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

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

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

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. Throughout the introduction, we will call such executions *leakage resilient.*

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

A crucial aspect of this question is how to model the partial information or *leakage* attack that an adversary can launch during executions. Proper modeling should simultaneously capture real world attacks and achieve the right level of theoretical abstraction. Furthermore, impossibility results on obfuscation [BGI$^+$01] imply inherent limitations on the leakage attacks which can be tolerated by general programs: [Imp10] observes that if the leakage attack model allows even a single bit of leakage to be computed by an adversarially chosen polynomial-time function applied to the entire internal state of the execution, then there exist programs P which cannot be executed in a leakage resilient manner. Thus, to enable any algorithm to run securely in the presence of continual leakage, we must put restrictions on the leakage attack model to rule out this impossibility result.

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

Wire Probe (ISW-L) The pioneering work of Ishai, Sahai, and Wagner [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.

$\mathcal{AC}^0$ **Bounded Leakage(CB-L).** Faust, Rabin, Reyzin, Tromer and Vaikuntanathan [FRR$^+$10] modify 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 $\mathcal{AC}^0$ computable function f computed on the values of all the wires. Similarly to the [ISW03] model they also place a total bound on the output length of this $\mathcal{AC}^0$ function f. To obtain results in this model, [FRR$^+$10] also augment the model to assume the existence of leak free hardware components that produce samples from a polynomial time sampleable distribution. It is assumed that there is no data leakage from the randomness generated and the computation performed inside of the device.

Only-Computation Leaks (OC-L). The Micali-Reyzin [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 computable length bounded function called a *leakage function*,[1] computed on each sub-computation's input and randomness.

To obtain results in this model, both [GR10] and [JV10] augment the model to assume the existence of leak free hardware components that produce samples from a polynomial time sampleable distribution. It is assumed that there is no data leakage from the randomness generated and the computation performed inside of the device. Similarly, in independent work, Dziembowski and Faust [DF12] also assume leak-free components. Unlike [GR10, JV10], they do not bound the computational power of the adversary.

RAM Cell Probe (RAM-L). The RAM model of Goldreich and Ostrovsky [GO96] considers an architecture that loads data from fully protected memory, and runs 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.[2] 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 (This is similar to the ISW-L model, in that a portion of memory-addresses used in a computation are either fully exposed or fully hidden, except that [Ajt11] works in the RAM model and divides the executions into sub-computations whereas [ISW03] work in the stateful circuit model). These are called the compromised memory accesses (or times). The contents of the un-compromised addresses, and the contents of the main memory not loaded into the CPU, are assumed to be perfectly hidden.

1.1 The New Work

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

Computationally Unbounded OC-L Leakage Adversary. The leakage attacks we address are in the "only computation leaks information" model of [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.

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

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

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

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

1. $Eval(x) = C(y,x)$ for all inputs x.
2. The execution of $Eval(x)$ for $|x| = n$, will consist of $O(|C_n|)$ sub-computations, each of complexity (time and space) $\tilde{O}(\kappa^{\omega})$, (where ω is the exponent in the best algorithm known for matrix multiplication).
3. There exists a simulator *Sim*, a leakage bound $\lambda(\kappa) = \tilde{\Theta}(\kappa)$, 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 poly(κ).

We emphasize that our result holds unconditionally, *without any leak-free hardware* or any computational assumptions. We stress that this is in contrast to all known works [FRR+10, GR10, JV10, DF12, Rot12] on resilience of general programs against continual leakage that consider non-trivial[3] leakage functions. See Section 3 for a fuller comparison with prior work on leakage resilience for general programs. See Section 1.2 for a description of the "leaky CPU" model, an alternative to the OC leakage model.

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

Doing Away with Computational Assumptions. Previous works relied on the existence of homomorphic properties of an underlying public-key encryption scheme with good leakage resilience properties and good key-refreshing possibilities, which helped carry out the computation

[3]By non-trivial, we mean that the leakage function performs some a non-trivial computation on wires or memory accesses in the execution, rather than simply releasing their values as in [ISW03, Ajt11].

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

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

We also mention that, whereas our focus is on enabling any algorithm to run securely in the presence of continual leakage, continual leakage on *restricted computations* (e.g. [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. See Section 3 for further discussions of related work.

1.2 Leaky CPU: An Alternative to OC-Leakage

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

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

1. A CPU which executes instructions from a fixed set of special universal instructions, each of size 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 $\Theta(\kappa)$ in our results). The leakage function is applied to the instruction executed in the CPU – namely, it is a function of all inputs, outputs, randomness, and intermediate wires of the CPU instruction being executed.

2. Contents of memory, when not loaded into the CPU, are hidden from the adversary.

Our result, stated in this model, provides a fixed set of CPU instructions, and a compiler which can take any polynomial time computation (say given in the form of a boolean circuit), and compile it into a program that can be run on this leaky CPU. A leakage adversary as above, who can specify inputs to the compiled program and observe its outputs, learns nothing from the execution beyond its input-out behavior. We elaborate on the set of instructions required by our compiler in Section 2.4

1.3 Applications of the Main Theorem

Application of Our Compiler to Obfuscation with Leaky Hardware. There is a fascinating connection between the problem of code obfuscation and leakage resilience for general programs. In a nut-shell, one may think of obfuscation of an algorithm as the ultimate "leakage resilient" transformation: If successful, it implies that the resulting algorithm can be *"fully leaked"* to the adversary – it is under the adversary's complete control! Since we know that full and general obfuscation is impossible [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 functionality) allow an adversary to have a "partial view" of the execution and only learn its black-box functionality. In our work, this "partial view" is as defined by the "only computation leaks" leakage attack model.

In a recent work, Bitansky *et al.* [BCG$^+$11] make the connection between the OCL attack model and obfuscation explicit. They use the compiler described here to obfuscate programs using simple secure hardware components which are "leaky": they may be subject to memory leakage attacks. At a high level, they run each "sub-computation" on a separate hardware component which is subject to memory leakage. Alternatively, viewing OC leakage as an attack in the leaky CPU model, each instruction of the leaky CPU is implemented in a separate hardware component. The main challenge in their setting is providing security even when the communication channels between the hardware components are observed and controlled by an adversary, which they address using non-committing encryption [CFGN96] and MACs .

Application of Our Compiler to Leakage Resilient Multi Party Computation. In a recent work, Boyle *et al.* [BGJK12] use our compiler (and the assumption that FHE exists) to build secure MPC protocols that can compute an unbounded number of polynomial time functions C_i on an input (which has been shared among the players a one-time leak free pre-processing stage) that are resilient to corruptions of a constant fraction of the players and to leakage on all of the rest of the players (separately). Intuitively, one can think of each player in the MPC as running one of the "sub-computations" in a compilation of C_i using our OC-L compiler. Alternatively, viewing of OC leakage as an attack in the leaky CPU model, operations of the leaky CPU are implemented in by different players in the MPC protocol. The additional challenges here are both adversarial monitoring/control of the communication channels and (more significantly) that the adversary may completely corrupt many of the players/sub-computations.

2 Compiler Overview and Technical Contributions

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

- In Section 2.1 we describe the first tool in our construction: a leakage-resilient one-time pad cryptosystem (LROTP), which is used as the subsidiary cryptosystem[4] in our construction, and is resilient to bounded leakage. In particular, we use this cryptosystem to encrypt the secret fixed input y.
- In Section 2.2, we show how to use these encryptions to compute the program's output *once* on a given input. This "one-time" safe evaluation is resilient to bounded OC leakage attacks. The main challenge is to develop a "safe homomorphic evaluation" procedure for computing the NAND of LROTP encrypted bits.
- In Section 2.3 we show how to extend the "one-time" safe evaluation to "any polynomial number" of safe evaluations on new inputs, i.e. to resist *continual* leakage. The main new technical tool introduced here, and where the bulk of technical difficulty of our paper lies, is in using what we call "ciphertext banks": they will allow the *repeated* generation of secure ciphertexts even under leakage.

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

2.1 Leakage-Resilient One Time Pad

One of the main components of our construction is the *"leakage resilient one-time pad"* cryptoscheme (LROTP). This simple private-key encryption scheme uses a vector $\mathit{key} \in \{0,1\}^\kappa$ 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: $\mathit{Decrypt}(\mathit{key}, \vec{c}) = \langle \mathit{key}, \vec{c} \rangle$. The scheme maintains the invariants that $\mathit{key}[0] = 1, \vec{c}[1] = 1$, 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 $\mathit{Decrypt}(\mathit{key}, \vec{c}) = b$.

The LROTP scheme is remarkably well suited for our goal of transforming general computations to resist leakage attacks. We use the following properties (see Section 5 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

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

ciphertext and the secret-key simultaneously (otherwise it could decrypt); second, the leakage from the key and from the ciphertext is of bounded length. This ensures, for example, that the adversary cannot learn enough of the ciphertext to be useful for it at a later time—when it could apply an adaptively chosen leakage function to the secret key (and, for example, decrypt).

Translating this reasoning into a proof, we show that semantic security is retained under concurrent attacks of bounded leakage $O(\kappa)$ length on *key* and $\vec{c}$. As long as leakage is of bounded length and operates separately on *key* and on $\vec{c}$, they remain (w.h.p.) high entropy sources, and are independent up to their inner product equaling the underlying plaintext. Since the inner product function is a two-source extractor, the underlying plaintext is statistically close to uniformly random even given the leakage. Moreover, this is true even for computationally unbounded adversaries and leakage functions. To ensure that the leakage operates separately on *key* and $\vec{c}$, we take care in our construction not to load ciphertexts and keys into working memory simultaneously.[5] We note that two-source extractors were used for enabling leakage-resilient cryptography in [GR10, HL11, DF12].

Key and Ciphertext Refreshing. We give procedures for "refreshing" an LROTP key and ciphertexts: the output key and ciphertext will be a "fresh" uniformly random encryption of the same underlying plaintext bit. Moreover, the refreshing procedure operates separately on the key and on the ciphertext, and so an OC leakage attack will not be able to determine the underlying plaintext. In fact, security of the underlying plaintext is maintained even under OC leakage from multiple composed applications of the refreshing procedure. Security is maintained as long as the accumulated leakage is a small constant fraction of the key and ciphertext length. After a large enough number of composed applications, however, security is lost: An OC leakage adversary can successfully reconstruct the underlying plaintext. This attack is described in Section 5.2. Intuitively, it "kicks in" once the length of the accumulated leakage is a large constant fraction of the key and ciphertext length.

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

We note that the construction in [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 One-Time Secure Evaluation

Next, 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.

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

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 to the adversary. The circuit takes as inputs the secret y, and also public input $x \in \{0,1\}^n$ (which may be chosen by the adversary), and produces a single bit output.[6] For example, C can be a public cryptographic algorithm (say for producing digital signatures), y a secret signing key, and x a public message to sign. More generally, to compile general algorithms, C can be the universal circuit, y the description of any particular algorithm that is to be protected, and x a public input to the protected algorithm.

The *output* of the compiler on C and y is a probabilistic stateful evaluation algorithm *Eval* (with a *state* that will be updated during each run of *Eval*), such that for all $x \in \{0,1\}^n$, $C(y,x) = Eval(y,x)$. The compiler is run exactly once at the beginning of time and is not subject to leakage. See Section 4.3 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, $i < k$. We use $v_i \in \{0,1\}$ to denote the bit value on wire i of the original circuit $C(y,x)$. *Eval* does not compute or load into memory the explicit v_i values for internal wires (or y-input wires) into memory: any such value loaded into memory might leak and expose non black-box information about the circuit C! Instead, *Eval* keeps track of each v_i value in LROTP encrypted form $(key_i, \vec{c_i})$. I.e., there is a key and a ciphertext (with underlying plaintext v_i) for each circuit wire, and v_i is protected because the key and ciphertext are never loaded into memory at once.

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

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

Recall that initialization is performed without any leakage. Looking ahead, the main challenge for multiple execution will be securely generating the keys and ciphertexts for each wire even in the presence of OC leakage. See Section 2.3.

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

Eval proceeds to compute, for each internal wire i of the circuit (and for the output wire *output*), a secure LROTP encryption $(key_i, \vec{c_i})$ of v_i. This is accomplished using a safe homomorphic evaluation procedure discussed below. The homomorphic evaluation follows the computation of the original circuit C gate by gate in lexicographic order (from the input wires to the output wire).

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

The adversary learns nothing about the v_i values, even under leakage. All the adversary "sees" are the input x and the output $v_{output} = C(y, x)$. The challenge is homomorphic evaluation of the internal NAND gates.

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

For a NAND gate with input wires i, j and output wire k, the input to the *SafeNAND* procedure is ciphertext-key pairs: $(key_i, \vec{c_i})$, $(key_j, \vec{c_j})$ (underlying plaintexts v_i, v_j), $(\vec{\ell_k}, \vec{d_k})$ (random underlying plaintext r_k), and $(\vec{o_k}, \vec{e_k})$ (underlying plaintext 1). The goal is to compute the "public bit" $a_k = (v_i \text{ NAND } v_j) \oplus r_k$, without leaking anything more about the underlying plaintexts (v_i, v_j, r_k).

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

We proceed with an overview of *SafeNAND*, see Section 7 for details. As a starting point, we first choose a single *key*, and compute from $(key_i, \vec{c_i})$, $(key_j, \vec{c_j})$, $(\vec{\ell_k}, \vec{d_k})$, $(\vec{o_k}, \vec{e_k})$, new ciphertexts $(\vec{c_i^*}, \vec{c_j^*}, \vec{d_k^*}, \vec{e_k^*})$ whose underlying plaintexts under this single *key* remain $(v_i, v_j, r_k, 1)$ (respectively). This uses homomorphic properties of the LROTP cryptosysm keys. Once the ciphertexts are all encrypted under the same *key*, our goal is to compute the public bit $a_k = (v_i \text{ NAND } v_j) \oplus r_k = v_k \oplus a_k$.

To compute a_k, we start with an idea of Sanders Young and Yung [SYY99] for homomorphically computing the NAND of two ciphertexts with underlying plaintexts v_i, v_j. They used homomorphic addition to create a 3-tuple of ciphertexts s.t. the number of ciphertexts with underlying plaintext 0 in this 3-tuple specifies whether $(v_i \text{ NAND } v_j)$ is 0 or 1. The locations of 0's and 1's in the 3-tuple expose information about v_i and v_j beyond their NAND, but [SYY99] permute the 3-tuple of ciphertexts using a random permutation (and also refresh each ciphertext). They showed that the resulting 3-tuple of permuted ciphertexts exposes only $(v_i \text{ NAND } v_j)$ and nothing more. They use this idea to build secure function evaluation protocols for NC^1 circuits.

We translate this idea to our setting. We use the homomorphic addition properties of LROTP

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

to compute a 4-tuple of encryptions (all under the same *key*):

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

the plaintexts underlying the 4 ciphertexts in C are:

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

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

Now, as was the case for [SYY99], the *locations* of 0's and 1's might reveal (via the adversary's leakage) information about (v_i, v_j, r_k) beyond just the value of a_k. Trying to follow [SYY99], we might try to *permute* the ciphertexts before decrypting. Our problem, however, is that *any permutation we use might leak.* What we seek, then, is a method for randomly permuting the ciphertexts even under leakage.

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

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

Sub-Computation 1: Duplicate-Refresh-Permute. Create κ copies of the input key and ciphertext 4-tuples. Refresh each tuple-copy using key-ciphertext refresh as in Section 2.1 (each refresh uses independent randomness). Finally, permute each tuple-copy using an independent uniformly random permutation $\pi_i^j \in_R S_4$ (π_i^j is used in iteration i on the j-th refreshed tuple-copy).

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

We first observe that *without leakage* from sub-computation 1, all κ permutations $(\pi_{i,1}, \ldots, \pi_{i,\kappa})$ look independently and uniformly random.[8] Thus, given λ bits of leakage from sub-computation 1, where $\lambda < 0.1\kappa$, most permutations still look "fairly random": by a counting argument, even given the λ bits of leakage, the entropy in many of the permutations $(\pi_{i,1}, \ldots, \pi_{i,\kappa})$ will remain

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

high. In other words, while significant leakage can occur on *some* of the permutations, it cannot occur on *all* of them. After this leakage occurs, in sub-computation 2 we choose one of the tuple-copies $j^* \in \{1, \dots, \kappa\}$ (and its permutation) uniformly at random and set $\pi_i \leftarrow \pi_i^{j^*}$. By the above, with constant probability we get that π_i has high entropy even given the leakage. Composing the permutations chosen in many iterations, with overwhelming probability in a constant fraction of the iterations the permutation chosen has high entropy. When this is the case, the composed permutation is statistically close to uniform. See Section 7 for further details on *Permute* and a formal statement and proof of its security properties.

2.3 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 various circuit wires.

