathcal{F}) = \exp(-\Omega(\kappa))$.

Lemma 5.19. *Take $\kappa, m, n \in \mathbb{N}$ s.t. $m, n \geq \kappa$. Set $\ell = 0.1\kappa$ and leakage bound $\lambda = 0.01\kappa \cdot (\ell/m)^2$. Let $\mathcal{A}$ be any (computationally unbounded) leakage adversary. Let $\mathcal{D}$ and $\mathcal{F}$ be the following two distributions, where in both distributions $key \in_r \{0,1\}^\kappa$ and $A \in_R \{0,1\}^{\kappa \times m}$:*[9]

$$\begin{aligned}
\mathcal{D} &= (key, C, \mathcal{A}^\lambda(key, A)[key, C \leftarrow PiecemealMM(A, \mathbf{B})])_{\mathbf{B} \in_R \{0,1\}^{m \times n} : \forall i, \oplus B^T[i] = 1} \\
\mathcal{F} &= (key, C, \mathcal{A}^\lambda(key, A)[key, C \leftarrow PiecemealMM(A, \mathbf{B})])_{\mathbf{B} \in_R \{0,1\}^{m \times n} : rank(B) = m-1, \forall i, \oplus B^T[i] = 1}
\end{aligned}$$

[9] In both distributions, we give $\mathcal{A}$ complete and explicit access to *key* and A. The piecemeal leakage attack here is on B, which has different distributions in the two cases.

then $\Delta(\mathcal{D}, \mathcal{F}) = \exp(-\Omega(\kappa))$.

5.5 Ciphertext Bank Security Proofs

We now prove Lemmas 5.1, 5.2, 5.3 , 5.4 and 5.5 from Section 5.1 (Lemma 5.1 is the most technically involved). These Lemmas consider leakage produced in an attack on a real or simulated sequence of T ciphertext generations. In proving statistical closeness of the leakage, we will use both the simulated views and additional hybrid views. We compute these views by running the T generations, under the leakage attack of $\mathcal{A}$, using biased random coins.

Internal Variables. We will use several internal variables as we run these T generations. For the t-th generation (where i goes from 0 to $T-1$): (key_i, C_i) denote the bank before the i-th generation, with underlying plaintexts $\vec{x}_i$. The randomness used to generate the i-th output ciphertext is $\vec{r}_i$, the matrix used to refresh the bank is R_i, and the key refresh value is σ_i. We use D_i to denote the intermediate ciphertext in the i-th generation after key refresh, but before multiplication with R_i. The output ciphertext of the i-th generation is $\vec{c}_i$ (the output key is key_i).

Proof of Lemma 5.1. We prove here the case that $b = 0$, the case $b = 1$ is similar. Recall that *Real* is the view of $\mathcal{A}$ given "real" generations of ciphertexts, using a bank of ciphertexts whose underlying plaintexts are 0, and generating ciphertexts whose underlying plaintexts are 0. *Simulated* is a view generated using a bank of ciphertexts whose underlying plaintexts are uniformly random, but choosing plaintexts using biased randomness so that their underlying plaintexts are always 0. The proof of statistical closeness uses a hybrid argument as follows.

We define hybrid views $\{\mathcal{H}_t\}$ for $t \in \{0, \ldots, T+1\}$. The output of each hybrid is T tuples, one for each ciphertext generation, each consisting of a key, a ciphertexts, and a leakage value. We compute the hybrids views by running the T generations, under the leakage attack of $\mathcal{A}$, using biased random coins. We specify the distribution of each of the internal variables described above, and these specify the hybrid view on the outputs and leakage from the T generations.

When generating $\mathcal{H}_t$, for $t > 0$ we initialize (key_0, C_0) as in *Simulated*. In $\mathcal{H}_0$ we initialize (key_0, C_0) as in *Real*. We then run T ciphertext generations under $\mathcal{A}$'s leakage attack. The key refresh value σ_i is always uniformly random. For the i-th generation, where $i \leq t$, we choose $\vec{r}_i$ uniformly at random s.t. it has odd parity and is in $kernel(\vec{x}_i)$. For $i \geq t$, we choose $\vec{r}_i$ to be uniformly random with odd parity (and no further restrictions). For $i \neq (t-1)$, we use a uniformly random R_i whose columns have odd parity. For $i = (t-1)$, we use a uniformly random R_i whose columns have odd parity and are in $kernel(\vec{x}_i)$. This completes the hybrids' specification.

By construction, we get that $\mathcal{H}_0 = Real$ and $\mathcal{H}_{T+2} = Simulated$. It remains to show that, for all t, $\Delta(\mathcal{H}_t, \mathcal{H}_{t+1}) = \exp(-\Omega(\kappa))$. We show this here for $2 < t < T$ (the borderline cases are handled similarly). We use an intermediate distribution $\mathcal{H}'_t$, which operates as $\mathcal{H}_t$, except that it chooses a $\vec{x}_t$ vector uniformly at random (recall that in $\mathcal{H}_t$ the columns of C_t are all in $kernel(key)$). It then chooses $\vec{r}_t$ and the columns of R_t to be uniformly random with odd parity and in $kernel(\vec{x}_t)$ (whereas in $\mathcal{H}_t$ these were uniformly random with odd parity and no further restriction).

Claim 5.20. $\Delta(\mathcal{H}'_t, \mathcal{H}_{t+1}) = \exp(-\Omega(\kappa))$

Proof. The differences between $\mathcal{H}'_t$ and $\mathcal{H}_{t+1}$ are: (i) the distribution of $\vec{r}_{t-1}$ and the columns of R_{t-1}: they are uniform (with odd parity) in $\mathcal{H}_{t+1}$, but in $\mathcal{H}'_t$ they are all in $kernel(\vec{x}_{t-1})$, (ii) in $\mathcal{H}'_t$ we have that the columns of C_t are orthogonal to key_t, whereas in $\mathcal{H}_{t+1}$ they are uniformly

random, and (*iii*) the distribution of $\vec{r}_t$ and the columns of R_t: they have odd parity in both $\mathcal{H}_{t+1}$ and $\mathcal{H}'_t$, but in $\mathcal{H}_{t+1}$ they are orthogonal to $\vec{x}_t$ that has the plaintext bits encrypted in C_t, whereas in $\mathcal{H}'_t$ they are orthogonal to a uniformly random $\vec{x}_t$ that is independent of (key_t, C_t).

We reduce the security game of Lemma 5.17 to distinguishing these two cases. There, a vector $\vec{x}$ is chosen uniformly at random. A matrix A either has columns orthogonal to key, or has uniformly random columns whose inner products with key equal the bits in $\vec{x}$. This A is multiplied by B with columns in the kernel of $\vec{x}$. To reduce the game of Lemma 5.17 to distinguishing $\mathcal{H}'_t$ and $\mathcal{H}_{t+1}$, we put key as key_t, A as C_t, $\vec{x}$ as $\vec{x}_t$, and B as $\mathcal{R}_t$.

We begin by showing that leakage from the t-th generation on, together with all keys and ciphertexts created in all generations, is statistically close in both hybrids. The leakage from the t-th generation takes as input the keys and ciphertexts produced in prior iterations, and so for each $i \in \{0, \ldots, t-1\}$, we pick $(key_i, \vec{c}_i)$ uniformly at random (independent of (key_t, C_t)) s.t. they have inner product 0. We also choose a uniformly random correlation value σ_t. Note that the distribution of these key-ciphertext pairs, in conjunction with (key_t, C_t) set as above, is exactly as in $\mathcal{H}'_t$ and $\mathcal{H}_{t+1}$ (respectively, depending on the distribution of key and A for the security game of Lemma 5.17).

