Leakage-Tolerant Computation with Input-Independent Preprocessing

Nir Bitansky[*] Dana Dachman-Soled [†] Huijia Lin[‡]

Abstract

Following a rich line of research on leakage-resilient cryptography, [Garg, Jain, and Sahai, CRYPTO11] and [Bitansky, Canetti, and Halevi, TCC12] initiated the study of *secure interactive protocols* in the presence of arbitrary leakage. They put forth notions of *leakage tolerance* for zero-knowledge and general secure multi-party computation that aim at capturing the best-possible security when the private inputs of honest parties are exposed to direct leakage. However, so far, only a handful of specific two-party functionalities have been successfully realized.

In this work, we present the first constructions of leakage-tolerant multi-party computation protocols for evaluating *general functions*. The protocols can tolerate continual leakage, throughout an unbounded number of executions, provided that leakage is bounded within any particular execution. The protocols rely on an *input-independent preprocessing* that is performed once and for-all, and is then maintained under continual leakage. In the malicious setting, we also require a common reference string, and a constant fraction of honest parties.

At the core of our construction, is a tight connection between secure compilers in the Only-Computation-Leaks (OCL) model and leakage-tolerant protocols. In particular, we show that leakage-tolerant protocols with input-independent preprocessing are essentially *equivalent* to OCL compilers with strong properties. We then show how to construct such *strong OCL compilers*.

[*]Tel Aviv University, E-Mail: `nirbitan@tau.ac.il`
[†]University of Maryland, Email: `danadach@ece.umd.edu`
[‡]UCSB, E-Mail: `rachel.lin@cs.ucsb.edu`.

Contents

1 Introduction

Starting with the seminal works of [Yao82, GMW87, BGW88, CCD88], secure two-party and multi-party computation (2PC and MPC) have played a central role in shaping modern cryptography, and are still the subject of a great body of research. An MPC protocol allows m mutually distrustful parties to securely compute any function $f(\bar{x})$ of their private inputs $\bar{x} = (x_1, ..., x_m)$. The security of such a protocol π is classically captured through the *simulation paradigm* (also known as the *real-ideal paradigm*). The paradigm stipulates that the adversarial effect and view in a "real-world" execution of π, can be simulated in an "ideal-world", where parties run an idealized protocol I_f. In the idealized protocol inputs are simply handed to a *trusted party* (or an *ideal functionality*) that performs the computation for the parties.

The *traditional attack model* against a real-world MPC protocol allows a monolithic adversary to corrupt any given set of parties and completely control their behavior, as well as to observe all network messages and control the scheduling of the delivery of messages. In the ideal world, in contrast, there is no communication between the parties, and the adversary (or simulator) is limited to learning the original inputs of corrupted parties and possibly replacing them.

A crucial assumption underlying the traditional attack model is that *the internal computation state of honest parties is kept completely secret,* and the sole way of affecting honest parties and gaining information regarding their secret state is through the communication interface. However, as witnessed in recent years, attackers may sometimes learn partial information—termed *leakage*—about the secret state of the honest parties, for example, via a myriad of side-channels (e.g. timing, radiation, etc., see [Sta09]). This growing threat has spurred a large body of research devoted to the development of *leakage-resilient cryptography.*

Recently, extending the notions of *non-interactive* leakage-resilient primitives [ISW03, MR04, DP08, AGV09, NS09, Pie09, ADN$^+$10, BKKV10, DHLAW10, FKPR10, BSW13], the works of [GJS11, BCH12] initiated the study of cryptographic protocols in the presence of leakage—termed *leaky protocols.* Here, in addition to usual attacks allowed, the adversary may obtain leakage on honest parties' secret states, modelled as the outputs of arbitrary leakage functions chosen adaptively (from some broad class) by the adversary.

Leakage-resilience vs leakage-tolerance. A fundamental question in the context of leaky protocols, concerns the level of security that can be achieved in the presence of leakage. A natural goal would be to achieve the same security properties as in the traditional attack model (where there is no leakage); such a strong guarantee means leakage on the secure states of honest parties causes no degradation of security and is called *leakage resilience.* In the context of MPC protocols, this means that despite leakage, the real world protocols retain the same security guarantees that the ideal world protocols have in a leakage-free environment, where honest parties' inputs are totally secret. However, such a guarantee is inherently *impossible* to achieve if the real world adversary can directly leak on parties' inputs. Previous works have taken two different approaches towards overcoming this difficulty.

The first approach by Boyle Goldwasser, Jain and Kalai [BGJK12] circumvents the difficulty by assuming a *leakage-free input preprocessing phase* per session to avoid direct leakage on the inputs. Very roughly, every input is first encrypted in a leakage-free input preprocessing phase, and only the ciphertexts are delivered to the parties; thus, the inputs themselves are never exposed to leakage (and only the ciphertexts are). Then, additionally assuming a leakage-free input-*independent* preprocessing phase that is executed once for all before any executions, they construct MPC protocols that are *leakage-resilient*, for polynomial number of parties among which a constant fraction of parties are uncorrupted.

A different approach, introduced in [GJS11, BCH12] and followed in this work, recognizes that, in many scenairos direct leakage on parties' inputs might be unavoidable. For instance, in modern information systems, parties' inputs often emerge only when the parties are already online, potentially as the outputs of other computations. Here, since leakage-resilience is unachievable, the goal shifts to devise protocols that achieve the "best possible" security guarantees. Under the real-ideal paradigm, this means the real-world executions subject to ℓ bits of leakage on honest parties' secret states, should emulate ideal-world executions, subject to comparable (e.g., in length) leakage on the ideal states of parties. Such protocols are *leakage-tolerant* in the sense that real-world leakage on the secret state of honest parties, including inputs and randomness, is intuitively not worst than ideal-world leakage on the parties' ideal state. More generally, different levels of leakage-tolerance can be captured by different specifications of the ideal functionality that, in particular, determines what is the *ideal leaky state* of parties.

Separate state vs. joint state ideal world leakage. The most natural leaky ideal functionality, as defined in [BCH12, GJS11], defines the *ideal state* of each party to consist of *only their own private input and output* (which is necessary), but nothing more. This *separate state model* intuitively captures the best possible achievable guarantee, in the presence of leakage. From hereon, we simply refer to protocols achieving this notion as leakage-tolerant protocols. In [BCG$^+$11, GJS11, BCH12], leakage-tolerant protocols for several basic tasks, including, secure message transmission, commitment, oblivious transfer and zero knowledge, are constructed. The question of constructing leakage-tolerant protocols for general tasks, however, was left open.

The work of Boyle et al. [BGJ$^+$13] made significant progress towards resolving the above question by considering a weaker notion of leakage tolerance. In particular, they formulated a leaky ideal functionality that includes in the ideal state of each party the *joint inputs and outputs of all parties*. Roughly speaking, this means that the effect of learning a leakage function on the *isolated* state of any single party can be "emulated" by a simulator that learns a *joint* leakage function on the joint inputs and outputs of all parties. While certainly meaningful for many scenarios, this *joint state model* is not as ideal as the separate state model. In particular, it means that the input-output privacy of one participant can be compromised due to the leakage attack launched on another participant.

Summarizing the state-of-the-art, for general multi-party computation tasks, known constructions either achieves only a weaker notion of leakage tolerance [BGJ$^+$13], or relies on the physical assumption that leakage-free input-processing is available *per session* [BGJK12]. Without leakage-free input-processing, the exsistence of (separate-state) leakage-tolerant protocols has only been demonstrated for specific two-party functionalities. Determining the feasibility of leakage-tolerant multi-party protocols for general tasks, in any model without leakage-free input-preprocessing, remains open.

1.1 Contributions

Our primary contribution is the construction of multi-party protocols for general functions, with *continual leakage-tolerance*, relying only on an *input-independent leakage-free preprocessing phase* that is carried out once for all before any executions. In this model, prior to the computation, each of the parties obtains an initial state to be used later in the computation. The initial states are sampled, without any leakage, from a fixed (joint) distribution that is independent of the inputs (or function), which are only determined online. The online phase proceeds in iterations, where in each iteration a function is computed on the new set of inputs. The entire state of each party, including its current inputs, randomness, and initial state, are all exposed to leakage at any point in the protocol's execution, with the restriction that, between every two iterations, leakage is bounded. The initial state produced in the preprocessing phase is updated between the iterations, under leakage, and previous state and inputs are erased (which is necessary in the continual setting).

At the heart of our constructions, is a strong connection that we establish between Leakage-Tolerant Computation (LTC for short) and secure compilers in the *Only Computation Leaks (OCL) model*. We next recall the basics of OCL compilers, and overview our main results.

OCL vs. LTC. The OCL model [MR04] considers a setting where computation is performed using leaky memory, under the assumption that only the parts "touched" by the computation might leak some bounded amount of information. The memory is assumed to be initialized ahead of time and without leakage, typically with secret information associated with the computation. A (continual) OCL scheme, is meant to take any computation represented by a circuit $C(k,\cdot)$ with an associated secret k, and compile it, offline and without leakage, into a new computation $C'(k,\cdot)$ that fully protects the secret k when executed using leaky memory. The model of computation is *semi-honest* in the sense that the evaluation of $C'(k,\cdot)$ is assumed to be performed correctly, and the adversarial power is limited to bounded leakage, and choice of inputs for the computation. The intuitive property that C' protects the secret k is formalized by the requirement that the adversary's view (i.e., the leakage) can be simulated from the input and output of the computation alone.

To see the connection with the setting of LTC, it is convenient to interpret the evaluation of an OCL-compiled circuit as a leaky distributed computation performed jointly by t honest parties (or components) [BCG$^+$11, DF12, BGJK12]: The parties' memories are initiated with some preprocessed information about the secret k, generated in private without leakage, and they communicate with each other via secret and authenticated secure channels (capturing communication through the secure memory); during the compuation, *bounded* leakage can be obtained from the different parties *separately*, but it is not possible to leak on the joint state of any two parties. Furthermore, in the basic OCL model, leakage is assumed to be *ordered*; namely, computations are done by the parties in a certain order, and at each point in time it is only possible to leak from the active party. [1]

We summarize the differences between the models of OCL and LTC. The first, and minor, difference is the secure communication assumption; this difference can be bridged using existing constructions of leakage-tolerant communication [BCG$^+$11] to replace the secure channels. A more crucial gap is the preprocessing of secret inputs: In OCL, a shared secret input k is preprocessed offline without leakage and split between the t parties; in contrast, in LTC, the parties receive their inputs online under direct leakage. Another difference is that in the LTC model leakage is *unordered*; namely, it is possible to leak from any party at any time. Finally, an essential difference is that the OCL model assumes that all parties are honest and only subject to bounded leakage, whereas in LTC, we will also want to deal with corruption of parties, in which case their entire state is leaked to the adversary.

Bridging the gap: LTC with input-independent preprocessing and strong OCL. As discussed above, the LTC model is meant to deal with a reality in which inputs are unavoidably subject to leakage, without input-dependent preprocessing. Our first contribution is a generic

[1]Different constructions of OCL shemes handle different settings. For instance the construction of 2-component OCL schemes using hardwares by [JV10, DF12] are secure even if the leakage is not "unordered". However, so far, the only continual OCL scheme without hardware by [GR12] only handles ordered leakage. As we will see later, our construction of leakage-tolerant protocols is based on the Goldwasser-Rothblum OCL scheme, and bridging from ordered leakage to unordered leakge is one of the main hurdles.

transformation from a strengthened form of two-party OCL, referred to as *strong OCL*, to two-party LTC with input-independent preprocessing. Roughly speaking, the main feature of strong OCL schemes is that they allow simulating the internal states of the two parties obliviously of the adversary's leakage functions, and moreover, simulation of the party that produces the output depends only on the output of the computation, *oblivious of the input.* (In contrast, basic OCL schemes allows simulating leakage on the two parties using both input and output). In addition, strong OCL security is guaranteed even under *unordered* leakage.

The transformation yields (continual) LTC protocols for the basic case of two-party LTC without corruptions, and is a crucial step towards getting stronger forms of security in the presence of corruptions and with multiple parties. (In fact, differing from constructing MPC protocols without leakeage, where security against no corruption is easy; when dealing with leakage, achieving security with no corruption is already a hard step.) While it may seem as a drawback that our transformation relies on a strengthened OCL scheme with special properties, we show that, in fact, strong OCL is necessary for LTC.

Theorem 1 (informal)**.** *Any two-party strong continual OCL scheme implies two-party continual LTC relying on input-independent preprocessing (and secure channels), and vice versa. The LTC protocol is secure when no party is corrupted and can tolerate the same amount of leakage on every party as the OCL scheme.*

Obtaining Strong continual OCL. There are several known (continual) OCL schemes in the literature [JV10, GR10, DF12, GR12]; however, none satisfy as is the requirements of strong OCL. Augmenting (some of) the schemes to satisfy the required strong properties is relatively straight forward, if one allows reliance on leak-free hardware, as in [JV10, GR10, DF12]. But the transformation presented in Theorem 1 carries the use of hardware in OCL to the resulting LTC protocol; in order to avoid this, we need strong OCL schemes without any reliance on hardware. However, so far, even for standard OCL, the only scheme in the literature that does not rely on hardware is by Goldwasser and Rothblum [GR12] (referred to as the GR scheme henthforth), which requries a polynomial number of components; and constructing two-component OCL without hardware is a long standing open question.

To circuimvent this, we resort to a weaker primitive—a strong 2-component OCL with *auxiliary parties*—where the computation is carried out by two main components with the assistance of several auxiliary parties; its strong security additionally requires that the states of the auxiliary parties can be simulated obliviously of both the input and output, (thus the sole purpose of the these auxiliary parties is to assist the computation and their view is completely independent of the input and output). With this relaxation, we construct such an OCL scheme without reliance on hardware, which also directly gives a multi-component OCL scheme without hardware.

Theorem 2 (informal)**.** *There exists a continual strong 2-component OCL scheme with $O(1)$ auxiliary parties that does not rely on any hardware. Moreover, the scheme is unconditionally secure.*

Given a strong two-component OCL scheme with $O(1)$ auxiliary parties, Theorem 1 is then generalized to yield two-party LTC, with $O(1)$ *auxiliary parties*, whose ideal state is empty. These LTC protocol (assisted by the auxiliary parties) would eventually lead to standard multi-party LTC, with no auxiliary parties.

Multiparty LTC and security against corruptions. We then leverage the two-party protocols, with $O(1)$ auxiliary parties, to obtain m-party (continual) LTC protocols that withstand up to $(1-\epsilon)m$ corrupted parties, for large enough m.

Concretely, we provide two transformations: The first is a generic transformation for the case of no party corruptions: it takes any m-party LTC protocol with (leakage-free) input-*dependent* preprocessing and obtains a new protocol relying only on input-*independent* preprocessing and two-party LTC (with auxiliary parties). The second achieves the same in the case of $(1-\epsilon)m$ corruptions and is based on the specific protocol of Boyle et al. [BGJK12] in the common reference string.

Theorem 3 (informal)**.** *Any m-party LTC protocol with input-***dependent** *preprocessing and two-party LTC with $O(1)$ auxiliary parties, both secure when no party is corrupted, imply an m-party LTC protocol with input-***independent** *preprocessing when no party is corrupted (and without additional auxiliary parties or hardware). Moreover, the [BGJK12] protocol, in the common reference string model, and any two-party LTC as above, imply security under $(1-\epsilon)m$ corruptions, for any constant ϵ, and large enough m. The resulting protocols can tolerate the same amount of leakage as the original protocols.*

Universal composability and oblivious simulation. All of our constructions are presented within the framework of universal composability (UC) with leakage [Can01, BCH12]. In particular, our protocols admit the strong form of emulation known as *leakage-oblivious simulation*. An oblivious simulator works obliviously of the actual leakage function that the adversary produces, and provides a way (more precisely, a state-translation function) that simulates the real world states of honest players using their ideal state; namely, inputs and outputs. An essential feature of protocols with oblivious simulation (and thus of the protocols constructed in this work) is that they respect the universal composition theorem.

1.2 Techniques

We now present our main ideas and techniques. Before that, we first give some intuition regarding the difficulties in constructing LTC protocols, exemplify why the classical paradigms of 2PC and MPC fall short of achieving leakage-tolerance, and motivate the core use of OCL in our techniques.

Why classical protocols are not leakage-tolerant. The common paradigm for (say, semi-honest) 2PC and MPC protocols is for parties to first *secret share* their inputs, and then homomorphically compute a given boolean circuit over their shares. For example, in two-party GMW [GMW87], the invariant is that throughout the computation each one of the parties holds one random additive share for each wire in the circuit, where the two shares together encode the actual value of the wire; then, addition is done locally over shares, and multiplication is done with the help of *oblivious transfer*.

A basic problem that arises when using such a protocol under leakage is that additive secret sharing has very poor leakage-resilience properties. Indeed, it is possible to learn the value of any intermediate value in the circuit, by simply leaking a single bit from every party. However, in the ideal world, where it is only possible to separately leak a single bit on the input and output of each party, learning the value of some intermediate wires might be impossible. For example, the value of an intermidiate wire might be the inner product of two uniformly random inputs, and thus statistically close to uniform, even under independent leakage as above. A rather similar problem also appears in other classical protocols (e.g., Yao [Yao82]), even if not as explicitly.

A plausible route towards circumventing this problem would be to use a *leakage-resilient secret sharing* [BGK11, DLWW11, DF11], such as the inner product two-source extractor. The challenge is, however, to be able to compute the circuit gates over such shares in a leakage-resilient way.

While this is not known in the plain model, this approach is successfully executed in existing OCL protocols (e.g. in [GR12, DF12] with the inner-product extractor), with the help of a leakage-free preprocessing phase. Only that there, all secret inputs are preprocessed offline, while online inputs are public. Thus, a natural question is whether we can import, perhaps even generically, the OCL techniques to the setting of LTC.

Before explaining how we bridge the gap between LTC an OCL, we first quickly cover some of the technical basics of OCL compiler, which will be instrumental for our technical exposition.

Strong OCL. It is convenient to consider two-party OCL schemes for universal circuits $U(k, \cdot)$ with a fixed secret input k. A continual strong OCL scheme Λ consists of a compiler algorithm Comp that preprocesses a secret k, and splits it into two shares, and a two party protocol between a left component P_L and a right component P_R whose memories are initiated with the two shares respectively; to evaluate a function f on k, the two components P_L and P_R interact with each other, where P_L receives the input f and P_R produces the output $y = f(k)$. The scheme may be assisted by additional *auxiliary parties* $P_{A_1}, \ldots, P_{A_a}$, who obtain an initial state from Comp participate together with P_L, P_R in the protocol for computing $f(k)$.

The protocol proceeds in iterations: In each iteration i, the adversary may specify $f = f_i$ and repeatedly, adaptively, and in no particular order, obtain leakage from any one of the parties $P_L, P_R, P_{A_1}, \ldots, P_{A_a}$. Leakage is separate on the individual state of each party, and the number of bits leaked from any given party during a single iteration is bounded by ℓ.

We require an oblivious simulator $\mathcal{S}$ that can simulate the states of all parties $P_L, P_R, P_{A_1}, \ldots, P_{A_a}$, obliviously of the leakage functions specified by the adversary; the leakage to the adversary is then simulated simply by evaluating these functions on the simulated states. We further require that $\mathcal{S}$ admits a special structure: the state of P_L in every iteration i can be simulated given the current input f_i and output $y_i = f_i(k)$. The state of P_R can be simulated given only the output y_i, and obliviously of f_i. The state of any auxiliary party P_{A_i} can be simulated obliviously of either f_i, y_i.

From strong OCL to LTC without corruption: To illustrate the idea behind our construction of LTC protocols secure without corruption, let us focus on the case where (P_0, P_1), assisted by $(P_{A_1}, \ldots, P_{A_a})$, jointly compute a single-output function f of their private inputs (x_0, x_1), and only one of them receives the output. (This can be shown to be without loss of generality—protocols computing single-output functions can be transformed into protocols computing functions with different outputs for different parties). Furthermore, let us first focus on the non-continual setting, where only a single execution is performed, and later on generalize to the continual setting.

Given a strong OCL scheme, obtaining a one-time leakage-tolerant protocol ρ is rather simple. An easy way to compute a function is to ask P_0 to send its input x_0 to P_1, who then computes $y = f(x_0, x_1)$ directly; however, this is obviously non-private. Instead, we may have P_0 encrypt its input x_0 using a one-time pad r and send the ciphertext $c = x_0 \oplus r$ to P_1. Now privacy is re-installed, but it becomes unclear how to perform the computation.