Ciphertext Generation under Continual Leakage. We seek a procedure for repeatedly generate secure LROTP key-ciphertext pairs with a fixed underlying plaintext bit. The underlying plaintexts will be as before in the construction of Section 2.2: for each y-input wire i corresponding to the j-th bit of y, the underlying plaintext should be $y[j]$. For each x-input wire and for the output wire *output*, the underlying plaintext should 0. For each internal wire (and for the output wire), we will generate a key-ciphertext pair with underlying plaintext 1. Finally, we also seek a procedure for repeatedly generating a two LROTP key-ciphertext pairs $(\vec{\ell_i}, \vec{c_i})$ and $(\vec{\ell'_i}, \vec{c'_i})$ whose underlying plaintexts are a uniformly random bit $r_i \in \{0, 1\}$ (the same bit in both pairs).

For security, the underlying plaintexts of the keys and ciphertexts produced should be completely protected even under continual leakage on the repeated generations. In previous works such as [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 used for repeated secure generations with a fixed underlying plaintext bit. This is what is needed for input wires and for the output wire. We then describe how to "randomize" the fixed underlying plaintext bit to be uniformly random (which is used to repeatedly generate two key-ciphertext pairs with a uniformly and independently random underlying plaintext).

A ciphertext bank is initialized once using a *BankInit*(b) procedure, where b is either 0 or 1 (there is no leakage during initialization). It can then be used, via a *BankGen* procedure, to repeatedly generate key-ciphertext pairs with underlying plaintext bit b, for an unbounded polynomial number of generations. We refer to b as the Ciphertext Bank's underlying plaintext bit. We also provide a *BankGenRand* procedure for generating pairs of key-ciphertext pairs with a uniformly random underlying plaintext bit. Informally, the ciphertext bank security property is that, even under leakage from the repeated generations, the plaintext underlying each key-ciphertext pair is protected. More formally, there are efficient simulation procedures that have arbitrary control over the plaintexts underlying all key-ciphertext pairs that the bank produces. Leakage from the simulated calls is statistically close to leakage from the "real" ciphertext bank calls. We outline these procedures below, see Section 6 for further details.

Using ciphertext banks, we modify the initialization and evaluation outlined in Section 2.2. In initialization, we initialize a ciphertext bank with a fixed underlying plaintext bit for each input wire, for each internal wire (with underlying plaintext 1), and two banks for the output wire (see

Section 2.2 for all the fixed underlying plaintexts). We also initialize a ciphertexts bank with a random underlying plaintext bit, which will be used for generating pairs of key-ciphertext pairs with a random underlying plaintext for the internal wires ((see Section 2.2. Before each evaluation, we use these ciphertext banks to generate all of the key-ciphertext pairs that are needed for each circuit wire. After this first step, evaluation proceeds as outlined in Section 2.2. The full *Init* and *Eval* procedures are in Section 8.

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

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

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

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

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

Observe, however, that computing an linear combination of C's columns does not require loading C into memory in its entirety. Instead, we can compute the linear combination in a "piecemeal" manner: first, load only $(c \cdot \kappa)$ columns of C into memory (for a small $0 < c < 0.5$). Compute their contribution $\vec{x}_1$ to the linear combination. Then, load $\vec{x}_1$ into memory together with the next $(c \cdot \kappa)$ columns of C, and add $\vec{x}_1$ to these columns' contribution to the linear combination. This gives $\vec{x}_2$, which is the contribution of the first $(2c \cdot \kappa)$ columns to the linear combination. We can continue this process for $(2/c)$ sub-steps, eventually computing the linear combination of C's columns without ever loading C into memory in its entirety. All we need to load into memory at one time is a collection of $((c \cdot \kappa) + 1)$ linear combinations of columns of C. We call each such collection a "sketch" (or a "piece") of C. We prove that *sketches of random matrices are leakage resilient*, and in particular leakage from sketches of C is statistically close in the real and simulated distributions (i.e. when C is of rank $\kappa - 1$ or uniformly random). Thus, the above procedure for computing a linear combination of C's rows is leakage resilient. Similarly, we show how to implement *BankGen* and *SimBankGen* using sub-computations, where each sub-computation only loads a single "sketch" of C into memory. We use this to show security of the ciphertext bank for any unbounded (polynomial) number of generations. We view these proofs as our most important technical contribution.

2.4 Leaky CPU: What are The Universal Instructions?

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

We elaborate here on the set of universal CPU instructions required. These are fairly simple and straightforward. They include instructions for generating a random matrix/vector of 0/1 bits, for vector-vector addition and multiplication (i.e. inner product), for matrix-matrix addition, and for matrix-vector and matrix-matrix multiplication. Beyond these, the only additional functionality used is permuting a sequence of vectors. This, in a nutshell, is a complete (high-level) list of the required instructions. This set of instructions suffices for LROTP operations such as decryption, key and ciphertext refresh, homomorphic operations (implemented using vector operations), as well as for the ciphertext banks outlined in Section 2.3 (implemented using matrix-matrix and matrix-vector multiplication). The *SafeNAND* and *Permute* procedures outlined above use these procedures, as well as a duplicate-refresh-and-permute operation. This operation can be implemented as a single atomic instruction (as described above), or as a sequence of instructions for duplication, refreshing and permuting. Both implementations are leakage resilient.

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

2.5 Organization and Roadmap

We provide further comparison to past work in Section 3. Definitions, notation and preliminaries are in Section 4. This includes the definitions of secure compilers against leakage and technical 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 5 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 6. The *SafeNAND* procedure for securely computing NAND gates on encrypted inputs is in Section 7. These ingredients are put together in Section 8, where we present the main construction and a proof (sketch) of its security.

3 Further Related Work

We provide a more detailed comparison prior work *leakage-resilience compilers for general programs* in various continual leakage attack models. Comparing to the work of Goldwasser and Rothblum [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 (DDH in [GR10] and the existence of fully homomorphic encryption scheme (FHE) in [JV10]) as well as secure hardware. Our result, on the other hand, is unconditional and uses no secure hardware components.

In terms of *quantitative* bounds, for security parameter κ, [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 sub-circuits is of size $\text{poly}(\kappa) \cdot |C|$). Assuming a fully-homomorphic encryption scheme that (for the security parameter κ) is secure against adversaries that run in time T, their construction can withstand $O(\log T)$ bits of leakage on each sub-circuit. For example, if the FHE is secure against $\text{poly}(L)$-time adversaries, then the leakage bound is $O(\log L)$. In our new construction, for any leakage parameter L, there are $O(|C|)$ sub-computations (i.e. more sub-computations), each of size $\tilde{O}(L^{\omega})$, where ω is the exponent is the exponent in the best algorithm known for matrix multiplication (i.e. smaller). The new construction can withstand L bits of leakage from each sub-computation (i.e. the amount of leakage we can tolerate, relative to the sub-computation size, is larger). The quantitative parameters of [GR10] are similar to ours (up to polynomial factors).

The work of Ishai, Sahai and Wagner [ISW03] in the ISW-L leakage model, may be viewed as converting any circuit C into $O(|C|)$ sub-circuits each of size $O(L^2)$, and allow the leakage of L wire values from each sub-circuit. Our transformation converts C into $O(|C|)$ sub-circuits, each of size $\tilde{O}(L^{\omega})$, and allow the leakage of L bits from each sub-circuit where these bits can be the output of arbitrary computations on the wire values (rather than the wire values themselves as in [ISW03]).

The work of Faust *et al.* [FRR+10] in the CB-L model, under the additional assumption that leak free hardware components exist, shows how to convert any circuit C into a new circuit C' of size $O(|C| \cdot L^2)$, which is resilient to leakage of the result of any $\mathcal{AC}^0$ function f of output length L computed on the entire set of wire values. Qualitatively, the main differences are (*i*) that construction used secure hardware, whereas we do not use secure hardware, and (*ii*) in terms of the class of leakage tolerated, they can handle bounded-length $\mathcal{AC}^0$ leakage *on the entire computation* of each execution. We, on the other hand, can handle length bounded OC-L leakage *of arbitrary complexity* that operates separately (if adaptively) on each sub-computation. A more recent result of [Rot12] removes the need for hardware component and shows how to convert C into C' of size $O(|C| \cdot poly(L)$ which is resilient against $\mathcal{AC}^0$ leakage functions of length L, under the computational assumption that computing inner-product cannot be done in $\mathcal{AC}^0$, even if polynomial time pre-processing (of the inputs to the inner product) is allowed. [Rot12] uses ciphertext banks, a tool introduced in this work.

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

3.1 Other Related Work

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.

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 another independent work by Dziembowski and Faust [DF11], they show how to compile a variety of well known cryptosystems (i.e. El Gamal PKC and Okamoto identification) into leakage resilient variants that resist OC-L attacks. Their results assumes the existence of hardware components (and retain the same computational assumption of the underlying cryptosystem).

In the context of a **bounded** number of executions, we remark that the work of Goldwasser, Kalai and Rothblum [GKR08] on one-time programs implies 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.

4 Definitions and Preliminaries

In this section we define leakage and multi-source leakage attacks (Section 4.1) and give a brief exposition about entropy, multi-source extractors, and facts about them that will be used throughout this work (Section 4.2). We then define and discuss the notion of independence up to orthogonality (Section 6.3.2).

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. We use $x^{-(i)}$ to denote the string formed from x by replacing the i-th symbol of x with $\perp$. Similarly, we use $x^{-(i_1,i_2,\dots,i_k)}$ to denote the string formed from x by replacing the i_1-th, i_2-th, ..., i_k-th symbols of x with $\perp$.

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.

4.1 Leakage Model

We build on 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 4.1. *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 4.2 (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 4.3 (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 4.2). 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. When the leakage bound on all k sources is identical we use a "global" leakage bound λ and denote this by $\mathcal{A}^\lambda(x)\{s_1, \ldots, s_k\}$. Finally, we remark that each source may be a data string or a computation. We occasionally use square braces, e.g. $[s_i]^{\lambda_i}$, to emphasize that source s_i is *not* exposed in its entirety, but rather only via a leakage attack.

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 4.4 (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 4.2). 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 4.2. 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 4.3, we drop the superscript if a source s exposed in its entirety.[9] 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.

4.2 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 4.5 (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 4.6 ((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:

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

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

[9]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.

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).

Lemma 4.7 (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 4.8 (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, $H_\infty(X'), H_\infty(Y') \geq k - \lambda - \log(1/\delta)$.*

4.3 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.[10]

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 4.9 ($(\lambda(\cdot), \delta(\kappa))$ Continuous Leakage Secure Compiler)**.** We say that a compiler $(\mathit{Init}, \mathit{Eval})$ 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: $\mathit{Init}(1^\kappa, C_n, y)$ runs in time $\mathrm{poly}(\kappa, n)$ and outputs an initial state state_0
- Evaluation: for every integer $t \leq \mathrm{poly}(\kappa)$, the evaluation procedure is run on the previous state 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:

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

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

[10]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 $\mathit{Real}_{\mathcal{A}}$ of $\mathcal{A}$ when adaptively choosing $T = \mathrm{poly}(\kappa)$ inputs $(x_1, x_2, \ldots x_T)$ while running a continuous leakage attack on the sequence $(\mathit{Eval}(\mathit{state}_0, x_1), \ldots, \mathit{Eval}(\mathit{state}_{T-1}, x_T))$, with adaptively and adversarially chosen x_t's, is $(\delta(\kappa))$-statistically close to the view $\mathit{Simulated}_{\mathcal{A}}$ 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 $\mathit{Eval}(\mathit{state}_{t-1}, x_t)$ (see Definition 4.2). $\mathit{Real}_{\mathcal{A},t}$ 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

  $$\mathit{Real}_{\mathcal{A}} = (\mathit{Real}_{\mathcal{A},1}, \ldots, \mathit{Real}_{\mathcal{A},T})$$

  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:

  $$\mathit{state}_0 \leftarrow \mathit{SimInit}(1^\kappa, C_n)$$

  $$x_t \leftarrow \mathcal{A}(\mathit{Simulated}_{\mathcal{A},1}, \ldots, \mathit{Simulated}_{\mathcal{A},t-1})$$

  $$(\mathit{state}_t, \mathit{Simulated}_{\mathcal{A},t}) \leftarrow \mathit{SimEval}(\mathit{state}_{t-1}, x_t, , C(y, x_t), \mathcal{A}, \mathit{Simulated}_{\mathcal{A},1}, \ldots, \mathit{Simulated}_{\mathcal{A},t-1})$$

  where $\mathit{Sim}_{\mathcal{A},t}$ 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

  $$\mathit{Simulated}_{\mathcal{A}} = (\mathit{Simulated}_{\mathcal{A},1}, \ldots, \mathit{Simulated}_{\mathcal{A},T})$$

  We require that the two views $\mathit{Real}_{\mathcal{A}}$ and $\mathit{Simulated}_{\mathcal{A}}$ 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.

5 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.

We use the following notation to denote key-ciphertext pairs

Definition 5.1 (LROTP$^\kappa_{\vec{b}}$ distribution)**.** For $\kappa \in \mathbb{N}$ and $\vec{b} \in \{0,1\}^m$, we use the following shorthand to denote drawing a fresh key and m fresh ciphertexts with underlying plaintexts as per $\vec{b}$:

$$\mathrm{LROTP}^\kappa_{\vec{b}} = (\mathit{key}, C)_{\mathit{key} \leftarrow \mathit{KeyGen}(1^\kappa), C[i] \leftarrow \mathit{Encrypt}(\mathit{key}, \vec{b}[i])}$$

Leakage-Resilient One-Time Pad (LROTP) Cryptosystem $(\mathit{KeyGen}, \mathit{Encrypt}, \mathit{Decrypt})$

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

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

5.1 Semantic Security under Multi-Source Leakage

Definition 5.2 (Semantic Security Under $\lambda(\cdot)$-Multi-Source Leakage)**.** An encryption scheme $(\mathit{KeyGen}, \mathit{Encrypt}, \mathit{Decrypt})$ 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 $b \in_R \{0,1\}$ and $(\mathit{key}, \vec{c}) \sim \mathrm{LROTP}_b^\kappa$.
2. The adversary launches a leakage attack on key and $\vec{c}$, and outputs a "guess" b':

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

the adversary wins if $b' = b$.

The LROTP cryptosystem is semantically secure in the presence of multi-source leakage with leakage bound $\lambda(\kappa) = \kappa/3$. This follows from Lemma 5.4 below.

Lemma 5.3. *Fix an integer $m > 0$. For every leakage bound $\lambda(\kappa)$, every multi-source adversary $\mathcal{A}$, every $\kappa \in \mathbb{N}$, and every $\vec{b} \in \{0,1\}^m$, consider:*

$$\mathcal{D} \quad = \quad \left(\mathcal{A}^{\lambda(\kappa)}[\mathit{key}, C]\right)_{(\mathit{key},C) \sim \mathit{LROTP}_{\vec{b}}^\kappa}$$

For any w in the support of $\mathcal{D}$, take $\mathcal{K}(w)$ to be the conditional marginal distribution of key, conditioned on $\vec{b}$ and on leakage w, and $\mathcal{C}(w)$ to be the conditional marginal distribution of C, conditioned on $\vec{b}$ and on leakage w. Then:

1. *The marginal distributions $\mathcal{K}(w)$ and $\mathcal{C}(w)$ are independent of $\vec{b}$. I.e., for every w in the support and $\vec{b} \in \{0,1\}^m$, these marginal distributions are identical.*
2. *The joint distribution of (key, C) conditioned on w equals the product distribution $\mathcal{K}(w) \times \mathcal{C}(w)$, conditioned on $\mathit{key}' \sim \mathcal{K}(w)$ and $C' \sim \mathcal{C}(w)$ satisfying $\langle \mathit{key}, C \rangle = \vec{b}$.*

Proof. Take $w = \mathcal{A}^\lambda[\mathit{key}, C]$. The leakage operates separately on key and on C, and thus there exist two sets $S_{\mathit{key}}(w) \subseteq \{0,1\}^\kappa$ and $S_C(w) \subseteq \{0,1\}^{\kappa \cdot m}$, s.t.:

$$w = \mathcal{A}^\lambda[\mathit{key}, C] \Leftrightarrow (\mathit{key}, C) \in S_{\mathit{key}}(w) \times S_C(w)$$

Note that for fixed leakage w, the sets $S_{key}(w)$ and $S_C(w)$ are well defined and completely independent of the underlying plaintexts $\vec{b}$ (though the distribution of w itself may depend on $\vec{b}$).

We take $\mathcal{K}(w)$ to be *key* conditioned on $key \in S_{key}(w)$, and $\mathcal{C}(w)$ to be $\mathcal{C}$ conditioned on $C \in S_C(w)$.