Using the above reduction, we conclude from Lemma 5.17 that the leakage from the t-th generation, together with $(key_t, \vec{c}_t, \sigma_t, C_{t+1})$ (and the list of key-ciphertext pairs from earlier generations), is statistically close when the random variables are drawn as in $\mathcal{H}'_t$ and $\mathcal{H}_{t+1}$. We can then use these to generate the leakage and key-ciphertext pairs for generations $(t+1)$ and up (these are just a function of $(key_t, \sigma_t, C_{t+1})$).

We need, however, to also generate the leakage for the ciphertext generations that precede the t-th. Recall that the $(key_i, \vec{c}_i)$ key-ciphertext pairs for all iterations $i < t$ were already chosen and fixed above. We compute the leakage from these iterations using piecemeal leakage from (key_t, C_t). In fact, for $i \in \{0, \ldots, t-3\}$ the leakage is independent of (key_t, C_t): we simply choose all of the randomness for these generations independently of (key_t, C_t). For generations $\{0, \ldots, t-2\}$, each C_i is sampled uniformly at random. The σ_i values are specified by $key_i \oplus \sigma_i = key_{i+1}$, and these in turn (together with the C_i's) specify the D_i key-refreshed banks. The R_i matrices are uniformly random s.t. their columns have odd parity and multiplying D_i by R_i yields C_{i+1}. $\vec{r}_i$'s are uniformly random s.t. they have odd parity and $C_i \times \vec{r}_i = \vec{c}_i$. This completely specifies the randomness for all iterations $i \in \{0, \ldots (t-3)\}$, and we can compute the leakage from those iterations using these values, independently of (key_t, C_t). Note that the randomness for iterations $t-2$ and $t-1$ *will depend on* (key_t, C_t), and so leakage from those iterations is not independent, and will be computed as follows using piecemeal leakage from (key_t, C_t).

For the $(t-1)$-th generation, we choose D_{t-1} uniformly at random. The variable σ_{t-1} is a function of key_t (can be accessed via leakage) and of key_{t-1} (which is fixed and public). The ciphertext bank C_{t-1} is a function of D_{t-1} and of σ_{t-1}. I.e. of public information and of (leakage from) key_t. The variable $\vec{r}_{t-1}$ is a function of C_{t-1} and $\vec{c}_{t-1}$, i.e. of public information and (leakage from) key_t. The only remaining variable which is not specified for iteration $t-1$ is R_{t-1}. We will show below how to compute the needed (piecemeal) leakage from R_{t-1} using D_{t-1} and *piecemeal leakage only* from C_t. Given this (see below), we conclude that leakage from each sub-computation in the $(t-1)$-th generation can be computed via piecemeal leakage from (key_t, C_t).

To compute each piece of R_{t-1} used in the piecemeal matrix multiplication, we observe that it suffices to use explicit access to all of D_{t-1} (a "public" uniformly random matrix), together with piecemeal leakage from C_t. We use here the fact that the pieces of R_{t-1} that are needed for

simulating matrix multiplication are all disjoint. Note that, in particular, the distributions of R_{t-1} that we will get in the scenarios of Lemma 5.17 are quite different (as they should be).

Finally, we also need to compute leakage from the $(t-2)$-th generation. Here we need to specify σ_{t-2}, which is a function of key_{t-2} and key_{t-1}: i.e., we can access is via leakage from key_t. This also specifies D_{t-2}. Finally, for R_{t-2} we use D_{t-2} and C_{t-1}, which can both be accessed via leakage from key_t.

In conclusion, we used a piecemeal attack on (key_t, C_t) to generate the key-ciphertext pairs and leakage up to the t-th generation, and an attack as in Lemma 5.17 to generate the leakage from the t-th generation on. This yielded the views $\mathcal{H}'_t$ and $\mathcal{H}_{t+1}$. By Lemma 5.17 we conclude that these views must be statistically close. ■

Claim 5.21. $\Delta(\mathcal{H}_t, \mathcal{H}'_t) = \exp(-\Omega(\kappa))$

Proof. The only difference between the hybrids is in the distribution of R_t (in the t-th generation). We reduce the attack game of Lemma 5.19 to distinguishing the two cases. To do so, we generate $(key_i, \vec{c}_i, C_i, w_i)_{i<t}$ identically as in both views. We put $key_{t+1} = key$, $D_t = A$, and $R_t = B$.

Now consider the t-th matrix update in $\mathcal{H}_t$ or $\mathcal{H}'_t$, performed via piecemeal multiplication of D_t with R_t. In $\mathcal{H}_t$ we have a uniformly random R_t whose columns have odd parity, and in $\mathcal{H}'_t$ we place the additional restriction that the columns are all orthogonal to $kernel(\vec{x}_t)$: i.e. they are all in a (random) subspace of dimension $(2\kappa - 1)$. By Lemma 5.19, the leakage obtained, together with (key_{t+1}, C_{t+1}), is statistically close in both cases. In both views, we can create the leakage from later rounds as a function of (key_{t+1}, C_{t+1}) (the same function in both cases). We can also create the leakage in the earlier rounds as a function of C_t, key_t as in Claim 5.20 (here this is even easier, because we have explicit access to both). ■

■

Proof of Lemma 5.2. We prove here the case that $b = 0$, the case $b = 1$ is similar. Recall that *Simulated′* and *Simulated″* are two simulated views of $\mathcal{A}$ on a sequence of T simulated generations, both using a bank of ciphertexts whose underlying plaintexts are all uniformly random. The views differ only in that the plaintexts underlying the ciphertexts that are generated, $\vec{b}'$ and $\vec{b}''$, might be different. The proof of statistical closeness uses a hybrid argument and follows below.

We define hybrid views $\{\mathcal{H}_t\}$ for $t \in \{0, \ldots, T\}$. The output of each hybrid is T leakage values, one for each ciphertext generation. We compute the hybrids views by running the T generations, under the leakage attack of $\mathcal{A}$, using biased random coins. We will use the internal variables described above as we run these T generations. When generating $\mathcal{H}_t$, we initialize (key_0, C_0) using *SimBankInit* (so the ciphertexts in the bank are uniformly random). In $\mathcal{H}_t$, for all i, in the i-th call to *SimBankInit*, we use uniformly random σ_i and R_i whose columns have parity 1. For $i < t$, we choose $\vec{r}_i$ uniformly at random s.t. its parity is 1 and the $\vec{c}_i$ produced has inner product $\vec{b}'[i]$ with key_i. For $i \geq t$, we choose $\vec{r}_i$ similarly, except its inner product with key_i is $\vec{b}''[i]$. This completes the specification of the hybrids.

By construction, we get that $\mathcal{H}_0 = Simulated''$ and $\mathcal{H}_T = Simulated'$. It remains to show that, for all t, $\Delta(\mathcal{H}_t, \mathcal{H}_{t+1}) = \exp(-\Omega(\kappa))$. This follows from Lemma 5.18. The Lemma shows that the leakage obtained in the t-th generation, together with (key_{t+1}, C_{t+1}), is statistically close in $\mathcal{H}_t$ and $\mathcal{H}_{t+1}$. In both views, we can create the leakage from later rounds as a function of (key_{t+1}, C_{t+1})

(the same function in both cases). We can also create the leakage in the earlier rounds using a piecemeal leakage attack on (key_t, C_t), as done in Claim 5.20 above. ■

Proof of Lemma 5.3. The distribution $\mathcal{D}$ of $((key_0, \ldots, key_{T-1}), (\vec{c}_0, \ldots, \vec{c}_{T-1}))$ in *Simulated* (without any leakage) is IuO, with orthogonality $\vec{b}$ and underlying distributions $\mathcal{K}$ and $\mathcal{C}$ on keys and ciphertexts: each key and ciphertext in the underlying distributions is uniformly and independently random. Now, observe that the ciphertext banks in *Simulated* are uniformly random, independent of the keys and ciphertexts. Thus, we can compute the leakage from all T generations as a (randomized) multi-source function operating on the ciphertext banks and separately on the keys and separately on each ciphertext. We conclude by Lemma 3.11 that for each i the distribution $\mathcal{D}_i(w)$ is indeed IuO, with orthogonality $\vec{b}[i]$ and with underlying distributions $\mathcal{K}_i(w)$ and $\mathcal{C}_i(w)$ that do not depend on $\vec{b}[i]$. The entropy bounds on each key and ciphertext (given w) follow by Lemma 3.8.