In remedy, the OCL scheme provides a way for the two parties to jointly decrypt x_0 and compute $f(x_0, x_1)$. More precisely, the preprocessing phase samples the initial states of the OCL scheme with respect to a random string r, which will set as the OCL secret (referred to before as k, and distributes the left-component initial state to P_1 and the right-component initial state together with r to P_0. During the protocol execution, P_0 sends $c = r \oplus x_0$ to P_1; then, jointly with the auxiliary parties $P_{A_1}, \ldots, P_{A_a}$, they perform an OCL evaluation, where P_0 acts as the right component and P_1 acts the left component with input function $g((c, x_1), \cdot) = f(c \oplus (\cdot), x_1) = y$.

The OCL evaluation computes the function g on the secret r, producing the desired output y at P_0.

Showing that the above protocol ρ is indeed leakage tolerant, boils down to showing that the states of P_0 and P_1 can be simulated using their own input and output. By construction, P_0's state consists of x_0, r and the right-component state of OCL, while P_1's state contains x_1, c and the left-component state of OCL. A simple yet crucial observation is that since r is truly random, so is c. Therefore, the ciphertext c can be simulated directly using a random string $\tilde{c} \leftarrow U$, and later, the secret r can be simulated as $\tilde{c} \oplus x_0$; and the pair $(\tilde{c}, \tilde{r})$ distributes identically to their counterparts (c, r) in the real execution. Next, it follows from the strong leakage resilience of the OCL scheme that the left-component state state_L in P_1 can be simulated using the input function $g(\tilde{c}, x_1), \cdot)$ and the output y, while the right-component state state_R in P_0 can be simulated using only y. Therefore, overall the simulated state $(x_0, \tilde{r}, \mathsf{state}_R)$ of P_0 and the simulated state of $(x_1, \tilde{c}, \mathsf{state}_L)$ of P_1 depend, respectively, on their own input and output. The state of the auxiliary parties is guaranteed, by strong OCL, to be simulatable independently of the input and output, as required. Thus, leakage-tolerance follows as required.

To generalize the above to the continual setting, requires a modification of the above protocol. Indeed, in the previous protocol ρ, the secret r underlying the OCL is fully revealed to P_0. In the one-time case, security still holds as the adversary is restricted to bounded leakage. However, in the continual case, r will eventually be revealed entirely and the preprocessed OCL state would no longer provide any leverage.

Instead, we present a slightly more complicated protocol ρ', in which even the ciphertext c is computed using the OCL scheme by evaluating the function $g'(x_0, \cdot) = x_0 \oplus (\cdot)$ on the secret r; To do this, the preprocessing stage is modified to sample an additional set of OCL initial states with respect to the secure r, and to distribute the left-component initial state to P_0 who later acts as the left component when evaluating g'. The protocol ρ' is still a one-time protocol, but in which r is not fully revealed.

Moving to the continual case, instead of directly using r as the one-time pad, in the i^{th} iteration, we use the pseudo-random string produced by $\mathsf{PRF}(r, i)$ as the one-time pad, where r is used as the seed. It follows from the continual strong leakage-resilience of the OCL scheme that the seed r is always kept secret, and thus all the one-time pads generated are pseudo-random.

From LTC with input-independent processing back to strong OCL. We now briefly sketch how LTC with input-independent processing can be used to obtain strong OCL, thus implying that OCL is necessary for our goal. For simplicity, we describe the transformation with two parties, and with no auxiliary parties. It is easy to see that if we start from an LTC with a auxiliary parties, we will get strong OCL with a auxiliary parties.

The idea relies on the properties of inner product as a two-source extractor [CG88]. For an OCL secret $k \in \{0,1\}^n$ we consider a two-party function $g(f, \mathbf{L}_i, \mathbf{L}'), (\mathbf{R}_, \mathbf{R}'))$ that takes as input a description of $f : \{0,1\}^n \to \{0,1\}^*$, matrices $\mathbf{L}_i, \mathbf{R}_i \in \mathbb{F}_2^{\kappa \times n}$, which will be inner product shares of the key k (That is, $\mathbf{L}_i[j], \mathbf{R}_i[j] \in \mathbb{F}_2^{\kappa}$ and $\langle \mathbf{L}_i[j], \mathbf{R}_i[j] \rangle = k_j$), and two random matrices $\mathbf{L}, \mathbf{R} \in \mathbb{F}_2^{\kappa' \times \kappa'}$, where $\kappa' = \mathrm{poly}(\kappa)$. The function computes $f(k) = f(\langle \mathbf{L}_i[1], \mathbf{R}_i[1] \rangle, \ldots, \langle \mathbf{L}_i[n], \mathbf{R}_i[n] \rangle)$, and in addition new random shares $\mathbf{L}_{i+1}[j], \mathbf{R}_{i+1}[j] \in \mathbb{F}_2^{\kappa}$ of the key k, which will be computed using randomness $\langle \mathbf{L}'[1], \mathbf{R}'[1] \rangle, \ldots, \langle \mathbf{L}'[\kappa'], \mathbf{R}'[\kappa'] \rangle$, derived by inner-product extraction.

At compilation, initial shares $\mathbf{L}_1, \mathbf{R}_1$ of the key k are sampled and distributed to the parties, and input-independent processing is done with respect to the function g. Then at each iteration i the parties compute the function where P_0 inputs $f, \mathbf{L}_i, \mathbf{L}'$, where $\mathbf{L}_i$ was produced in the previous iteration and $\mathbf{L}'$ was sampled uniformly at random by P_0 itself. P_1 accordingly inputs $\mathbf{R}_i, \mathbf{R}'$. The properties of the LTC ensure that throughout all the different shares $\mathbf{L}, \mathbf{L}', \mathbf{R}, \mathbf{R}'$ are only leaked on separately, within some small bound. Strong OCL simulation then follows directly by the LTC simulation guarantee.

Obtaining strong OCL. At high level, our construction combines the two-component OCL scheme of Dziembowski and Faust [DF12] (referred to as the DF scheme henceforth), which relies on a leakage-free hardware that samples random orthogonal vectors, with the key ciphertext bank module in the Goldwasser-Rothblum OCL scheme [GR12] (Hencefort, the GR scheme). The ciphertext bank allows continual sampling of random orthogonal vectors at the presence of leakage using multiple components.

A natural idea is, indeed, to use auxiliary parties to emulate the GR ciphertext bank in order to implement the hardware needed for the DF-scheme. However, combining the two schemes and showing that the joint scheme admits strong simulation turns out to be quite challenging.

First, the GR-scheme is only proven secure in a weaker model of OCL, where at any moment, the leakage adversary can only obtain leakage from the component that is *currently activated*; in other words, leakage occurs in the same order as the sequence of sub-computations. As a first step towards our construction, we argue in a relatively modular way that the GR-scheme is also secure agasint "unordered leakage".

Second, we provide new simulation procedures for both the DF-scheme and the GR ciphertext banks that are structured as required by strong OCL. In fact, the need for specially-structured simulation is the reason we rely on the DF-scheme (besides from its simplicity), rather than relying entirely on the GR scheme. (So far, we do not know whether the GR OCL scheme can be simply adapted to satisfy the requirements of strong OCL schemes.) On the positive side, the resulting OCL scheme enjoys the relative simplicity of the DF scheme, while leveraging only the features of the GR scheme that are required to circumvent hardware (and are relatively more complex).

Due to the fact that the joint OCL schemes relies on details of DF-scheme and the GR ciphertext bank and the lack of space, we defer more detailed description of the joint scheme to Section C.

From two-party LTC to multiparty LTC. We now briefly explain our transformations from m-party LTC protocols with (leakage-free) input-*dependent* preprocessing to protocols, relying only on input-*independent preprocessing* and two-party LTC (even with auxiliary parties).

The high-level idea behind our transformations is as follows: the input processing of the multiparty LTC can be performed online, and under leakage, jointly by two parties. Namely, to process an input x_i of a given party P_{i_0}, it will use the help of another party P_{i_1}, and possibly of other auxiliary parties $P_{i'_1}, \ldots, P_{i'_a}$. The two parties would each sample independently a long enough random string r_{i_0} and r_{i_1}, respectively, and will use the LTC to compute the two-party function $g((x_i, r_{i_0}), r_{i_1})$ that computes the processing function $\bar{x}_i = \Pi(x_i; \mathsf{Ext}(r_{i_0}, r_{i_1}))$, where the randomness $r = \mathsf{Ext}(r_{i_0}, r_{i_1})$ is derived from the two random strings using a two-source extractor (e.g., inner product).

Once each party obtains this processed input, the parties then run the original protocol (which used to rely on a leakage-free processing of inputs). Intuitively, by the guarantees of two-source extraction, provided that there's only bounded separate leakage on each of the random strings, the randomness $r = \mathsf{Ext}(r_{i_0}, r_{i_1})$ is statistically independent of the leakage, achieving the same effect as leakage-free input preprocessing.

The above intuition holds only assuming that the party P_{i_1} assisting P_{i_0}, as well as the other assisting parties $P_{i'_1}, \ldots, P_{i'_a}$, are all honest. In particular, we can get a protocol in the no corruption (but only bounded leakage) model. Indeed, assuming P_{i_0} is (even semi-honestly) corrupted, the adversary, who now knows r_{i_0} can learn any ℓ-bounded function of r, by leaking on r_{i_0}. Furthermore, a malicious party may even bias the result and hurt the correctness of the protocol.

An appealing approach towards overcoming this problem is to have each party P_i jointly process its input with all other parties P_j, and then somehow aggregate the processed inputs, some of which computed with dishonest parties, into one processed input, which is safe to use. While we do not know how to do this in general, we observe that the input-processing in the [BGJK12] protocol possesses some additional properties, which give rise to a very similar approach. Specifically, in the [BGJK12] protocol the input processing function $\Pi(x_i, \mathsf{pk}, \mathsf{crs}) := (\mathsf{Enc}_{\mathsf{pk}}(x_i), \pi)$ samples an encryption of the input x_i under a public key pk for a fully-homomorphic encryption scheme, and a NIZK of knowledge π of the input x_i. Here the public key pk and the common reference string crs are determined as part of the input-independent processing (in particular, in the [BGJK12] scheme there is no leakage on the randomness for the encryption).

We implement the above idea is follows, let $a = O(1)$ be the number of auxiliary parties required for the two-party LTC. We let each P_i jointly compute with each coalition C of parties of size $a+1$ an encryption c_C of **zero**, and a NIZK for it being and encryption of zero with respect to pk. The randomness for this computation is computed by a two-source extractor, as above. Then, P_i aggregates all these ciphers by adding them together to a new zero encryption $\mathsf{c} = \sum_{C \in \binom{[m]}{a+1}} \mathsf{c}_C$, and uses them to get a rerandomnized encryption c^{x_i} of his input x_i, by encrypting x_i under leakage (and thus non-securely) and then adding to it the aggregated zero encryption c. In addition, P_i computes a NIZK of knowledge that it knows x_i and that c^{x_i} was generated by adding an encryption of x_i to ciphers c, and that NIZKs for the fact that they're zero ciphers were verified.

It can be shown that, in known fully homomorphic encryption schemes, the eventual encryption of x_i is semantically-secure provided that any one of the zero encryptions c_C, which follows, assuming there exists some non-corrupted coalition C of parties. Moreover, the NIZKs guarantee that malicious parties cannot bias the result of the computation.

In summary, the above transformation, can withstand the same number of corruptions as the [BGJK12] protocol: it allows for $(1-\epsilon)m$ parties to be corrupted, $m > \lambda$.

References

[ADN+10] Joël Alwen, Yevgeniy Dodis, Moni Naor, Gil Segev, Shabsi Walfish, and Daniel Wichs, *Public-key encryption in the bounded-retrieval model*, EUROCRYPT, 2010, pp. 113–134.

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

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

[BCH12] Nir Bitansky, Ran Canetti, and Shai Halevi, *Leakage-tolerant interactive protocols*, TCC, 2012, pp. 266–284.

[BGJ+13] Elette Boyle, Sanjam Garg, Abhishek Jain, Yael Tauman Kalai, and Amit Sahai, *Secure computation against adaptive auxiliary information*, CRYPTO, 2013.

[BGJK12] Elette Boyle, Shafi Goldwasser, Abhishek Jain, and Yael Tauman Kalai, *Multiparty computation secure against continual memory leakage*, STOC, 2012, pp. 1235–1254.

[BGK11] Elette Boyle, Shafi Goldwasser, and Yael Tauman Kalai, *Leakage-resilient coin tossing*, DISC, 2011, available at `http://eprint.iacr.org/2011/291`, pp. 181–196.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson, *Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract)*, STOC, 1988, pp. 1–10.

[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*, FOCS, 2010, pp. 501–510.

[BSW13] Elette Boyle, Gil Segev, and Daniel Wichs, *Fully leakage-resilient signatures*, J. Cryptology **26** (2013), no. 3, 513–558.

[Can01] Ran Canetti, *Universally composable security: A new paradigm for cryptographic protocols*, FOCS, 2001, pp. 136–145.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård, *Multiparty unconditionally secure protocols (extended abstract)*, STOC, 1988, pp. 11–19.

[CG88] Benny Chor and Oded Goldreich, *Unbiased bits from sources of weak randomness and probabilistic communication complexity*, SIAM J. Comput. **17** (1988), no. 2, 230–261.

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

[DF12] ———, *Leakage-resilient circuits without computational assumptions*, TCC, 2012, pp. 230–247.

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

[DLWW11] Yevgeniy Dodis, Allison B. Lewko, Brent Waters, and Daniel Wichs, *Storing secrets on continually leaky devices*, FOCS, 2011, pp. 688–697.

[DP08] Stefan Dziembowski and Krzysztof Pietrzak, *Leakage-resilient cryptography*, FOCS, IEEE Computer Society, 2008, pp. 293–302.

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

[GJS11] Sanjam Garg, Abhishek Jain, and Amit Sahai, *Leakage-resilient zero knowledge*, CRYPTO, 2011, pp. 297–315.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson, *How to play any mental game or a completeness theorem for protocols with honest majority*, STOC, 1987, pp. 218–229.

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

[GR12] ———, *How to compute in the presence of leakage*, FOCS, 2012, pp. 31–40.

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

[JV10] Ali Juma and Yevgeniy Vahlis, *Protecting Cryptographic Keys against Continual Leakage*, CRYPTO, 2010, pp. 41–58.

[MR04] Silvio Micali and Leonid Reyzin, *Physically observable cryptography*, TCC, 2004, pp. 278–296.

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

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

[Sta09] Francois-Xavier Standaert, *Introduction to side-channel attacks*, Secure Integrated Circuits and Systems (Ingrid M.R. Verbauwhede, ed.), Springer, 2009, pp. 27–44.

[Yao82] Andrew Chi-Chih Yao, *Protocols for secure computations (extended abstract)*, FOCS, 1982, pp. 160–164.

A Definitions

In this section, we provide the definition for leakage tolerant secure computation.

A.1 Leakage-Tolerant Protocols

We present our results within the UC framework [Can01] and its extension to the setting of leakage [BCH12]. We now provide a brief overview of the basic concepts and definitions of UC execution, emulation and composition in the face of leakage. For more details, we refer the reader to [BCH12].

Protocol execution. The model of executing protocol π with environment $\mathbf{Z}$ and adversary $\mathcal{A}$ is the same as in the basic UC framework, with the exception that the adversary is allowed to execute an additional *leakage* instruction. Such a leakage instruction includes the identity of a given party P (given by some pid) and a leakage function L, represented by a circuit. The leakage function is to be evaluated on the *entire state* of P:

$$\mathsf{state}_P = (\mathsf{state}_{\rho_1}, \dots, \mathsf{state}_{\rho_k}) \quad ,$$

which includes its local state state_{ρ_i}, with respect to any subroutine ρ_i that the party P is running. (Intuitively, the total state of P corresponds to the entire state that is physically available for joint leakage.) The leakage operation is completed once the adversary obtains $L(\mathsf{state}_P)$, and the environment $\mathbf{Z}$ is notified regarding the identity of the leaking party, and the number of bits leaked. (Technically, to deal with the non-modular leakage-operation and enable joint leakage on separate processes, corresponding to the same party pid, an *aggregator* entity is added to the UC framework. See [BCH12] for details.)

Leaky ideal functionalities and leakage in hybrid models. As in the standard (non-leaky) UC framework, the level of security of a given protocol, in the presence of leakage, is captured by the ideal functionality that it implements. For each party P, associated with an ideal functionality $\mathcal{F}$, the functionality defines a corresponding *ideal leaky state* $\mathsf{state}_{\mathcal{F}}$. (In principle the functionality, might also change its behavior after leakage occurs, e.g., disclose extra information to the adversary; however, in this work, all the behavior of functionalities will not change due to leakage.)

As in standard UC, we also consider hybrid protocols where a real protocol π might be invoking an ideal functionality $\mathcal{F}$. In such hybrid protocols, the entire leaky state of a party P:

$$\mathsf{state}_P = \left(\mathsf{state}_{\rho_1}, \ldots, \mathsf{state}_{\rho_k}, \mathsf{state}_{\mathcal{F}_1}, \ldots, \mathsf{state}_{\mathcal{F}_{k'}}\right)$$

also include its ideal leaky state $\mathsf{state}_{\mathcal{F}_i}$ with respect to any ideal functionality $\mathcal{F}_i$ that the party is associated with.

Emulation with leakage. Protocol emulation is defined as in the standard UC framework: for any adversary $\mathcal{A}$, there should exist a simulator $\mathcal{S}$, such that no environment will be able to distinguish between an interaction with $\mathcal{A}$ and an implementation protocol π or with $\mathcal{S}$ and an ideal specification protocol ϕ (typically, representing some ideal functionality $\mathcal{F}$).

Throughout most of this paper, we will consider a restricted class of *ℓ-bounded leakage adversaries* that are guaranteed to leak at most ℓ bits from any party throughout the protocol.

In the following definitions $\mathsf{EXEC}_{\pi,\mathcal{A},\mathbf{Z}}(\lambda, z)$ denotes the output distribution of environment $\mathbf{Z}$ when interacting with parties running protocol π on security parameter λ and auxiliary input $\mathbf{Z}$. Let $\mathsf{EXEC}_{\pi,\mathcal{A},\mathbf{Z}}$ denote the ensemble $\{\mathsf{EXEC}_{\pi,\mathcal{A},\mathbf{Z}}(\lambda, z)\}_{\lambda\in\mathbb{N}, z\in\{0,1\}^*}$.

Definition 1 (Emulation for leaky protocols [BCH12]). *Let π and ϕ be leaky protocols. Then π UC-emulates ϕ if, for any* **PPT** *adversary $\mathcal{A}$, there exists a* **PPT** *simulator $\mathcal{S}$, such that for any* **PPT** *environment $\mathbf{Z}$, the environment cannot distinguish an execution with π and $\mathcal{A}$ from one with ϕ and $\mathcal{S}$, i.e.*

$$\mathsf{EXEC}_{\pi,\mathcal{A},\mathbf{Z}} \approx_c \mathsf{EXEC}_{\phi,\mathcal{S},\mathbf{Z}} \ .$$

We say that π UC-emulates ϕ under ℓ-bounded leakage, if the above is only guaranteed for ℓ-leakage adversaries, which leak at most $\ell(\lambda)$ bits from any given party during any execution.

Remark 1 (Non-triviality of leakage bound). *We consider a model where the adversary will be able to ask for additional leakage whenever parties receive new messages or toss fresh random coins, or more generally, whenever the new state state_P^i cannot be deterministically computed from the previous state state_P^{i-1}. In particular, the leakage bound ℓ will always be greater than the number of such state updates.*

We note that when the above restriction is removed and some "leakage-free state updates" are allowed, then designing leakage-resilient protocols becomes easier (albeit still not trivial, because parties do not know which of their state updates are the leakage-free ones.)

Remark 2 (The dummy adversary). *A useful technicality in the UC framework is that it is sufficient to obtain protocol emulation only with respect to the* dummy adversary *$\mathcal{D}$ that simply reports all the information it receives to the environment and follows all the instructions of the environment regarding sending messages to parties and ideal functionalities. In the setting of leakage, the dummy adversary also executes any leakage instructions provided by the environment, and forwards to the environment their result. Relying on the fact that any adversary can be emulated by the adversarial environment itself, it is easy to show that simulation of the dummy adversary $\mathcal{D}$ implies simulation for any adversary.*

Thus, from hereon when discussing protocol emulation in UC framework, we can restrict attention to the dummy adversary.