Let $\mathcal{X} = \mathrm{LROTP}^{\kappa}_{\vec{b}}$ be the initial joint distribution of (key, C). I.e., *key* and C drawn uniformly at random s.t. the inner products equal $\vec{b}$. For w in the support of $\mathcal{D}$, let $\mathcal{X}(w) = (\mathcal{X}|w)$ be the distribution of (key, C), conditioned on leakage w. By the above, $\mathcal{X}(w)$ is $\mathcal{X}$ conditioned on $(key, C) \in S_{key}(w) \times S_C(w)$. Thus, $\mathcal{X}(w)$ is the product distribution $\mathcal{K}(w) \times \mathcal{C}(w)$, conditioned on $\langle key, C\rangle = \vec{b}$. ■

Lemma 5.4. *For $\kappa \in \mathbb{N}$, for an integer $m \le 0.1\kappa$, and leakage bound $\lambda(\kappa) = \kappa/3$,for every multi-source adversary $\mathcal{A}$, and for every $\vec{b}, \vec{b}' \in \{0,1\}^m$, take:*

$$\begin{aligned}\mathcal{D} &= \left(\mathcal{A}^{\lambda(\kappa)}[key, C]\right)_{(key,C)\sim LROTP^{\kappa}_{\vec{b}}}\\ \mathcal{D}' &= \left(\mathcal{A}^{\lambda(\kappa)}[key, C]\right)_{(key,C)\sim LROTP^{\kappa}_{\vec{b}'}}\end{aligned}$$

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

Proof of Lemma 5.4. For $w = \mathcal{A}^{\lambda}[key, C]$. Let $\mathcal{X}(w)$ be the distribution of $(key, C) \sim \mathrm{LROTP}^{\kappa}_{\vec{b}}$ conditioned on leakage w, and $\mathcal{X}'(w)$ the distribution of $(key, C) \sim \mathrm{LROTP}^{\kappa}_{\vec{b}'}$ conditioned on leakage w. By Lemma 5.3, the conditional distributions $\mathcal{K}(w)$ and $\mathcal{C}(w)$ of *key* and of C (respectively) are independent of $\vec{b}$.

For fixed w, define $\beta(w)$ to be the statistical distance of the inner product $\langle key, C\rangle_{key\sim\mathcal{K}(w), C\sim\mathcal{C}(w)}$ from uniform. We will show that with overwhelming probability $\beta(w)$ is exponentially small (Claim 5.6), and also:

Claim 5.5. *For any $w \in Support(\mathcal{D})$:*

$$1 - O(\beta(w)) \le \frac{\mathcal{D}'[w]}{\mathcal{D}[w]} \le 1 + O(\beta(w))$$

Proof. By Lemma 5.3 and Bayes' Rule:

$$\begin{aligned}\mathcal{D}[w] &= \Pr_{key\sim\mathcal{K}, C\sim\mathcal{C}: \langle key, C\rangle = \vec{b}}[(key, C) \in S_{key}(w) \times S_C(w)]\\ &= \frac{\Pr_{key\sim\mathcal{K}, C\sim\mathcal{C}}[((key, C) \in S_{key}(w) \times S_C(w)) \bigwedge (\langle key, C\rangle = \vec{b})]}{\Pr_{key\sim\mathcal{K}, C\sim\mathcal{C}}[\langle key, C\rangle = \vec{b}]}\\ &= 2^m \cdot \Pr_{key\sim\mathcal{K}, C\sim\mathcal{C}}[((key, C) \in S_{key}(w) \times S_C(w)) \bigwedge (\langle key, C\rangle = \vec{b})]\end{aligned}$$

Similarly:

$$\mathcal{D}'[w] = 2^m \cdot \Pr_{key\sim\mathcal{K}, C\sim\mathcal{C}}[((key, C) \in S_{key}(w) \times S_C(w)) \bigwedge (\langle key, C\rangle = \vec{b}')]$$

The inner product of $key \sim \mathcal{K}(w)$ and $C \sim \mathcal{C}(w)$ is $\beta(w)$-close to uniform, and so:

$$\frac{1}{2^m} + \frac{\beta(w)}{2} < \Pr_{key\sim\mathcal{K}, C\sim\mathcal{C}}[((key, C) \in S_{key}(w) \times S_C(w)) \bigwedge (\langle key, C\rangle = \vec{b})] < \frac{1}{2^m} - \frac{\beta(w)}{2}$$

$$\frac{1}{2^m} + \frac{\beta(w)}{2} < \Pr_{key \sim \mathcal{K}, C \sim \mathcal{C}}[((key, C) \in S_{key}(w) \times S_C(w)) \bigwedge (\langle key, C \rangle = \vec{b}')] < \frac{1}{2^m} - \frac{\beta(w)}{2}$$

And the claim follows. ■

Claim 5.6. *With all but* $\exp(-\Omega(\kappa))$ *probability over* $w \sim \mathcal{D}$*,* $\beta(w) = \exp(-\Omega(\kappa))$*.*

Proof. By Lemma 4.8, with all but δ probability over $w \sim \mathcal{D}$, we have that $H_\infty(\mathcal{K}(w)) + H_\infty(\mathcal{C}(w)) \geq (m + 1 - 0.5) \cdot \kappa$. When this is the case, by Lemma 4.7 we have $\beta(w) = \exp(-\Omega(\kappa))$. ■

By Claim 5.5 and 5.6 we conclude that $\Delta(\mathit{Real}, \mathit{Simulated}) = \exp(-\Omega(\kappa))$. ■

5.2 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 $key' \leftarrow \mathit{KeyRefresh}(key, \sigma)$, 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.

LROTP key and ciphertext refresh

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

Figure 2: LROTP key and ciphertext refresh Cryptosystem

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. $\sigma \leftarrow \mathit{KeyEntGen}(1^\kappa)$
2. $key' \leftarrow \mathit{KeyRefresh}(key, \sigma)$
3. $\vec{c}' \leftarrow \mathit{CipherCorrelate}(\vec{c}, \sigma)$
4. $\pi \leftarrow \mathit{CipherEntGen}(1^\kappa)$

5. $\vec{c}'' \leftarrow \mathit{CipherRefresh}(\vec{c}', \pi)$

6. $\mathit{key}'' \leftarrow \mathit{KeyCorrelate}(\mathit{key}', \pi)$

The output of the refresh operation is $(\mathit{key}'', \vec{c}'')$. 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 2.3 and 7.3). *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 $(\mathit{key}_1, \vec{c}_1)$ 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 $\mathit{key}'_1 = (\mathit{key}_1 + \sigma)$ 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 $\mathit{key}_2 = \mathit{KeyCorrelate}(\mathit{key}'_1, \pi)$ 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.

5.3 "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 2.3 and 7 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).

6 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 for an overview.

This section is organized as follows. In Section 6.1 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 (and their proofs) make use of secure procedures for piecemeal matrix multiplication and for refreshing collections of ciphertexts, which are in section Section 6.2. In Section 6.3 we define piecemeal attacks on matrices and prove that random matrices are resilient to piecemeal leakage. In Section 6.4 we state and prove security properties of piecemeal matrix multiplication (these are used in the security proofs of Section 6.1 to prove security of the ciphertext bank).

6.1 Ciphertext Bank: Interface and Security

We present a full description of the ciphertext bank procedures and simulator. Recall that (as in Section 5), 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. The *BankGen* procedure then injects new entropy into the bank's internal state. 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 *BankGenRand* procedure. This procedure re-draws a uniformly random plaintext bit that will underly ciphertexts produced by the bank, and then produces two key-ciphertext pairs with this underlying plaintext bit. 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 begins by injecting new entropy into the key (see below). It then 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*. It then injects new entropy into C and (again) into the key. Key refresh procedures are performed 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 *BankGenRand* 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. It then generates two key-ciphertext pairs with this new underlying plaintext bit using the *BankGen* procedure.

The full ciphertext bank procedures are in Figure 3. The piecemeal matrix multiplication, addition, and key refresh procedures are below in Section 6.2.

Simulated Ciphertext Bank. 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,

```
$\underline{\mathit{BankInit}(1^\kappa, b)}$: initializes a ciphertext bank; No leakage

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

$\underline{\mathit{BankGen}(\mathit{Bank})}$: generates a new ciphertext; Under leakage

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

$\underline{\mathit{BankGenRand}(\mathit{Bank})}$: generates a ciphertext pair with random plaintext; Under leakage

    1. redraw the underlying plaintext: pick $\vec{v} \in_R \{0,1\}^\kappa$, compute $D \leftarrow \mathit{PiecemealAdd}(C, \vec{v})$
       let $\mathit{Bank} \leftarrow (\mathit{key}, D)$
    2. generate the first ciphertext: $(\mathit{key}, \vec{c}) \leftarrow \mathit{BankGen}(\mathit{Bank})$
    3. generate the second ciphertext: $(\mathit{key}', \vec{c}') \leftarrow \mathit{BankGen}(\mathit{Bank})$
    4. output $(\mathit{key}, \vec{c}, \mathit{key}', \vec{c}')$
```

Figure 3: Ciphertext Bank

unlike in the real ciphertext bank, the plaintexts underlying C's columns are independent and uniformly 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, and keeps track of changes to the vector $\vec{x}$ of underlying plaintexts when new entropy is injected into the bank. Finally, we also provide a *SimBankGenRand* procedure, which operates similarly to *BankGenRand*, except that that it too keeps track of changes to the vector $\vec{x}$ of plaintext bits underlying C. The simulation procedures are in Figure 4.