We note that independence up to orthogonality and high entropy hold even given the explicit lists of ciphertexts in the bank (in all calls), as these are just uniformly random matrices, and even given the random coins used to compute leakage from the target generations (given the ciphertext bank and the target ciphertext). ■

6 Safe Computations

In this section we present the *SafeNAND* procedure, see Section 2.2 for an overview. The (simpler) treatment of duplications gates is omitted.

This section is organized as follows: the *SafeNAND* procedure and its security properties are in Section 6.1. This procedure uses a leakage-resilient permutation procedure, *Permute*, which is presented and proved secure in Section 6.2. We then use *Permute*'s security in the proof of *SafeNAND*'s security, which follows in Section 6.3.

6.1 Safe Computations: Interface and Security

In this section we present the procedure for safely computing NAND gates. The full procedure is in Figure 8. Correctness follows from the description (see the introduction). For security, we show that a view of the NAND computation can be simulated, given only the output (and the underlying distributions of the input keys and the input ciphertexts). This is formalized in Lemma 6.1. See the subsequent sections for details of the *Permute* procedure and the proof of Lemma 6.1.

Security of *SafeNAND*. We provide a simulator for producing the leakage on the *SafeNAND* procedure, when the inputs to *SafeNAND* are chosen from an IuO distribution. The simulator is given a_k, and the underlying distributions for the which the *SafeNAND* inputs were drawn. It outputs a complete view of the leakage from *SafeNAND*. This includes the leakage from the *Decrypt* operation (which loads keys and ciphertexts into memory simultaneously). The security claim is below in Lemma 6.1. We note that *the SafeNAND simulator is not efficient*, its running time might be exponential in that of the leakage adversary. The descriptions of the underlying input distributions themselves might already be of exponential size. This does not pose a problem, because the security of our main construction is statistical, and we never use the *SafeNAND*

$\underline{SafeNAND(a_i, key_i, \vec{c}_i, a_j, key_j, \vec{c}_j, key_k, \vec{c}_k)}$**:** Safe NAND computation

1. Correlate the ciphertexts to a new key. Pick a new key $key \leftarrow KeyGen(1^\kappa)$
   $\sigma_i \leftarrow key_i \oplus key$, $\sigma_j \leftarrow key_j \oplus key$, $\sigma_k \leftarrow key_k \oplus key$
   $c'_i \leftarrow CipherCorrelate(c_i, \sigma_i)$, $c'_j \leftarrow CipherCorrelate(c_j, \sigma_j)$, $c'_k \leftarrow CipherCorrelate(c_k, \sigma_k)$
   leakage on $[(key_i, \sigma_i), (key_j, \sigma_j), (key_k, \sigma_k), (\vec{c}_i, \sigma_i), (\vec{c}_j, \sigma_j), (\vec{c}_k, \sigma_k)]$
2. $c''_i \leftarrow c'_i \oplus (a_i, 0, \ldots, 0)$, $c''_j \leftarrow c'_j \oplus (a_j, 0, \ldots, 0)$
   $C \leftarrow (\vec{c}'_k, \vec{c}'_k \oplus \vec{c}''_i, \vec{c}'_k \oplus \vec{c}''_j, \vec{c}'_k \oplus \vec{c}''_i \oplus \vec{c}''_j \oplus (1, 0, \ldots, 0))$
   leakage on ciphertexts
3. $(K', C') \leftarrow Permute(key, C)$
   leakage from Permute (see below)
4. Decrypt the four ciphertexts in C' using the four keys in K'. If there is one 0 plaintext in the results, then output $a_k \leftarrow 0$. Otherwise, output $a_k \leftarrow 1$
   leakage on C' *and* K' *(jointly)*

Figure 8: *SafeNAND* procedure. The *Permute* procedure is in Figure 9.

simulator for the (efficient) *SimEval* simulation procedure, only for creating hybrid distributions in the security proof.

Lemma 6.1. *There exist: an* exponential time *simulator SimNAND, a leakage bound* $\lambda(\kappa) = \tilde{\Omega}(\kappa)$*, and a distance bound* $\delta(\kappa) = \mathrm{negl}(\kappa)$ *s.t. for every* $\kappa \in \mathbb{N}$ *and leakage adversary* $\mathcal{A}$*:*

Let $\mathcal{D}$ *be a distribution on two 3-tuples: a key-tuple* $(key_i, key_j, key_k) \in \{0,1\}^{3\times\kappa}$*, and a ciphertext-tuple* $(\vec{c}_i, \vec{c}_j, \vec{c}_k) \in \{0,1\}^{3\times\kappa}$*. Suppose that* $\mathcal{D}$ *is IuO with orthogonality* $(b_i, b_j, b_k) \in \{0,1\}^3$*. Let* $\mathcal{D}$*'s underlying distributions on the key-tuple and on the ciphertext-tuple be* $\mathcal{K}$ *and* $\mathcal{C}$*. I.e.* $\mathcal{D} = \mathcal{K} \perp_{(b_i,b_j,b_k)} \mathcal{C}$*. Suppose further that* $H_\infty(\mathcal{K}), H_\infty(\mathcal{C}) \geq 3\kappa - O(\lambda(\kappa))$*.*

For any $(a_i, a_j) \in \{0,1\}^2$*, take:*

$$Real = \left(\mathcal{A}^{\lambda(\kappa)}[a_k \leftarrow SafeNAND(a_i, key_i, \vec{c}_i, a_j, key_j, \vec{c}_j, key_k, \vec{c}_k)]\right)_{((key_i,key_j,key_k),(\vec{c}_i,\vec{c}_j,\vec{c}_k))\sim\mathcal{D}}$$

$$Simulated = (SimNAND(a_i, a_j, a_k, \mathcal{K}, \mathcal{C}))_{a_k \leftarrow (((a_i \oplus b_i)\ NAND\ (a_j \oplus b_j)) \oplus b_k)}$$

then $\Delta(Real, Simulated) \leq \delta(\kappa)$*.*

6.2 Leakage-Resilient Permutation

The *Permute* procedure receives as input a key and a 4-tuple of ciphertexts. It outputs a "fresh" pair of 4-tuples of keys and ciphertexts. The correctness property of the permute procedure is that the plaintexts underlying the output ciphertexts (under the respective output keys) are a (random) permutation of the plaintexts underlying the input ciphertexts. The intuitive security guarantee is that, even to a computationally unbounded leakage adversary, the permutation looks uniformly random. The procedure is below in Figure 9. Correctness is immediate. Security is formalized by the existence of a simulator that generates a complete view of the leakage *and the output keys and ciphertexts*. The simulator only gets: (*i*) descriptions of the marginal distribution from which *key* and the input ciphertext are drawn, and (*ii*) a random permutation of the plaintexts underlying the

input ciphertexts. We show that, under the appropriate conditions on the distribution from which the key and ciphertexts are drawn, the real and simulated joint distributions of leakage and output from *Permute* will be statistically close. In particular, on an intuitive level, the joint distribution of the leakage and the outputs is independent of the permutation that was used. This security property is stated in Lemma 6.2 below. We note that the simulator is not efficient, and may run in exponential time (as was the case for the *SimNAND* simulator, it is only used in the security proof of our main construction).