Oblivious emulation. [BCH12] also consider a stronger form of emulation called *leakage-oblivious emulation* where the simulator works obliviously of the actual leakage function that the adversary produces, and the same leakage functions that are evaluated in the real world are also evaluated in the ideal world, but on a *simulated state.* Specifically, an oblivious simulator $\mathcal{S}$ has a separate subroutine $\widetilde{\mathcal{S}}$ for handling leakage. When $\mathcal{S}$ receives from the emulated adversary a request to apply a leakage function L to a party P, the subroutine $\widetilde{\mathcal{S}}$ is invoked to produce a *state translation function* T. This function is meant to transform the internal state of P in the ideal protocol ϕ into "a real state" in the implementation π. Once T is produced, the composed leakage function $L \circ T$ is evaluated on the state of the party state_P. Finally, when the leakage-result is returned, it is forwarded directly back to the emulated adversary and $\mathcal{S}$ returns to its state prior to the leakage event. The subroutine $\widetilde{\mathcal{S}}$ should operate **independently of the leakage function** L, and its only input is the state of $\mathcal{S}$ (prior to the leakage query) and the identity of the leaking party. Also, the leakage operation has no side effects on $\mathcal{S}$. That is, following the leakage event $\mathcal{S}$ return to the state that it had before that event.

For simplicity, we define oblivious simulation with respect to the dummy adversary $\mathcal{D}$.

Definition 2 (Oblivious simulation [BCH12])**.** *Let π and ϕ be leaky protocols. Then π strongly UC-emulates ϕ if there exists an* **oblivious PPT** *simulator $\mathcal{S}$ so for any* **PPT** *environment* $\mathbf{Z}$*:*

$$\mathsf{EXEC}_{\pi,\mathcal{D},\mathbf{Z}} \approx_c \mathsf{EXEC}_{\phi,\mathcal{S},\mathbf{Z}} \ ,$$

where $\mathcal{D}$ is the dummy adversary.

We say that π strongly UC-emulates ϕ under ℓ-bounded leakage, if the above is only guaranteed for ℓ-leakage environments, which leak at most $\ell(\lambda)$ bits from any given party during the execution.

Universal composition with leakage. Under Definition 2, [BCH12] extend the universal composition theorem [Can01] to the setting of leakage. The theorem directly extends to the case where all adversaries are ℓ-bounded.

Let π be a "real" implementation protocol and let ϕ be an "ideal" specification protocol, corresponding to an ideal functionality $\mathcal{F}$. We denote by $\rho = \rho[\pi]$ a protocol that includes subroutine calls to π. Also, we denote by ρ^π the system where the subroutine calls to π are processed by π and by $\rho^{\mathcal{F}/\pi}$ the system where these subroutine calls are processed by $\mathcal{F}$. Also, we say that a protocol is *modular up to leakage* if it only interacts directly with its caller and its subroutines (and the adversary).

Theorem 4 (UC-composition with leakage [BCH12])**.** *Let ρ, π, ϕ be protocols as above, all modular up to leakage, such that π UC-emulates ϕ with an oblivious simulator. Then ρ^π UC-emulates $\rho^{\mathcal{F}/\pi}$ with an oblivious simulator.*

Protocols with leakage-tolerant secure communication. All of our protocols rely on a leakage-tolerant secure communication functionality $\mathcal{F}_{\mathsf{LSC}}$, i.e., they are constructed in the $\mathcal{F}_{\mathsf{LSC}}$-hybrid model. The $\mathcal{F}_{\mathsf{LSC}}$ functionality (Figure 1) is similar to the standard secure communication functionality where parties can transmit messages to each other in a private and authenticated manner; however, unlike the standard case where privacy implies that only the length of messages is leaked, here the adversary can obtain additional leakage on the parties. Concretely, the ideal leaky state of parties consist of the transmitted message alone. (Thus, Any protocol that implements the $\mathcal{F}_{\mathsf{LSC}}$ functionality guarantees that leakage on the message and parties' randomness does not reveal more than leakage on the message alone.)

The $\mathcal{F}_{\mathsf{LSC}}$ functionality can be implemented using *non-committing encryption* and *leakage-resilient MACS* [BCG$^+$11, BCH12], and thus we shall freely use it throughout this paper.

Functionality $\mathcal{F}_{\mathsf{LSC}}$

Running with parties S, R and an adversary $\mathcal{S}$, $\mathcal{F}_{\mathsf{LSC}}$ operates as follows:

- Given input (send, S, R, m), send a message $(\mathsf{send}, S, R, |m|)$ to the adversary $\mathcal{S}$. In case S or R are corrupted, allow $\mathcal{S}$ to replace the message m with a new message m' of his choice. Once $\mathcal{S}$ allows forwarding the message, send (sent, S, m') to R.
- The ideal leaky state of each one of the parties is m.

Figure 1: The Secure Communication Functionality

A.2 Leakage-Tolerant Computation

Like in standard secure computation, we consider a setting where m parties $P_1, \ldots, P_m$ are interested in computing a joint function of their inputs $(y_1, \ldots, y_m) = f(x_1, \ldots, x_m)$ of their inputs $x_1, \ldots, x_m$. Our technical results are mainly focused on a specific two-party setting, where two parties (P_0, P_1) jointly compute a function $(y_0, y_1) = f(x_0, x_1)$ of their inputs (x_0, x_1), and where *auxiliary parties* without input or output are available to help the two parties compute the function. In Section E, we show how to extend our results from the specific two-party (with auxiliary parties) setting to the standard multiparty (without auxiliary parties) setting.

We consider two attack models. In the basic model, we consider the two-party with auxiliary party setting and only allow the adversary to leak from any party, but not to corrupt parties. We call this the "no-corruption" setting. Then, when we consider multiparty protocols, we allow the adversary to statically corrupt some subset of parties (or no parties) and simultaneously leak on the remaining honest parties' states.

A.2.1 Leakage-Tolerant Computation with Input-Independent Preprocessing

In the model of leakage-tolerant computation with input-independent preprocessing (LTIIP), the parties $(P_1, P_2, \ldots, P_m)$ rely on a leakage-free preprocessing phase. The preprocessing is done once and is **independent of the future inputs for the computation**.

We construct leakage-tolerant protocols that, after a single preprocessing stage, support an arbitrary number of evaluations on arbirary inputs $(x_0^1, x_1^1), (x_0^2, x_1^2), \ldots$. We call this setting the *continual* setting and call such protocols *continual leakage-tolerant protocols*. Note that the distinction between continual and one-time is only relevant in the preprocessing model. Formally, continual leakage-tolerant protocols are protocols that implement a *continual leakage-tolerant computation (LTC) functionality*, at the presence of an adversary who can obtain independent ℓ-bounded leakage in each evaluation—we refer to such adversaries as *continual ℓ-bounded adversaries.*

Definition 3 (continual ℓ-bounded adversaries)**.** *$\mathcal{A}$ is a continual ℓ-bounded adversary, against a stateful protocol π, if between any two consecutive executions of π , $\mathcal{A}$ leaks at most $\ell(\lambda)$ bits on any given party.*

The leakage-free preprocessing is captured by a leakage-free sampling functionality $\mathcal{F}_{\mathsf{LFS}}^{\Delta}$ (Figure 3) that is used (in the real world) to implement $\mathcal{F}_{\mathsf{2LTC}}$. The functionality $\mathcal{F}_{\mathsf{LFS}}$ is associated with efficiently samplable distribution Δ that samples states $(\mathsf{init}_0, \mathsf{init}_1)$ for each one of the parties. The distribution is fixed, and only gets as input the security parameter, and possibly other length parameters; in particular, it is independent of the protocol's execution and parties' inputs. We think of this distribution as being sampled offline and distributed to the parties. (Alternatively, in case

Functionality $\mathcal{F}^{f,\infty}_{\mathsf{2LTC\text{-}AUX}}$

Running with parties $(P_0, P_1, P^{\mathsf{aux}}_1, \ldots, P^{\mathsf{aux}}_a)$ and an adversary $\mathcal{S}$, $\mathcal{F}^{f,\infty}_{\mathsf{2LTC\text{-}AUX}}$ operates as follows:

- Given inputs (x^i_1, x^i_2) from (P_0, P_1), it computes $(y^i_0, y^i_1) = f(x^i_0, x^i_1)$, notifies $\mathcal{S}$ of the computation, and once $\mathcal{S}$ allows, sends y^i_j to P_j.
- The ideal leaky state of party P_b, $b \in \{0, 1\}$ after obtaining the i input and before obtaining $i+1$-st, is (x^i_b, y^i_b). The ideal leaky state of auxiliary party P^{aux}_j, $j \in [a]$ is $\perp$.

Figure 2: The continual 2-party LTC Functionality with auxiliary parties

the sampler is not trusted, the parties can run a non-leaky protocol to compute their preprocessed state.)

The ideal leaky state of the parties in the functionality is empty modeling the fact that the preprocessing is leakage free. The functionality can be invoked once during the execution in order to retrieve the initial states $(\mathsf{init}_0, \mathsf{init}_1)$.

Our protocols will thus be constructed in the $\mathcal{F}_{\mathsf{LFS}}$-hybrid model. (All previous features, such as universal composition, and the completeness of emulation with respect to the dummy adversary, also hold in this model.)

Functionality $\mathcal{F}^{\Delta}_{\mathsf{LFS}}$

Running with parties $(P_1, \ldots P_m)$ and an adversary $\mathcal{A}$, $\mathcal{F}^{\Delta}_{\mathsf{LFS}}$ operates as follows:

- When activated with security parameter λ, and possibly another length parameter n, sample states $(\mathsf{init}_1, \ldots, \mathsf{init}_m) \leftarrow \Delta(1^\lambda, 1^n)$.
- When invoked for the first time, send init_i to P_i. In subsequent invocations, return $\perp$.
- The ideal leaky states of each party P_i only includes its output init_i (if it has already been generated).

Figure 3: The Leakage-Free Sampling Functionality

B Strong Only-Computation-Leaks Compilation

In this section, we discuss the *only computation leaks model* (OCL) and OCL schemes, which will play a central role in our constructions.

At high-level, the OCL model [MR04] considers a scenario where a given computation is performed using leaky memory; however, the assumption is that, at any given point in time, only the parts currently "touched" by the computation might leak some bounded amount of information. The memory is assumed to be initialized ahead of time and without leakage, possibly with secret information associated with the computation. The model is *semi-honest* in the sense that the computation is assumed to be performed correctly, and the adversarial power is limited to leakage, and choice of inputs for the computation. The security requirement is that the adversary's view (i.e., the leakage) can be simulated from the input and output of the computation alone, independent of the secret of the computation.

Different variants of OCL have been considered in the literature. One varient is the leaky distributed system (LDS) considered by [BCG+11, DF12, BGJK12]. In a LDS, the leaky computation is performed jointly by several parties (or components) communicating with each other via secure channels—in other words, the communication between different parties or components is secret and authenticated (modeling communication through the secure memory). In this modeling, bounded leakage can be obtained from different parties *separately*, but it is not possible to leak on the joint state of any two parties.

The LDS model can be viewed as a strengthening of the classical OCL model [MR04], where the attackers cannot obtain leakage in an arbitrary order, but are restricted to leak only from the component (or sub-computation) that is currently active. We call this model the *ordered OCL model.*

In this work, our definition and construction follow the LDS model. Below we recall its formalization and simply refer to it as the OCL model. Our construction in Section C relies on a previous construction [GR12] in the ordered OCL model; we provide more detailed comparison relavant to our construction in Section C.

N-component OCL. A N-component OCL scheme for a circuit $C(k,\cdot)$, associated with a secret k, consists of an efficient compiler Comp and a N-party protocol $\Pi = (P_1^{\mathrm{OCL}}, P_2^{\mathrm{OCL}}, \cdots, P_N^{\mathrm{OCL}})$. To compute $C(k,\cdot)$ in a leakage-resilient way, the circuit is *compiled* ahead of time by $\mathsf{Comp}(C(k,\cdot))$ that produces an initial state $(\mathsf{init}_1^{(k)}, \cdots, \mathsf{init}_N^{(k)})$ for each one of the N parties, and this compilation is done "in the dark" without any leakage. Then, at computation time, the parties can compute together $y = C(k,x)$ for any input x by running the protocol Π.

Below we provide the formal definition of OCL schemes for *universal circuits*; this is w.l.o.g since to evaluate any circuit C, one can provide C as a part of the input to the universal circuit. Furthermore, towards our end goal of constructing composable leakage tolerant protocols, where the simulator is oblivious of the leakage queries from the adversary, we consider directly strengthened OCL schemes that have obvious simulators.

OCL schemes with oblivious simulation: Let $\{U_T(k,f)\}_{T\in\mathbb{N}}$ denote the family of *universal circuits*: U_T takes two inputs k and f of length at most T, where f represents a T-step deterministic computation, and computes $f(k)$. (If the computation does not complete in T steps, we assume w.l.o.g. that the output of $U_T(k,f)$ is $\perp$).

Definition 4 (Continual N-component OCL schemes)**.** *We say that* $\Lambda = (\mathsf{Comp}, \Pi = \langle P_1^{OCL}, \cdots, P_N^{OCL}\rangle)$ *is a* continual, *N-component OCL scheme for the universal circuit family $\{U_T(k,f)\}_{T\in N}$ if it satisfies the following properties.*

Initialization: *For every security parameter λ and $T \in \mathbb{N}$, $k \in \{0,1\}^T$, the compiler* $\mathsf{Comp}(1^\lambda, U_T, k)$ *runs in time* $\mathrm{poly}(\lambda, T)$ *and outputs N initial states* $\mathsf{init}_1, \mathsf{init}_2, \cdots, \mathsf{init}_N$.

Unbounded-time evaluation: *The evaluation procedure invokes the protocol Π between the components $P_1^{OCL}(\mathsf{init}_1)$, $P_2^{OCL}(\mathsf{init}_2)$ to $P_N^{OCL}(\mathsf{init}_N)$, which interact in an* arbitrary *polynomial number of iterations: In the i^{th} iteration, P_1^{OCL} receives an input $f_i \in \{0,1\}^T$ and P_2^{OCL} produces an output y_i. At the end of the evaluation, an update procedure is carried out, producing the new initial states for the next iteration; then all information other than the new initial states are erased.*

For every component $j \in [N]$, denote by $\mathsf{init}_{i,j}$ the initial states of component j at the onset of the i^{th} iteration (in the first iteration, $\mathsf{init}_{1,j} = \mathsf{init}_j$), and $\mathsf{evl}_{i,j}$ the random coins tossed *and messages exchanged by each P_j^{OCL} during the i^{th} iteration (including its state during the update phase).*

Correctness with adaptive input selection: *For every $\lambda \in \mathbb{N}$, every $T \in \mathbb{N}$ polynomially related to λ, every $k \in \{0,1\}^T$, every auxiliary input $z \in \{0,1\}^{\mathrm{poly}(\lambda)}$, and every PPT adversary $\mathcal{A}$, consider the following real experiment $\mathsf{RealExp}_{\mathcal{A}}^{\infty}(1^\lambda, T, k, z)$ where $\mathcal{A}$ initiates an arbitrary number of evaluations with adaptively chosen inputs. It holds that with all but negligible probability, the outputs of all evaluations in $\mathsf{RealExp}_{\mathcal{A}}^{\infty}(1^\lambda, T, k, z)$ are correct.*

We say that Λ has perfect correctness, if the above holds with probability 1.

We now describe the security experiments of OCL schemes. Λ is said to be ℓ-leakage-resilient with oblivious simulation if there is a simulator $\mathcal{S}$, such that, for every $\lambda \in \mathbb{N}$, $T \in \mathbb{N}$ polynomially related to λ, every $k \in \{0,1\}^T$, and auxiliary input $z \in \{0,1\}^{\mathrm{poly}(\lambda)}$, the views of the adversary in the following real and ideal experiments are indistinguishable. In the real world, the adversary has the power of obtaining leakage independently from each component, during honest OCL evaluations; the inputs to the evaluations are chosen adaptively by the adversary. In the ideal world, the adversary obtains leakage from states of the components simulated by an oblivious simulator. More formally,

$\mathsf{RealExp}_{\mathcal{A}}^{\infty}(1^\lambda, T, k, z)$ (Real experiment): The adversary $\mathcal{A}(1^\lambda, T, k, z)$ proceeds as follows:

1. The initial states $(\mathsf{init}_1, \cdots, \mathsf{init}_N) \leftarrow \mathsf{Comp}(1^\lambda, U_T, k)$ are sampled.
2. $\mathcal{A}$ launches ℓ-bounded leakage attacks on an unbounded number of evaluations of its choice: In the i^{th} iteration,
   (a) $\mathcal{A}$ submits an input function $f_i \in \{0,1\}^T$, which is evaluated on k by resuming the protocol execution of Π between the components $P_1^{\mathrm{OCL}}(\mathsf{init}_{i,1}), \cdots, P_N^{\mathrm{OCL}}(\mathsf{init}_{i,N})$ with input f_i to the first component P_1^{OCL}.
   (b) $\mathcal{A}$ launches an ℓ-bounded leakage attack on the i^{th} evaluation: It issues an arbitrary number of leakage queries (P_j^{OCL}, L) for $j \in [N]$ adaptively, and obtains leakage answers $L(\mathsf{init}_{i,j}, \mathsf{evl}_{i,j})$, as long as the total amount of leakage on each P_j^{OCL} in this iteration is smaller than $\ell(\lambda)$ bits.
   (c) $\mathcal{A}$ obtains the output of the evaluation, which is the output of P_2^{OCL}.

Denote by $\mathsf{view}_{\mathcal{A}}^{\ell,\infty}(1^\lambda, T, k, z)$ the view of $\mathcal{A}$ in the above experiment.

$\mathsf{IdealExp}_{\mathcal{S},\mathcal{A}}^{\infty}(1^\lambda, T, k, z)$ (Ideal experiment): The adversary $\mathcal{A}(1^\lambda, T, k, z)$ participates in the same experiment as above, except that during its ℓ-bounded leakage attacks, it is given simulated answers: In the i^{th} iteration,

(a) $\mathcal{A}$ submits an input function $f_i \in \{0,1\}^T$. $\mathcal{S}(1^\lambda, T, i, f_i, f_i(k);\ \mathbf{w}_i)$ is invoked, producing simulated states $(\widetilde{\mathsf{intl}}_{i,1}, \cdots, \widetilde{\mathsf{intl}}_{i,N}, \widetilde{\mathsf{evl}}_{i,1}, \cdots, \widetilde{\mathsf{evl}}_{i,N})$, where w_i is the fresh random coins tossed for the simulation in iteration i and $\mathbf{w}_i = w_1, \cdots, w_i$ is all the random coins that have been tossed for simulation in the first i iterations.

(b) Whenever $\mathcal{A}$ issues a leakage query (P_j^{OCL}, L) for $j \in [N]$, it is given the simulated answer $L(\widetilde{\mathsf{intl}}_{i,j}, \widetilde{\mathsf{evl}}_{i,j})$, as long as the total amount of leakage on each P_j^{OCL} in this iteration is smaller than $\ell(\lambda)$ bits.

(c) $\mathcal{A}$ obtains the simulated output of the evaluation in $\widetilde{\mathsf{evl}}_{i,2}$.

Denote by $\widetilde{\mathsf{view}}^{\ell,\infty}_{\mathcal{S},\mathcal{A}}(1^\lambda, T, k, z)$ the view of $\mathcal{A}$ in the above experiment.

Definition 5 (Continual ℓ-Leakage-resilience with oblivious simulation)**.** *We say that a continual OCL scheme Λ is continually ℓ-leakage-resilient with oblivious simulation if there is a* **PPT** *simulator $\mathcal{S}$, such that, for every* **PPT** *adversary $\mathcal{A}$, the following two ensembles are indistinguishable.*

- $\{\mathsf{view}^{\ell,\infty}_{\mathcal{A}}(1^\lambda, T, k, z)\}_{\lambda\in\mathbb{N}, T\in\mathbb{N}, k,z\in\{0,1\}^{\mathrm{poly}(n)}}$
- $\{\widetilde{\mathsf{view}}^{\ell,\infty}_{\mathcal{S},\mathcal{A}}(1^\lambda, T, k, z)\}_{\lambda\in\mathbb{N}, T\in\mathbb{N}, k,z\in\{0,1\}^{\mathrm{poly}(n)}}$

Strong OCL schemes: In the above definition, the oblivious simulator simulates the states of all N components in each evaluation i depending on both the input f_i and output $f_i(k)$. We consider the following strengthening: Only the simulation of the first component depends on both the input and output, whereas the simulation of the second component depends solely on the output, and simulation of the rest components depends on neither the input nor output.