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*. A *sequence of simulated generations* is similar, except that initialization is performed using *SimBankInit*, each

```
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 $\vec{r}$)
    3. update $\vec{x}$ to contain the new bits underlying the updated $C$

SimBankGenRand(Bank, b, b')

    1. run exactly as in BankGenRand, except replace the first call to SimBankGen with a call to
       SimBankGen(b), and replace the second call with a call to SimBankGen(b')
       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

generation is performed by specifying an underlying plaintext bit b and then calling *SimBankGen*.

We also consider sequences of generations of pairs of key-ciphertext pairs, each pair of pairs has the same uniformly random underlying plaintext bit. 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 pair generations, each performed by a call to *BankGenRand* to get two keys and two ciphertexts with the same underlying plaintexts. A *sequence of simulated random generations* is performed similarly, except that *BankInit* and *BankGenRand* are replaced by *SimBankInit* and *SimBankGenRand* plaintext bits b, b' (we will not always use the same plaintext bit 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 6.1 below, which we view as the most important technical contribution in this section. The proof builds on the security of piecemeal matrix computations (see sections 6.2 and 6.4 for the full procedures and their security properties).

Lemma 6.1. *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 $\leftarrow$ BankInit(b), and in Simulated we begin by running Bank $\leftarrow$ SimBankInit (both without leakage):

$$\begin{aligned} \mathit{Real} \;=\; & \mathcal{A}\big\{[(\mathit{key}_0, \vec{c}_0) \leftarrow \mathit{BankGen}(\mathit{Bank})]^{\lambda(\kappa)}, \mathit{key}_0, \vec{c}_0, \\ & [(\mathit{key}_1, \vec{c}_1) \leftarrow \mathit{BankGen}(\mathit{Bank})]^{\lambda(\kappa)}, \mathit{key}_1, \vec{c}_1, \\ & \ldots \\ & [(\mathit{key}_{T-1}, \vec{c}_{T-1}) \leftarrow \mathit{BankGen}(\mathit{Bank})]^{\lambda(\kappa)}, \mathit{key}_{T-1}, \vec{c}_{T-1}\big\} \end{aligned}$$

$$\begin{aligned} \mathit{Simulated} \;=\; & \mathcal{A}\big\{[(\mathit{key}_0, \vec{c}_0) \leftarrow \mathit{SimBankGen}(\mathit{Bank}, b)]^{\lambda(\kappa)}, \mathit{key}_0, \vec{c}_0, \\ & [(\mathit{key}_1, \vec{c}_1) \leftarrow \mathit{SimBankGen}(\mathit{Bank}, b)]^{\lambda(\kappa)}, \mathit{key}_1, \vec{c}_1, \\ & \ldots \\ & [(\mathit{key}_{T-1}, \vec{c}_{T-1}) \leftarrow \mathit{SimBankGen}(\mathit{Bank}, b)]^{\lambda(\kappa)}, \mathit{key}_{T-1}, \vec{c}_{T-1}\big\} \end{aligned}$$

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

Proof. We prove here the case that $b = 0$, the case $b = 1$ is similar. We consider the T generations, real or simulated, and keep track of the values of various internal variables as the computation proceeds.

Internal Variables. For the t-th generation (where t goes from 0 to $T-1$): (key_t, C_t) denote the bank before the t-th generation, with underlying plaintexts $\vec{x}_t$. The randomness used to generate the t-th output ciphertext is $\vec{r}_t$, the matrix used to refresh the bank is R_t, and the key refresh value is σ_t. We use D_t to denote the intermediate ciphertext bank in the t-th generation after key refresh, but before multiplication with R_t. The output of the t-th generation is the key-ciphertext pair $(\mathit{key}_t, \vec{c}_t)$.

Hybrids. 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 ciphertext, 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 as follows.

For $t > 0$, $\mathcal{H}_t$ is initialized using (key_0, C_0) as in *Simulated* (n $\mathcal{H}_0$ we initialize (key_0, C_0) as in *Real*. We then run T ciphertext generations under $\mathcal{A}$'s leakage attack. For the i-th generation in $\mathcal{H}_t$, the key refresh value σ_i is uniformly random. For $i < t$, we choose $\vec{r}_i$ uniformly at random s.t. it has odd parity and is in $\mathit{kernel}(\vec{x}_i)$. 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 $\mathit{kernel}(\vec{x}_i)$. In particular, this means that for $i < t$, the distribution of C_i is uniformly random, and $\vec{x}_i$ specifies the underlying plaintexts in C's columns (as in *Simulated*). For $i \geq t$, the columns of C_i are orthogonal to *key*, and $\vec{x}_i$ is the zero vector (and has no effect on the distribution).

By construction, we get that $\mathcal{H}_0 = \mathit{Real}$ and $\mathcal{H}_{T+1} = \mathit{Simulated}$. It remains to show that, for all $t \in [T+1]$, $\Delta(\mathcal{H}_t, \mathcal{H}_{t+1}) = \exp(-\Omega(\kappa))$. We show this here for $2 < t < T-1$ (the borderline cases are handled similarly).

We use an intermediate distribution $\mathcal{H}'_t$, which operates as $\mathcal{H}_t$, except that it chooses the vector $\vec{x}_t$ uniformly at random (recall that in $\mathcal{H}_t$ the columns of C_t are all in $\mathit{kernel}(\mathit{key})$, and $\vec{x}_t$ is the zero vector). It then chooses $\vec{r}_t$ and the columns of R_t to be uniformly random with odd parity and in $\mathit{kernel}(\vec{x}_t)$ (whereas in $\mathcal{H}_t$ these were uniformly random with odd parity and no further restriction).

The Lemma will follow from Claims 6.2 and 6.3 below (the proofs will depend on technical lemmas on leakage-resilience of piecemeal matrix operations, see Sections 6.2 6.4).

Claim 6.2. $\Delta(\mathcal{H}'_t, \mathcal{H}_{t+1}) = \exp(-\Omega(\kappa))$

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 have odd parity in both $\mathcal{H}_{t+1}$ and $\mathcal{H}'_t$, but in $\mathcal{H}'_t$ they are also all in $\mathit{kernel}(\vec{x}_{t-1})$, (ii) in $\mathcal{H}'_t$, 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).

Statistical closeness of the views will follow from Lemma 6.22 in Section 6.4. That Lemma considers the multiplication of two matrices A and B under piecemeal leakage from the matrices. The attack draws uniformly random vectors key and $\vec{x}$, and takes B to have columns that are orthogonal to $\vec{x}$, and considers two cases. In the first case, A has columns orthogonal to key (and is independent of $\vec{x}$). In the second case, $\vec{x}$ specifies the inner products of A's columns with key. Lemma 6.22 shows that piecemeal leakage from the multiplication of A and B (together with piecemeal leakage from key and A), even in conjunction with key and with the product of A and B. To reduce the attack of Lemma 6.22 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 also include the vector $\vec{r}_t$ in B, its distribution is always identical to that of R_t's columns).

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 $(\mathit{key}_i, \vec{c}_i)$ 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 6.22).

Using the above reduction, we conclude from Lemma 6.22 that the leakage from the t-th generation, together with $(\mathit{key}_t, \vec{c}_t, \sigma_t, C_{t+1})$ (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 $(\mathit{key}_t, \sigma_t, C_{t+1})$).

We need, however, to also generate the leakage for the ciphertext generations that precede the t-th. Recall that the $(\mathit{key}_i, \vec{c}_i)$ 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, the reduction chooses D_{t-1} uniformly at random ("in public") . 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. I.e., for each piece of R_{t-1} in the the computation $C_t \leftarrow D_{t-1} \times R_{t-1}$ (this access several rows of R_{t-1} at a time), the reduction can choose a uniformly collection of rows that satisfy the equation $C_t \leftarrow D_{t-1} \times R_{t-1}$ for the piece being computed. Note that, in particular, the distributions of R_{t-1} that we will get in the two scenarios of Lemma 6.22 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 6.22 to generate the leakage from the t-th generation on. This yields the views $\mathcal{H}'_t$ and $\mathcal{H}_{t+1}$. By Lemma 6.22, these views are statistically close. The full statement and proof of Lemma 6.22 are in Section 6.4. ■

Claim 6.3. $\Delta(\mathcal{H}_t, \mathcal{H}'_t) = \exp(-\Omega(\kappa))$

Proof. The only difference between the hybrids is in the distribution of $\vec{r_t}$ and R_t (in the t-th generation). In $\mathcal{H}_t$ the vector $\vec{r_t}$ and columns of R_t are uniformly random with odd parity. In $\mathcal{H}'_t$ there is an additional restriction that these vectors are all orthogonal to $kernel(\vec{x_t})$: i.e. they are all in a (random) subspace of dimension $(2\kappa - 1)$ ($\vec{x_t}$ is a uniformly random vector independent of any of the other variables in the construction). Note that $\vec{r_t}$ and R_t are independent of key_t, C_t, σ_t.

Lemma 6.23 in Section 6.4 shows that bounded-length piecemeal leakage from a matrix is statistically close in the cases where the matrix columns are uniformly random with odd parity, and where the matrix columns are uniformly random with odd parity and in a subspace of rank $2\kappa - 1$.

We generate the variables for all generations, up to (key_{t+1}, D_t) in the t-th generation. By Lemma 6.23, the distributions of these variables (and the leakage from generations $0, \ldots, t+1$), and the leakage from the t-th generation (using piecemeal leakage on R_t) are statistically close in $\mathcal{H}_t$ and $\mathcal{H}'_t$. We can create the leakage from the latter rounds as a function of the public key_{t+1}, D_t, and piecemeal leakage from R_t. This is done similarly to the proof of Claim 6.2 (there we used similar ideas to compute leakage from earlier generations). See Section 6.4 for the statement and proof of Lemma 6.23. ■

■

Single simulated sequence: each generation is *strongly* secure. Consider an OC leakage adversary that attacks a sequence of T *simulated* generations, generating key-ciphertext pairs with underlying plaintexts $\vec{b} \in \{0,1\}^T$. The adversary "targets" a single generation in this sequence, say the i-th generation with underlying plaintext $\vec{b}[i]$. We show that the entire view of the adversary, in conjunction with all the generated key-ciphertexts pairs except the i-th pair, and the final state of the (simulated) ciphertext bank after the generations are complete, can be generated by a simulator that is only given multi-source leakage access to a freshly generated LROTP key-ciphertext pair $(key^*, \vec{c}^*)$ with underlying plaintext $\vec{b}[i]$, and is also given all bits of $\vec{b}$ except the i-th bit. This is stated formally in Lemma 6.4 below.

Notice that, in particular, this means that a adversary cannot determine the i-th underlying plaintext bits *even given all other keys and ciphertexts generated*: if the adversary could determine the i-th underlying plaintext, then it could also break the security of the LROTP scheme using multi-source leakage (which is impossible by Lemma 5.4).

Lemma 6.4. *There exist: a simulator Sim s.t. for any leakage bound λ, security parameter $\kappa \in \mathbb{N}$, execution bound $T = \mathrm{poly}(\kappa)$, vector $\vec{b} \in \{0,1\}^T$, "target" round $i \in [T]$, and any (computationally unbounded) leakage adversary $\mathcal{A}$, the following holds:*

Let $\mathcal{D}$ and $\mathcal{E}$ be the following distribution, where in $\mathcal{D}$ we begin by running Bank $\leftarrow$ SimBankInit (without leakage):

$$\begin{aligned}
\mathcal{D} \;=\; & \mathcal{A}^{\lambda(\kappa)}\big\{[(key_0, \vec{c}_0) \leftarrow \mathit{SimBankGen}(\mathit{Bank}, \vec{b}[0])], key_0, \vec{c}_0, \\
& [(key_1, \vec{c}_1) \leftarrow \mathit{SimBankGen}(\mathit{Bank}, \vec{b}[1])], key_1, \vec{c}_1, \\
& \ldots, \\
& [(key_1, \vec{c}_1) \leftarrow \mathit{SimBankGen}(\mathit{Bank}, \vec{b}[i-1])], key_{i-1}, \vec{c}_{i-1}, \\
& [(key_1, \vec{c}_1) \leftarrow \mathit{SimBankGen}(\mathit{Bank}, \vec{b}[i])], \\
& [(key_1, \vec{c}_1) \leftarrow \mathit{SimBankGen}(\mathit{Bank}, \vec{b}[i+1])], key_{i+1}, \vec{c}_{i+1}, \\
& \ldots, \\
& [(key_{T-1}, \vec{c}_{T-1}) \leftarrow \mathit{SimBankGen}(\mathit{Bank}, \vec{b}[T-1])], key_{T-1}, \vec{c}_{T-1}, \\
& [key_i, \vec{c}_i], \mathit{Bank}\big\}
\end{aligned}$$

$$\mathcal{E} \;=\; \mathit{Sim}^{O(\lambda(\kappa))}(\vec{b}^{-(i)})[key_i, \vec{c}_i]_{(key_i, \vec{c}_i) \sim \mathit{LROTP}^{\kappa}_{\vec{b}[i]}}$$

then the distributions $\mathcal{D}$ and $\mathcal{E}$ are identical *(i.e. statistical distance 0).*

Proof. The simulator Sim has $\vec{b}^{-(i)}$ and multi-source leakage access to key_i and $\vec{c}_i$ (with underlying plaintext bit $\vec{b}[i]$), and wants to generate the adversary's view in the leakage attack. To do this, Sim chooses uniformly random matrix C_t of LROTP ciphertexts for each of the T generations (note that these matrices are independent of the underlying plaintexts in the simulated generations!). Sim also chooses a uniformly random LROTP key key_t for each of the T generations except the i-th (the key for the i-th generation, key_i, is only accessed via bounded length leakage). Sim also chooses uniformly random output ciphertext $\vec{c}_t$ s.t. $(key_t, \vec{c}_t)$ have underlying plaintext bit $\vec{b}[t]$ for all generations except the i-th generation.

By construction, the joint distribution of $\{(C_t, key_t, \vec{c}_t)\}_{t\in[T]}$ created by the simulator is identical to their joint distribution in $\mathcal{D}$. Now Sim simulates the sequence of generations with these values of $\{(C_t, key_t, \vec{c}_t)\}$. The only issue is simulating the leakage from the i-th generation, where Sim does not know the explicit values of key_i or of $\vec{c}_i$. Another (smaller) issue is simulating the leakage from the key refresh in the $(i-1)$-th generation, where Sim does not know the "target" key. This leakage is simulated using multi-source leakage access to key_i and to $\vec{c}_i$ as follows.

For the i-th generation, leakage from the generation of $\vec{c}_i$ is a function of $(\vec{c}_i, C_i)$, where C_i is explicitly known to Sim. Thus, this can be simulated using (bounded length) leakage from $\vec{c}_i$ only: the leakage function chooses a random linear combination $\vec{r}$ of odd weight s.t. $C_i \times \vec{r} = \vec{c}_i$, and then computes leakage as a function of C_i and $\vec{r}$. Leakage from the key refresh is only a function of (key_i, key_{i+1}, C_i), where key_{i+1} and C_i are explicitly known to Sim. Thus this (bounded length) leakage can be simulated using (bounded length) leakage from key_i only. The Finally, refreshing the ciphertexts in the bank is only a function of C_i and C_{i+1}, which are explicitly known to Sim. Thus, the leakage from the i-th generation can be computed via separate bounded-length leakage from key_i and from $\vec{c}_i$.

Similarly, for the $(i-1)$-th generation, leakage from the key refresh step can be simulated as a bounded-length function of key_i only (since all variables, including the $(i-1)$-th output ciphertext, are explicitly known to Sim). ∎

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 via *BankGenRand*. Consider also a "simulated" view for the same attacker, given leakage from a simulated sequence of calls, where each pair of calls to *SimBankGenRand* specify a uniformly random bit (the same bit for both generations, and 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 6.5 below. The proof is similar to that of Lemma 6.1 and is omitted.

Lemma 6.5. *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. In Real, we begin by running $Bank \leftarrow BankInit(0)$. In

Simulated we begin by running Bank $\leftarrow$ SimBankInit and we choose $\vec{b} \in_R \{0,1\}^T$:

$$\begin{aligned}
\mathit{Real} \quad = \quad & \mathcal{A}\{[(key_0, \vec{c}_0, key'_0, \vec{c}'_0) \leftarrow \mathit{BankGenRand}(\mathit{Bank})]^{\lambda(\kappa)}, (key_0, \vec{c}_0, key'_0, \vec{c}'_0), \\
& [(key_1, \vec{c}_1, key'_1, \vec{c}'_1) \leftarrow \mathit{BankGenRand}(\mathit{Bank})]^{\lambda(\kappa)}, (key_1, \vec{c}_1, key'_1, \vec{c}'_1), \\
& \ldots \\
& [(key_{T-1}, \vec{c}_{T-1}, key'_{T-1}, \vec{c}'_{T-1}) \leftarrow \mathit{BankGenRand}(\mathit{Bank})]^{\lambda(\kappa)}, \\
& (key_{T-1}, \vec{c}_{T-1}, key'_{T-1}, \vec{c}'_{T-1})\}
\end{aligned}$$

$$\begin{aligned}
\mathit{Simulated} \quad = \quad & \mathcal{A}\{[(key_0, \vec{c}_0, key'_0, \vec{c}'_0) \leftarrow \mathit{SimBankGenRand}(\mathit{Bank}, \vec{b}[0], \vec{b}[0])]^{\lambda(\kappa)}, (key_0, \vec{c}_0, key'_0, \vec{c}'_0), \\
& [(key_1, \vec{c}_1, key'_1, \vec{c}'_1) \leftarrow \mathit{SimBankGenRand}(\mathit{Bank}, \vec{b}[1], \vec{b}[1])]^{\lambda(\kappa)}, (key_1, \vec{c}_1, key'_1, \vec{c}'_1), \\
& \ldots \\
& [(key_{T-1}, \vec{c}_{T-1}, key'_{T-1}, \vec{c}'_{T-1}) \leftarrow \mathit{SimBankGenRand}(\mathit{Bank}, \vec{b}[T-1], \vec{b}[T-1])]^{\lambda(\kappa)}, \\
& (key_{T-1}, \vec{c}_{T-1}, key'_{T-1}, \vec{c}'_{T-1})\}
\end{aligned}$$

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

Single simulated sequence of random generations: each triplet is *strongly* secure. Consider an OC leakage adversary that attacks a sequence of T *simulated* generations of random pairs via *SimBankGenRand*, where each generation generates a pair of key-ciphertext pairs, with underlying plaintexts $\vec{b} \in \{0,1\}^{2T}$. The adversary "targets" *three* generation in this sequence, say $(i,j,k) \in [2T]^3$, with underlying plaintexts $\vec{b}[i], \vec{b}[j], \vec{b}[k]$. We show that the entire view of the adversary, in conjunction with all the generated key-ciphertexts pairs except the pairs from the (i,j,k) generations, and the complete state of the (simulated) ciphertext bank after all generations are complete, can be generated by a simulator that is only given multi-source leakage access to freshly generated LROTP key-ciphertext pairs $(key_i, \vec{c}_i), (key_j, \vec{c}_j), (key_k, \vec{c}_k)$ with underlying plaintext $\vec{b}[i], \vec{b}[j], \vec{b}[k]$, and is also given all bits of $\vec{b}$ except bits (i,j,k). This is stated formally in Lemma 6.6 below.

We note that, as was the case for a single sequence of (standard) generations, by the security of LROTP under multi-source leakage, this means that the underlying plaintext bits for the targeted keys and ciphertexts are strongly protected (even given all other keys and ciphertexts that were generated).

Lemma 6.6. *There exists a simulator Sim, s.t. for any leakage bound λ, security parameter $\kappa \in \mathbb{N}$, execution bound $T = \mathrm{poly}(\kappa)$, vector $\vec{b} \in \{0,1\}^{2T}$, "target" generations $(i,j,k) \in [2T]$, and any (computationally unbounded) leakage adversary $\mathcal{A}$, the following holds:*

Let $\mathcal{D}$ and $\mathcal{E}$ be the following distribution, where in $\mathcal{D}$ we begin by running Bank $\leftarrow$ SimBankInit *(without leakage):*

$$\begin{aligned}
\mathcal{D} \;\; = \;\; & \mathcal{A}^{\lambda(\kappa)}\big\{((key_0, \vec{c}_0, key_0', \vec{c}_0') \leftarrow \mathit{SimBankGenRand}(\mathit{Bank}, \vec{b}[0], \vec{b}[1])), key_0, \vec{c}_0, key_0', \vec{c}_0' \\
& ((key_1, \vec{c}_1, key_1', \vec{c}_1') \leftarrow \mathit{SimBankGenRand}(\mathit{Bank}, \vec{b}[2], \vec{b}[3])), , key_1, \vec{c}_1, key_1', \vec{c}_1' \\
& \ldots \\
& \qquad\qquad \textit{for generations of } i\textit{-th, } j\textit{-th, and } k\textit{-th key-ciphertext pairs, only leakage is released} \\
& \qquad\qquad \textit{(the explicit keys and ciphertexts are not released)} \\
& \ldots \\
& ((key_{T-1}, \vec{c}_{T-1}, key_{T-1}', \vec{c}_{T-1}') \leftarrow \mathit{SimBankGenRand}(\mathit{Bank}, \vec{b}[2T-2], \vec{b}[2T-1])), \\
& \quad key_{T-1}, \vec{c}_{T-1}, key_{T-1}', \vec{c}_{T-1}', \\
& [(key_i, key_j, key_k), (\vec{c}_i, \vec{c}_j, \vec{c}_k)], \mathit{Bank}\big\} \\
\mathcal{E} \;\; = \;\; & \mathit{Sim}^{O(\lambda(\kappa))}(\vec{b}^{-(i,j,k)})[(key_i, key_j, key_k), (\vec{c}_i, \vec{c}_j, \vec{c}_k)]_{((key_i, key_j, key_k), (\vec{c}_i, \vec{c}_j, \vec{c}_k)) \sim \mathit{LROTP}^{\kappa}_{(\vec{b}[i], \vec{b}[j], \vec{b}[k])}}
\end{aligned}$$

Proof. The proof is very similar to that of Lemma 6.4. The only difference is that here there are the key-ciphertext pairs that *Sim* cannot generate explicitly (instead of just one pair). As in the proof of Lemma 6.4, the simulator can explicitly generate all other key-ciphertext pairs and the ciphertext matrices in the bank in all generations. Leakage from the i-th, j-th and k-th generations can then be computed using bounded-length multi-source leakage from (key_i, key_j, key_k) and from $(\vec{c}_i, \vec{c}_j, \vec{c}_k)$. ■

6.2 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 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.

6.3 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

```
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 $(A_i, \sigma)$ for each $i$ separately
3. $key' \leftarrow KeyRefresh(key, \sigma)$; leakage on $(key, \sigma)$
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 $\kappa, \ell \in \mathbb{N}$

(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 6.7 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 6.3.1 with a warmup for the case of an attack on a single piece (Lemma 6.9). We then show security for large number of pieces in Section 6.3.3 (Lemma 6.14).

Definition 6.7 (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. We omit key in cases when the adversary does not get any access to it (not even leakage access).

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 6.8 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 6.3.4 and Lemma 6.19.

Definition 6.8 (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 6.7.

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 6.7), 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.

6.3.1 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 6.9, is a warm-up for the results in later sections.

Lemma 6.9 (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:*

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

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

Remark 6.10. *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 6.9 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 6.9. 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 4.7), 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*.

Now consider the distribution of $\vec{v}$ given $(M_{-(i+1)} \times Lin)$. $\vec{v}$ is a uniformly random vector in a subspace of rank $\kappa - 1$ that includes the ℓ columns of $(M_{-(i+1)} \times Lin)$. Thus, with probability $2^{\ell}/2^{\kappa-1}$, $\vec{v}$ is spanned by the columns of $(M_{-(i+1)} \times Lin)$, and otherwise it is uniformly random outside of the span of the columns of $(M_{-(i+1)} \times Lin)$. We conclude that the distribution of $\vec{v}$ given $(M_{-(i+1)} \times Lin)$ is $O(2^{\ell-\kappa})$-close to uniformly random.