Permute(key, C)**:** leakage-resilient permutation for key and a 4-tuple C of ciphertexts

Take $K_0 \leftarrow (key, key, key, key), C_0 \leftarrow C$, and $\ell = \text{polylog}(\kappa)$

For $i \in [\ell]$, repeat:

1. for $j \in [\kappa], k \in [4]$: $\sigma_i[j][k] \leftarrow \mathit{KeyEntGen}(1^\kappa)$, $L_i[j][k] \leftarrow \mathit{KeyRefresh}(K_i[k], \sigma_i[j][k])$ *leakage on* (K_i, σ_i)
2. for $j \in [\kappa], k \in [4]$: $D_i[j][k] \leftarrow \mathit{CipherCorrelate}(C_i[k], \sigma_i[j][k])$
   leakage on (C_i, σ_i)
3. for $j \in [\kappa], k \in [4]$: $\tau_i[j][k] \leftarrow \mathit{CipherEntGen}(1^\kappa)$, $D'_i[j][k] \leftarrow \mathit{CipherRefresh}(D_i[j][k], \tau_i[j][k])$
   leakage on (D_i, τ_i)
4. for $j \in [\kappa], k \in [4]$: $L'_i[j][k] \leftarrow \mathit{KeyCorrelate}(L_i[j][k], \tau_i[j][k])$
   leakage on (L_i, τ_i)
5. pick $\pi_i \in_R S_4^\kappa$, for $j \in [\kappa]$: $L''_i[j] \leftarrow \pi_i[j](L'_i[j])$, $D''_i[j] \leftarrow \pi_i[j](D'_i[j])$
   leakage on $[(L'_i, \pi_i), (D'_i, \pi_i)]$
6. pick $j_i^* \in_R [\kappa]$. Save $K_{i+1} \leftarrow L''_i[j_i^*]$, and $C_{i+1} \leftarrow D''_i[j_i^*]$
   leakage on $[(L''_i[j_i^*], j_i^*), (K''_i[j_i^*], j_i^*)]$

Output (K_ℓ, C_ℓ)

Figure 9: Leakage-Resilient Ciphertext Permutation for $\kappa \in \mathbb{N}$

Lemma 6.2. *There exists an* exponential-time *simulator SimPermute, a leakage bound* $\lambda(\kappa) = \tilde{\Omega}(\kappa)$*, and a distance bound* $\delta(\kappa) = \text{negl}(\kappa)$*, s.t for any* $\kappa \in \mathbb{N}$ *and leakage adversary* $\mathcal{A}$*:*

Let $\mathcal{D}$ *be a distribution on key* $\in \{0,1\}^\kappa$ *and a ciphertext 4-tuple* $C \in \{0,1\}^{4\times\kappa}$*. Suppose that* $\mathcal{D}$ *is IuO with orthogonality* $\vec{b} \in \{0,1\}^4$*. Let* $\mathcal{K}$ *and* $\mathcal{C}$ *be* $\mathcal{D}$*'s underlying distributions on key and on* C*. Suppose further that* $H_\infty(\mathcal{K}) \geq \kappa - O(\lambda(\kappa))$ *and* $H_\infty(\mathcal{C}) \geq 3\kappa - O(\lambda(\kappa))$*.*

Take Real and Simulated to be the following views:

$$\mathit{Real} = \left(K', C', \mathcal{A}^{\lambda(\kappa)}[(K', C') \leftarrow \mathit{Permute}(key, C)]\right)_{(key,C)\sim\mathcal{D}}$$

$$\mathit{Simulated} = \left(\mathit{SimPermute}(\vec{b}', \mathcal{K}, \mathcal{C})\right)_{\mu\in_R S_4, \vec{b}'\leftarrow\mu(\vec{b})}$$

then $\Delta(\mathit{Real}, \mathit{Simulated}) \leq \delta(\kappa)$*.*

Proof. We begin by describing the *SimPermute* Simulator. The proof that *Real* and *Simulated* are statistically close follows.

SimPermute. The simulator samples $key \sim \mathcal{K}$ and $C \sim \mathcal{C}$, conditioned on the inner product of key with C being $\vec{0}$ (rather than $\vec{b}[i]$, as in $\mathcal{D}$). The simulator then runs *Permute* on (key, C), under $\mathcal{A}$'s leakage attack, to compute the leakage w. To compute the output (K', C'), the simulator first samples an input key and randomness $\vec{r}$ for *Permute*, from the conditional underlying distribution of key and the randomness given leakage w. Note that, as in Lemma 3.11, this conditional distribution depends only on $\mathcal{K}$ (and not on $\mathcal{C}$). Using key and $\vec{r}$, the simulator can compute the K' that *Permute* would output. Similarly, the simulator computes the conditional distribution of C', given w and $\vec{r}$. Again, as in Lemma 3.11, this depends only on $\mathcal{C}$ (and not on $\mathcal{K}$). The simulator samples C' from this conditional distribution, under the additional condition that the inner products of C' with K' equal $\vec{b}'$. The output is (K', C', w).

Statistical Closeness of *Real* and *Simulated*. We first observe that w is $\mathcal{A}$'s output in a leakage attack that operates separately on key and on C. Moreover, the leakage on key and on C is of bounded total length $O(\ell \cdot \lambda(\kappa)) << \kappa$. Since the "real" distribution $\mathcal{D}$ of (key, C) is IuO, by Lemma 3.11 the distributions of w in *Real* and in *Simulated* are $\delta(\kappa)$-statistically close.

The more difficult part of the proof is arguing that (w.h.p. over w), the distributions of (K', C'), *conditioned on* w, in *Real* and in *Simulated* are statistically close. For this, we consider a hybrid distribution *Real*$'$. To generate *Real*$'$, we compute w as in *Simulated*, by running $\mathcal{A}$'s leakage attack on *Permute*, activated on key and C chosen s.t. their inner products equal $\vec{0}$. Let π be the composition of the permutations chosen in the ℓ iterations of *Permute*. In *Real*$'$ we generate (K', C') as in *Simulated*, but conditioning the underlying output distributions on the inner products of K' and C' equalling $\vec{b}' = \pi(\vec{b})$, rather than $\vec{b}'$ which is a uniformly random permutation of $\vec{b}$ in *Simulated*. We show that *Real*$'$ is statistically close to both *Real* and *Simulated*.

Proposition 6.3. $\Delta(Real, Real') = O(\delta(\kappa))$

Proof. We re-cast *Real* by considering the following procedure for generating it. This alternate generation operates as *Real*, except that when drawing the input (key, C) from the underlying distributions, we condition on the inner products equalling $\vec{b}$ (rather than $\vec{0}$). These (key, C) are used to compute w, and then (K', C') are drawn (as in *Real*$'$) from their conditional distributions, conditioned on inner products $\pi(\vec{b})$ (where π is the composition of the permutations used in all ℓ iterations of *Permute*). Since the input distribution $\mathcal{D}$ used in *Real* is IuO, this procedure generates exactly the view *Real*.