Definition 6 (Continual strong OCL Schemes)**.** *We say that $\Lambda = (\mathsf{Comp}, \Pi = (P_1^{OCL}, P_2^{OCL}, \cdots, P_N^{OCL}))$ is a* continually ℓ-leakage-resilient strong OCL scheme *if it satisfies the following property.*

Strong ℓ-leakage resilience: *Λ admits an oblivious simulator $\mathcal{S}$ satisfying Definition 5 with the following structure: $\mathcal{S}$ consists of three sub-algorithms $(\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3)$ and on input $(1^\lambda, T, i, f_i, f_i(k)\ ;\ \mathbf{w}_i)$, $\mathcal{S}$ invokes these sub-algorithms as follows:*

- $\mathcal{S}_1(1^\lambda, T, i, f_i, f_i(k);\ \mathbf{w}_i) = (\widetilde{\mathsf{intl}}_{i,1}, \widetilde{\mathsf{evl}}_{i,1})$
- $\mathcal{S}_2(1^\lambda, T, i, f_i(k);\ \mathbf{w}_i) = (\widetilde{\mathsf{intl}}_{i,2}, \widetilde{\mathsf{evl}}_{i,2})$
- $\mathcal{S}_3(1^\lambda, T, i;\ \mathbf{w}_i) = (\widetilde{\mathsf{intl}}_{i,3}, \cdots, \widetilde{\mathsf{intl}}_{i,N}, \widetilde{\mathsf{evl}}_{i,3}, \cdots, \widetilde{\mathsf{intl}}_{i,N})$

and outputs $(\widetilde{\mathsf{intl}}_{i,1}, \cdots, \widetilde{\mathsf{intl}}_{i,N}, \widetilde{\mathsf{evl}}_{i,1}, \cdots, \widetilde{\mathsf{evl}}_{i,N})$.

Strong two-component OCL with auxiliary components In this work, we often consider the special case of a strong *two-component* OCL scheme, and refer to the two components the left and right components, denoted as P_L^{OCL} and P_R^{OCL}. The strong oblivious simulation property ensures that the state of the left component in each evaluation can be simulated using both the input and output, whereas the state of the right component can be simulated using only the output. We sometimes view a strong $(N+2)$-component OCL scheme as a strong 2-component OCL scheme using N auxiliary parties $P_{A_1}^{\mathrm{OCL}}, \cdots, P_{A_N}^{\mathrm{OCL}}$; they are called auxiliary parties since their states can be simulated independent of both the input and output. In this view, we denote the strong oblivious simulator as $S = (S_L, S_R, S_A)$. Viewing strong N-component OCL as strong two-component OCL with auxiliary components is instrumental for our construction of leakage tolerant protocols.

C Obtaining Strong OCL

In this section, we construct a *continually $\ell(\lambda)$-leakage-resilient strong OCL scheme*, for $\ell(\lambda) = \lambda^{\Omega(1)}$, where λ is the security parameter. Our scheme combines together the OCL scheme of Dziembowski-Faust [DF12] OCL schemes, and the *ciphertext bank* construction from the OCL scheme of Goldwasser and Rothblum [GR12].

The core of our scheme relies on the DF scheme, which is relatively simple to describe and analyze. The original DF scheme, however, relies crucially on leakage-free hardware for *orthogonal* *share generation.* Here, instead of relying on hardware, we implement orthogonal share generation using the ciphertext bank component from the GR scheme, and thus avoid any reliance on hardware. Achieving the extra properties of strong OCL (comparing to standard OCL), requires dealing with extra technical challenges (mainly in simulation). For this purpose, we modify the schemes and their analysis. As a side product, our modfication actually simplifes the constructions and analysis of [DF12, GR12] and leads to a simpler OCL compiler without using hardware.

In Section C.1, we define the orthogonal share generation required in our scheme, and show how to achieve it based on the GR ciphertext bank in Section C.2. In Section C.3, we present the full scheme.

C.1 Definition of Orthogonal Share Generation Protocol

At high-level, an orthogonal share generation procedure is a protocol that is meant to enable two parties P_L and P_R to repeatedly generate pairs of orthogonal vectors $L_i, R_i \in \mathbb{F}_2^m$ in a leakage tolerant way. Here, by leakage tolerance, we mean that it is possible to simulate an execution of the protocol, such that, an adversary that obtains (separate) leakage on each of the participating parties, cannot tell the difference from a real execution. As before, we require the simulation to be "oblivious" (of the leakage functions), that is, the simualtor can translate the left shares $\{L_i\}$ to the state of the left component and the right shares $\{R_i\}$ to that of the right component. Given such an oblivious simulator, it is easy to see that the simulator can *bias* (L_i, R_i) to have arbitrary inner product; the produced simulated states would still be indistinguishable to any adversary that obtains only bounded leakge from each component in each iteration. (An update procedure is executed between iterations, in order to inject new entropy to the system.) Also, while ideally we would like to obtain a protocol with only two participating parties, so far the only known leakage tolerant way of generating orthognal shares is by GR, which requires multiple components. Correspondingly, our solution will require some constant number of additional auxiliary parties.

We next present the interface of an orthogonal share generation protocol in more detail. The definition is very similar to the definition of strong OCL; but we formalize the use of erasure more explicitly since it is crucial for the construction.

Definition 7 (Orthogonal share generation)**.** *We say that* $(\mathsf{Comp}^{\mathsf{OG}}, \mathsf{OG} = \langle P_L, P_R, P_{A_1}, \ldots, P_{A_a} \rangle)$ *is a* continual *orthogonal share generation protocol if it satisfies the following properties.*

Initialization: $\mathsf{Comp}(1^{m(\lambda)})$ *is a* **PPT** *compiler that produces initial states* $\mathsf{init}_L, \mathsf{init}_R, \mathsf{init}_{A_1}, \ldots, \mathsf{init}_{A_a}$.

Unbounded-time evaluation: *The evaluation procedure invokes the protocol* OG *between the parties, which interact in an* arbitrary *polynomial number of iterations: In the* i^{th} *iteration,* P_L *receives an output* $L_i \in \mathbb{F}_2^m$ *and* P_R *receives an output* R_i. *At the end of the evaluation, an update procedure is carried out, producing the new initial states for the next iteration; then all information other than the new initial states are erased.*

For every party P, *denote by* $\mathsf{init}_{i,P}$ *the initial states of* P *at the onset of the* i^{th} *iteration, and* $\mathsf{evl}_{i,P}$ *the random coins tossed and messages exchanged by each* P *during the* i^{th} *iteration, including its state in the update phase. In case,* P *performs erasures within a given iteration we view* $\mathsf{evl}_{i,P} = (\mathsf{evl}_{(i,1),P}, \ldots, \mathsf{evl}_{(i,e),P})$ *as a collection of evaluation states, where* e *is the number of erasures.*

Orthogonality: *In any iteration* i, *it holds that* $\langle L_i, R_i \rangle = 0$.

We now describe the security properties that we require from the orthogonal generation protocol. We say the an orthogonal generation protocol $(\mathsf{Comp}^{\mathsf{OG}}, \mathsf{OG})$ is ℓ-leakage-resilient with *strong* *simulation* if there is a simulator $\mathcal{S} = (\mathcal{S}_L, \mathcal{S}_{R,A})$, such that, for every $\lambda \in \mathbb{N}$, polynomial $m(\lambda)$, and auxiliary input $z \in \{0,1\}^{\text{poly}(\lambda)}$, the views of the adversary in the following real and ideal experiments are indistinguishable.

In the real world, the adversary obtains continual (bounded) leakage independently from each party in the execution of OG, whereas in the ideal world, it obtains leakage from simulated party states. More formally,

$\mathsf{RealExp}^{\infty}_{\mathcal{A}}(1^\lambda, m, z)$ (Real experiment): The adversary $\mathcal{A}(1^\lambda, m, z)$ proceeds as follows:

1. The initial states $(\mathsf{init}_L, \mathsf{init}_R, \mathsf{init}_{A_1} \cdots, \mathsf{init}_{A_a}) \leftarrow \mathsf{Comp}(1^{m(\lambda)})$ are sampled.
2. $\mathcal{A}$ launches ℓ-bounded leakage attacks on an unbounded number of executions: In the i^{th} iteration, $\mathcal{A}$ launches an ℓ-bounded leakage attack on the i^{th} evaluation. It issues an arbitrary number of leakage queries (P, Φ) for any party P adaptively, and obtains leakage answers $\Phi(\mathsf{init}_{i,P}, \mathsf{evl}_{i,P})$. *If P performs erasures within a given iteration, then Φ is only applied to its current state* $\mathsf{evl}_{(i,j),P}$. The total amount of leakage on each P in this iteration is smaller than $\ell(\lambda)$ bits.

Denote by $\mathsf{view}^{\ell,\infty}_{\mathcal{A}}(1^\lambda, T, k, z)$ the view of $\mathcal{A}$ in the above experiment.

$\mathsf{IdealExp}^{\infty}_{\mathcal{S},\mathcal{A}}(1^\lambda, m, z)$ (Ideal experiment): The adversary $\mathcal{A}(1^\lambda, m, z)$ participates in the same experiment as above, except that during its ℓ-bounded leakage attacks, it is given simulated answers: In the i^{th} iteration, $\mathcal{S}$ samples coins w_i to be jointly used by $(\mathcal{S}_L, \mathcal{S}_{R,A})$, and samples uniformly random orthogonal vectors (L_i, R_i).

(a) $\mathcal{S}(1^\lambda, m, i; \mathbf{w}_i, \mathbf{L}_i, \mathbf{R}_i)$ is invoked, producing simulated states

$$(\widetilde{\mathsf{intl}}_{i,L}, \widetilde{\mathsf{intl}}_{i,R}, \widetilde{\mathsf{intl}}_{i,A_1} \cdots, \widetilde{\mathsf{intl}}_{i,A_a}, \widetilde{\mathsf{evl}}_{i,L}, \widetilde{\mathsf{evl}}_{i,R}, \widetilde{\mathsf{evl}}_{i,A_1} \cdots, \widetilde{\mathsf{evl}}_{i,A_a}).$$

where $\mathbf{w}_i = w_1, \cdots, w_i$ is all the random coins that have been tossed for simulation in the first i iterations, and and $(\mathbf{L}_i, \mathbf{R}_i) = (L_1, R_1), \ldots, (L_i, R_i)$ consists of all the previous sampled orthognal shares. (In case, there are erasures within the iteration the states $\widetilde{\mathsf{evl}}$ are split according to the erasures as noted above.)

We further require that the simulator S consists of algorithms $(\mathcal{S}_L, \mathcal{S}_{R,A})$ as follows:

- $\mathcal{S}_L(1^\lambda, m, i; \mathbf{w}_i, L_i)$ generates any state corresponding to P_L.
- $\mathcal{S}_{R,A}(1^\lambda, m, i; \mathbf{w}_i, \mathbf{R}_i)$ generates any state corresponding to $P_R, P_{A_1}, \ldots, P_{A_a}$.

Note that the simulation of the state of the left party relies on only the current left share, whereas the simulation of the right and auxiliary parties can depend on all right shares generated so far.

(b) Whenever $\mathcal{A}$ issues a leakage query (P, Φ), it is given the simulated answer $\Phi(\widetilde{\mathsf{intl}}_{i,P}, \widetilde{\mathsf{evl}}_{i,P})$, or of the relevant sub-state $\widetilde{\mathsf{evl}}_{(i,j),P}$ in case there are erasures within the iteration. As in the real experiment, the total amount of leakage on each P in the iteration is smaller than $\ell(\lambda)$ bits.

Denote by $\widetilde{\mathsf{view}}^{\ell,\infty}_{\mathcal{S},\mathcal{A}}(1^\lambda, m, z)$ the view of $\mathcal{A}$ in the above experiment.

Definition 8 (Continual ℓ-Leakage-resilience with oblivious simulation)**.** *We say that an orthogonal share generation scheme* $(\mathsf{Comp}^{\mathsf{OG}}, \mathsf{OG})$ *is continually ℓ-leakage-resilient if the following two ensembles are indistinguishable.*

- $\{\mathsf{view}^{\ell,\infty}_{\mathcal{A}}(1^\lambda, m(\lambda), z)\}_{\lambda\in\mathbb{N}, z\in\{0,1\}^{\mathrm{poly}(n)}}$
- $\{\widetilde{\mathsf{view}}^{\ell,\infty}_{\mathcal{S},\mathcal{A}}(1^\lambda, m(\lambda), z)\}_{\lambda\in\mathbb{N} z\in\{0,1\}^{\mathrm{poly}(n)}}$.

Remark 3. *In our application below, we need to generate orthogonal vectors of different dimension. All of the above definitions naturally extend to this case, in which we will denote by* $\mathsf{Comp}^{\mathsf{OG}}(1^{m_1}, \ldots, 1^{m_k})$ *a procedure that can generate shares for any of the dimensions m_i, and assume that m_i is specified every time* OG *operates.*

C.2 An Orthogonal Share Generation Protocol via a Modifed GR Ciphertext Bank.

We now introduce an orthogonal share generation procedure based on the Goldwasser and Rothblum [GR12] ciphertext bank procedures. We start with a quick recap of their main constructs, focusing on the ones required for our purpose. We then discuss the security properties of their bank, and pinpoint the obstructions towards the strong simulation properties described in the previous subsection. We then describe an augmented version of the bank that bridges this gap.

C.2.1 The GR Ciphertext Bank - A Recap.

In the base of the GR OCL scheme is an inner product leakage-resilient encryption scheme. Where the key is a random (or high-entropy) vector $\mathsf{key} \in \mathbb{F}_2^\kappa$, and an encryption of a given plaintext $b \in \mathbb{F}_2$ consists of a random $\mathsf{c} \in \mathbb{F}_2^\kappa$ such that $\langle \mathsf{key}, \mathsf{c}\rangle = b$.

They then describe an "updatable" ciphertext bank that allows generating, in every iteration i, fresh key-ciphertext pairs $(\mathsf{key}_i, \mathsf{c}_i)$ that encrypt some predetermined $b \in \mathbb{F}_2$, all this under continual leakage on the generation and update procedure (below, we discuss what kind of leakage resilience is achieved). Eventually, the key-ciphertext pairs $(\mathsf{key}_i, \mathsf{c}_i)$ will essentially correspond to our orthogonal shares (L_i, R_i).

Before we proceed to describing GR, we first highlight a subtle, but important, difference between the OCL settings handled in GR and this work. As mentioned earlier in Section B, GR, as well as [MR04, GR10], considers the ordered OCL model, whereas this work, as well as [JV10, BCG+11, DF12], considers the model of LDS with unordered leakage; we detail below:

OCL with "ordered" leakage: In GR, an OCL computation consists of a sequence of sub-computations and a shared, secret, memory. An evaluation is carried out by performing the sub-computations sequentially, each of which reads from and writes to only a part of the memory. At any moment, the adversary can only leak from the state of the sub-computation that is currently "active". We say that in this model, the leakage is "ordered" according to the computation. Note that the memory keeps some intermediate states of the computation that will be loaded for a later subcomputation; however, the adversary cannot leak from them until this occurs.

OCL with "unordered" leakage: In our work, as well as in [JV10, DF12], an OCL computation consists of components that interact with each other, without access to a shared, secret, memory. At any moment, the adversary can leak from the current state of components separately, even those that are not currently "active". We say that in this model, the leakage is "unordered", irrespective to the order of computation. We also remark that since the components must keep the intermediate states of the computation themselves, which are subject to leakage at any point.

As we shall see later, when using the GR ciphertext bank to implement our orthogonal share generation protocol, this difference in the model will be one of the obstructions preventing us from using the GR ciphertext bank in a black-box way. By a closer examination of the GR analysis, we will see how to overcome this in a relatively modular way, towards proving the security of our orthogonal share generation protocol.

In what follows, when we describe GR and stating their claims, we stick to with their ordered OCL model. Later on, we augment the scheme, and in particular translate it to our unordered leakage setting.

Ciphertext Bank Procedures. We now recap the main procedures of the ciphertext bank and their implementation. Most of the text is taken verbatim from [GR12], excluding procedures that are not required in our context, and noting other simplifications that we shall allow.

The bank is initialized (without leakage) using a $\mathsf{BankInit}$ procedure that takes as input a bit $b \in \mathbb{F}_2$. It can then be accessed (repeatedly) using a $\mathsf{BankGen}$ procedure, which produces a key-ciphertext pair whose underlying plaintext is b. Between generations, the banks internal state is updated using a $\mathsf{BankUpdate}$ Procedure. Leakage from a sequence of $\mathsf{BankGen}$ and $\mathsf{BankUpdate}$ 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.

These functionalities are implemented as follows. The ciphertext bank consists of key and a collection C of 2κ ciphertexts. We view C as an $\kappa \times 2\kappa$ matrix, whose columns are the ciphertexts. In the $\mathsf{BankInit}$ procedure, on input b, key is drawn uniformly at random conditioned on $\mathsf{key}[1] = 1$ (this will be useful for key updates), and the columns of C are drawn uniformly at random such that their inner product with key is b. We denote by b as the ciphertext bank's underlying plaintext bit.

The $\mathsf{BankGen}$ procedure outputs a linear combination of C's columns. In GR, the linear combination is chosen uniformly at random such that 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 $\mathsf{PiecemealMM}$.

The $\mathsf{BankUpdate}$ procedure injects new entropy into key and into C. We refresh the key using a (piecemeal) key refresh procedure $\mathsf{PiecemealRefresh}$. We refresh C by multiplying it with random matrix $R \in \mathbb{F}_2^{2\kappa\times 2\kappa}$ whose columns all have parity 1 (again, in for our purpose, the parity restriction is not necessary). Matrix multiplication is again performed securely using $\mathsf{PiecemealMM}$.

Simplification we allow: For our purpose, we will be interested in the special case where the underlying plaintext b is always 0, namely, we will always sample truly orthogonal vectors. Because of this, the $\mathsf{BankGen}$ and $\mathsf{PiecemealRefresh}$ procedures can be simplified to use any random linear combintations and any random matrix R without the restriction of having parity 1.

Piecemeal matrix procedures. The piecemeal matrix procedures access matrices by dividing them into pieces, and separating the pieces between separate parties in the computation. Each piece is a collection of linear combinations of the matrix's columns; namely, a small subspace of the matrix's columns. We now overview the GR piecemeal procedures for matrix multiplication, and for refreshing the key under which the ciphertexts in a matrix's columns are encrypted. In all these procedures, no matrix is ever present in its entirety in the active memory.

The $\mathsf{PiecemealMM}$ procedure is given two matrices A, B, where say $A \in \mathbb{F}_2^{\kappa\times 2\kappa}$ and $B \in \mathbb{F}_2^{2\kappa\times 2\kappa}$ as above. A is divided to a sub-matrices $A_1, \ldots, A_a$, each with only $\ell = 2\kappa/a \ll \kappa$ columns, which are divided between different parties, and leaked on separately. Each column B_i of B is accordingly separated to a parts $B_{i,1}, \dots, B_{i,a}$, each of dimension ℓ. Then, the column C_i of the resulting matrix is aggregated by multiplying each A_j with $B_{i,j}$, the result $D_{i,j}$ is then carried on to the next computation where A_{j+1} is multiplied by $B_{i,j+1}$ and added $D_{i,j}$, and so on. (More generally, we can deal with several B-columns at a time, rather than a single one.)

The PiecemealRefresh procedure is given key and a ciphertext matrix A. It first refreshes the key to key' by adding a uniformly random vector σ conditioned on $\sigma[1] = 0$, so that the invariant $\mathsf{key}'[1] = 1$ is maintained. Then, given σ, the procedure resets each of A's columns A_j to maintain their inner product with the original key. This is done by adding $\langle A_j, \sigma \rangle$ to the first coordinate $A_j[1]$ of A_j. (Again, more generally, we deal with several A-columns at a time.)

The procedures are described in Figures 4,5,6 with the restriction that $b \equiv 0$, and without the parity 1 restrictions referred to above. The description still does not assign the different tasks to parties. This will be done later on, according to our concrete simulation requirements.

$\mathsf{BankInit}(1^\kappa)$: **initializes a ciphertext bank; No leakage.**

1. key is sampled uniformly at random from $\mathbb{F}_2^\kappa$, so that $\mathsf{key}[1] = 1$.
2. For $i \in [2\kappa], C[i]$ is a random vector orthogonal to key.
3. Output $\mathsf{Bank} = (\mathsf{key}, C)$.