The reduction uses $\lambda = 0.05\kappa$ bits of multi-source leakage, and so by lemma 4.8 with all but $2^{-0.2\kappa}$ probability, even given $(M_{-(i+1)} \times Lin)$ and the leakage, *key* and $\vec{v}$ are still independent random sources, and have min entropy at least 0.7κ (or rather, $\vec{v}$ is statistically close to uniformly random). When this is the case, by lemma 4.7 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 $\delta \leq 2 \cdot 2^{-0.2\kappa}$.

One remaining subtlety, is that the lemma doesn't consider giving $(M_{-(i+1)} \times Lin)$ in its entirety, but rather only the leakage (a function of $(M_{-(i+1)} \times Lin)$ and of *key* and $\vec{v}$ separately). The joint distribution of *key* and $\vec{v}$ given the leakage, however, is a convex combination of independent high-entropy sub-distributions (one for each possible value of $(M_{-(i+1)} \times Lin)$). By the above, the leakage on each pair of sub-distributions is close, and the lemma follows. ■

6.3.2 Independence up to Orthogonality

To prove leakage resilience to a piecemeal leakage attack that targets many pieces, we introduce and use the notion of "independence up to orthogonality"

Definition 6.11 (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 6.12 (Independent up to Orthogonality (IuO) Distribution on Matrices)**.** Generalizing Definition 6.12, 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 6.12, 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). This Lemma follows similarly to the proof of Lemmas 5.3 and 5.4 (security of the LROTP cryptosystem).

Lemma 6.13. *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:*

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

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

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 proceeding, 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 6.13, 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).

6.3.3 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 6.14 (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:*

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

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

For each i, we define $\mathcal{H}_i$ by drawing $(\mathit{key}, M) \sim \mathcal{V}_0$, and simulating the piecemeal leakage attack $\mathcal{A}(\mathit{key}, M)$. 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 $j \in \{1, \dots, i\}$, we define $P_j = (\mathbf{M} \times \mathit{Lin}_j)$.
- For $j \in \{i+1, \dots a\}$, 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 6.15. 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 6.15 below shows that, because the leakage operates separately on *key* and on M, these two distributions are identical.

Claim 6.15. $\mathcal{H}_0 = \mathit{Simulated}$

Proof of Claim 6.15. 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 6.14, 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 $((i+2),\dots,a)$, 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 $(key, M) \sim \mathcal{V}_i$. In $\mathcal{H}_i$ we use independent $key \sim \mathcal{K}_i, M \sim \mathcal{M}_i$. 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 $(key, M) \sim \mathcal{V}_i$ could looks very different from leakage on $(key \sim \mathcal{K}_i, M \sim \mathcal{M}_i)$. 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 6.12). We will also show it holds when $\mathcal{V}_i$ is statistically close to IuO, as defined below.

Definition 6.16 (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 6.17 below. We also show that $\mathcal{V}_i$ is (w.h.p over w_i) an IoU distribution, see Claim 6.18. Statistical closeness of the hybrids $\mathcal{H}_i$ and $\mathcal{H}_{i+1}$ follows.

Claim 6.17. *Take $a, \kappa, m, \ell, \lambda$ as in Lemma 6.14. 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 6.17 is below.

Claim 6.18. *Take $a, \kappa, \ell, \lambda, \mathcal{V}, L, \mathcal{A}$ as in Claim 6.17. Suppose here that $\mathcal{V}$: (i) has min-entropy at least $(\kappa + (\kappa-1)\cdot 2\kappa - 0.15\kappa)$ (as in Claim 6.17), and (ii) is α-close to independence up to orthogonality (see Definition 6.16). 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 6.18 is below. We now complete the proof of Lemma 6.14:

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 4.8, 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 6.18 (and the min-entropy bound in Item 1).

Finally, we use Claim 6.17 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 6.17. 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:

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

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 $\mathcal{K}_{i+1} = (\mathcal{K}|w, P[1], \ldots, P[i], \mathbf{P}[\mathbf{i}+\mathbf{1}])$. Since $\mathcal{V}$ is independent up to orthogonality, we have:

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

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 $\Delta(\mathcal{H}_i, \mathcal{H}_{i+1}) \le 2\Delta(\mathcal{H}_i^+, \mathcal{H}_i^R)$.

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 $(1.85\kappa - i) \ge 1.75\kappa$. This is simply by the min-entropy of $\mathcal{V}$. By Lemma 4.8, 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 4.7, 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 6.18. $\mathcal{V}$ is α-close to IuO. Let $\mathcal{V}'$ be an IuO distribution s.t. $\Delta(\mathcal{V}, \mathcal{V}') \le \alpha$. 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 6.17, we know that with all but δ probability over $w \sim \mathcal{Z}'$ we have that $\Delta(\mathcal{Z}'(w), \mathcal{Z}''(w)) \le \delta$. Claim 6.17 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 $\Delta(\mathcal{Z}'(w), \mathcal{Z}''(w)) \le \delta$ 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 $\Delta(\mathcal{V}, \mathcal{V}') \le \alpha$, we know that:

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

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. ∎

6.3.4 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 6.14 (piecemeal leakage resilience of random matrices) and on a "pairwise independence" property under piecemeal leakage, stated separately in Claim 6.21 below.

Lemma 6.19 (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:*

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

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

Before proving Lemma 6.19, we state a useful corollary. In a nutshell, Corollary 6.20 states that a piecemeal leakage attack on a matrix M cannot distinguish between a uniformly random matrix, and one whose columns are orthogonal to a vector $\vec{x}$. This is true even when the leakage is combined with the vector $\vec{x}$.

Corollary 6.20. *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:*

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

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

Proof. The proof follows from Lemma 6.19. Taking $key, M, \vec{v}$ as in that lemma, we reduce to the security game of corollary 6.20 by using the same *key*, choosing a uniformly random vector $\vec{x}$ and taking $B \leftarrow M + (\vec{x} \times \vec{v})$. When $\vec{v}$ is in the kernel of *key*, we get that B's columns are uniformly distributed in the kernel, but when $\vec{v}$ is not in the kernel, B's columns are uniformly distributed. Now, by Lemma 6.19, we get that piecemeal leakage from B is indistinguishable in these two cases, even in conjunction with *key* (but note that here we cannot release B, as its distribution depends on $\vec{v}$ and differs in the two cases!). ■

Proof of Lemma 6.19. 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):

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

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

Claim 6.21 (Pairwise Independence under Piecemeal Leakage)**.** *Take $a, \kappa, m, \ell, \lambda, \vec{Lin}, \mathcal{A}$ as in Lemma 6.21. 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. $\mathcal{A}(key, (M, \vec{v}_1) = \mathcal{A}(key, (M, \vec{v}_2))$.*

$$\begin{aligned}\mathcal{F} &= (\vec{v}_1, \vec{v}_2, b_1, b_2, \mathcal{A}(key, (M, \vec{v}_1)))_{key,M,\vec{v}_1,\vec{v}_2,\mathbf{b_1}=\langle key,\tilde{\mathbf{v}}_\mathbf{1}\rangle,\mathbf{b_2}=\langle key,\tilde{\mathbf{v}}_\mathbf{2}\rangle} \\ \mathcal{F}' &= (\vec{v}_1, \vec{v}_2, b_1, b_2, \mathcal{A}(key, (M, \vec{v}_1)))_{key,M,\vec{v}_1,\vec{v}_2,\mathbf{b_1},\mathbf{b_2}\in_R\{0,1\}}\end{aligned}$$

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

The proof of Claim 6.21 is below.

We will show that if $\mathbb{E}_{(M,w)\sim H}[|\mathit{bias}(key, M, w)|]$ 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 6.21). 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 6.21, 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 $\mathbb{E}_{(M,w)\sim\mathcal{D}}[|\mathit{bias}(key, M, w)|]$ 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 $1/2 + 2|\mathit{bias}(key, M, w)|^2$ (notice that the advantage over 1/2 is always "in the same direction"). Since (by Claim 6.21) the advantage of $\mathcal{DIS}$ is at most δ, we get that:

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

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

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

We also have that:

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

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

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

which completes the proof. ■

Proof of Claim 6.21. 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. $\mathcal{A}(\mathit{key},(M,\vec{v}_1)) = \mathcal{A}(\mathit{key},(M,\vec{v}_2))$:

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

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

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

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 6.14).
2. In $\mathcal{H}$, the advantage in distinguishing $(\langle \mathit{key}, \vec{v}_1\rangle, \langle \mathit{key}, \vec{v}_2\rangle)$ 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 6.14. 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 6.14, 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 $\mathcal{A}(\mathit{key},(M,\vec{v}_1))$, and also computes whether $\mathcal{A}(\mathit{key},(M,\vec{v}_1)) = \mathcal{A}(\mathit{key},(M,\vec{v}_2))$ (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

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

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 6.14 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 4.8 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 4.8 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 4.7, 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). ■

6.4 Piecemeal Matrix Multiplication: Security

In this section we use security of random matrices under piecemeal leakage to prove security properties for piecemeal matrix multiplication. These are the claims used above to prove the security of the ciphertext bank as a whole (specifically, Claims 6.2 and 6.3).

Lemma 6.22. *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 $\Delta(\mathcal{D}, \mathcal{F}) = \exp(-\Omega(\kappa))$.

Proof. The Claim follows from Lemma 6.19 (strong resilience to matrix-vector piecemeal leakage). We use the random variables $key, M, \vec{v}$ from Lemma 6.19 to generate the views $\mathcal{D}$ or $\mathcal{F}$, depending on whether the inner product of key and $\vec{v}$ is 0 or 1. We assume w.l.o.g that the parity of $\vec{v}$ is 0, and that the columns of M all have parity 1 (we can always extend a given $\vec{v}$ and M by a single coordinate to guarantee that this is the case).

To reduce to the game of Lemma 6.22, we use the same key, and we pick a uniformly random "public" $\vec{x} \in_R \{0,1\}^m$. Now take:

$$A \leftarrow M + (\vec{x} \times \vec{v}^T)$$

so that A is a function of M and $\vec{v}$ only (together with the public $\vec{x}$). Observe that if $\vec{v}$ is in the kernel of key, then A is a uniformly random matrix whose columns are in the kernel of key (as in $\mathcal{D}$). If $\vec{v}$ is *not* in the kernel of key, then A is a uniformly random matrix whose columns have inner products $\vec{x}$ with key (as in $\mathcal{F}$). Finally, the reduction chooses a uniformly random B s.t. its columns are in the kernel of $\vec{x}$. Now piecemeal leakage as in Lemma 6.22 can be used to compute the leakage:

$$w \leftarrow \left(\mathcal{A}^\lambda_{\kappa,\ell,m,Lin}(key, A), \mathcal{A}'^\lambda(w, \vec{x}, B)[key, C \leftarrow PiecemealMM(A, B)]\right)$$

Given the above reduction, by Lemma 6.19 we conclude that the joint distributions of $(\vec{x}, B, key, M, w)$ are statistically when they are drawn by $\mathcal{D}$ and by $\mathcal{F}$. Finally, observe that:

$$C = A \times B = (M + (\vec{x} \times \vec{v}^T)) \times B = M \times B$$

(we use here the fact that the columns of B are orthogonal to $\vec{x}$). We conclude that the joint distributions of key, C, w are close when they are drawn by $\mathcal{D}$ and by $\mathcal{F}$. ■

Lemma 6.23. *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 leakage adversary, and $\mathcal{A}'$ any piecemeal adversary. Let $\mathcal{D}$ and $\mathcal{F}$ be the following* *two distributions, where in both distributions key* $\in_r \{0,1\}^\kappa$ *and* $A \in_R \{0,1\}^{\kappa \times m}$ *s.t. the columns of* A *are orthogonal to key:*

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

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

Proof. The proof follows from Corollary 6.20. ■

7 Safe Computations

In this section we present the *SafeDup* and *SafeNAND* procedures, see Section 2.2 for an overview. We begin in Section 7.1 with the *SafeDup* procedure and its security, a warup for the considerably more complex *SafeNAND* procedure. *SafeNAND* and its security are in Section 7.2. *SafeNAND* uses a leakage-resilient permutation procedure, *Permute*, which is presented and proved secure in Section 7.3.

7.1 *SafeDup*: Interface and Security

In this section we present the procedure for safely computing Duplication gates. The input is two key-ciphertext pairs, and the output is the XOR of their underlying ciphertexts. For security, we show that a view any leakage attack on a *SafeDup* computation (with freshly drawn LROTP keys and ciphertexts as its input) can be simulated, given only the output bit of *SafeDup*. This is formalized in Lemma 7.1. The full procedure is in Figure 8. Correctness follows from the description.

Security of *SafeDup*. We provide a simulator for simulating leakage observed in an OC leakage attack on the *SafeDup* procedure. The attack considers on two freshly drawn key-ciphertext pairs, where the underlying plaintexts bits are a (v_i, v_j). The attack proceeds in two phases: first, an adversary $\mathcal{A}_1$ mounts a leakage attack operating separately on the input keys and on the input ciphertexts (with bounded length leakage). $\mathcal{A}_1$ generates an output view V as a function of this leakage (the leakage is of bounded length, but V might be long). Then, a second adversary $\mathcal{A}_2$ mounts an OC leakage attack on the execution of *SafeDup* with those same inputs. $\mathcal{A}_2$'s attack can be adaptive, and depends on the output V generated by $\mathcal{A}_1$. The Simulator *SimDup* is given only the output bit $a = v_i \oplus v_j$ (but not any of the plaintext bits underlying the input), and simulates the leakage generated by $\mathcal{A}_1$ and $\mathcal{A}_2$ in their two-step attack. Note that the leakage attack includes the leakage from the *Decrypt* operation (which loads keys and ciphertexts into memory simultaneously). The security claim is in Lemma 7.1.

Lemma 7.1. *There exist: a simulator SimDup, a leakage bound* $\lambda(\kappa) = \Theta(\kappa)$, *and a distance bound* $\delta(\kappa) = \exp(-\Omega(\kappa))$ *s.t. for every* $\kappa \in \mathbb{N}$ *and leakage adversaries* $\mathcal{A}_1, \mathcal{A}_2$, *and for any bit*

```
SafeDup(key_i, c_i, key_j, c_j): Safe NAND computation

  1. Correlate the ciphertexts to a new key. Pick a new key $key^* \leftarrow KeyGen(1^\kappa)$:
     $\sigma_i \leftarrow key_i \oplus key^*, \vec{c}_i^* \leftarrow CipherCorrelate(\vec{c}_i, \sigma_i)$
     $\sigma_j \leftarrow key_j \oplus key^*, \vec{c}_j^* \leftarrow CipherCorrelate(\vec{c}_j, \sigma_j)$
     leakage on $[(key_i, key_j, \sigma_i, \sigma_j), (\vec{c}_i, \vec{c}_j, \sigma_i, \sigma_j)]$
  2. $\vec{c}^* \leftarrow \vec{c}_i^* \oplus \vec{c}_j^*$
     leakage on $(\vec{c}_i^*, \vec{c}_j^*)$
  3. Pick a new key $key \leftarrow KeyGen(1^\kappa)$:
     $\sigma \leftarrow key^* \oplus key, \vec{c} \leftarrow CipherCorrelate(\vec{c}^*, \sigma)$
     leakage on $[(key^*, \sigma), (\vec{c}^*, \sigma)]$
  4. Output $a \leftarrow Decrypt(key, \vec{c})$.
     leakage on $(key, \vec{c})$ (jointly)
```

Figure 8: *SafeDup* procedure.

values $v_i, v_j \in \{0,1\}$, taking:

$$\begin{aligned} Real &= (V \leftarrow \mathcal{A}_1^{\lambda(\kappa)}[(key_i, key_j), (\vec{c}_i, \vec{c}_j)], \\ &\qquad \mathcal{A}_2^{\lambda(\kappa)}(V)[a \leftarrow SafeNAND(key_i, \vec{c}_i, key_j, \vec{c}_j)]) \\ &\quad : ((key_i, key_j), (\vec{c}_i, \vec{c}_j)) \sim LROTP^{\kappa}_{(v_i, v_j)} \\ Simulated &= SimDup(a)_{a \leftarrow (v_i \oplus v_j)} \end{aligned}$$

it is the case that $\Delta(Real, Simulated) \leq \delta(\kappa)$.

Proof. The simulator *SimDup* chooses $v_i' \in_R \{0,1\}$. It generates $(key^*, \vec{c}_i^*) \sim \text{LROTP}^\kappa(v_i')$, and simulates the view of $\mathcal{A}_1, \mathcal{A}_2$ using $O(\lambda(\kappa))$ bits of multi-source leakage from key^* and from $\vec{c}_i^*$.