We now show that *Real* and *Real*$'$ are statistically close. These two distributions differ only in the joint distribution of (w, π); given w and π, the distributions of (K', C') derived in *Real* and *Real*$'$ are identical. (w, π) are generated via a multi-source leakage attack, operating separately on key and on C, with a total of $O(\ell \cdot \lambda(\kappa)) << \kappa$ bits of leakage. Moreover, the distributions of key and C in *Real* and *Real*$'$ are both IuO (by construction), and differ only in their orthogonalities ($\vec{b}$ or $\vec{0}$ respectively). By Lemma 3.11, we get that the distributions of (w, π) in *Real* and *Real*$'$ are $\delta(\kappa)$-statistically close, and thus so are *Real* and *Real*$'$ themselves. ■

Proposition 6.4. $\Delta(Real', Simulated) = \text{negl}(\kappa)$

Proof. Recall that the distributions of w in *Real*$'$ and *Simulated* are identical. The underlying distributions on K' and on C' (given w) are also identical. The difference is in the distribution (given w) of the permutation used to compute $\vec{b}'$ from the given vector $\vec{b}$ ($\vec{b}'$ is then used to jointly sample (K', C')). In *Real*$'$ we have $\vec{b}' = \pi(\vec{b})$, where π is the composition of permutations chosen

by *Permute* in its ℓ iterations. In *Simulated* we have $\vec{b}' = \mu(\vec{b})$, where μ is a uniformly random permutation in S_4, independent of w. We will show that, for *any* input (key, C), the distribution of π in *Real*$'$ (conditioned on w), is negl(κ)-close to uniformly random (w.h.p over the leakage w). It follows that *Real*$'$ and *Simulated* are negl(κ)-statistically close.

The intuition, loosely speaking, is that for each $i \in [\ell]$, the permutation $\pi_i^* = \pi_i[j_i^*]$ chosen in *Permute*'s i-th iteration, looks "fairly random" even given w. Moreover, these ℓ permutations are drawn independently from their "fairly random" distributions. The composition, over all ℓ iterations of *Permute*, of the permutations chosen in each iteration, is thus statistically close to uniformly random. We formalize this intuition below, starting with the notion of "well-mixing" distributions over in S_4.

Definition 6.5 (Well-Mixing Distribution on Permutations)**.** A distribution P over S_4 is said to be *well-mixing* if:

$$H_\infty(P) \geq 0.99 \log |S_4|$$

Next, we observe that the composition of a sequence of permutations drawn from well-mixing distributions is itself very close to uniform.

Claim 6.6. *For any sequence $P_0, \ldots, P_{\ell-1}$ of well-mixing distributions, let P be:*

$$P \triangleq (\pi_0 \circ \ldots \circ \pi_{\ell-1})_{\pi_0 \sim P_1, \ldots, \pi_{\ell-1} \sim P_{\ell-1}}$$

then P is $\exp(-\Omega(\ell))$-close to uniform over S_4.

For *Permute*'s i-th iteration, let w_i be the leakage in that iteration. We define P_i to be the distribution of the permutation $\pi_i^* = \pi_i[j_i^*]$ chosen in the i-th iteration, conditioned on $(w_0, \ldots, w_i)$ and also on the keys and ciphertexts $(K_i, C_i, K_{i+1}, C_{i+1})$. We show that in *Real*$'$, with overwhelming probability over the random coins up to (but not including) the choice of j_i^*, with probability at least $1/2$ over *Permute*'s choice of j_i^*, the distribution P_i is well-mixing.

Claim 6.7. *For the view Real$'$, for any $i \in [\ell]$, and for any $(K_i, C_i, (w_0, \ldots, w_{i-1}))$, with all but $O(\delta(\kappa))$ probability over Permute's random choices in iteration i up to Step 6, with probability at least $1/2$ over Permute's choice of j_i^* in Step 6, the distribution P_i is well-mixing.*

Proof. Examine the distribution of the vector π_i of permutations used in iteration i, conditioned on $(K_i, C_i, (w_0, \ldots, w_{i-1}))$, and conditioned also on (L_i'', D_i'') (but without conditioning on the leakage w_i in the i-th iteration or on j_i^*). Here the randomness is over $(\sigma_i, \tau_i, \pi_i)$. We observe that in this conditional distribution,the marginal distribution on $(\pi_i[0], \ldots, \pi_i[\kappa - 1])$ is uniformly random over S_4^κ. This is because for each $j \in [\kappa]$, the pair $(\sigma_i[j], \tau_i[j])$ are uniformly random (under the condition that they maintain the underlying 0 plaintext bits in $\vec{b}_i$). Thus, $\sigma_i[j], \tau_i[j]$ completely "mask" the permutation $\pi_i[j]$ that was used: all permutations are equally likely. Note that here we use the fact that the plaintext bits $\vec{b}_i$ underlying (K_i, C_i) in *Real*$'$ are all identical (they all equal 0). Otherwise, since *Permute* preserves the set of underlying plaintexts (if not their order), there would be information about each $\pi_i[j]$ in the plaintexts underlying $(L_i''[j], D_i''[j])$.

By Lemma 3.8, since the leakage w_i on $(\sigma_i, \tau_i, \pi_i)$ is of length at most $O(\lambda(\kappa))$ bits, with all but $\delta(\kappa)$ probability, the min-entropy of the vector π_i given $(K_i, C_i, L_i'', D_i'', (w_0, \ldots, w_{i-1}, w_i))$ is at least $0.995 \cdot \kappa \cdot \log |S_4|$. By an averaging argument, with probability at least $1/2$ over *Permute*'s (uniformly random) choice of j_i^*, we get that the min entropy of $\pi_i^* = \pi_i[j_i^*]$, given $(K_i, C_i, L_i'', D_i'', (w_0, \ldots, w_{i-1}, w_i))$, is at least $0.99 \log |S_4|$. The claim about P_i follows (in P_i we

condition π_i^* on the same information as above, except we replace (L_i'', D_i'') with just $(K_{i+1}, C_{i+1}) = (L_i''[j_i^*], D_i''[j_i^*])$). ■

To complete the proof of Proposition 6.4, we examine the composed distribution $(\pi = (\pi_0^* \circ, \ldots \circ \pi_{\ell-1}^*)|w)$. Each π_i^* is drawn from P_i, and these draws are all independent of each other. By Claim 6.7, we get that with all but $\exp(-\Omega(\ell))$ probability over the random coins, fixing the sequence $((K_0, C_0), \ldots, (K_{\ell-1}, C_{\ell-1}))$ and the leakage w, at least 1/3 of the distributions P_i are well-mixing. When this happens, by Claim 6.6, the distribution of $(\pi|w)$ is $\exp(-\Omega(\ell))$-close to uniform, where $\ell = \text{polylog}(\kappa)$. ■

■

6.3 Proof of *SimNAND* Security (Lemma 6.1)

Remark 6.8. *We will assume throughout this section that the leakage w from SafeNAND includes Permute's output in its entirety. This is a strengthening of the leakage adversary (it gets more leakage "for free"), and so it strengthens our security claim for SafeNAND.*

Proof of Lemma 6.1. We begin by describing the *SimNAND* simulator, and then proceed with a proof of statistical closeness of *Real* and *Simulated*.

***SimNAND*.** Let $\mathcal{D}^\times$ be the independently drawn variant of $\mathcal{D}$ (as in Definition 3.10, i.e. with independent draws from $\mathcal{K}$ and from $\mathcal{C}$). *SimNAND* samples $((key_i, key_j, key_k), (\vec{c}_i, \vec{c}_j, \vec{c}_k)) \sim \mathcal{D}^\times$, and $key \leftarrow KeyGen(1^\kappa)$. It runs Steps 1 and 2 of the *SafeNAND* procedure, on the keys and ciphertexts it drew, under $\mathcal{A}$'s leakage attack. Let $w_{1,2}$ be the leakage generated in this partial execution, and let $\vec{\sigma} = (\sigma_i, \sigma_j, \sigma_k)$ be the correlation values computed by *SafeNAND* in this simulated execution.