$\mathsf{BankGen}(C)$: **generates a new ciphertext; Under leakage.**

1. Generate a random $r \in \mathbb{F}_2^\kappa$.
2. $\mathsf{c} \leftarrow \mathsf{PiecemealMM}(C, r)$.
3. Output c.

$\mathsf{BankUpdate}(\mathsf{Bank})$: **updates the bank between generations; Under leakage.**

1. Refresh the key: $(\mathsf{key}', D) \leftarrow \mathsf{PiecemealRefresh}(\mathsf{key}, C)$.
2. Refresh the ciphertexts: sample a random $R \in \mathbb{F}_2^{2\kappa \times 2\kappa}$, $C' \leftarrow \mathsf{PiecemealMM}(D, R)$.
3. $\mathsf{Bank}' = (\mathsf{key}', C')$.

Figure 4: The ciphertext bank procedures

The GR simulated ciphertext bank. Next, we recall the GR 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 $\mathbb{F}_2^\kappa$. Here, unlike in the real ciphertext bank, the plaintexts underlying C's columns are independent and uniformly random.

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. SimBankUpdate operates identically to BankUpdate.

The simulation procedures are in Figure 7.

```
PiecemealMM$(A, B)$: multiplies matrices $A \in \mathbb{F}_2^{\kappa \times m}$ and $B \in \mathbb{F}_2^{m \times n}$; Under leakage.
  1. Let $A = [A_1, \ldots, A_a]$, 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.
  2. For $i \in [b]$:
      (a) Set $D_0 = 0^{\kappa \times \ell}$.
      (b) For $j \in [a] : D_j = D_{j-1} + (A_j \times B_{i,j})$; leakage on each tuple $(D_{j-1}, A_j, B_{i,j})$ separately.
      (c) $C_i = D_a$.
  3. Output the product matrix $C = [C_1, \ldots, C_b]$.
```

Figure 5: Piecemeal matrix multiplication

```
PiecemealRefresh$(\mathsf{key}, A)$: refreshes the key for matrix $A \in \mathbb{F}_2^{\kappa \times m}$.
  1. Parse: $A = [A_1, \ldots, A_a]$, where each $A_i$ is a $\kappa \times \ell$ matrix.
  2. Sample a uniformly random $\sigma \in \mathbb{F}_2^{\kappa}$ such that $\sigma[1] = 0$. (leakage on $\sigma$)
  3. Set $\mathsf{key}' = \mathsf{key} + \sigma$. (leakage on $\mathsf{key}, \sigma$)
  4. For $j \in [a]$: $A_j' = \mathsf{CorrelateKey}(A_j, \sigma)$, where $A_j'$ is set to be the same as $A_j$ except the 1st row $A_j'[1] = A_j[1] + A_j^t \times \sigma$.
     (leakage on $(A_j, \sigma)$ for each $j \in [a]$ separately).
  5. Output $\mathsf{key}'$ and the refreshed matrix $A' = [A_1', \ldots, A_a']$.
```

Figure 6: Piecemeal matrix refresh

$\mathsf{SimBankInit}(1^\kappa)$**:**

1. key is sampled uniformly at random from $\mathbb{F}_2^\kappa$.
2. For $i \in [2\kappa], C[i]$ is a uniformly random vector.
3. Output $\mathsf{Bank} = (\mathsf{key}, C)$.

$\mathsf{SimBankGen}(C, b)$**:**

1. Generate a random $r \in \mathbb{F}_2^\kappa$ such that $\langle r, C^t \times \mathsf{key}\rangle = \langle C \times r, \mathsf{key}\rangle = b$.
2. $\mathsf{c} \leftarrow \mathsf{PiecemealMM}(C, r)$.
3. Output c.

$\mathsf{SimBankUpdate}(\mathsf{Bank})$**: identical to** $\mathsf{BankUpdate}(\mathsf{Bank})$**.**

1. Refresh the key: $(\mathsf{key}', D) \leftarrow \mathsf{PiecemealRefresh}(\mathsf{key}, C)$.
2. Refresh the ciphertexts: sample a random $R \in \mathbb{F}_2^{2\kappa \times 2\kappa}$, $C' \leftarrow \mathsf{PiecemealMM}(D, R)$.
3. $\mathsf{Bank}' = (\mathsf{key}', C')$.

Figure 7: The simulated ciphertext bank procedures

C.2.2 Overview of Our Orthogonal Share Generation Protocol

To build an orthogonal share generation scheme from the GR ciphertext bank, as described in the previous subsections, we need to bridge three gaps:

1. **Difference in models of computation:** As discussed above, the OCL setting in GR considers computations performed using a sequence of sub-computation with shared, secret, memory, whereas in our setting, computations are done by separate components without shared memory.

   Thus our first task is to design a protocol $\Lambda = (P_L, P_R, P_{A_1}, \ldots, P_{A_{2a}})$ that carries out the sub-computations, and stores the intermediate memory states of GR using different components. A naive one-to-one assingment, mapping every subcomputation to a separate component, would require a polynomial number of components. To maintain a constant number of components, we use a many-to-one assignment and apply erasure to "separate" different sub-computations assigned to one component. The invariant that we maintain is that with erasure, at any point, the state of any component consists of only the state of a single sub-computation, so that states of different sub-computations would never be leaked on jointly. We describe our protocol Λ in Section C.2.3.

2. **Ordered v.s. unordered leakage:** In GR, at any moment, the adversary can only leak from the sub-computation that is currently active. In our language, the adversary can only leak from the component that is currently computing. However, the model of leakage we handle is more stringent, and allows the adversary to leak from any component at anytime.

   Our second task is to show that unordered leakage does not break the security of GR scheme. In Section C.2.4, we transcribe the GR simulation procedures ($\mathsf{SimBankInit}(1^\kappa)$), ($\mathsf{SimBankGen}(C, b)$), and ($\mathsf{SimBankUpdate}(C)$) into a *monolitic* simulator $\mathcal{S}'$ of our protocol

Λ that simualtes the state of all parties in one shot (using both the left and right shares $\{L_i\}, \{R_i\}$). The GR analysis directly ensures to that the adversarial views in an honest execution of Λ and in simulation by $\mathcal{S}'$ are statistically close, *provided that the adversary performs only "ordered" leakage.*

Next, we show that this statistical closeness holds even under "unordered" leakage, that is, the simulator $\mathcal{S}'$ generates a statistically close adversarial view, *even if the adversary performs "unordered" leakage.* This was previously claimed, e.g. in [BCG+11], but without a proof. For the sake of completeness, we give here a rather crude, but relatively modular and simple, argument to why this is the case, at cost of getting slightly worse leakage bounds.

3. **Strong simulation:** Our final task is go beyond a monolithic simulator $\mathcal{S}'$, and obtain a strong simulator $\mathcal{S} = (\mathcal{S}_L, \mathcal{S}_{R,A})$ that has two special structures, which are crucial for eventually getting a *strong* OCL scheme.

   - The monolithic simulator $\mathcal{S}'$ entailed by GR simulation samples the orthogonal shares $\{(L_i, R_i)\}$ (or in GR terminology, $(\mathsf{c}_i, \mathsf{key}_i)$) internally. In contrast, we require the strong simulator $\mathcal{S}$ to receive these shares externally, and "reverse sample" the rest of the state consistently.
   - Moreover, the "reverse sampling" must be done in a way that uses $\{L_i\}$ (or $\{\mathsf{c}_i\}$) and $\{R_i\}$ (or $\{\mathsf{key}_i\}$) *separately*: $\mathcal{S}_L$ simulates the state of P_L and only gets the shares $\{L_i\}$ (or $\{\mathsf{c}_i\}$), while $\mathcal{S}_{R,A}$ simulats the state of $P_R, P_{A_1}, \dots, P_{A_{2a}}$ and only gets the shares $\{R_i\}$ (or $\{\mathsf{key}_i\}$).

     Note that since the left and right shares are never used together, they can be simulated to have arbitrary underlying plaintext bits $\{b_i\}$, by sampling $\{(L_i, R_i)\}$ (or $\{\mathsf{c}_i, \mathsf{key}_i\}$) with inner products $\{b_i\}$. Relying on the fact that inner products are resilient to bounded separate leakage, the view simulted by $\mathcal{S}_L, \mathcal{S}_{R,A}$ using these shares is statistically close to the view generated using orthognal shares. In other words, we can "program" these shares however we want, so long that their distribution withstands separate bounded leakage, which will be leveraged when proving the security of our strong OCL scheme.

   We describe and prove security of our strong simulator $\mathcal{S}$ in Section C.2.5.

We next move to provide details for each of the steps. In what follows we denote $\ell = 2\kappa/a$, and adopt the GR terminology.

C.2.3 Our Orthogonal Share Generation Protocol Λ

Our orthogonal share generation protocol $\Lambda = (\mathsf{Comp}^{\mathsf{OG}}, \mathsf{OG})$ with $\mathsf{OG} = (P_L, P_R, P_{A_1}, \dots, P_{A_{2a}})$ starts with preprocessing that runs $\mathsf{BankInit}$ and then proceeds in many iterations, where in each iteration, $\mathsf{BankUpdate}$ is executed first followed by $\mathsf{BankGen}$. Note that, the protocol uses $2a$ auxiliary components, twice the number of pieces in GR ciphertext banck. Let κ be the security parameter.

Preprocessing $\mathsf{Comp}^{\mathsf{OG}}$: Sample $(C_0, \mathsf{key}_0) \leftarrow \mathsf{BankInit}(1^\kappa)$ without leakage. P_R is given key_0, while the bank $C_0 = [C_{01}, \dots, C_{0,a}]$ is divided into a pieces and assigned to $P_{A_1}, \dots, P_{A_a}$ respectively. More precisely, the initial states of components are

$$\mathsf{init}_{1,L} = \mathsf{null}, \qquad \mathsf{init}_{1,R} = \mathsf{key}_0, \qquad \forall i \in [a],\ \mathsf{init}_{0,A_i} = C_{0,i}, \qquad \forall i \in [a],\ \mathsf{init}_{0,A_{a+i}} = \mathsf{null}$$

Evaluation protocol OG: In the t^{th} evaluation, an execution of $\mathsf{BankUpdate}$ is carried out first by P_R and $P_{A_1}, \ldots, P_{A_a}$, followed by an execution of $\mathsf{BankGen}$ by P_L. Note here, the update procedure is done before the actual share generation; as we will see in Section C.2.5, the order is important for ensuring that the protocol admits a strong simulator. More specifically:

(1) Executing $\mathsf{BankUpdate}(1^\kappa)$: P_R and $P_{A_1}, \cdots, P_{A_a}$ jointly perform $\mathsf{BankUpdate}$ as follows: To do $(\mathsf{key}_t, H) = \mathsf{PiecemealRefresh}(\mathsf{key}_{t-1}, C_{t-1})$, P_R samples a uniformly random σ_t so that $\sigma_t[1] = 0$, sets $\mathsf{key}_t = \mathsf{key}_{t-1} + \sigma_t$, sends σ_t to each P_{A_j}, and erases σ_t. Each P_{A_j} computes $H_j = \mathsf{CorrelateKey}(C_{t-1,j}, \sigma_t)$ as described in Figure 6 and erases $C_{t-1,j}$.

Note the seperation of sub-computations $\{H_j = \mathsf{CorrelateKey}(C_{t-1,j}, \sigma_t)\}$ in $\mathsf{PiecemealRefresh}$ *is ensured via the use of different parties and erasure.*

To compute the new bank $C' = [C'_1, \cdots, C'_a] = \mathsf{PiecemealMM}(H, B)$, $P_{A_1}, \cdots, P_{A_a}$ compute each piece C'_j for $j \in [a]$ in three steps.

1. P_{A_1} to P_{A_a} computes in order, where P_{A_i} upon receiving $D_{j,i-1}$ from $P_{A_{i-1}}$ (for P_{A_1}, $D_{j,0}$ is set to the all zero matrix) samples a random matrix $B_{j,i} \in \mathbb{F}_2^{\ell \times \ell}$, computes $D_{j,i} = H_i \times B_{j,i} + D_{j,i-1}$, and sends $D_{j,i}$ to $P_{A_{i+1}}$.
2. P_{A_a} sets the j^{th} piece of the new bank C'_j to $D_{j,a}$, which equals to $H \times B_j$, and sends it to store at $P_{A_{a+j}}$.
3. Each P_{A_j} erases $(D_{j,i}, B_{j,i})$ from its memory after its computation

Finally, after the new bank C' has been computed entirely and stored at $P_{A_{a+1}}, \ldots, P_{A_{2a}}$ piece by piece. Each party P_{A_j} erases the piece H_j first, loads the new piece C'_j from $P_{A_{a+j}}$, who then erases C'_j from its memory. The new bank C_t is C' and the random update matrix R_t in this iteration is B.

Note the seperation of sub-computations $\{D_{j,i} = D_{j,i-1} + H_i \times B_{j,i}\}$ in $\mathsf{PiecemealMM}$ *is done via erasure and the use of different parties. By storing the new bank pieces C'_j at $P_{A_{a+j}}$ before the old piece H_j is erased from P_{A_j}, the new bank pieces and the old bank pieces are never leaked together, except in P_{A_a}, which is the same as in GR.*

(2) Executing $\mathsf{BankGen}(1^\kappa)$: P_L performs $\mathsf{PiecemealMM}(C, r)$ in a iterations: In iteration i, it "loads" the new bank piece $C_{t,i} \in \mathbb{F}_2^{\kappa \times \ell}$ from P_{A_i}; it then samples a random $r_{t,i} \in \mathbb{F}_2^{\ell}$, and computes $D_i = D_{i-1} + C_{t,i} \times r_{t,i}$ (in the first iteration D_0 is set to the zero vector). P_L then erases everything, but D_i for the next iteration. At the end of all iterations, P_L obtains $\mathsf{c} = D_a = C_t \times r_t$.

Note the sub-computations $\{D_i = D_{i-1} + C_{t,i} \times r_{t,i}\}$ in $\mathsf{PiecemealMM}$ *is done sequentially in P_L and their separation is ensured via erasure.*

(3) Outputting: P_L outputs the left share $L_t = \mathsf{c}_t$, and P_R outputs the right share $R_t = \mathsf{key}_t$.

Theorem 5. *Let $\gamma(\kappa) = o(\kappa^{1/2})$, and $a = 20$. The orthogonal share generation protocol* $(\mathsf{Comp}^{\mathsf{OG}}, \mathsf{OG})$ *with parties $P_L, P_R, P_{A_1}, \cdots, P_{A_{2a}}$ is continual γ-leakage-resilient.*

Towards this theorem, we need to construct a strong simulator $\mathcal{S}$ as in Definition 8. As an intermediate step, below we first construct a monolithic simulator $\mathcal{S}'$ that does not have the structure of a strong simulator; instead, $\mathcal{S}'$ on input all the shares $\{L_t, R_t\}$ simulates the states of all components $P_L, P_R, P_{A_1}, \ldots, P_{A_{2a}}$. When discussing the security of this simulator $\mathcal{S}'$, we will consider both ordered and unordered leakage adversaries. Below we describe their behaviors when attacking the above protocol Λ.

Ordered leakage adversary: An ordered (leakage) adversary $\mathcal{A}^{ord}$ attacking Λ, at any moment, can only leak from the component that is currently "activated". As noted above already, the constrution of Λ uses erasure and the separation of parties to ensure that, at any moment, the current state of the activated component corresponds to the state of a sub-computation in the GR scheme. Thus, an ordered adversary $\mathcal{A}^{ord}$ obtains leakage in the same way as an adversary attacking the GR scheme does in [GR12].
Unordered leakage adversary: An unordered adversary $\mathcal{A}$ attacking Λ (as described in experiment $\mathsf{RealExp}^{\infty}_{\mathcal{A}}(1^\kappa, m, z)$), at any moment, can obtain separate leakage from the current state of all parties, even ones that are not activated. We note that, informally, in Λ, the currently activated party is equivalent to the current sub-computation in GR, while the parties that are not currently activated serves as the memory in GR. The protocol Λ makes sure that parties immediately erase all information that is no longer needed for later, and it keeps the invariant that, at any point, the state at every inactive party is the input of a single sub-computation to be performed later. Thus, an unordered adversary $\mathcal{A}$ can be viewed as an adversary that at any moment can obtains leakage from all sub-computations of GR, as long as the leakage on different sub-computations is separate.

C.2.4 A Monolithic Simulator of Λ

In this section, we describe a monolithic oblivious simualtor $\mathcal{S}'$ that simulates the state of all parties in each evaluation using the GR simulation procedures:

- In the first evaluation: $\mathcal{S}'$ calls $\mathsf{SimBankInit}(1^\kappa)$ to generated simulated initial bank and key $(\tilde{C}_0 = [\tilde{C}_{0,1}, \cdots, \tilde{C}_{0,a}], \tilde{\mathsf{key}}_0)$, and sets the initial states of all parties as

$$\mathsf{init}_{0,L} = \mathsf{init}_{0,P_{A_{a+1}}} = \mathsf{null}, \quad \mathsf{init}_R = \tilde{\mathsf{key}}_0, \quad \forall\, j \in [a],\ \mathsf{init}_{0,A_j} = \tilde{C}_{0,j},\ \mathsf{init}_{0,A_{a+j}} = \mathsf{null}$$

- In every evaluation $t \geq 1$: $\mathcal{S}'$ simulates the evaluation states $\mathsf{evl}_{t,R}, \mathsf{evl}_{t,A_1}, \cdots, \mathsf{evl}_{t,A_{2a}}$ of the right and auxiliary parties by calling $\mathsf{SimBankUpdate}(\tilde{C}_{t-1}, \tilde{\mathsf{key}}_{t-1})$ to simulate the sub-computations for updateing the key and bank, and assign the simulated state of each sub-computation to the appropriate party that performs it. This yields the simulated bank and key $(\tilde{C}_t, \tilde{\mathsf{key}}_t)$ for the next iteration; it then sets the initial states $\mathsf{init}_{t,L}, \mathsf{init}_{t,R}, \mathsf{init}_{t,A_1}, \cdots, \mathsf{init}_{t,A_a}$ of all parties for the next iteration using $(\tilde{C}_t, \tilde{\mathsf{key}}_t)$ as in the first iteration.

  Next, $\mathcal{S}'$ simulates the evaluation state $\mathsf{evl}_{t,L}$ of P_L by calling $\mathsf{SimBankGen}(\tilde{C}_t)$ to simulate the sub-computations for sampling a new ciphertext $\tilde{c}_t$. (Note that, $\mathcal{S}'$ split the simulated evaluation states of parties according to the erasure performed in Λ.)

Consider an ideal experiment $\mathsf{IdealExp}^{\infty}_{\mathcal{S}',\mathcal{A}}(1^\kappa, m, z)$ using the monolithic simulator, where the adversary obtains bounded independent leakage from the simulated state of each party, by $\mathcal{S}'$. Denote by $\widetilde{\mathsf{view}}^{\gamma,\infty}_{\mathcal{S}',\mathcal{A}}(1^\kappa, m, z)$ the view of an $\mathcal{A}$ in this experiment.

Since the monolithic simulator $\mathcal{S}'$ uses the GR simulation procedures exactly as in [GR12], and as discussed before, ordered leakage adversary attacking Λ obtains leakage in the same as in the ciphertext bank scheme, it follows from the GR analysis that the simulation by $\mathcal{S}'$ is indistinguishable to any ordered leakage adversary. Formally,

Lemma 4 (Security of Λ against ordered adversaries [GR12])**.** *There exists a function $\gamma(\kappa) = \Theta(\kappa)$, such that, for every* ordered *γ-leakage adversary $\mathcal{A}^{ord}$, the following ensembles are indistinguishable.*

- $\{\mathsf{view}^{\gamma,\infty}_{\mathcal{A}^{ord}}(1^\kappa, m(\kappa), z)\}_{\kappa\in\mathbb{N}, z\in\{0,1\}^{\mathrm{poly}(n)}}$

- $\{\widetilde{\mathsf{view}}^{\gamma,\infty}_{\mathcal{S}',\mathcal{A}^{ord}}(1^\kappa, m(\kappa), z)\}_{\kappa\in\mathbb{N}z\in\{0,1\}^{\mathrm{poly}(n)}}$.

Below we sketch the proof of Lemma 4 from [GR12] and then show how to extend their analysis to handle also unordered adversaries, at the price of reducing the leakage bound.