SimDup chooses auxiliary random variables: some of these will be "public", meaning they are completely independent of $(key^*, \vec{c}_i^*)$. Other random variables are either computed using $O(\lambda(\kappa))$ bits of leakage from key^* (in fact, 2 bits of leakage are sufficient), in which case we think of them as "public" too, or are each a function *either* of key^* *or* of $\vec{c}_i^*$ (but never of both). *SimDup* computes the leakage from $(\mathcal{A}_1, \mathcal{A}_2)$'s attack using bounded multi-source leakage from key^* and from $\vec{c}_i^*$. Finally, when $v_i' = v_i$, the view computed is *identical* to the *Real* view produced in $\mathcal{A}_1$'s and $\mathcal{A}_2$'s attack on *SafeDup*. By leakage-resilient security of LROTP, the view is statistically close to *Real* even when v_i' is a uniformly random bit.

We now specify the random variables used by *SimDup* as a function of key^* and of $\vec{c}_i^*$:

- Choose a public uniformly random $\vec{c}^*$, s.t. the underlying plaintext of $(key^*, \vec{c}^*)$ is a. This requires a single bit of leakage from key^* (to guarantee the underlying plaintext value).

- Take $\vec{c}_j^* = \vec{c}^* \oplus \vec{c}_i^*$ (a function of the public $\vec{c}^*$ and of $\vec{c}_i$).

- Choose public uniformly random correlation values $\sigma_i \leftarrow KeyEntGen(1^\kappa), \sigma_j \leftarrow KeyEntGen(1^\kappa)$. Take $key_i = key^* \oplus \sigma_i, key_j = key^* \oplus \sigma_j$, so that key_i, key_j are functions of key^* and of the public σ_i, σ_j (respectively). Similarly, $\vec{c}_i, \vec{c}_j$ are a function of $\vec{c}_i^*$ and of the public σ_i, σ_j (respectively) (recall that $\vec{c}_j^*$ is itself a function of $\vec{c}_i^*$ and of the public $\vec{c}^*$).

- Choose a public uniformly random $key \leftarrow KeyGen(1^\kappa)$. Take $\sigma = key \oplus key^*$, a function of the public key^* and of key. Note that $\vec{c}$ is a function of the public $\vec{c}^*$, and of $\langle \vec{c}^*, \sigma \rangle$, where this inner product can be computed using a single bit of leakage from key^*. Thus, $\vec{c}$ is also public.

Once these random variables are chosen as above, the leakage from an execution of *SafeDup* can be computed using multi-source leakage from key^* and from $\vec{c}_i^*$:

- In the real execution, the leakage from Step 1 is a function of $[(key_i, key_j, \sigma_i, \sigma_j), (\vec{c}_i, \vec{c}_j, \sigma_i, \sigma_j)]$. In the simulation, σ_i, σ_j are public, and this leakage can be simulated using multi-source leakage from $[key^*, \vec{c}_i^*]$.
- In the real execution, the leakage from Step 2 is a function of $[(\vec{c}_i^*, \vec{c}_j^*)]$.

  In the simulation, it can be simulated using access to $\vec{c}_i^*$.
- In the real execution, the leakage from Step 3 is a function of $[(key, \sigma), (\vec{c}^*, \sigma)]$.

  In the simulation, key and $\vec{c}^*$ are public, and this leakage can be simulated using leakage from key^*.
- In the real execution, the leakage from Step 4 is *a joint function* of $(key, \vec{c})$.

  In the simulation, both key and $\vec{c}$ are public, and so this leakage is public too.

By construction, when $v_i' = v_i$ this simulation gives the view *Real*, and the proof follows from leakage-resilience of the LROTP cryptosystem (Lemma 5.4). ■

7.2 *SafeNAND*: Interface and Security

In this section we present the procedure for safely computing NAND gates. The full procedure is in Figure 9. Correctness follows from the description (see Section 2.2). For security, we show that a view any leakage attack on a *SafeNAND* computation (with freshly drawn LROTP keys and ciphertexts as its input) can be simulated, given only the output bit of *SafeNAND*. This is formalized in Lemma 7.2.

Security of *SafeNAND*. We provide a simulator for simulating leakage observed in an OC leakage attack on the *SafeNAND* procedure. We consider attacks on two freshly drawn 4-tuples of keys and ciphertexts, where the underlying plaintexts bits are a 4-tuple $(v_i, v_j, r_k, 1)$ (note that the last underlying plaintext bit is always fixed to 1). An attack proceeds in two phases: first, an adversary $\mathcal{A}_1$ mounts a leakage attack operating separately on the input keys and on the input ciphertexts (with bounded length leakage). $\mathcal{A}_1$ generates an output view V as a function of this leakage (the leakage is of bounded length, but V might be long). Then, a second adversary $\mathcal{A}_2$ mounts an OC leakage attack on the execution of *SafeNAND* with those inputs. $\mathcal{A}_2$'s attack can be adaptive, and depends on the output V generated by $\mathcal{A}_1$. The Simulator *SimNAND* is given only the output bit $a_k = (v_i \text{ NAND } v_j) \oplus r_k$ (but not any of the plaintext bits underlying the input), and simulates the

```
$\underline{SafeNAND(key_i, \vec{c}_i, key_j, \vec{c}_j, \vec{\ell}_k, \vec{d}_k, \vec{o}_k, \vec{e}_k)}$: Safe NAND computation

  1. Correlate the ciphertexts to a new key. Pick a new key $key \leftarrow KeyGen(1^\kappa)$:
     $\sigma_i \leftarrow key_i \oplus key, \vec{c}_i^* \leftarrow CipherCorrelate(\vec{c}_i, \sigma_i)$
     $\sigma_j \leftarrow key_j \oplus key, \vec{c}_j^* \leftarrow CipherCorrelate(\vec{c}_j, \sigma_j)$
     $\sigma_k \leftarrow \vec{\ell}_k \oplus key, \vec{d}_k^* \leftarrow CipherCorrelate(\vec{d}_k, \sigma_k)$
     $\sigma'_k \leftarrow \vec{o}_k \oplus key, \vec{e}_k^* \leftarrow CipherCorrelate(\vec{e}_k, \sigma'_k)$
     $leakage\ on\ [(key_i, key_j, \vec{\ell}_k, \vec{o}_k, \sigma_i, \sigma_j, \sigma_k, \sigma'_k), (\vec{c}_i, \vec{c}_j, \vec{d}_k, \vec{e}_k, \sigma_i, \sigma_j, \sigma_k, \sigma'_k)]$
  2. $C \leftarrow (\vec{d}_k^*, \vec{c}_i^* \oplus \vec{d}_k^*, \vec{c}_j^* \oplus \vec{d}_k^*, \vec{c}_i^* \oplus \vec{c}_j^* \oplus \vec{d}_k^* \oplus \vec{e}_k^*)$
     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 $(K', C')$ (jointly)
```

Figure 9: *SafeNAND* procedure. The *Permute* procedure is in Figure 10.

leakage generated by $\mathcal{A}_1$ and $\mathcal{A}_2$ in their two-step attack. Note that the leakage attack includes the leakage from the *Decrypt* operation (which loads keys and ciphertexts into memory simultaneously). The security claim is below in Lemma 7.2.

Lemma 7.2. *There exist: a 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 adversaries $\mathcal{A}_1, \mathcal{A}_2$, and for any bit values $v_i, v_j, r_k \in \{0,1\}$, taking:*

$$\begin{aligned}
Real &= (V \leftarrow \mathcal{A}_1^{\lambda(\kappa)}[(key_i, key_j, \vec{\ell}_k, \vec{o}_k), (\vec{c}_i, \vec{c}_j, \vec{d}_k, \vec{e}_k)], \\
&\qquad \mathcal{A}_2^{\lambda(\kappa)}(V)[a_k \leftarrow SafeNAND(key_i, \vec{c}_i, key_j, \vec{c}_j, \vec{\ell}_k, \vec{d}_k, \vec{o}_k, \vec{e}_k)]) \\
&\quad : ((key_i, key_j, \vec{\ell}_k, \vec{o}_k), (\vec{c}_i, \vec{c}_j, \vec{d}_k, \vec{e}_k)) \sim LROTP^\kappa_{(v_i, v_j, r_k, 1)} \\
Simulated &= SimNAND(a_k)_{a_k \leftarrow ((v_i \ NAND \ v_j) \oplus r_k)}
\end{aligned}$$

it is the case that $\Delta(Real, Simulated) \leq \delta(\kappa)$.

Proof. The proof of security for *SafeNAND* will follow directly from the security of *Permute*, which is stated in Lemma 7.3 of Section 7.3. We begin by describing the *SimNAND* simulator, and then proceed with a proof of statistical closeness of *Real* and *Simulated*.

SimNAND **Simulator.** The simulator gets as input a bit $a_k \in \{0,1\}$. It chooses (arbitrarily) bit values $(v'_i, v'_j, r'_k) \in \{0,1\}^3$ s.t. $a_k = ((v'_i \text{ NAND } v'_j) \oplus r'_k)$ and runs the leakage attack on freshly generated keys and ciphertexts encrypting these bit values (note that the simulator does not know the "real" (v_i, v_j, r_k), and we expect that $(v'_i, v'_j, r'_k) \neq (v_i, v_j, r_k)$). The simulator's output is the leakage in the attack:

$$\begin{aligned} \mathit{Simulated} &= (V \leftarrow \mathcal{A}_1^{\lambda(\kappa)}[(\mathit{key}_i, \mathit{key}_j, \vec{\ell}_k, \vec{o}_k), (\vec{c}_i, \vec{c}_j, \vec{d}_k, \vec{e}_k)], \\ & \quad \mathcal{A}_2^{\lambda(\kappa)}(V)[a_k \leftarrow \mathit{SafeNAND}(\mathit{key}_i, \vec{c}_i, \mathit{key}_j, \vec{c}_j, \vec{\ell}_k, \vec{d}_k, \vec{o}_k, \vec{e}_k)])) \\ & \quad : ((\mathit{key}_i, \mathit{key}_j, \vec{\ell}_k, \vec{o}_k), (\vec{c}_i, \vec{c}_j, \vec{d}_k, \vec{e}_k)) \sim \mathrm{LROTP}^{\kappa}_{(v'_i, v'_j, r'_k, 1)} \end{aligned}$$

Statistical closeness of *Real* and *Simulated*. We reduce the leakage attacks of $\mathcal{A}_1$ and $\mathcal{A}_2$ on *SafeNAND* in *Real* and in *Simulated*, to attacks on *Permute*. The two cases (*Real* and *Simulated*) reduce to attacks on *Permute* that differ only in the plaintext bits underlying *Permute*'s input ciphertexts, and the numbers of 0-plaintexts and 1-plaintexts are identical in the two cases. By the security of *Permute*, we conclude that the views generated in the leakage attack on *Permute* launched in the *Real* and *Simulated* cases are statistically close, and so the views *Real* and *Simulated* must also be statistically close. Note that we will assume from here on that the leakage w from *SafeNAND* includes *Permute*'s output in its entirety. This is a strengthening of the leakage adversaries (they get more leakage "for free"), and so it strengthens our security claim for *SafeNAND*.

The security game for *Permute* is set up as follows: for the case of the *Real* attack, we use the bit-vector $\vec{b} = (r_k, v_i \oplus r_k, v_j \oplus r_k, v_i \oplus v_j \oplus r_k \oplus 1)$. For the case of the *Simulated* attack, we use the bit-vector $\vec{b} = (r'_k, v'_i \oplus r'_k, v'_j \oplus r'_k, v'_i \oplus v'_j \oplus r'_k \oplus 1)$. Note that for both (v_i, v_j, r_k) used in *Real*, and (v'_i, v'_j, r'_k) used in *Simulated*, the number of 0's and 1's in $\vec{b}$ is identical (three 0's if $a_k = 1$, and one 0 if $a_k = 0$). Given these bit values, we generate an input $(\mathit{key}, C) \sim \mathrm{LROTP}^{\kappa}_{\vec{b}}$ for the *Permute* security game of Lemma 7.3.

We now construct from the *SafeNAND* adversaries $\mathcal{A}_1$ and $\mathcal{A}_2$, two new adversaries $\mathcal{A}'_1$ and $\mathcal{A}'_2$ for *Permute* as follows. First, we choose correlation values $(\sigma_i, \sigma_j, \sigma_k, \sigma'_k)$. The adversaries $\mathcal{A}'_1$ and $\mathcal{A}'_2$ will attack *Permute* by simulating an attack of $\mathcal{A}_1$ and $\mathcal{A}_2$ on *SafeNAND*. The inputs $(\mathit{key}_i, \vec{c}_i, \mathit{key}_j, \vec{c}_j, \vec{\ell}_k, \vec{d}_k, \vec{o}_k, \vec{e}_k)$ to *SafeNAND*, as well as the internal variables, are set up in the natural way. E.g key_i and $\vec{c}_i$ are set as:

$$\begin{aligned} &\mathit{key}_i \leftarrow \mathit{key} \oplus \sigma_i \\ &\vec{c}^{\,*}_i \leftarrow C[1] \oplus C[0] \\ &\vec{c}_i \leftarrow \mathit{CipherCorrelate}(\vec{c}^{\,*}_i, \sigma_i) \end{aligned}$$

In this way, the joint distributions of input keys and ciphertexts, correlation values, internal variables, and the resulting (key, C) are identical in this simulation and in an execution of an attack on *SafeNAND* on inputs with underlying plaintexts $(v_i, v_j, r_k, 1)$ or $(v'_i, v'_j, r'_k, 1)$ (in the *Real* and *Simulated* cases respectively).

Now $\mathcal{A}'_1$ runs $\mathcal{A}_1$ and $\mathcal{A}_2$ on Steps 1 and 2 of *SafeNAND*, to generate leakage w_1. Observe that (since $\mathcal{A}'_1$ "knows" the correlation values), the leakage computed by $\mathcal{A}_1$ from the inputs to *SafeNAND* and the leakage computed by $\mathcal{A}_2$ in Steps 1 and 2can be computed as a multi-source functions of length $O(\lambda(\kappa))$, operating separately on *key* and on C.

Next, $\mathcal{A}'_2$ runs $\mathcal{A}_2$ on Step 3 of *SafeNAND* to compute leakage w_2 from *Permute*'s operation. This uses multi-source leakage of $\tilde{O}(\kappa)$ bits from *key* and from C.

Finally, we add to the leakage values computed by $\mathcal{A}_1$ and $\mathcal{A}_2$ the keys and ciphertexts (K', C') as generated by *Permute* in Step 3 of *SafeNAND*.

By the security of *Permute*, we conclude that (w_1, w_2, K', C') comprising both the leakage computed by $\mathcal{A}'_1$ and $\mathcal{A}'_2$ and the output of *Permute*, are statistically close in the *Real* and *Simulated* cases. This is because the number of 0 plaintexts and 1 plaintexts in the the ciphertexts C given as input to *Permute* in these two views are identical, and so both views will be close to the once generated by *Permute*'s simulator *SimPermute* (which is only given a uniformly random permutation of these plaintext bits). ■

7.3 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 10.

Correctness is immediate.

Security is formalized by the existence of a simulator that generates a complete view of the leakage observed in an OC leakage attack on *Permute and the output keys and ciphertexts.* We consider attacks on a freshly drawn *key* and four LROTP ciphertexts C, where the underlying plaintexts are some four-tuple of bits $\vec{b}$. An attack proceeds in two phases (similarly to an attack on *SafeNAND*): first, an adversary $\mathcal{A}_1$ mounts a (bounded length) leakage attack operating separately on *key* and on C. Then, a second adversary $\mathcal{A}_2$ mounts an OC leakage attack on the execution of *Permute* with those inputs. Finally, we also include *Permute*'s outputs (K', C') in the attacker's view. The simulator only gets a random permutation of the plaintexts underlying the input (key, C), and simulates the leakage generated by $\mathcal{A}_1$ and $\mathcal{A}_2$ in their attack *together with the output* (K', C'). The security claim is below in Lemma 7.3.

We note that the Simulator we provide here is not efficient, and may run in exponential time. This is not a problem because the *SimPermute* simulator is only be used in the *security proof* of our main construction, and never in the main construction's simulator (the main construction's simulator is efficient). We will use *SimPermute* when generating hybrid distributions, and since the hybrids will all be *statistically* close to each other, we do not mind that their generation requires exponential time.