Next, *SimNAND* computes $\mathcal{K}_{SimPermute}$ and $\mathcal{C}_{SimPermute}$, the conditional distributions of *key* and of C in Step 3, given $w_{1,2}$ and $\vec{\sigma}$. *SimNAND* proceeds to simulate Step 3 by calling the *Permute* simulator, *SimPermute* (see Lemma 6.2), on input $(\vec{b}_{SimPermute}, \mathcal{K}_{SimPermute}, \mathcal{C}_{SimPermute})$, where $\vec{b}_{SimPermute}$ is a uniformly random permutation of the vector $(a_k, a_k \oplus 1, a_k \oplus 1, a_k \oplus 1)$. *SimPermute*'s output includes the leakage w_3 from Step 3 and an output (K', C') from *Permute*. *SimNAND* completes the simulation by running Step 4 on (K', C') under $\mathcal{A}$'s leakage attack, producing leakage w_4. The leakage that *SimNAND* outputs is the accumulated leakage $w = (w_{1,2} \circ w_3 \circ (K', C') \circ w_4)$ from all the simulated steps of *SafeNAND* (recall from Remark 6.8 that we include (K', C') in the leakage).

Statistical closeness of *Real* and *Simulated*. We examine the distributions of the leakage $w_{1,2}$ in Steps 1 and 2 in both views. In both *Real* and *Simulated* we have $(key_i, key_j, key_k) \sim \mathcal{K}$ and $key \leftarrow KeyGen(1^\kappa)$. These determine the correlation values $\vec{\sigma} = (\sigma_i, \sigma_j, \sigma_k)$ computed in Step 1 of *SafeNAND*. Note that the correlation values are a function of the keys only (and not the ciphertexts), and thus they are identically distributed in both *Real* and *Simulated*. The difference is in the conditional distribution of $(\vec{c}_i, \vec{c}_j, \vec{c}_k)$ given $(key, \vec{\sigma})$.

We focus on the joint conditional distribution of $(key, (\vec{c}_i, \vec{c}_j, \vec{c}_k))$, conditioned on $\vec{\sigma}$. We will show that this joint distribution, conditioned on $\vec{\sigma}$, is: (i) IuO in *Real*, and (ii) its independently drawn variant in *Simulated*. Given $\vec{\sigma}$, the leakage is a multi-source function of *key* and of the ciphertexts. We will conclude, using Lemma 3.11, that: (i) the leakage in *Real* and in *Simulated*

is statistically close, and (ii) for a fixed leakage value $w_{1,2}$ that can occur in *Real*, the conditional distribution of $(key, (\vec{c}_i, \vec{c}_j, \vec{c}_k))$ in *Real*, given $(\vec{\sigma}, w_{1,2})$, will remain IuO. The distribution of $(key, (\vec{c}_i, \vec{c}_j, \vec{c}_k))$ in *Simulated*, given $(\vec{\sigma}, w_{1,2})$, will be the independently drawn variant of the same distribution in *Real*. Note that C is a (linear) function of $(\vec{c}_i, \vec{c}_j, \vec{c}_k)$ (and a_i, a_j), and so we get the same guarantees for the distributions of (key, C) in *Real* and *Simulated*.

It remains to show that the requirements of Lemma 3.11 hold. Namely, that $(key, (\vec{c}_i, \vec{c}_j, \vec{c}_k))$ in *Real*, conditioned on $\vec{\sigma}$, is IuO, and that the same distribution in *Simulated* is its independently drawn variant. For this, observe first that in *Simulated*, the distribution of $(key, (\vec{c}_i, \vec{c}_j, \vec{c}_k))$ given $\vec{\sigma}$ is the product distribution of *key* given $\vec{\sigma}$, and of $\mathcal{C}$ (without any conditioning). The fixed values of $\vec{\sigma}$ do not effect the marginal distribution on ciphertexts, because (in *Simulated*) the ciphertexts are drawn independently of the keys. In *Real*, on the other hand, the keys and ciphertexts are no longer drawn independently. However, even in *Real*, $\mathcal{D}$ is IuO. In particular, $\mathcal{D}$'s marginal conditional distribution on $(\vec{c}_i, \vec{c}_j, \vec{c}_k)$, given *key* and $\vec{\sigma}$, is equal to $\mathcal{C}$, conditioned on $\langle key \oplus \sigma_i, \vec{c}_i \rangle = b_i$, on $\langle key \oplus \sigma_j, \vec{c}_j \rangle = b_j$, and on $\langle key \oplus \sigma_k, \vec{c}_k \rangle = b_k$. We conclude that in *Real*, the distribution of $(key, (\vec{c}_i, \vec{c}_j, \vec{c}_k))$, conditioned on $\vec{\sigma}$, is also IuO. Moreover, by Lemma 3.8, with all but $O(\delta(\kappa))$ probability over $\vec{\sigma}$, the min-entropy of $(key, (\vec{c}_i, \vec{c}_j, \vec{c}_k))$ given $\vec{\sigma}$ is at least $4\kappa - O(\lambda(\kappa))$.

By Lemma 3.11, we conclude that the distributions $((key, C)|(\vec{\sigma}, w_{1,2}))$ in *Real* and in *Simulated* satisfy all the conditions of Lemma 6.2 (security of *Permute*). By construction, the vector $\vec{b}_{SimPermute}$ of plaintext values given as input to the *SimPermute* simulator in *Simulated*, is a uniformly random permutation of the plaintexts underlying (key, C), the input to *Permute* in *Real*. By Lemma 6.2, we conclude that the distributions of $(w_{1,2} \circ w_3 \circ (K', C'))$, in conjunction with $\vec{\sigma}$, are statistically close in *Real* and *Simulated*. Statistical closeness of *Real* and *Simulated* follows, because the leakage w_4 from Step 4 is a function of (K', C'). ■

7 Putting it Together: The Full Construction

In this section we show how to compile any circuit into a secure transformed one that resists OC side-channel attacks, as per Definition 3.14 in Section 3.4. See Section 2 for an overview of the construction.

The full initialization and evaluation procedures are presented below in Figures 10 and 11. The evaluation procedure is separated into sub-computations (which may themselves be separated into sub-computations of the cryptographic algorithms). Ciphertext bank procedures are in Section 5. The procedures for safely computing NAND and duplication are in Section 6. Theorem 7.1 states the security of the compiler.

Theorem 7.1. *There exist a leakage bound $\lambda(\kappa) = \tilde{\Omega}(\kappa)$ and a distance bound $\delta(\kappa) = \mathrm{negl}(\kappa)$, s.t. for every $\kappa \in \mathbb{N}$, the $(Init, Eval)$ compiler specified in Figures 10 and 11 is a (λ, δ)-continuous leakage secure compiler, as per Definition 3.14.*

Proof Sketch. We first specify the simulator and then provide a sketch of statistical security.