Lemma 5 (Security of Λ against unordered adversaries [GR12])**.** *There exists a function $\gamma(\kappa) = o(\kappa^{1/2})$, such that, for every (potentially unordered) γ-leakage adversary $\mathcal{A}$, the following ensembles are indistinguishable.*

- $\{\mathsf{view}^{\gamma,\infty}_{\mathcal{A}}(1^\kappa, m(\kappa), z)\}_{\kappa\in\mathbb{N},z\in\{0,1\}^{\mathrm{poly}(n)}}$
- $\{\widetilde{\mathsf{view}}^{\gamma,\infty}_{\mathcal{S}',\mathcal{A}}(1^\kappa, m(\kappa), z)\}_{\kappa\in\mathbb{N}z\in\{0,1\}^{\mathrm{poly}(n)}}$.

Proof Skech of Lemma 4 The GR analysis can be divided into two steps. In the first step, they fomulate piecemeal leakage attacks on matrices and vectors—we call them "piecemeal games"—which capture the leakage that can be computed via a leakage attack on the piecemeal matrix procedures (multiplication, refresh), and then show that random matrices are resilient to several flavors of such piecemeal attacks. In the second step, they show how to base the security of the GR simulation (and thus the security of $\mathcal{S}'$) on the resilience of random matrices to piecemeal leakage. We remark that, though not presented explicitly, their proof of the second step actually constructs a non-uniform black-box reduction that converts an ordered leakage adversary attacking the GR simulation (or an ordered adversary $\mathcal{A}^{ord}$ attacking simulation by $\mathcal{S}'$) to an adversary in the piecemeal matrix game; moreover the reduction satisfies certain special property that are crucial for adapting the analysis of [GR12] to handle unordered leakage adversaries.

Piecemeal Games:
Piecemeal leakage attack on a matrix is parameterized with a key $\mathsf{key} \in \{0,1\}^\kappa$ and matrix $M \in \{0,1\}^{\kappa\times m}$ with $m \geq \kappa$. An γ-leakage attacker $\mathcal{A}^{Mtrx}$ can obtain γ-bit leakage on key and on pieces $\{E_i\}$ of the matrix M separately, where each piece E_i is a small collection $E_i = M \times Lin_i$ of ℓ linear combinations of the columns of the matrix M (represented by $Lin_i \in \{0,1\}^{m\times\ell}$), with the restriction that leakage on $E_1, \ldots, E_L$ must occur in order (that is, after the adversary starts to leak on E_j it cannot go back to leak from any E_k for $k < j$). We call this game the *piecemeal matrix game.*

It was shown in [GR12] that when each piece is small enough $\ell \leq 0.1\kappa$ and the leakage bound is small enough $\gamma = 0.05\kappa/L$, a random matrix M is resilient to piecemeal leakage. That is, the leakage obtained by $\mathcal{A}^{Mtrx}$ in the above game is statistically close in the following settings: (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). This statistical closeness holds even if key is later revealed to $\mathcal{A}^{Mtrx}$ in its entirety. Furthermore, the statistical distances between the distrubtions of view of $\mathcal{A}^{Mtrx}$ in the above three cases are bounded by $\exp(-O(\kappa/L))$. (See Lemma 5.10 in [GR12] for more details.)

Piecemeal leakage attack on a matrix and vector extends the above attack and allow the adversary $\mathcal{A}^{Mtrx}$ to additionally leak from a vector $v \in \{0,1\}^\kappa$ jointly with each piece E_i of the matrix M. We call this game the *piecemeal matrix-vector game*

It was shown in [GR12] that when each piece is small enough $\ell \leq 0.1\kappa$ and the leakage bound is small enough $\gamma = 0.05\kappa/L^2$, for a matrix M with columns in the kernel of key, the leakage obtained by $\mathcal{A}^{Mtrx}$ in the above extended game is statistically close in the following two settings: (i) the vector v is in the kernel of key, and (ii) the vector v is not in the kernel of key. Moreover, this statistical closeness holds even if key is later exposed to $\mathcal{A}^{Mtrx}$ in its entirety (as above) and also

M is later exposed in its entirety. Again, the distance between the distrubutions of the view of $\mathcal{A}^{Mtrx}$ in the above two cases is bounded by $\exp(-O(\kappa/L))$. (See Lemma 5.15 in [GR12] for more details.)

Overall as long as the number of pieces that $\mathcal{A}^{Mtrx}$ can leak from is bounded $L = o(\kappa^{1/2})$, the statistical distances between the views of $\mathcal{A}^{Mtrx}$ in the piecemeal matrix game and piecemeal matrix-vector games with different settings are exponentially small. In this case, we say that the two games are secure.

Reduction to the security of piecemeal games: In the second step, the security of the GR simulation is reduced to the security of the two games. Though not presented explicitly, their proof actually establishes the exitence of a black-box reduction *Red* that converts an (ordered) adversary that distinguishes the GR simulation from the real ciphertext bank execution—or in our language an adversary $\mathcal{A}^{ord}$ that violates Lemma 4—to an adversary $\mathcal{A}^{Mtrx} = Red(\mathcal{A}^{ord})$ that violates the security of one of the piecemeal games. Furthermore, *Red* has the following special properties:

- *Red* externally participates in a piecemeal game, obtaining leakage from a *constant* number of matrix pieces $L = O(1)$; internally it emualtes the view of $\mathcal{A}^{ord}$ with leakage from the real execution of Λ or from the simulation by $\mathcal{S}'$ (depending on the setting of the piecemeal game it participates in).

- It emulates the view of $\mathcal{A}^{ord}$ *in a straightline and oblivious of the leakage queries from* $\mathcal{A}^{ord}$. More specifically, *Red* handles each query $q = (P, \Phi)$ from $\mathcal{A}^{ord}$ in the t^{th} evaluation in one of the two ways: Either, the state $(\mathsf{init}_{t,P}, \mathsf{evl}_{t,P})$ that q applies to is emualted by *Red* internally; then $\mathcal{A}^{ord}$ receives as the answer, the output of Φ evaluated on the simulated state. Or *Red* outputs a translation functions T, which translates q into a leakage query $q' = (q \circ T, E_i)$ on some matrix piece E_i in the external piecemeal game, and feeds the answer a it receives to $\mathcal{A}^{ord}$. We remark that this special property of *Red* will be crucial for showing that the security of GR extends to handling unordered adversaries.

When $L = O(1)$ and the leakage bound is $\gamma = 0.05\kappa/L^2 = \Theta(\kappa)$, the security of the external piecemeal games hold against *Red*. Thus, it follows that no ordered adversary $\mathcal{A}^{ord}$ can distinguish leakage obtained from the simulation by $\mathcal{S}'$, from that obtained in the honest execution of Λ. Thus Lemma 4 follows.

Proof Skech of Lemma 5 We show that the above analysis can be adapted to prove Lemma 5. Towards this, we argue first that the piecemeal games described above are secure even against unordered adversaries, and second the reduction *Red* converts an undordered adversary $\mathcal{A}$ distinguishing the simualtion by $\mathcal{S}'$ to an unordered adversary breaking the security of the piecemeal games. Then Lemma 5 follows.

Recall that an ordered adversary in the piecemeal game can only obtain leakage from the matrix pieces $\{E_i\}$ in order. In contrast, an unordered adversary does not respect the order and can leak from all pieces at any moment: That is, the γ-bit leakage from each piece E_i consists of γ 1-bit leakage queries, which can be interleaved in an arbitrary way with the 1-bit leakage queries from other pieces. We observe that the view of an *unordered* adversary with leakage from L pieces, can be emulated an *ordered* adversary with leakage from γL appropriate pieces (by simply duplicating the matrix pieces according to unordered leakage queries). Thus, by the security of the piecemeal games against ordered adversaries, we have that piecemeal games remain secure at the presence of unordered leakage attacks, as long as γL is bounded by $o(\kappa^{1/2})$.

Next, we show how to modify the reduction *Red* to work with unordered adversaries $\mathcal{A}$ (that distinguish simulation by $\mathcal{S}'$ from the honest execution of Λ). To see this, recall that *Red* is completely oblivious of the leakage queries from the adversary. Thus for each query q from an unordered adversary $\mathcal{A}$,

- if q leaks from a state $(\mathsf{init}_{t,P}, \mathsf{evl}_{t,P})$ that has not been previously leaked from, emulate an answer just as described above;
- if q leaks from a state $(\mathsf{init}_{t,P}, \mathsf{evl}_{t,P})$ that has been previously leaked from, do: If the previous leakage is answered using internally simulated state, then answer the current query by evaluating on the same simualted state again; otherwise, if the previous leakage is answered using translation T and leakage from E_i in the external piecemeal game, simply issue an out-of-order leakage $q \circ T$ to E_i, and feed the answer a to $\mathcal{A}$.

It is easy to see that the oblivious emulation property of *Red* allows for handling unordered leakage queries from $\mathcal{A}$ with the help of obtaining unordered leakage from the external piecemeal games. Furthermore, the external game remains the same with $L = O(1)$ matrix pieces and a bound of γ on the total length of leakage from each piece, except that now unordered leakage is allowed.

When $L = O(1)$ and $\gamma L = o(\kappa^{1/2})$, that is $\gamma = o(\kappa^{1/2})$, the security of the external piecemeal games hold against *Red* (making unordered queries). Thus it follows that the simulation by $\mathcal{S}'$ is indistinguishable from a real world execution of Λ even to unordered γ-leakage adversaries. Lemma 5 follows.

C.2.5 A Strong Simulator of Λ

In this final step, we describe a strong simulator $\mathcal{S}$ that can simulate the same distribution of the states of parties as $\mathcal{S}'$ does. Recall that $\mathcal{S} = (\mathcal{S}_L, \mathcal{S}_{R,A})$ must observe that

- $\mathcal{S}_L(1^\kappa, m, i; \mathbf{w}_i, L_i)$ generates any state corresponding to P_L.
- $\mathcal{S}_{R,A}(1^\kappa, m, i; \mathbf{w}_i, \mathbf{R}_i)$ generates any state corresponding to $P_R, P_{A_1}, \ldots, P_{A_a}$.

We now describe the simulation procedures:

Simulate the state of P_R: In each iteration t, the state of P_R consists of $\widetilde{\mathsf{intl}}_{t,R} = \tilde{\mathsf{key}}_{t-1}$ and $\widetilde{\mathsf{evl}}_{t,R} = (\sigma_t, \tilde{\mathsf{key}}_t)$. To bias the output right share in this iteartion to R_t, $\mathcal{S}_R$ with input $\mathbf{R}_t$ sets $\tilde{\mathsf{key}}_t$ to R_t, and computes $\sigma_t = \tilde{\mathsf{key}}_t - \tilde{\mathsf{key}}_{t-1} = R_t - R_{t-1}$. In the first iteration, $\tilde{\mathsf{key}}_0$ is set to a random string such that $\tilde{\mathsf{key}}_0[0] = 1$.

Simulate the state of $P_{A_1}, \cdots, P_{A_{2a}}$: In each iteration t, the intitial states of $P_{A_1}, \cdots, P_{A_a}$ are the previous bank pieces, $\mathsf{init}_{t,j} = C_{t-1,j}$; in the first iteration, C_0 is a ramdom matrix. (The initial states of the latter a auxiliary parties are empty.) The joint evaluation states $(\mathsf{evl}_{t,A_1}, \cdots, \mathsf{evl}_{t,A_{2a}})$ of $P_{A_1}, \cdots, P_{A_2}a$ contain the states related to the computation of $H = \mathsf{PiecemealRefresh}(C_{t-1}, \sigma_t)$ and that related with $C_t = \mathsf{PiecemealMM}(H, R_t)$. To simulate these states $\mathcal{S}_A$ with input $\mathbf{R}_t$ compute σ_t as $\mathcal{S}_R$ does, and then performs the two computation honestly; the states of the sub-computations are then assigned to $(\mathsf{evl}_{t,A_1}, \cdots, \mathsf{evl}_{t,A_{2a}})$ appropriately.

Simulate the state of P_L: In each iteration t, the state of P_L consists of initial state $\widetilde{\mathsf{intl}}_{t,L} = \mathsf{null}$ and evaluation state $\widetilde{\mathsf{evl}}_{t,L}$ for computing the new ciphertext $\tilde{c}_t$. To bias $\tilde{c}_t$ to the new left share L_t, $\mathcal{S}_L$ with input L_t, samples at random a linear combination $\tilde{Lin}_t$ such that $\tilde{C}_t \times \tilde{Lin}_t = L_t$. Then it performs the computation $\tilde{c}_t = \mathsf{PiecemealMM}(\tilde{C}_t, \tilde{Lin}_t)$ honestly, and set $\mathsf{evl}_{t,L}$ accordingly.

Concluding Theorem 5: By examining the construction of the monolitic simulator $\mathcal{S}'$ and GR simulation procedures $\mathsf{SimBankInit}$, $\mathsf{SimBankGen}$, and $\mathsf{SimBankUpdate}$, the strong simulator $\mathcal{S}$ indeed produces the same distribution of the simulated states of all parties as $\mathcal{S}'$ does. Thus it follows directly from Lemma 5 that when leakage bound is bounded by $\gamma = o(\kappa^{1/2})$ and $a = 20$ such that $\ell = 2\kappa/a = 0.1\kappa$, we have that no adversary can distinguish leakage obtained from the simulation by $\mathcal{S}$ or that obtained from the honest execution of Λ. Furthemore, $\mathcal{S}$ has the right structure as required in Definition 8. Therefore we conclude that Λ is indeed continually γ-leakage resilient.

C.3 The Full Scheme

Next, we construct a strong continual OCL scheme, based on the Dziembowski-Faust [DF12] OCL scheme and the orthogonal share generation scheme presented in the previous subsection.

Overview of the scheme. At high level, the DF scheme follows the classic GMW [GMW87] paradigm for semi-honest two-party computation: the parties (P_L, P_R) secret share the secret key k, and then, to compute $f(k)$, evaluate the circuit $U(\cdot, f)$ gate by gate over the shares. To guarantee leakage-resilience, the DF scheme relies on *a leakage-resilient secret sharing scheme* - the inner product two-source extractor.

We now overview how computation over the inner product shares is done. For simplicity, we describe a simple instance of the DF scheme over the binary field $\mathbb{F}_2$. We assume that each share is of length m (we later describe how this length is related to the amount of leakage that can be tolerated). Throughout, we denote by $\mathcal{O}^m_v$ the uniform distribution on $(L, R) \in \mathbb{F}_2^m \times \mathbb{F}_2^m$ conditioned on $\langle L, R\rangle = v$.

At the onset of the t^{th} iteration, the parties share each bit k_i of the key k; namely, P_L holds L_{k_i}, and P_R holds R_{k_i}. These shares are refreshed in every iteration (as will be described later), with the invariant that $\langle L_i, R_i\rangle = k_i$. Then, when P_L obtains the current input $f = f_t$, it computes $(L_{f_i}, R_{f_i}) \leftarrow \mathcal{O}^m_{f_i}$, and sends $\mathbf{R}_f = (R_{f_i})_{i\in[m]}$ to P_R. The parties then homomoprhically compute the boolean circuit $U(\cdot,\cdot)$ on the underlying input (k, f).

We assume that the circuit consists of NAND gates that are performed one at a time (or one level at a time). Concretely, for a NAND gate with input wires i, j and output wire k, the parties initially hold the shares $(L_i, R_i), (L_j, R_j)$ for the input wires, and would like to compute shares (L_k, R_k) for the output wire, where

$$\langle L_k, R_k\rangle = 1 + \langle L_i, R_i\rangle \cdot \langle L_j, R_j\rangle \ ,$$

and all operations are over $\mathbb{F}_2$. Thus, it is enough to be able to perform multiplication and addition of the constant 1 over shares. We next describe how this is done.

Given (L_i, R_i), (L_j, R_j), encoding v_i and v_j, the parties first locally compute the tensor product of their shares $L_{\otimes,k} = L_i \otimes L_j$, and $R_{\otimes,k} = R_i \otimes R_j$. While $\langle L_{\otimes,k}, R_{\otimes,k}\rangle = v_i \cdot v_j$, the size of $L_{\otimes,k}, R_{\otimes,k}$ is now m^2 each, and thus we cannot go on multiplying shares in this manner, without incurring exponential blowup. To shrink back the size of shares, the parties first run a refresh procedure that rerandomizes their shares, at the end of which they posses refreshed shares $(L^{\mathsf{fr}}_{\otimes,k}, R^{\mathsf{fr}}_{\otimes,k})$. Then, P_R sends the last $m^2 - m + 1$ entries $R^{\mathsf{fr}}_k[m, \ldots, m^2]$ of its refreshed share $R^{\mathsf{fr}}_{\otimes,k}$. (Refreshing is indeed necessary, since the non-refreshed $R_{\otimes,k}[m, \ldots, m^2]$, combined with even one bit of leakage may reveal for example the entire share R_i). Then, P_L computes

$$v := \langle L^{\mathsf{fr}}_{\otimes,k}[m, \ldots, m^2], R^{\mathsf{fr}}_{\otimes,k}[m, \ldots, m^2]\rangle \ .$$

Noting that at this point $(L^{\mathsf{fr}}_{\otimes,k}[1,\ldots,m-1], R^{\mathsf{fr}}_{\otimes,k}[1,\ldots,m-1])$ encode $v_i \dot{v}_j - v$, the parties now only need to add $v+1$. Thus P_L sets its share L_k to $(1+v|L^{\mathsf{fr}}_{\otimes,k}[1,\ldots,m-1])$, and P_R sets its share R_k to $(1|R^{\mathsf{fr}}_{\otimes,k}[1,\ldots,m-1])$. At the end of this protocol, the parties hold new shares of the right length m, encoding the right value $v_k = 1 + v_i \cdot v_j$.

At the end of the evaluation process, the parties hold vectors of shares $(\mathbf{L}_{f(k)}, \mathbf{R}_{f(k)})$ encoding the output result. At this point, P_L sends his vector of shares $\mathbf{L}_{f(k)}$ to P_R who produces the output. Finally, towards the next iteration, the parties refresh their shares $(\mathbf{R}_k, \mathbf{R}_k)$ of the secret key k.

Refreshing based on orthogonal share generation. We now describe the DF refresh procedure $\mathsf{Refresh}(L,R)$, when instantiated with the an orthogonal share generation scheme $(\mathsf{Comp}^{\mathsf{OG}}, \mathsf{OG})$ described in the previous subsection. Given two arbitrary shares $(L,R) \in \mathbb{F}_2^\ell \times \mathbb{F}_2^\ell$, the $\mathsf{Refresh}$ procedure outputs new shares $(L^{\mathsf{fr}}, R^{\mathsf{fr}}) \in \mathbb{F}_2^\ell \times \mathbb{F}_2^\ell$ such that $\langle L^{\mathsf{fr}}, R^{\mathsf{fr}} \rangle = \langle L, R \rangle$. For the sake of each refresh operation, two uniformly random orthogonal shares $((L'|L''),(R'|R'')) \leftarrow \mathcal{O}_0^{2\ell}$ should be generated and split between P_L and P_R. In the original DF scheme, these are sampled using designated **leakage-free** hardware. Here we shall use, for this purpose, the GR-based orthogonal generation scheme $(\mathsf{Comp}^{\mathsf{OG}}, \mathsf{OG})$. The reliance of leakage-free hardware is replaced with **leaky** auxiliary parties $P_{A_1}, \ldots, P_{A_a}$, for some constant a.

When executing $\mathsf{Refresh}(L,R)$, P_L first samples an invertible matrix $M' \in \mathbb{F}_2^{\ell \times \ell}$ such that $M'L = L'$, and sends M' to P_R. P_R then sets $R^{\mathsf{fr}} = R + M'^t R'$. Then, symmetrically, P_R samples an invertible matrix $M'' \in \mathbb{F}_2^{\ell \times \ell}$ such that $M'' R^{\mathsf{fr}} = R''$, and sends M'' to P_L. P_L then sets $L^{\mathsf{fr}} = L + M''^t L''$. (For simplicitly of exposition, assume for now that in the above neither L, L', R', R'' are zero.) Correctness follows by linearity:

$$\begin{aligned}
\langle L^{\mathsf{fr}}, R^{\mathsf{fr}} \rangle = \langle L, R^{\mathsf{fr}} \rangle + \langle M''^t L'', R^{\mathsf{fr}} \rangle &= \\
\langle L, R^{\mathsf{fr}} \rangle + \langle L'', M'' R^{\mathsf{fr}} \rangle &= \\
\langle L, R^{\mathsf{fr}} \rangle + \langle L'', R'' \rangle &= \\
\langle L, R \rangle + \langle L, M'^t R' \rangle + \langle L'', R'' \rangle &= \\
\langle L, R \rangle + \langle M' L, R' \rangle + \langle L'', R'' \rangle &= \\
\langle L, R \rangle + \langle L', R' \rangle + \langle L'', R'' \rangle &= \\
\langle L, R \rangle + \langle (L'|L''), (R'|R'') \rangle &= \\
\langle L, R \rangle + 0 \ .
\end{aligned}$$

As explained in [DF12], the refresh procedure guarantees that $(L^{\mathsf{fr}}, R^{\mathsf{fr}})$ are independent of (L,R) upto their similar inner product. Moreover, a simulator can "reprogram" $((L'|L''),(R'|R''))$ so that $(L^{\mathsf{fr}}, R^{\mathsf{fr}})$ will be distributed according to $\mathcal{O}_b^{2m^2}$, for any $b \in \mathbb{F}_2$ of its choice, in a way that is indistinguishable under leakage. Our proof of strong OCL, will rely in an essential way on the strong simulation guarantee for the orthogonal share generation procedure, where the simulator is guaranteed to work, even given "externally programed shares".