Lemma 7.3. *There exist: an (exponential time) simulator SimPermute, 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 adversaries $\mathcal{A}_1, \mathcal{A}_2$, and for any vector of bit values $\vec{b} = (b_1, b_2, b_3, b_1 + b_2 + b_3 + 1)$, taking:*

$$\begin{aligned} Real &= (\mathcal{A}_1^{\lambda(\kappa)}[key, C], \\ &\quad \mathcal{A}_2^{\lambda(\kappa)}[(K', C') \leftarrow Permute(key, C)], \\ &\quad K', C')_{(key,(\vec{c}_1,\vec{c}_2,\vec{c}_3)) \sim LROTP^{\kappa}_{(b_1,b_2,b_3)}, C=(\vec{c}_1,\vec{c}_2,\vec{c}_3,(\vec{c}_1 \oplus \vec{c}_2 \oplus \vec{c}_3 \oplus (1,0,\ldots,0)))} \\ Simulated &= SimPermute(\vec{b}')_{\mu \in_R S_4, \vec{b}' \leftarrow \mu(\vec{b})} \end{aligned}$$

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

Proof. The *SimPermute* simulator takes as input $\vec{b}' = \mu(\vec{b})$. It outputs leakage w and an output (K', C') of *Permute* as follows:

```
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 $j \in [\kappa], k \in [4]$: $\sigma_i[j][k] \leftarrow KeyEntGen(1^\kappa)$, $L_i[j][k] \leftarrow KeyRefresh(K_i[k], \sigma_i[j][k])$ leakage on $(K_i, \sigma_i)$
    2. for $j \in [\kappa], k \in [4]$: $D_i[j][k] \leftarrow CipherCorrelate(C_i[k], \sigma_i[j][k])$
       leakage on $(C_i, \sigma_i)$
    3. for $j \in [\kappa], k \in [4]$: $\tau_i[j][k] \leftarrow CipherEntGen(1^\kappa)$, $D'_i[j][k] \leftarrow CipherRefresh(D_i[j][k], \tau_i[j][k])$
       leakage on $(D_i, \tau_i)$
    4. for $j \in [\kappa], k \in [4]$: $L'_i[j][k] \leftarrow KeyCorrelate(L_i[j][k], \tau_i[j][k])$
       leakage on $(L_i, \tau_i)$
    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 $(K_\ell, C_\ell)$
```

Figure 10: Leakage-Resilient Ciphertext Permutation for $\kappa \in \mathbb{N}$

1. Sample $(key, C) \sim \text{LROTP}_0^\kappa$ (i.e. LROTP encryptions of 0, rather than of $\vec{b}$ as in *Real*).

   Fix randomness for the adversaries $\mathcal{A}_1, \mathcal{A}_2$ and compute:

$$w = (\mathcal{A}_1^{\lambda(\kappa)}[key, C], \mathcal{A}_2^{\lambda(\kappa)}[(K', C') \leftarrow Permute(key, C)])$$

   Let π be the composed distribution used by *Permute*.

2. Compute the conditional distribution $\mathcal{K}'$ of K' given: (*i*) the fixed randomness used by the adversaries, (*ii*) the leakage w, (*iii*) the permutation π, and (*iv*) that $(key, C) \sim \text{LROTP}_0^\kappa$

   Sample $K' \sim \mathcal{K}'$

3. Compute the conditional distribution $\mathcal{C}'$ of C' given: (*i*) the fixed randomness used by the adversaries, (*ii*) the leakage w, (*iii*) the permutation π, and (*iv*) that $(key, C) \sim \text{LROTP}_0^\kappa$

   Sample $C' \sim \mathcal{C}'$, under the additional condition that the inner products of C' with K' are $\vec{b}'$

4. The simulator's output is (w, K', C')

We remark again, as noted above, that the complexity of the simulator is super-polynomial in κ (this is required for sampling from the conditional distributions $\mathcal{K}'$ and $\mathcal{C}'$ above).

The *Hybrid* distribution. Observe that (key, C) chosen in *Real* and in *Simulated* consist of an LROTP key and four ciphertexts. The only difference in their distributions is that in *Real* the underlying plaintexts are $\vec{b}$, and in *Simulated* they are $\vec{0}$. The *Permute* procedure operates separately on *key* and on C (as does the leakage computed by $\mathcal{A}_1$), and the total leakage computed by $\mathcal{A}_1$ and $\mathcal{A}_2$ is bounded by $O(\ell \cdot \lambda(\kappa)) << \kappa$ bits. By Lemma 5.4 (security of the LROTP cryptosystem), the distributions of leakage w generated in *Real* and *Simulated* are statistically close (this remains true even if we include the randomness used by $\mathcal{A}_1$ and $\mathcal{A}_2$).

The more difficult part of the proof is arguing that (w.h.p. over w), even given w, the joint distributions of (K', C') in *Real* and in *Simulated* are statistically close. For this, we consider a hybrid distribution *Hybrid*. We generate *Hybrid* exactly as does *SimPermute*, except that in Step 3, we draw C' from $\mathcal{C}'$ conditioning also on the inner products of K' and C' being $\pi(\vec{b})$ (rather than $\vec{b}' = \mu(\vec{b})$ for a random μ as in *Simulated*).

We now show that *Hybrid* is statistically close to both *Real* and *Simulated*. In what follows, we always fix the randomness used by $\mathcal{A}_1$ and $\mathcal{A}_2$: statistical closeness holds for any fixed randomness used by the adversaries, and so it will also hold over the choice of randomness.

Proposition 7.4. $\Delta(\mathit{Real}, \mathit{Hybrid}) = O(\delta(\kappa))$

Proof. We re-cast *Real*, as follows. We generate *Real* using a similar procedure to the generation of *Hybrid*, except that in Step 1, we sample $(key, C) \sim \mathrm{LROTP}^{\kappa}_{\vec{b}}$ (i.e. encryptions of the real $\vec{b}$ bits). In Steps 2 and 3, we condition $\mathcal{K}'$ and $\mathcal{C}'$ (respectively) on $(key, C) \sim \mathrm{LROTP}^{\kappa}_{\vec{b}}$. Other than these changes to the bits encrypted in (key, C), we proceed as in *Hybrid*.

To see that this indeed generates the *Real* view, observe that the joint distribution of (w, π) is exactly as in *Real*. In Step 2 we are sampling K' from its true conditional distribution in *Real* (given (w, π)), and similarly C' is drawn from its true conditional distribution in *Real* (given (w, π)).Fixing all random choices of *Permute* (including the composed permutation π), the leakage w operates separately on *key* and on C. Thus, by Lemma 5.3, the conditional distribution of C' given (w, π) *and given also* K' equals its conditional distribution given (w, π) conditioned only on the inner product of C' and K' equalling $\pi(\vec{b})$.

To argue that *Real* and *Hybrid* are statistically close, we fix all the randomness used by *Permute* (including the composed distribution π), and view these two distributions as a multi-source leakage attack operating separately on *key* and on C. In *Real*, (key, C) are generated as LROTP encryptions of $\vec{b}$, and in *Hybrid* they are encryptions of $\vec{0}$. By Lemma 5.4, the joint distributions of w and the randomness used by *Permute* in *Real* and *Simulated* are statistically close. Moreover, by Lemma 5.3, w.h.p. over w the conditional distributions of *key* and of C (each separately), conditioned on w and *Permute*'s randomness, are *identical*. Once *Permute*'s randomness is fixed, K' and C' are deterministic functions of *key* and of C (respectively). We conclude that w.h.p. over w, the conditional distributions of K' and of C' (each separately) in *Real* and in *Hybrid* (conditioned on (w, π)) are also identical. We emphasize that this is true even though we are conditioning on different plaintexts encrypted in (key, C) in *Real* and in *Hybrid*.

Real and *Hybrid* both draw K' from its conditional distribution, so these draws (together with (w, π)) will be statistically close. They then draw C' from its conditional distribution, conditioning further on the same inner products $\pi(\vec{b})$ with K'. By Lemma 5.3, for fixed (w, π, K'), the draws of C' conditioned on (w, π, K') (and inner products $\pi(\vec{b})$) in *Real* and in *Hybrid* will be statistically close. ■

Proposition 7.5. $\Delta(\mathit{Simulated}, \mathit{Hybrid}) = \mathrm{negl}(\kappa)$

Proof. The main claim we will show is that, when $(\mathit{key}, C) \sim \mathrm{LROTP}^{\kappa}_{\vec{0}}$, w.h.p. over w, even given leakage w on *Permute* and given also the output (K', C'), the composed permutation π used by *Permute* is indistinguishable from uniformly random. This is stated in Claim 7.6 below.

The only difference between *Hybrid* and *Simulated*, is that in *Hybrid* we condition (K', C') on the permutation π (and on w), whereas in *Simulated*, we effectively condition (K', C') on a uniformly random composed permutation $(\mu \circ \pi)$. By Claim 7.6, with overwhelming probability over the leakage w, drawing (K', C') from these two conditional distributions yields statistically close views. ■

Claim 7.6. *Fix any randomness for the adversaries $\mathcal{A}_1, \mathcal{A}_2$ and consider:*

$$w = (\mathcal{A}_1^{\lambda(\kappa)}[\mathit{key}, C], \mathcal{A}_2^{\lambda(\kappa)}[(K', C') \leftarrow \mathit{Permute}(\mathit{key}, C)])_{(\mathit{key},C) \sim \mathit{LROTP}^{\kappa}_{\vec{0}}}$$

Let π be the composed distribution used by Permute. Consider the conditional distributions:

$$\begin{aligned} D_0(w) &= ((\pi, K', C')|w) \\ D_1(w) &= (((\mu \circ \pi), K', C')|w)_{\mu \in_R S_4} \end{aligned}$$

then with all but $\exp(-\Omega(\kappa))$ *probability over w, it is the case that* $\Delta(D_0(w), D_1(w)) = \mathrm{negl}(\kappa)$.

Proof. 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 7.7 (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.

Proposition 7.8. *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 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.

Proposition 7.9. *In Permute's execution in Claim 7.6, for any $i \in [\ell]$, and for any $(K_i, C_i, (w_0, \ldots, w_{i-1}))$, with all but* $\exp(-\Omega(\kappa))$ *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) here 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 4.8, since the leakage w_i on $(\sigma_i, \tau_i, \pi_i)$ is of length at most $O(\lambda(\kappa))$ bits, with all but $\exp(-\Omega(\kappa))$ 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 7.5, we examine the composed distribution:

$$(\pi = (\pi^*_0 \circ, \ldots \circ \pi^*_{\ell-1}) | w, K_0, C_0, \ldots, K_\ell, C_\ell)$$

Each π^*_i is drawn from P_i, and these draws are all independent of each other. By Proposition 7.9, we get that with all but $\exp(-\Omega(\ell))$ probability over the random coins, fixing the sequence $((K_0, C_0), \ldots, (K_\ell, C_\ell))$ and the leakage w, at least 1/3 of the distributions P_i are well-mixing. When this happens, by Proposition 7.8, the distribution of $(\pi | w, K_0, C_0, \ldots, K_\ell, C_\ell)$ is $\exp(-\Omega(\ell))$-close to uniform, where $\ell = \text{polylog}(\kappa)$. ■

■

8 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 4.9 in Section 4.3. *See Section 2 for an overview of the construction and its security.*

The full initialization and evaluation procedures are presented below in Figures 11 and 12. 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 6. The procedures for safely computing NAND and duplication are in Section 7. Theorem 8.1 states the security of the compiler.

Theorem 8.1. *There exist a leakage bound $\lambda(\kappa) = \tilde{\Theta}(\kappa)$ and a distance bound $\delta(\kappa) = \text{negl}(\kappa)$, s.t. for every $\kappa \in \mathbb{N}$, the (Init, Eval) compiler specified in Figures 11 and 12 is a (λ, δ)-continuous leakage secure compiler, as per Definition 4.9.*

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

```
Initialization Init(1^κ, C, y)

  1. for every y-input wire $i$, corresponding to $y[j]$:
       $Bank_i \leftarrow BankInit(1^\kappa, y[j])$
  2. for every x-input wire $i$:
       $Bank_i \leftarrow BankInit(1^\kappa, 0)$
  3. for the output wire output:
       $Bank_{output} \leftarrow BankInit(1^\kappa, 0)$
  4. for the internal wires:
       $Bank_{random} \leftarrow BankInit(1^\kappa, 0)$
       $Bank_{fixed} \leftarrow BankInit(1^\kappa, 1)$
  5. output: $state_0 \leftarrow (\{Bank_i\}_{i \text{ is an input wire}}, Bank_{output}, Bank_{random}, Bank_{fixed})$
```

Figure 11: *Init* procedure, to be run in an offline stage on circuit C and secret y.

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, with input x_t and output $C(y, x_t)$, the simulator computes the values on all internal wires for the "dummy" circuit computation $C(\vec{0}, x_t)$, let v'_i be the value on wire i in this dummy computation. We emphasize that in the simulated view, the v'_i values are always stored and manipulated in LROTP encrypted form, and are never exposed to the adversary (similarly to the v_i wire values in the real execution).

The simulator also picks "public bits" $a_i \in \{0, 1\}$ for the internal and output wires. This bit determines the (public) output of the *SafeNAND* or *SafeDup* call whose output is on wire i. For each internal wire i, the bits a_i is uniformly random. For the output wire, the simulator sets $a_{output} = C(y, x_t)$. In particular, the outputs of all *SafeNAND* and *SafeDup* calls are identically distributed in the real and simulated executions (as they should be, because these are visible to the adversary).

Once the v'_i and a_i values are picked, the simulator uses the *SimBankGen* simulation procedure to generate key-ciphertext pairs for all circuit wires, where the underlying plaintexts are consistent with the v'_i and a_i values: for the input wires, the y-input wire keys-ciphertext pairs have underlying plaintext bit 0 (the x-input wire ciphertexts are unchanged). Proceeding layer by layer, the keys and ciphertexts on the input wires of each NAND gate have the dummy v'_i wire values as their underlying plaintexts. For an internal (or output) wire i, the key-ciphertext pair $(\vec{\ell}_i, \vec{d}_i), (\vec{\ell}'_i, \vec{d}'_i)$ encrypts the bit $(v'_i \oplus a_i)$. When the simulator runs a *SafeNAND* call for the NAND gate whose output wire is i, it will get output a_i, and will "toggle" $(\vec{\ell}'_i, \vec{d}'_i)$ to get $(key_i, \vec{c}_i)$ with underlying plaintext v'_i (we note that there is no change to the underlying plaintext of $(\vec{o}_i, \vec{e}_i)$, it remains 1). Finally, after evaluating all the NAND gates in order, the output gate's *SafeNAND* evaluation yields a_{output}, the correct output value. The leakage is generated as it would be from an execution of *Eval* using the ciphertexts generated by *SimBankGen*. The *SimInit* and *SimEval* procedures are specified below in Figures 13 and 14.

```
Evaluation $Eval(state_{t-1}, x_t)$
$state_{t-1} = (\{Bank_i\}_{i \text{ is an input wire}}, Bank_{output}, Bank_{random}, Bank_{fixed})$
  1. Generate keys and ciphertexts for all circuit wires:
      (a) $y$ input wire $i$:
          $(key_i, \vec{c}_i) \leftarrow BankGen(Bank_i)$
      (b) $x$ input wire $i$, carrying bit $x_t[j]$:
          $(key_i, \vec{c}_i) \leftarrow BankGen(Bank_i)$
          $\vec{c}_i \leftarrow \vec{c}_i \oplus (x_t[j], 0, \ldots, 0)$
      (c) output wire $output$:
          $(\vec{\ell}_{output}, \vec{d}_{output}) \leftarrow BankGen(Bank_{output})$
          $(\vec{o}_{output}, \vec{e}_{output}) \leftarrow BankGen(Bank_{fixed})$
      (d) each internal wire $i$ (in sequence):
          $(\vec{\ell}_i, \vec{d}_i, \vec{\ell}'_i, \vec{d}'_i) \leftarrow BankGenRand(Bank_{random})$
          $(\vec{o}_i, \vec{e}_i) \leftarrow BankGen(Bank_{fixed})$
  2. Proceed layer by layer (from input to output):
      (a) for each NAND gate with input wires $i, j$ and output wire $k$, compute:
              $a_k \leftarrow SafeNAND(key_i, \vec{c}_i, key_j, \vec{c}_j, \vec{\ell}_k, \vec{d}_k, \vec{o}_k, \vec{e}_k)$
          for internal NAND gates, also compute:
              $key_k \leftarrow \vec{\ell}'_k, \vec{c}_k \leftarrow \vec{d}'_k \oplus (a_k, 0, \ldots, 0)$
      (b) for each duplication gate with input wire $i$ and output wires $j, k$, compute:
              $a_j \leftarrow SafeDup(key_i, \vec{c}_i, \vec{\ell}_j, \vec{d}_j), key_j \leftarrow \vec{\ell}'_j, \vec{c}_k \leftarrow \vec{d}'_j \oplus (a_j, 0, \ldots, 0)$
              $a_k \leftarrow SafeDup(key_i, \vec{c}_i, \vec{\ell}_k, \vec{d}_k), key_i \leftarrow \vec{\ell}'_k, \vec{c}_k \leftarrow \vec{d}'_k \oplus (a_k, 0, \ldots, 0)$
     After completing these evaluations, output $a_{output}$
  3. the new state is: $state_t \leftarrow (\{Bank_i\}_{i \text{ is an input wire}}, Bank_{output}, Bank_{random}, Bank_{fixed})$
```

Figure 12: *Eval* procedure performed on input x_t, under OC leakage. See Section 6.1 for ciphertext bank procedures, Section 7 for the full *SafeNAND* and *SafeDup* procedures.

```
Simulator Initialization $SimInit(1^\kappa, C)$
Proceed exactly as in $Init$, but replace each call to $BankInit$ with a call to $SimBankInit$.
```

Figure 13: Simulator Initialization *SimInit*

Statistical Security. The intuition for security is that the "public" a_i values computed by *SafeNAND* in the simulated execution are distributed exactly as they are in the real execution—

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

The *SimEval* simulator sets values as follows:
for each circuit wire i, compute its value v'_i by evaluating $C(\vec{0}, x_t)$.
for each internal circuit wire i, choose a share $a_i \in_R \{0, 1\}$
for the output wire *output*, set $a_{output} \leftarrow C(y, x_t)$
It then runs a simulated execution of *Eval* under leakage as follows:

- In Step 1:

  for each y-input wire i, replace each call to *BankGen* with a call to *SimBankGen*(0)

  for each x-input wire i, replace each call to *BankGen* with a call to *SimBankGen*(0)

  for the output wire *output*, replace the call to *BankGen* for generating $(\vec{\ell}_{output}, \vec{d}_{output})$ with a call to *SimBankGen*$(v'_{output} \oplus a_{output})$. Replace the call to *BankGen* for generating $(\vec{o}_{output}, \vec{e}_{output})$ with a call to *SimBankGen*(1).

  for each internal wire i, replace the call to *BankGenRand* for generating $(\vec{\ell}_i, \vec{d}_i, \vec{\ell}'_i, \vec{d}'_i)$ with a call to *SimBankGenRand*$(v'_i \oplus a_i, v'_i \oplus a_i)$. Replace the call to *BankGen* for generating $(\vec{o}_i, \vec{e}_i)$ with a call to *SimBankGen*(1).