Simulator. Let $\mathcal{A}$ be a (continuous) leakage adversary. The simulator, using *SimInit* and *SimEval*, creates a view of repeated executions of *Eval*, on different inputs, under a (continuous) leakage attack by $\mathcal{A}$. It mimics the operation of the "real" *Eval* procedure. The *SimInit* procedure starts by initializing all ciphertext banks using *SimBankInit*. Within the t-th execution,

Initialization $\mathit{Init}(1^\kappa, C, y)$

1. for every y-input wire i, corresponding to $y[j]$:

$$\mathit{Bank}_i \leftarrow \mathit{BankInit}(1^\kappa, y[j])$$

2. for the output wire *output*:

$$\mathit{Bank}_{\mathit{output}} \leftarrow \mathit{BankInit}(1^\kappa, 0)$$

3. for the internal wires:

$$\mathit{Bank}_{\mathit{internal}} \leftarrow \mathit{BankInit}(1^\kappa, b), \text{ where } b \in_R \{0,1\}$$

4. output: $\mathit{state}_0 \leftarrow (\mathit{Bank}_{\mathit{internal}}, \mathit{Bank}_{\mathit{output}}, \{\mathit{Bank}_i\}_{i \text{ is a } y\text{-input wire}})$

Figure 10: *Init* procedure, to be run in an offline stage on circuit C and secret y.

with input x_t and output $C(y, x_t)$, the simulator picks all of the (a_i, b_i) shares for each wire i in advance. To do so, the simulator first evaluates $C(\vec{0}, x_t)$ and takes v'_i to be the bit value on wire i in this evaluation. For y-input wires, the simulator sets $a_i = b_i = 0$. For internal wires, the a_i shares are uniformly random, and each b_i is set so that $a_i \oplus b_i = v'_i$. For the output wire *out*, the simulator sets $a_{out} = C(y, x_t)$, and $b_{out} = v'_{out} \oplus a_{out}$.

Once the (a_i, b_i) values are picked, the simulator generates the ciphertexts $\vec{c}_i$ so that the plaintext underlying $\vec{c}_i$ is indeed b_i. This is done using the *SimBankGen* simulation procedures, which gives the simulator control over the plaintext underlying the ciphertext that it generates. The rest of the simulator's operation follows the *Eval* procedure on the generated ciphertexts, and the leakage is generated as it would be from an execution of *Eval*. The *SimInit* and *SimEval* procedures are specified below in Figures 12 and 13.

Statistical Security (Sketch). The intuition for security is that the "public" a_i shares in the simulated execution are distributed exactly as they are in the real execution. The "private" b_i shares differ between the real and simulated execution, but these shares are in protected LROTP encrypted form $(\mathit{key}_i, \vec{c}_i)$, where the key and ciphertext are never loaded into memory together.

The full proof that *Real* and *Simulated* are statistically close uses several hybrids:

Real **to** *HybridReal***: replacing real generations with simulated ones.** The first hybrid is *HybridReal*. It is obtained from *Real* by replacing each "real" generation with a "simulated" generation that produces a key-ciphertext pair with the same underlying plaintext. In particular, we replace each $\mathit{BankInit}(b_i)$ call for an output or y-input wire i, with a *SimBankInit* call, and we replace the *BankInit* call for $\mathit{Bank}_{\mathit{internal}}$ with a *SimBankInit* call. We then replace each *BankGen* call for an output or y-input wire i with a call to $\mathit{SimBankGen}(b_i)$, where b_i is the appropriate private share for wire i. We replace each pair of *BankGen* calls to $\mathit{Bank}_{\mathit{internal}}$ with a pair of calls to $\mathit{SimBankGen}(b)$, where b is independent and uniformly random in $\{0,1\}$. Finally, we replace each call to *BankUpdate* and *BankRedraw* with a call to *SimBankUpdate* and *SimBankRedraw* (respectively). Other than these changes to the ciphertext bank calls, we run exactly as in *Real*.

Evaluation $\underline{Eval(state_{t-1}, x_t)}$

$state_{t-1} = (Bank_{internal}, Bank_{output}, \{Bank_i\}_{i \text{ is a } y\text{-input wire}})$

1. Generate keys and ciphertexts for all circuit wires:

   (a) y input wire i:
   $(key_i, \vec{c}_i^{in}) \leftarrow BankGen(Bank_i)$
   $Bank_i \leftarrow BankUpdate(Bank_i)$

   (b) output wire *output*:
   $(key_{output}, \vec{c}_{output}^{out}) \leftarrow BankGen(Bank_{output})$
   $Bank_{output} \leftarrow BankUpdate(Bank_{output})$

   (c) each internal wire i (in sequence):
   $(key_i, \vec{c}_i^{in}) \leftarrow BankGen(Bank_{internal})$
   $(key_i, \vec{c}_i^{out}) \leftarrow BankGen(Bank_{internal})$
   $Bank_{internal} \leftarrow BankRedraw(Bank_{internal})$
   $Bank_{internal} \leftarrow BankUpdate(Bank_{internal})$

   (d) x_t-input wire i: $key_i \leftarrow KeyGen(1^\kappa), \vec{c}_i^{in} \leftarrow Encrypt(key_i, 0)$

2. Compute the public shares on all wires.

   For the input wires: for each y-input wire i, $a_i \leftarrow 0$. For each x-input wire i corresponding to $x_t[j]$, $a_i \leftarrow x_t[j]$.

   Proceed layer by layer (from input to output) to compute the remaining public shares:

   (a) for each NAND gate with input wires i, j and output wire k, compute:

   $$a_k \leftarrow SafeNAND(a_i, key_i, \vec{c}_i^{in}, a_j, key_j, \vec{c}_j^{in}, key_k, \vec{c}_k^{out})$$

   (b) for each duplication gate with input wire i and output wires j, k, compute:

   $$a_j \leftarrow SafeDup(a_i, key_i, \vec{c}_i^{in}, key_j, \vec{c}_j^{out})$$

   $$a_k \leftarrow SafeDup(a_i, key_i, \vec{c}_i^{in}, key_k, \vec{c}_k^{out})$$

   (c) output a_{output}

3. the new state is: $state_t \leftarrow (Bank_{internal}, Bank_{output}, \{Bank_i\}_{i \text{ is a } y\text{-input wire}})$

Figure 11: *Eval* procedure performed on input x_t, under OC leakage. See Section 5.1 for ciphertext bank procedures, Section 6.1 for *SafeNAND*, *SafeDup*.

The two views *Real* and *HybridReal* differ only in that in *Real* we have calls to *BankInit*, *BankGen*, *BankUpdate*, *BankRedraw*, whereas in *HybridReal* we have calls to the corresponding simulated procedures. Note that the b_i values given as input to *SimBankGen* in *HybridReal* are distributed *identically* to the plaintexts underlying the ciphertexts generated in the corresponding calls to *BankGen* in *Real*: for y-input wire i, corresponding to the j-th bit of y, b_i is equal to $y[j]$ in both views. For each internal wire i, b_i is an independently uniformly random bit in both views. For the output wire *output*, b_{output} equals 0 in both views. By Lemmas 5.1 and 5.4, we get that the

Simulator Initialization $SimInit(1^\kappa, C)$

1. for every y-input wire i, corresponding to $y[j]$:

$$Bank_i \leftarrow SimBankInit(1^\kappa)$$

2. for the output wire *output*:

$$Bank_{output} \leftarrow SimBankInit(1^\kappa)$$

3. for the internal wires:

$$Bank_{internal} \leftarrow SimBankInit(1^\kappa)$$

4. output: $state_0 \leftarrow (Bank_{internal}, Bank_{output}, \{Bank_i\}_{i \text{ is a } y\text{-input wire}})$