The scheme in detail. We now provide a more detailed description of the scheme and each of its components, and show that it satisfies the properties required by a strong OCL scheme. The full scheme is given in Figure 8 below, where $U(\cdot,\cdot)$ denotes the universal circuit, and is assumed to have T NAND gates labeled $G = \{1,\ldots,T\}$, among them n input gates labeled $I = \{1,\ldots,n\}$, and w output gates labeled $O = \{T-w+1,\ldots,T\}$. For a set G we often denote the collection of shares $(L_g)_{g \in G}$ as $\mathbf{L}_G$; we often treat strings $k \in \{0,1\}^n$ as their corresponding bit set $\{k_1,\ldots,k_n\}$. We denote by $\mathcal{O}_b^\ell$ the distribution on vector pairs in $\mathbb{F}_2^\ell \times \mathbb{F}_2^\ell$ whose inner product is b.

The scheme proceeds in iterations, we often denote the t^{th} iteration by superscript t. Within, every iteration, we invoke multiple iterations of a (continual) orthogonal share generation. We denote by superscript $\mathsf{OG}(t,i)$ the i^{th} orthogonal share generation, within the t^{th} iteration of the main protocol. We denote by $\mathsf{OG}(t)$ the sequence of all orthogonal share generation, within the t^{th} iteration of the main protocol.

Correctness. The correctness of the scheme follows readily, as outlined in the above overview. The only thing to note is that if during refreshing any of $L, L', R^{\mathsf{fr}}, R''$ are zero, there might not exist invertible matrices M', M'' as required and the parties would have to abort. However, it is not hard to see that all of these shares throughout the execution have very high min-entropy, and thus the above event occurs with neglgible probability. (It is also possible to augment the scheme so that the above event never occurs, by repeated sampling when necessary. For simplicitly of exposition, we describe and analyze the scheme without making this change.)

We now go on to give a proof sketch the security analysis.

Oblivious simulation. We show that Λ has a strong oblivious simulator $\mathcal{S} = (\mathcal{S}_R, \mathcal{S}_L, \mathcal{S}_A)$.

Recall that we need to describe how, in the t^{th} iteration, $\mathcal{S}_L$, given $(f, f(k)) = (f_t, f_t(k))$ and randomness $\mathbf{w}_i$, simulates

$$\begin{aligned}&\{\mathsf{init}_{t,L}, \mathsf{evl}_{t,L}\} = \\ &\{\mathbf{L}_k^t, \mathsf{init}_L^{\mathsf{OG}(t,1)}, f, \mathbf{L}_f, \mathbf{R}_f\mathbf{R}^{\mathsf{fr}}_{\otimes,z(O)}[m,\ldots,m^2], \mathbf{L}', \mathbf{L}'', \mathbf{M}', \mathbf{M}'', \mathsf{init}_L^{\mathsf{OG}(t)}, \mathsf{evl}_L^{\mathsf{OG}(t)}, \mathsf{init}_{t+1,L}\} \ ,\end{aligned}$$

how $\mathcal{S}_R$, given only $f(k)$ and randomness $\mathbf{w}_i$, simulates

$$\begin{aligned}&\{\mathsf{init}_R, \mathsf{evl}_R\} = \\ &\{\mathbf{R}_k^t, \mathsf{init}_R^{\mathsf{OG}(t,1)}, f(k), \mathbf{R}_f, \mathbf{R}', \mathbf{R}'', \mathbf{M}', \mathbf{M}'', \mathbf{L}_{z(O)}, \mathsf{init}_R^{\mathsf{OG}(t)}, \mathsf{evl}_R^{\mathsf{OG}(t)}, \mathsf{init}_{t+1,R}\} \ ,\end{aligned}$$

and how $\mathcal{S}_A$, given only the randomness $\mathbf{w}_i$, simulates

$$\begin{aligned}&\{\mathsf{init}_{A_i}, \mathsf{evl}_{A_i}\}_{i\in[a]} = \\ &\{\mathsf{init}_{A_i}^{\mathsf{OG}(t)}, \mathsf{evl}_{A_i}^{\mathsf{OG}(t)}, \mathsf{init}_{t+1,A_i}\}_{i\in[a]} \ .\end{aligned}$$

The simulator $\mathcal{S}$ will invoke the strong simulator $\mathcal{S}^{\mathsf{OG}} = (\mathcal{S}_L^{\mathsf{OG}}, \mathcal{S}_{R,A}^{\mathsf{OG}})$ for the orthogonal share generation scheme. We start by describing how the joint randomness w_t is sampled towards simulation of the t^{th} iteration.

- $(\widetilde{\mathbf{L}}_k, \widetilde{\mathbf{R}}_k) = (\widetilde{\mathbf{L}}_k^t, \widetilde{\mathbf{R}}_k^t)$ are sampled so that $\widetilde{L}_{k_i}, \widetilde{R}_{k_i} \in \mathbb{F}_2^m$ are independently uniformly random.
- $\widetilde{\mathbf{R}}_f$ is sampled so that $\widetilde{R}_{f_i} \in \mathbb{F}_2^m$ is uniformly random.
- $\widetilde{\mathbf{R}}^{\mathsf{fr}}_{\otimes,z(G)}$ is sampled so that, for each $g \in G$, $\widetilde{R}^{\mathsf{fr}}_{\otimes,z(g)} \in \mathbb{F}_2^{m^2}$ is uniformly random.
- $\widetilde{\mathbf{L}}^{\mathsf{fr}}_{\otimes,z(G\setminus O)}$ is sampled so that, for each $g \in G \setminus O$, $\widetilde{L}^{\mathsf{fr}}_{\otimes,z(g)} \in \mathbb{F}_2^{m^2}$ is uniformly random.
- $\widetilde{\mathbf{M}}', \widetilde{\mathbf{M}}''$ are sampled so that
  - When refreshing the output of each gate $g \in G$, the corresponding matrices $\widetilde{M}'_g, \widetilde{M}''_g \in \mathbb{F}_2^{m^2\times m^2}$ are uniformly random invertible matrices.

OCL scheme $\Lambda = (\mathsf{Comp}, P_L, P_R, P_{A_1}, \ldots, P_{A_a})$

Initial Compilation by Comp**:**

- For each bit k_i of the secret $k \in \{0,1\}^n$, sample $(L^1_{k_i}, R^1_{k_i}) \leftarrow \mathcal{O}^m_{k_i}$.
- Invoke $\mathsf{Comp}^{\mathsf{OG}}(1^{2n}, 1^{2m^2})$ with parties $P_L, P_R, P_{A_1}, \ldots, P_{A_a}$.

Initial state: $\mathsf{init}_L = (\mathbf{L}_k, \mathsf{init}^{\mathsf{OG}(1,1)}_L)$, $\quad \mathsf{init}_R = (\mathbf{R}_k, \mathsf{init}^{\mathsf{OG}(1,1)}_R)$, $\quad \mathsf{init}_{A_i} = \mathsf{init}^{\mathsf{OG}(1,1)}_{A_i}$.

Evaluation during the t^{th} iteration by $P_L, P_R, P_{A_1}, \ldots, P_{A_a}$**:**

- Given input $f = f_t$, P_L computes: $\{(L_{f_i}, R_{f_i}) \leftarrow \mathcal{O}^m_{f_i}\}_{i \in [|f|]}$, and sends $\mathbf{R}_f$ to P_R.
- For every gate $g \in G$, with input wires $x = x(g), y = y(g)$ and output wire $z = z(g)$, and input shares (L_x, L_y, R_x, R_y), each in $\mathbb{F}^m_2$, the parties run a homomorphic NAND procedure:

$$(L_z, R_z) \leftarrow \mathsf{HNand}(L_x, L_y, R_x, R_y) \ .$$

- Having computed the shares $(\mathbf{L}_{z(O)}, \mathbf{R}_{z(O)}) = \{(L_{z(g)}, R_{z(g)})\}_{g \in O}$ for the output wires, P_L sends $\mathbf{L}_{z(O)}$ to P_R, who outputs $\langle \mathbf{L}_{z(O)}, \mathbf{R}_{z(O)} \rangle = \{\langle L_{z(g)}, R_{z(g)} \rangle\}_{g \in O}$.

State update towards iteration $t+1$:

- The parties refresh their shares $(\mathbf{L}^t_k, \mathbf{R}^t_k)$ of the secret key k:

$$(\mathbf{L}^{t+1}_k, \mathbf{R}^{t+1}_k) \leftarrow \mathsf{Refresh}(\mathbf{L}^t_k, \mathbf{R}^t_k) \ .$$

- Previous state is erased, except for the initial state $\mathsf{init}_{t+1,L}, \mathsf{init}_{t+1,R}, \mathsf{init}_{t+1,A_1}, \ldots, \mathsf{init}_{t+1,A_a}$ for iteration $t+1$, including $(\mathbf{L}^{t+1}_k, \mathbf{R}^{t+1}_k)$, and $\mathsf{init}^{\mathsf{OG}(t+1,1)}_L, \mathsf{init}^{\mathsf{OG}(t+1,1)}_R, \mathsf{init}^{\mathsf{OG}(t+1,1)}_{A_1}, \ldots, \mathsf{init}^{\mathsf{OG}(t+1,1)}_{A_a}$.

Procedure $\mathsf{HNand}(L_x, L_y, R_x, R_y)$**:**

- Each party locally computes $L_{\otimes,z} = L_x \otimes L_y \in \mathbb{F}^{m^2}_2$, $R_{\otimes,z} = R_x \otimes R_y \in \mathbb{F}^{m^2}_2$.
- The parties then run a refresh procedure:

$$(L^{\mathsf{fr}}_{\otimes,z}, R^{\mathsf{fr}}_{\otimes,z}) \leftarrow \mathsf{Refresh}(L_{\otimes,z}, R_{\otimes,z}) \ .$$

- P_R sends $R^{\mathsf{fr}}_{\otimes,z}[m, \ldots, m^2]$ to P_L, and the parties set their shares to:

$$R_z = (1 | R^{\mathsf{fr}}_{\otimes,z}[1, \ldots, m-1]), \qquad L_z = (1 + v | L^{\mathsf{fr}}_{\otimes,z}[1, \ldots, m-1]) \ ,$$

  where $v = \langle L^{\mathsf{fr}}_{\otimes,z}[m, \ldots, m^2], R^{\mathsf{fr}}_{\otimes,z}[m, \ldots, m^2] \rangle$.

Procedure $\mathsf{Refresh}(L, R)$**:**

- For $\ell = \dim(L) = \dim(R)$, $P_L, P_R, P_{A_1}, \ldots, P_{A_a}$ generate orthogonal vector shares for (P_L, P_R):

$$\mathbb{F}^{2\ell}_2 \times \mathbb{F}^{2\ell}_2 \ni ((L', L''), (R', R'')) \leftarrow \mathsf{OG} \ .$$

- P_L samples a random invertible matrix $M' \in \mathbb{F}^{\ell \times \ell}_2$ such that $M'L = L'$.
- M' is sent to P_R who sets $R^{\mathsf{fr}} = R + M'^t R'$.
- P_R samples a random invertible matrix $M'' \in \mathbb{F}^{m^2 \times m^2}_2$ such that $M'' R^{\mathsf{fr}} = R''$.
- M'' is sent to P_L who sets $L^{\mathsf{fr}} = L + M''^t L''$.

Overall evaluation state:

- $\mathsf{evl}_{t,L} = f, \mathbf{L}_f, \mathbf{R}_f \mathbf{R}^{\mathsf{fr}}_{\otimes,z(G)}[m, \ldots, m^2], \mathbf{L}', \mathbf{L}'', \mathbf{M}', \mathbf{M}'', \mathsf{init}^{\mathsf{OG}(t)}_L, \mathsf{evl}^{\mathsf{OG}(t)}_L, \mathsf{init}_{t+1,L}$.
- $\mathsf{evl}_{t,R} = f(k), \mathbf{R}_f, \mathbf{R}', \mathbf{R}'', \mathbf{M}', \mathbf{M}'', \mathbf{L}_{z(O)}, \mathsf{init}^{\mathsf{OG}(t)}_R, \mathsf{evl}^{\mathsf{OG}(t)}_R, \mathsf{init}_{t+1,R}$.
- $\mathsf{evl}_{t,A_i} = \mathsf{init}^{\mathsf{OG}(t)}_{A_i}, \mathsf{evl}^{\mathsf{OG}(t)}_{A_i}, \mathsf{init}_{t+1,A_i}$.

Figure 8: A strong OCL scheme

- When refreshing shares corresponding to the i^{th} bit of the secret key k, during the update phase, the corresponding matrices $\widetilde{M}'_{k_i}, \widetilde{M}''_{k_i} \in \mathbb{F}_2^{m \times m}$ are also uniformly random invertible matrices.

- $\widetilde{\mathbf{R}}_{\otimes,z(G)}, \widetilde{\mathbf{R}}', \widetilde{\mathbf{R}}''$ are then set consistently and deterministically. That is,
  - For every gate $g \in G$, with input wires $x = x(g), y = y(g)$, and output wire $z = z(g)$, first figure out the pre-refreshed share $\widetilde{R}_{\otimes,z} \in \mathbb{F}_2^{m^2}$. If $g \in I$ is an input gate, this can be figured out from the values $\widetilde{\mathbf{R}}_k, \widetilde{\mathbf{R}}_f$. If $g \in G \setminus I$, this can be figured out from $\widetilde{R}^{\mathsf{fr}}_{\otimes,x}$, which determines $\widetilde{R}_x = (1|\widetilde{R}^{\mathsf{fr}}_{\otimes,x}[1,\ldots,m-1])$, and from $\widetilde{R}^{\mathsf{fr}}_{\otimes,y}$, which determines $\widetilde{R}_y$ in the same way. Thus, $\widetilde{R}_{\otimes,z} = \widetilde{R}_x \times \widetilde{R}_y$ is also determined.
  - When refreshing the output of each gate $g \in G$,
  $$\widetilde{R}'_g = (\widetilde{M}'^t_g)^{-1}(\widetilde{R}^{\mathsf{fr}}_{\otimes,z} - \widetilde{R}_{\otimes,z}) \qquad\qquad \widetilde{R}''_g = \widetilde{M}''_g \widetilde{R}^{\mathsf{fr}}_{\otimes,z} \ .$$
  - When refreshing shares corresponding to the i^{th} bit of the secret key k,
  $$\widetilde{R}'_{k_i} = (\widetilde{M}'^t_{k_i})^{-1}(\widetilde{R}^{t+1}_{k_i} - \widetilde{R}^t_{k_i}) \qquad\qquad \widetilde{R}''_{k_i} = \widetilde{M}''_{k_i} \widetilde{R}^{t+1}_{k_i} \ .$$

- $\mathsf{init}^{\mathsf{OG}(t)}_R, \mathsf{evl}^{\mathsf{OG}(t)}_R, \{\mathsf{init}^{\mathsf{OG}(t)}_{A_i}, \mathsf{evl}^{\mathsf{OG}(t)}_{A_i}\}_{i\in[a]}$ are all sampled by invoking $\mathcal{S}^{\mathsf{OG}}_{R,A}$ with input $\widetilde{\mathbf{R}}'_t, \widetilde{\mathbf{R}}''_t$, which include all the vectors $\widetilde{\mathbf{R}}', \widetilde{\mathbf{R}}''$ produced in iterations $1,\ldots,t$.

We next describe how each one of the simulators work.

Simulator $\mathcal{S}_R$: All the values that simulator $\mathcal{S}_R$ is required to simulate appear in the joint state $\mathbf{w}_t$, except for the left shares $\widetilde{\mathbf{L}}_{z(O)}$ corresponding to the output gates.

To simulate the above:

1. $\mathcal{S}_R$ first simulates $\widetilde{\mathbf{L}}^{\mathsf{fr}}_{\otimes,z(O)}$ as follows. For every gate $g \in O$, with input wires $x = x(g), y = y(g)$, and output wire $z = z(g)$, let $f_z(k)$ denote the value of this output gate. $\widetilde{L}^{\mathsf{fr}}_{\otimes,z}$ is sampled uniformly at random conditioned on $\langle \widetilde{L}^{\mathsf{fr}}_{\otimes,z}, \widetilde{R}^{\mathsf{fr}}_{\otimes,z}\rangle = f_z(k)$. (Recall that $\mathcal{S}_R$ gets the output $f(k)$.)

2. Then, $\widetilde{L}_{z(g)}$ can be computed from $(\widetilde{L}^{\mathsf{fr}}_{\otimes,z}, \widetilde{R}^{\mathsf{fr}}_{\otimes,z})$. Recall that it is simply $(1+v|\widetilde{L}^{\mathsf{fr}}_{\otimes,z}[1,\ldots,m-1])$, where $v = \langle \widetilde{L}^{\mathsf{fr}}_{\otimes,z}[m,\ldots,m^2], \widetilde{R}^{\mathsf{fr}}_{\otimes,z}[m,\ldots,m^2]\rangle$.

Simulator $\mathcal{S}_A$: The values that simulator $\mathcal{S}_A$ needs to simulate all appear in the joint state $\mathbf{w}_t$.

Simulator $\mathcal{S}_L$: All the values that simulator $\mathcal{S}_L$ is required to simulate appear in the joint state $\mathbf{w}_t$, except for $\widetilde{\mathbf{L}}_f, \widetilde{\mathbf{L}}^{\mathsf{fr}}_{\otimes,z(O)}, \widetilde{\mathbf{L}}', \widetilde{\mathbf{L}}'', \mathsf{init}^{\mathsf{OG}(t)}_L, \mathsf{evl}^{\mathsf{OG}(t)}_L$ that are computed using the joint state $\mathbf{w}_i$ and $f = f_t, f(k)$ as follows:

1. For input wire $i \in I$ with value f_i, sample $\widetilde{L}_{f_i}$ uniformly at random from $\mathbb{F}_2^m$, subject to $\langle \widetilde{L}_{f_i}, \widetilde{R}_{f_i}\rangle = f_i$.

2. $\widetilde{\mathbf{L}}^{\mathsf{fr}}_{\otimes,z(O)}$ is sampled consistently with how $\mathcal{S}_{R,A}$ samples it (by rleying on shared randomness).