- In Step 2, proceed as in *Eval* and compute, for each internal wire i the bit a_i and the key-ciphertext pair $(\vec{c}_i, key_i)$. For the output wire, compute a_{output}, the output for this execution.

- In Step 3, update the state as in *Eval*

Figure 14: *Sim* procedure performed on input x_t and circuit output $C(y, x_t)$

they are uniformly random for all internal wires, and for the output wire a_{output} equals the circuit output $C(y, x_t)$. The "hidden" underlying plaintexts for internal wires may be quite different, but the ciphertext bank security guarantees that the leakage adversary cannot distinguish the simulated generations from real ones, and the security of *SafeNAND* implies that the adversary learns no more than the output a_i of *SafeNAND* for the NAND gate with out wire i (and these values are identically distributed in the real and simulated executions). The full proof that *Real* and *Simulated* are statistically close uses several hybrids:

Real **and** *HybridReal***: replacing real generations with simulated ones.** The view *HybridReal* 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 *BankInit* call of *Init* with a call to *SimBankInit*. We then replace each *BankGen* call for an x-input wire with a call to *SimBankGen*(0), each call to *BankGen* for a y-input wire i carrying the j-th bit of y with a call to *SimBankGen*$(y[j])$, and the call to *BankGen* for the output wire with a call to *SimBankGen*(0). For each internal wire, we replace each call to *BankGen* for generating a ciphertext with underlying plaintext 1 with a call to *SimBankGen*(1), and each call to *BankGenRand* with a call to *SimBankGenRand* with a uniformly random plaintext (the same plaintext for both *SimBankGenRand*). Other than these changes to the ciphertext bank calls, we run exactly as in *Real*.

The two views *Real* and *HybridReal* differ only in that in *Real* we have calls to *BankInit*, *BankGen*, *BankGenRand*, whereas in *HybridReal* we have calls to the corresponding simulated procedures. Note that the b-values given as input to *SimBankGen* and *SimBankGenRand* in

HybridReal are distributed *identically* to the plaintexts underlying the ciphertexts generated in the corresponding calls to *BankGen* in *Real*. By Lemmas 6.1 and 6.5, the joint distributions of the leakage in all of these calls, *together with all keys and ciphertexts produced*, are negl(κ)-statistically close in *Real* and in *HybridReal*. For each execution of *Eval*, we can replace the ciphertext generations as above, and then complete the adversary's view by generating the leakage from *SafeNAND* and *SafeDup* as a function of the keys and ciphertexts produced. Thus, we conclude that *Real* and *HybridReal* are negl(κ)-statistically close.

HybridReal **to** *Simulated*: **simulated generations, different underlying plaintexts.** In both the *HybridReal* and the *Simulated* views, all ciphertexts are generated using simulated ciphertext bank calls. The same computations are performed on the key-ciphertext pairs that are generated in both views (namely the *Eval* procedure's *SafeNAND* and *SafeDup* computations). The only difference between the views is in the underlying plaintexts specified as inputs for simulated ciphertext bank generations.

The difference between *HybridReal* and *Simulated* in one execution of *Eval* on input x_t is as follows. The input x_t specifies, for each internal wire i: a value $v_i \in \{0,1\}$, the bit on wire i in the evaluation of $C(y, x_t)$ (as in *HybridReal*), and a value $v'_i \in \{0,1\}$, the bit on wire i in the evaluation of $C(\vec{0}, x_t)$ (as in *Simulated*). Similarly, v_{output}, v'_{output} are the bits on the output wire in $C(y, x_t)$ and $C(\vec{0}, x_t)$.

In both views the execution is also determined by "public bits", where for each internal wire i there is a public bit $a_i \in \{0,1\}$. These public bits are *identically distributed* in *HybridReal* and *Simulated*: for each internal wire i, the public bit a_i is uniformly random, and for the output wire *output* we have $a_{output} = C(y, x_t)$ (this is the distribution in both views).

The values v_i, v'_i for each circuit wire, and the public bits a_i for the internal and output wires determine the plaintexts underlying each simulated ciphertext generations as follows:

- For an input wire i, we use w_i to denote the underlying plaintext of the (single) key-ciphertext pair generated for that wire.

  In *HybridReal* the underlying plaintext is the corresponding bit of y or x_t, i.e. $w_i = v_i$, whereas in *Simulated* we have $w_i = v'_i$ (which is 0 for a y-input wire, or the correct corresponding bit of x_t for an x-input wire).

- For an internal wire i, we use u_i, w_i to denote the underlying plaintexts of the (two) key-ciphertext pairs $(\vec{\ell}_i, \vec{d}_i), (\vec{\ell}'_i, \vec{d}'_i)$ (respectively) generated by the *BankGenRand* call for wire i. We note that both in *HybridReal* and in *Simulated*, these two key-ciphertext pairs have the same underlying plaintexts (within each view), and $u_i = w_i$. In the security proof below, however, we will consider further hybrids where this is not the case and $u_i \neq w_i$.

  In *HybridReal* we have $u_i = w_i = (v_i \oplus a_i)$, whereas in *Simulated* we have $u_i = w_i = (v'_i \oplus a_i)$.

- For the output wire *output*, we use u_{output} to denote the underlying plaintext of the (single) key-ciphertext pair generated for that wire.

  In *HybridReal* we have $u_{output} = (v_{output} \oplus a_{output}) = (C(y, x_t) \oplus C(y, x_t)) = 0$, whereas in *Simulated* we have $u_{output} = (v'_{output} \oplus a_{output}) = (C(\vec{0}, x_t) \oplus C(y, x_t))$.

Gate-By-Gate Hybrids. To prove that *HybridReal* and *Simulated* are statistically close, we consider a sequence of hybrid views. We view C as a layered circuit of depth D, where layer 0 is the layer of input wires and layer D is the layer of the output wire (we layer the wires in the circuit, which also imposes a layering on the gates). We take S to be a bound on the number of circuit gates, and also on the number of gates in each layer (we assume a numbering on the gates within each circuit layer). We take T to be the total number of *Eval* calls. We then have a sequence of hybrids:

$$\{\mathcal{H}_{t,d,k}\}_{t \in [T+1], d \in [D], s \in [S]}$$

Each hybrid is associated with a single execution of *Eval* (within the T executions), and a single gate in a single layer of the circuit C. The hybrids differ in the plaintexts underlying the calls to the simulated ciphertext bank, i.e. in the values of u_i and w_i. We emphasize that in all hybrids *the joint distributions of the public a_i values are identical* and as in *HybridReal* and *Simulated*. Moreover, the hybrids differ only in the underlying plaintext for the key-ciphertext pairs that are generated. The subsequent computations on these key-ciphertext pairs (e.g *SafeNAND*, *SafeDup*) are performed identically in all hybrids (as in *HybridReal* and *Simulated*).

In $\mathcal{H}_{t,d,k}$, all calls up to (but not including) the calls for wires in layer d in the t-th execution of *Eval* use underlying plaintexts (u_i, w_i) as in *HybridReal*. All calls after (but not including) layer $(d+1)$ in the t-th execution use underlying plaintexts (u_i, w_i) as in *Simulated*. It remains to specify the underlying plaintexts for wires in layers d and $d+1$ of the t-th execution. Take g_s to be the s-th gate between wire layers d and $d+1$. Without loss of generality, let g_s be a *SafeNAND* gate, with input wires i and j and output wire k (*SafeDup* gates are handled similarly). The main case is when wires i, j, k are internal circuit wires, but we also specify the distributions when i, j are circuit input wires:

- If the d-th layer is an "internal" layer, i.e. $d \in \{1, 2, \ldots, D-1\}$, then the underlying plaintexts are determined as follows.

  in layer d: all of the u_i values are as in *HybridReal*. For wires going into gates up to (but not including) gate g_s, the w_i values are as in *HybridReal*. For wires going into gate g_s or higher, the w_i values are as in *Simulated*.

  in layer $(d+1)$: all of the w_i values are as in *Simulated*. For wires going into gates up to (but not including) gate g_s, the u_i values are as in *HybridReal*. For wires going into gate g_s or higher, the u_i values values are as in *Simulated*.

- If the d-th layer is the input layer, i.e. $d = 0$.

  in layer d: For wires going into gates up to (but not including) gate g_s, the w_i values are as in *HybridReal*. For wires going into gate g_s or higher, the w_i values are as in *Simulated*.

  in layer $d+1$, all of the w_i values are as in *Simulated*. For wires going into gates up to (but not including) gate g_s, the u_i values are as in *HybridReal*. For wires going into gate g_s or higher, the u_i values are as in *Simulated*.

By definition, $\mathcal{H}_{0,0,0} =$ *Simulated* and $\mathcal{H}_{T,0,0} =$ *HybridReal*. Propositions 8.2, 8.3, and 8.4 below show that adjacent hybrids are statistically close. Statistical closeness of *HybridReal* and *Simulated* follows by a hybrid argument.

Proposition 8.2. *For every $t \in [T], d \in [D-1], s \in [S-1]$:*

$$\Delta(\mathcal{H}_{t,d,s}, \mathcal{H}_{t,d,s+1}) = \text{negl}(\delta)$$

Proposition 8.3. *For every $t \in [T], d \in [D-1]$:*

$$\Delta(\mathcal{H}_{t,d,S-1}, \mathcal{H}_{t,d+1,0}) = \text{negl}(\delta)$$

Proposition 8.4. *For every $t \in [T]$:*

$$\Delta(\mathcal{H}_{t,D-1,0}, \mathcal{H}_{t+1,0,0}) = \text{negl}(\delta)$$

We prove Proposition 8.2, the proofs of Propositions 8.3 and 8.4 are similar.

Proof of Proposition 8.2. Let $\mathcal{A}$ be *Eval*'s OC leakage attacker. Consider the hybrids $\mathcal{H}_{t,d,s}$ and $\mathcal{H}_{t,d,s+1}$, and the executions of *Eval* up to (and including) the t-th execution. Fix values for the a_i "public bits". Let g_s be the s-th gate between wire layers d and $d+1$, with input wires i, j, and output wire k (w.l.o.g. we assume that g_s is a NAND gate). The only difference between the hybrids is in the underlying plaintexts (w_i, w_j and u_k) of the key-ciphertexts pairs $(\ell'_i, \vec{d}'_i)$,$(\ell'_j, \vec{d}'_j)$,$(\ell_k, \vec{d}_k)$. In what follows, we call these the *target key-ciphertext pairs.* In $\mathcal{H}_{t,d,s}$, the underlying plaintexts for the target pairs are $(a_i \oplus v'_i)$, $(a_j \oplus v'_j)$, and $(a_k \oplus v'_k)$ (respectively). In $\mathcal{H}_{t,d,s+1}$ they are $(a_i \oplus v_i)$, $(a_j \oplus v_j)$, and $(a_k \oplus v_k)$ (respectively). The generations of all other key-ciphertext pairs are identical in the two hybrids.

$\mathcal{A}$'s OC leakage attack on *Eval* is viewed as an attack on the (simulated) ciphertext bank and on the *SafeNAND* procedure. Consider the leakage on the first $t-1$ executions of *Eval*, and on the ciphertexts generations of the t-th execution (but not yet on the *SafeNAND* and *SafeDup* operations in the t-th execution). This leakage is a function of all the key-ciphertext pairs produced in the first t executions. We can cast this leakage attack as an attack on the simulated ciphertext bank, and in particular on the generation of the target key-ciphertext pairs in the t-execution of *Eval*.

In particular, the computation of: (i) the leakage in the first $t-1$ executions, (ii) all generations in the t-th execution, (iii) the list of all key-ciphertext pairs produced in the t-th execution except the target ones, and (iv) the explicit values of all ciphertext banks at the end of the t-execution's generation, can be viewed as an attack on the (simulated) ciphertext bank's generation of the target key-ciphertext pairs. By Lemma 6.6 (security of the simulated ciphertext bank), there exists a simulator Sim_1 that can simulate this entire computation. Sim_1 only needs to know the underlying plaintexts for the non-target key-ciphertext pairs (which are identical in both hybrids), and multi-source leakage access to the target key-ciphertext pairs.

Let V be the view generated by Sim_1. It now remains to generate: (i) the leakage from the *SafeNAND* and *SafeDup* operations on all gates but g_s in the t-th execution (these do not involve any of the target pairs), (ii) the leakage from the *SafeNAND* operation on gate g_s (and the target pairs), and (iii) the leakage form all subsequent executions of *Eval* (a function of the ciphertext banks at the end of the t-th execution). The view V generated by Sim_1 can be used to compute items (i) and (iii) above, but for item (ii), generating the leakage from gate g_s's *SafeNAND* operation, we need access to the target keys and ciphertexts. Moreover, the *SafeNAND* operation can load LROTP keys and ciphertexts into memory together, and so multi-source leakage access to the target pairs may not be sufficient.

This is where we use the security of *SafeNAND*. Let $\mathcal{A}_2$ be the leakage attack that the adversary mounts on the *SafeNAND* operation for gate g_s (this attack is a function of V). The computation of all leakage seen by the adversary, can now be computed as a function of V and of this leakage attack. By Lemma 7.2 (security of *SafeNAND*), generating the view V (using Sim_1 and multi-source access to the target keys and the target ciphertexts), and then generating the leakage from *SafeNAND* on the target pairs (using $\mathcal{A}_2$ and OC leakage on this operation), can can be simulated using only the output of *SafeNAND*. This output, in both of the hybrids, is simply the (identical) bit a_k, and so we conclude that the hybrids are 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.

[CFGN96] Ran Canetti, Uriel Feige, Oded Goldreich, and Moni Naor. Adaptively secure multi-party computation. In *STOC*, pages 639–648, 1996.

[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.

[DF11] Stefan Dziembowski and Sebastian Faust. Leakage-resilient cryptography from the inner-product extractor. In *ASIACRYPT*, pages 702–721, 2011.

[DF12] Stefan Dziembowski and Sebastian Faust. Leakage-resilient circuits without computational assumptions. In *TCC*, pages 230–247, 2012.

[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 B. Lewko, Brent Waters, and Daniel Wichs. Storing secrets on continually leaky devices. In *FOCS*, pages 688–697, 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.

[HL11] Shai Halevi and Huijia Lin. After-the-fact leakage in public-key encryption. In *TCC*, pages 107–124, 2011.

[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 $\mathcal{AC}^0$ leakage without secure hardware. In *CRYPTO*, 2012.

[SYY99] Tomas Sander, Adam Young, and Moti Yung. Non-interactive cryptocomputing for $\mathcal{NC}^1$. In *FOCS*, pages 554–567, 1999.