Figure 12: Simulator Initialization *SimInit*

Simulator $SimEval(state_{t-1}, x_t, C(y, x_t))$

The simulator first computes v'_i values for each wire i in the circuit by evaluating $C(\vec{0}, x_t)$.
For each circuit wire i, choose shares (a_i, b_i) for each wire:

x_t input wire corresponding to $x_t[j]$: $a_i \leftarrow x_t[j], b_i \leftarrow 0$
y-input wire: $a_i, b_i \leftarrow 0$
internal wire: $a_i \leftarrow_R \{0,1\}, b_i \leftarrow v'_i \oplus a_i$
output wire: $a_{output} \leftarrow C(y, x_t), b_{output} \leftarrow v'_{output} \oplus a_{output}$

After the a_i, b_i shares have been computed for each wire, simulate *Eval* as follows:

- in Step 1, for each wire i, replace each call to *BankGen* for wire i with a call to *SimBankGen* with b_i. Replace each call to *BankUpdate* and *BankRedraw* with a call to *SimBankUpdate* or *SimBankRedraw* (respectively).
- in Step 2, for each NAND gate with input wires i, j and output wire k, compute:

$$a_k \leftarrow SafeNAND(a_i, key_i, \vec{c}_i^{in}, a_j, key_j, \vec{c}_j^{in}, key_k, \vec{c}_k^{out})$$

  for each duplication gate with input wire i and output wires j, k, compute:

$$a_j \leftarrow SafeDup(a_i, key_i, \vec{c}_i^{in}, key_j, \vec{c}_j^{out})$$

$$a_k \leftarrow SafeDup(a_i, key_i, \vec{c}_i^{in}, key_k, \vec{c}_k^{out})$$

- as in *Eval*, the new state is $state_t \leftarrow (Bank_{internal}, Bank_{output}, \{Bank_i\}_{i \text{ is a } y\text{-input wire}})$

Figure 13: *Sim* procedure performed on input x_t and circuit output $C(y, x_t)$

joint distributions of the leakage in all of these calls, *together with all keys and ciphertexts produced*, are statistically close in *Real* and in *HybridReal*. We can complete the generation of the view in both cases (the leakage from *SafeNAND* and *SafeDup*) as a function of the keys and ciphertexts produced, and we conclude that the two views are statistically close.

HybridReal **to** *HybridReal′* **and** *Simulated* **to** *Simulated′***: replacing safe computations with simulated leakage.** Next, we obtain *HybridReal′* from *HybridReal* by replacing the *SafeNAND* calls, in each execution and for each internal and output wire i, with calls to the *SimNAND* simulator from Lemma 6.1, using *HybridReal*'s a_k public shares and the underlying distributions on keys and ciphertexts (the underlying distributions are a function of the leakage values in prior computations).

Similarly, we obtain a hybrid *Simulated′* from *Simulated* by replacing the *SafeNAND* calls with calls to the *SimNAND* simulator, using *SimEval*'s a_i public shares and its underlying distributions on keys and ciphertexts.

Note that, in particular, *HybridReal′* and *Simulated′* can no longer be generated efficiently (because the *SafeNAND* simulator is not efficient). By Lemmas 5.3 and 5.5, the conditions of Lemma 6.1 all hold for each replacement of *SafeNAND* by *SimNAND* in both views (given the leakage in prior computations). In particular, the keys and ciphertexts involved in each *SafeNAND* come from IuO distributions whose underlying distributions have high entropy (w.h.p.). This is where we use the fact that, for each internal wire i, even given the leakage, the i-th wire's ciphertexts $\vec{c}_i^{out}$ and $\vec{c}_i^{in}$ are independent up to having the same orthogonality w.r.t. *key*. We conclude that *HybridReal* and *HybridReal′* are statistically close, as are *Simulated* and *Simulated′*.

Closeness of *HybridReal′* **and** *Simulated′***.** *HybridReal′* and *Simulated′* are both obtained as a function of leakage from a sequence of *SimBankGen* calls: the leakage from these generations is then used to compute the leakage for *SimNAND* calls (the leakage from the generations specifies the underlying distributions used by *SimNAND*). In particular, the actual keys and ciphertexts generated are never again accessed after their generation. The same post-processing is performed on the leakage from the generations in both cases: namely, calls to *SimNAND* on the same underlying distributions, and with identically distributed a_i values. The two sequences of generations differ only in the orthogonalities of the underlying plaintexts that are generated in the *SimBankGen* calls for the output and the y-input wires (the plaintexts for internal wires are identically distributed). By Lemma 5.2, we conclude that the leakage from the generations is statistically close in both cases, and so *HybridReal′* and *Simulated′* are also statistically close. ■

References

[Ajt11] Miklos Ajtai. Secure computation with information leaking to an adversary. In *STOC*, 2011.

[BCG+11] Nir Bitansky, Ran Canetti, Shafi Goldwasser, Shai Halevi, Yael Tauman Kalai, and Guy N. Rothblum. Program obfuscation with leaky hardware. In *ASIACRYPT*, pages 722–739, 2011.

[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.

[BGJK12] Elette Boyle, Shafi Goldwasser, Abhishek Jain, and Yael Tauman Kalai. Multiparty computation secure against continual leakage. In *STOC*, 2012.

[BHHO08] Dan Boneh, Shai Halevi, Michael Hamburg, and Rafail Ostrovsky. Circular-secure encryption from decision diffie-hellman. In *CRYPTO*, pages 108–125, 2008.

[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.

[CG88] Benny Chor and Oded Goldreich. Unbiased bits from sources of weak randomness and probabilistic communication complexity. *SIAM J. Comput.*, 17(2):230–261, 1988.

[DHLAW10] Yevgeniy Dodis, Kristiyan Haralambiev, Adriana López-Alt, and Daniel Wichs. Cryptography against continuous memory attacks. In *FOCS*, pages 511–520, 2010.

[DLWW11] Yevgeniy Dodis, Allison Lewko, Brent Waters, and Daniel Wichs. Storing secrets on continually leaky devices. Cryptology ePrint Archive, Report 2011/369, 2011.

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

[DRS04] Yevgeniy Dodis, Leonid Reyzin, and Adam Smith. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. In *EUROCRYPT*, pages 523–540, 2004.

[FKPR10] Sebastian Faust, Eike Kiltz, Krzysztof Pietrzak, and Guy N. Rothblum. Leakage-resilient signatures. In *TCC*, pages 343–360, 2010.

[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.

[GIS+10] Vipul Goyal, Yuval Ishai, Amit Sahai, Ramarathnam Venkatesan, and Akshay Wadia. Founding cryptography on tamper-proof hardware tokens. In *TCC*, pages 308–326, 2010.

[GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. One-time programs. In *CRYPTO*, pages 39–56, 2008.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431–473, 1996.

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

[Imp10] Russel Impagliazzo. Personal communication, 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.

[LLW11] Allison Lewko, Mark Lewko, and Brent Waters. How to leak on key updates. In *STOC*, 2011.

[LRW11] Allison Lewko, Yannis Rouselakis, and Brent Waters. Achieving leakage resilience through dual system encryption. In *TCC*, 2011.

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

[NS09] Moni Naor and Gil Segev. Public-key cryptosystems resilient to key leakage. In *CRYPTO*, pages 18–35, 2009.

[Pie09] Krzysztof Pietrzak. A leakage-resilient mode of operation. In *EUROCRYPT*, pages 462–482, 2009.

[Rao07] Anup Rao. An exposition of bourgain's 2-source extractor. *Electronic Colloquium on Computational Complexity (ECCC)*, 14(034), 2007.

[Rot12] Guy N. Rothblum. How to compute under ac^0 leakage without secure hardware. In *Manuscript*, 2012.

[SYY99] Tomas Sander, Adam Young, and Moti Yung. Non-interactive cryptocomputing for nc^1. In *FOCS*, pages 554–567, 1999.

ECCC ISSN 1433-8092

http://eccc.hpi-web.de