3. For every gate $g \in G$, with input wires $x = x(g), y = y(g)$, and output wire $z = z(g)$, figure out the pre-refreshed share $\widetilde{L}_{\otimes,z} \in \mathbb{F}_2^{m^2}$. If $g \in I$ is an input gate, this can be figured out from the values $\widetilde{\mathbf{L}}_k, \widetilde{\mathbf{L}}_f$. If $g \in G \setminus (I \cup O)$, this can be figured out from $\widetilde{L}^{\mathsf{fr}}_{\otimes,x}, \widetilde{R}^{\mathsf{fr}}_{\otimes,x}[m, \ldots, m^2], \widetilde{L}^{\mathsf{fr}}_{\otimes,y}, \widetilde{R}^{\mathsf{fr}}_{\otimes,y}[m, \ldots, m^2]$, which determine $\widetilde{L}_x, \widetilde{L}_y$. Indeed, $\widetilde{L}_x = (1 + v|\widetilde{L}^{\mathsf{fr}}_{\otimes,x}[1, \ldots, m-1])$, where $v = \langle \widetilde{L}^{\mathsf{fr}}_{\otimes,z}[m, \ldots, m^2], \widetilde{R}^{\mathsf{fr}}_{\otimes,z}[m, \ldots, m^2] \rangle$, and $\widetilde{L}_y$ is computed similarly. Then, $\mathcal{S}_L$ sets

$$\widetilde{L}'_g \leftarrow \widetilde{M}'_g \widetilde{L}_{\otimes,z} \qquad \widetilde{L}''_g \leftarrow (\widetilde{M}''^{t}_g)^{-1}(\widetilde{L}^{\mathsf{fr}}_{\otimes,z} - \widetilde{L}_{\otimes,z}) \ .$$

4. For refreshing the shares corresponding to the i^{th} bit of the secret key k,

$$\widetilde{L}'_{k_i} = (\widetilde{M}'^{t}_{k_i})^{-1}(\widetilde{L}^{t+1}_{k_i} - \widetilde{L}^{t}_{k_i}) \qquad \widetilde{L}''_{k_i} = \widetilde{M}''_{k_i} \widetilde{L}^{t+1}_{k_i} \ .$$

5. $\mathsf{init}^{\mathsf{OG}(t)}_L, \mathsf{evl}^{\mathsf{OG}(t)}_L$ are sampled by invoking $\mathcal{S}^{\mathsf{OG}}_L$ with input $\widetilde{\mathbf{L}}', \widetilde{\mathbf{L}}''$, which include all the vectors $\widetilde{\mathbf{L}}', \widetilde{\mathbf{L}}''$ produced the t^{th} iteration.[2]

Proposition 6. Λ *is continually ℓ-leakage-resilient strong OCL scheme against unbounded adversaries, with $2 + a$ components, where $\ell(\lambda) = m(\lambda)/10$, $m(\lambda) = \omega(\log(\lambda))$, and $a = O(1)$.*

The proof naturally relies on the properties of inner product as a *two-source extractor* [CG88]. Concretely, we will use the fact that an (unbounded) adversary $\mathcal{A}$ that executes an $\frac{m}{10}$-leakage attack on two sources (L, R),[3] cannot tell whether they are sampled uniformly at random from $\mathbb{F}_2^m \times \mathbb{F}_2^m$, or from the correlated distribution $\mathcal{O}_b^m$, where $\langle L, R \rangle = b$.

Lemma 7 (Following [CG88, GR12])**.** *Let $\mathsf{view}^{(L,R)}_{\mathcal{A}}$ denote the view of an adversary $\mathcal{A}$ in an $m/10$-leakage attack on sources $(L, R) \in \mathbb{F}_2^m \times \mathbb{F}_2^m$. Then for any $b \in \mathbb{F}_2$, the following two distributions are $2^{-\Omega(m)}$-statistically close:*

$$\{\mathsf{view}^{(L,R)}_{\mathcal{A}} : (L, R) \leftarrow \mathbb{F}_2^m \times \mathbb{F}_2^m\} \ , \tag{1}$$

$$\{\mathsf{view}^{(L,R)}_{\mathcal{A}} : (L, R) \leftarrow \mathcal{O}_b^m\} \ . \tag{2}$$

The lemma is in proved in [GR12][Lemma 3.11].

We shall also use the following simple corollary.

Corollary 8 (of Lemma 7)**.** *Let $\mathsf{view}^{(L,R)}_{\mathcal{A}}(R[m, \ldots, m^2])$ denote the view of an adversary $\mathcal{A}$ in an $m/10$-leakage attack on sources $(L, R) \in \mathbb{F}_2^{m^2} \times \mathbb{F}_2^{m^2}$, where it also gets all but the first $m - 1$ entries of R. Then, for any $b \in \mathbb{F}_2$, the following two distributions are $2^{-\Omega(m)}$-statistically close:*

$$\{\mathsf{view}^{(L,R)}_{\mathcal{A}}(R[m, \ldots, m^2]) : (L, R) \leftarrow \mathbb{F}_2^{m^2} \times \mathbb{F}_2^{m^2}\} \ , \tag{1}$$

$$\{\mathsf{view}^{(L,R)}_{\mathcal{A}}(R[m, \ldots, m^2]) : (L, R) \leftarrow \mathcal{O}_b^{m^2}\} \ . \tag{2}$$

[2]Here we rely on the fact that, unlike $\mathcal{S}^{\mathsf{OG}}_{R,A}$, who needs the values $\widetilde{\mathbf{R}}', \widetilde{\mathbf{R}}''$ also from previous iterations, $\mathcal{S}^{\mathsf{OG}}_L$ can do only with those of the current iteration (see Definition 8). Indeed, in order to compute $\widetilde{\mathbf{L}}', \widetilde{\mathbf{L}}''$ as above, the current input $f = f_t$ and output $f(k)$ are needed.

[3]A ℓ-leakage attack on two sources (L, R), sampled from some correlated distribution, is defined analogously to an ℓ-leakage attack on two OCL components.

Proof sketch: Given a distinguisher $\mathcal{A}$ that breaks the above for some $b \in \mathbb{F}_2$, we can construct a distinguisher $\mathcal{A}'$ that breaks Lemma 7. $\mathcal{A}'$ samples $(L[m, \ldots, m^2], R[m, \ldots, m^2]) \leftarrow \mathbb{F}_2^{m^2-m+1} \times \mathbb{F}_2^{m^2-m+1}$; we shall denote $v = \langle L[m, \ldots, m^2], R[m, \ldots, m^2] \rangle$. Then it emulates

$$\mathcal{A}^{(L[1,\ldots,m-1],R[1,\ldots,m-1])}(R[m, \ldots, m^2]) \ ,$$

where $(L[1, \ldots, m-1], R[1, \ldots, m-1])$ are sampled either independently from $\mathbb{F}_2^{m-1} \times \mathbb{F}_2^{m-1}$, or from $\mathcal{O}_{b-v}^{m-1}$. In the first case, (L, R) are distributed uniformly and independently. In the second case they are $2^{-\Omega(m)}$ close to being sampled from $\mathcal{O}_b^{m^2}$. The corollary follows. $\square$

We now turn to give a proof sketch of Proposition 6.

Proof sketch: We show that the view of any $\mathcal{A}$, when executing a continual ℓ-leakage attack in the real world, is $2^{-\Omega(m)}$-close to its view in the ideal world, where the state of (P_L, P_R) is simulated by $\mathcal{S} = (\mathcal{S}_L, \mathcal{S}_R)$. For this purpose, we consider a sequence of hybrid experiments and show their statistical indistinguishability. Throughout, let $t^* = t^*(\lambda)$ be the polynomial bound on the number of iterations that $\mathcal{A}$ chooses to execute.

hyb_0: Describes the ideal world where leakage is simulated by $\mathcal{S}$.

hyb_1: This experiment differs from hyb_0 in that, in every iteration t, with input $f = f_t$, instead of sampling first $\widetilde{R}_{f_i}$ and then sampling $\widetilde{L}_{f_i}$ conditioned on $\langle \widetilde{L}_{f_i}, \widetilde{R}_{f_i} \rangle = f_i$, the two are sampled together directly from $\mathcal{O}_{f_i}^m$ as in the real world. It is easy to see that the distribution of the entire continual execution, including the full states of all parties, is the same up to a $2^{-\Omega(m)}$ statistical distance.

$\mathsf{hyb}_{2,t}$, for $0 \le t \le t^*$: In this experiment, $\mathsf{hyb}_{2,0} = \mathsf{hyb}_1$, and every $\mathsf{hyb}_{2,t-1}$ differs from $\mathsf{hyb}_{2,t}$ in that, in the t^{th} execution, instead of sampling $(\widetilde{L}_{k_i}^t, \widetilde{R}_{k_i}^t)$ uniformly and independently, they are sampled from $\mathcal{O}_{k_i}^m$ as in the real world. By Lemma 7, each two such hybrids are $2^{-\Omega(m)}$-close. Indeed, we can transform any $\mathcal{A}$ that distinguishes $\mathsf{hyb}_{2,t-1}$ from hyb_{2_t} to a distinguisher $\mathcal{A}'$ in Lemma 7 for $b = k_i$. $\mathcal{A}'$ would emulate $\mathcal{A}$ and simulate its leakage using leakage on $(\widetilde{L}_{k_i}, \widetilde{R}_{k_i})$. Indeed, the simulated leaky state of P_L can be fully simulated from $\widetilde{L}_{k_i}$, and that of P_R can be fully simulated from $\widetilde{R}_{k_i}$.

$\mathsf{hyb}_{3,t,g}$, for $1 \le t \le t^*$, for $0 \le g \le T - w$: In this experiment, for any t, $\mathsf{hyb}_{3,t,g-1}$ differs from $\mathsf{hyb}_{3,t,g}$ in that, instead of sampling $(\widetilde{L}_{\otimes,z(g)}^{\mathsf{fr}}, \widetilde{R}_{\otimes,z(g)}^{\mathsf{fr}})$ uniformly at random they are sampled from $\mathcal{O}_{b(g)}^{m^2}$, where $b(g) = \langle \widetilde{L}_{\otimes,z(g)}, \widetilde{R}_{\otimes,z(g)} \rangle$; namely, refreshing preserves the inner product as in the real world. $\mathsf{hyb}_{3,1,0} = \mathsf{hyb}2, t^*$ and $\mathsf{hyb}_{3,t,0} = \mathsf{hyb}_{3,t-1,T-w}$.

We show how to turn any $\mathcal{A}$ that distinguishes $\mathsf{hyb}_{3,t,g-1}$ from $\mathsf{hyb}_{3,t,g}$, to a distinguisher $\mathcal{A}'$ in *Corollary* 8 for the value $b = b(g)$, which is the real-world output value of gate g. $\mathcal{A}'$ attacks $(\widetilde{L}_{\otimes,z(g)}^{\mathsf{fr}}, \widetilde{R}_{\otimes,z(g)}^{\mathsf{fr}})$ that are sampled either independently or from $\mathcal{O}_b^{m^2}$; in addition, it gets the extra input $\widetilde{R}_{\otimes,z(g)}^{\mathsf{fr}}[m, \ldots, m^2]$. $\mathcal{A}'$ emulates $\mathcal{A}$, simulating its leakage queries using leakage on $(\widetilde{L}_{\otimes,k(g)}^{\mathsf{fr}}, \widetilde{R}_{\otimes,z(g)}^{\mathsf{fr}})$. The simulated leaky state of P_L is simulated from $\widetilde{L}_{\otimes,z(g)}^{\mathsf{fr}}$ and $\widetilde{R}_{\otimes,z(g)}^{\mathsf{fr}}[m, \ldots, m^2]$, and that of P_R is simulated from $\widetilde{R}_{\otimes,z(g)}^{\mathsf{fr}}$. It is easy to see that the view of the emulated $\mathcal{A}$ corresponds to either $\mathsf{hyb}_{3,t,g-1}$ if the shares are sampled independently, and according to $\mathsf{hyb}_{3,t,g}$ if the shares are sampled from $\mathcal{O}_b^{m^2}$.

We now note that in $\mathsf{hyb}_{3,t^*,T-w}$, all the shared values along the wires are consistent with the real world. Moreover, for each output gate $g \in O$ the simulator already samples $(\widetilde{L}_{\otimes,k(g)}^{\mathsf{fr}}, \widetilde{R}_{\otimes,k(g)}^{\mathsf{fr}})$ with inner product $[f(k)]_g$, which will now be the same as $\langle \widetilde{L}_{\otimes,k(g)}, \widetilde{R}_{\otimes,k(g)} \rangle$. Thus, throughout the entire circuit the simulated refresh procedure preserves inner product as in the real world.

hyb_4: This hybrid differs from $\mathsf{hyb}_{3,t^*,T-w}$ in that the refresh procedure is done in a different order. Recall that in the simulated $\mathsf{hyb}_{3,t^*,T-w}$, to refresh $(\widetilde{L}, \widetilde{R})$, we first sample $\widetilde{M}', \widetilde{M}'' \leftarrow \mathbb{F}_2^{m^2 \times m^2}$, $(\widetilde{L}^{\mathsf{fr}}, \widetilde{R}^{\mathsf{fr}}) \leftarrow \mathcal{O}^{\ell}_{\langle \widetilde{L}, \widetilde{R} \rangle}$, where $\ell \in \{m, m^2\}$, and then derive $\widetilde{L}', \widetilde{L}'', \widetilde{R}', \widetilde{R}''$ according to the equations:

$$\begin{aligned} \widetilde{L}' &= \widetilde{M}'\widetilde{L}, & \widetilde{R}' &= (\widetilde{M}'^t)^{-1}(\widetilde{R}^{\mathsf{fr}} - \widetilde{R}) \\ \widetilde{L}'' &= (\widetilde{M}''^t)^{-1}(\widetilde{L}^{\mathsf{fr}} - \widetilde{L}), & \widetilde{R}'' &= \widetilde{M}''\widetilde{R}^{\mathsf{fr}} \ . \end{aligned}$$

Now, instead of the above, we first sample $((\widetilde{L}'|\widetilde{L}''), (\widetilde{R}'|\widetilde{R}'')) \leftarrow \mathcal{O}_0^{2m^2}$, then we sample $\widetilde{M}'$ uniformly conditioned on $\widetilde{L}' = \widetilde{M}'\widetilde{L}$, and set $\widetilde{R}^{\mathsf{fr}}$ according to $\widetilde{R}' = (\widetilde{M}'^t)^{-1}(\widetilde{R}^{\mathsf{fr}} - \widetilde{R}$, then we sample $\widetilde{M}''$ uniformly conditioned on $\widetilde{R}'' = \widetilde{M}''\widetilde{R}^{\mathsf{fr}}$, and set $\widetilde{L}^{\mathsf{fr}}$ according to $\widetilde{L}'' = (\widetilde{M}''^t)^{-1}(\widetilde{L}^{\mathsf{fr}} - \widetilde{L})$.

It is not hard to check that the joint distributions of

$$\widetilde{L}', \widetilde{L}'', \widetilde{R}', \widetilde{R}'', \widetilde{M}', \widetilde{M}'', \widetilde{L}^{\mathsf{fr}}, \widetilde{R}^{\mathsf{fr}}$$

in all of the t^* iterations, in the two cases are $2^{-\Omega(m^2)}$-close.

hyb_5: Identical to the real world. Note that the previous hybrid hyb_4 is identical to the real world, with the exception that the refreshing operation is done using simulated orthogonal generation generated by $\mathcal{S}^{\mathsf{OG}}$, instead of real orthogonal generation by OG. The two hybrids are statistically close by the simulation guarantee of $\mathcal{S}^{\mathsf{OG}}$. Indeed, we can easily convert a distinguisher between hyb_4 and hyb_5 into a distinguisher between the real and ideal orthogonal generation, as we can fully simulate the state of P_L, respectively P_R, from that of P_L^{OG}, respectively P_R^{OG}.

This completes the proof sketch.

□

D Two-Party Leakage-Tolerant Protocols without Corruption

In this section, we show how to construct a two-party, a auxiliary-party, continual leakage-tolerant protocol ρ in the input-independent pre-processing model based on any strong, continual 2-component OCL scheme with a auxiliary parties. Our tranformation works for any number a of auxiliary parties, and, in particular works for the special case of $a = 0$. The protocol is secure against adversaries that leak a bounded amount of ℓ bits of information on the state of each honest party (separately) in each time period, but do not corrupt any of the parties. We note that using the strong continual OCL constructions of Section C, we achieve our result without requiring the use of any trusted hardware.

Theorem 6. *Assume the existence of a ℓ-continual-leakage-resilient strong OCL Λ scheme with some number, a, of auxiliary components for the universal circuit family and the existence of one-way functions. Then for every efficiently computable deterministic two-input two-output function $f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* \times \{0,1\}^*$, there is a protocol ρ that strongly UC-emulates the functionality $\mathcal{F}^f_{\mathsf{2LTC\text{-}AUX}}$ under ℓ-bounded continual leakage, with a auxiliary parties, when no party is corrupted, in the $(\mathcal{F}_{\mathsf{LSC}}, \mathcal{F}_{\mathsf{LFS}})$-hybrid model (i.e. with secure communication and input-independent leakage-free preprocessing). Furthermore, if Λ has perfect correctness, ρ also has perfect correctness.*

Towards proving the theorem, we first observe that it suffices to consider only functions with a single output and design leakage-tolerant protocols where both parties obtain this output. For brevity of notation, below we use $\mathcal{F}^{f,\infty}_{\mathsf{2LTC\text{-}AUX}}$ for a single output function f to denote the ideal functionality that proceeds identically to $\mathcal{F}^{g,\infty}_{\mathsf{2LTC\text{-}AUX}}$ for two-output functions as described in Figure 2, except that it distributes the single output to both parties.

Proposition 9. *Assume the existence of a ℓ-continual-leakage-resilient strong OCL Λ scheme with a auxiliary components for the universal circuit family. Then, for every efficiently computable deterministic two-input function $f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$, there is a protocol ρ that strongly UC-emulates the functionality $\mathcal{F}^{f,\infty}_{\mathsf{2LTC\text{-}AUX}}$ under ℓ-bounded continual leakage, when no party is corrupted, in the $(\mathcal{F}_{\mathsf{LSC}}, \mathcal{F}_{\mathsf{LFS}})$-hybrid model (i.e. with secure communication and input-independent leakage-free preprocessing). Furthermore, if Λ has perfect correctness, ρ also has perfect correctness.*

We show that Theorem 6 directly follows from Proposition 9 using standard techniques.

Proof. Fix any two-input two-output function g; let $m(n)$ be a bound on the length of each output of the function given two n-bit input strings. Translate g into a single output function f as follows: For every $x_0^j, x_1^j \in \{0,1\}^n$, $f((x_0^j, r_0^j), (x_1^j, r_1^j)) = y_0^j \oplus r_0^j \| y_1^j \oplus r_1^j$, where $r_0^j, r_1^j, y_0^j, y_1^j \in \{0,1\}^m$ and (y_0^j, y_1^j) are the outputs of $g(x_0^j, x_1^j) = (y_0^j, y_1^j)$. By Proposition 9, there is a $a+2$-party protocol $\rho = (P_0, P_1, P_1^{\mathsf{aux}}, \ldots, P_a^{\mathsf{aux}})$ that strongly UC-emulates the functionality $\mathcal{F}^{f,\infty}_{\mathsf{2LTC\text{-}AUX}}$ under ℓ-bounded continual leakage, when no party is corrupted, in the $(\mathcal{F}_{\mathsf{LSC}}, \mathcal{F}_{\mathsf{LFS}})$-hybrid model. Then using ρ, we construct a protocol $\rho' = (P_0', P_1', P_1^{\mathsf{aux}}, \ldots, P_a^{\mathsf{aux}})$ that strongly UC-emualtes $\mathcal{F}^{g,\infty}_{\mathsf{2LTC\text{-}AUX}}$ under ℓ-bounded leakage, when no party is corrupted in the $(\mathcal{F}_{\mathsf{LSC}}, \mathcal{F}_{\mathsf{LFS}})$-hybrid model.

In the j-th iteration, the protocol ρ' simply has the two parties with input, P_0' and P_1', given security parameter 1^λ and private inputs $x_0^j, x_1^j \in \{0,1\}^n$, choose two random strings r_0^j and r_1^j of length m, respectively, and invoke ρ to compute $f((x_0^j, r_0^j), (x_1^j, r_1^j))$. At the end of the interaction, P_0' obtains an output $z_0^{j,0} \| z_1^{j,0}$ and outputs $\tilde{y}_0^j = z_0^{j,0} \oplus r_0^j$, while P_1' obtains $z_0^{j,1} \| z_1^{j,1}$ and outputs $\tilde{y}_1^j = z_1^{j,1} \oplus r_1^j$. The (perfect) correctness of ρ follows direct from the (perfect) correctness of ρ.

To see that ρ' indeed strongly UC-emulates $\mathcal{F}^{g,\infty}_{\mathsf{2LTC\text{-}AUX}}$ under ℓ-bounded continual leakage, we need to show that there is an oblivious simulator $\mathcal{S}_{\rho'}$ such that no malicious environment $\mathbf{Z}$ that, on input 1^λ and auxiliary input $z \in \{0,1\}^{\mathrm{poly}(\lambda)}$, launches an ℓ-bounded continual leakage attack on an execution of ρ' (without corrupting any party), can distinguish whether it interacts with an honest execution of ρ', or with $\mathcal{S}_{\rho'}$ and $\mathcal{F}^{g,\infty}_{\mathsf{2LTC\text{-}AUX}}$. By the fact that ρ strongly UC-emulates $\mathcal{F}^{f,\infty}_{\mathsf{2LTC\text{-}AUX}}$, there exists such an oblivious simulator $\mathcal{S}_\rho$ for ρ; moreover, by the definition of the oblivious simulators, $\mathcal{S}_\rho$ contains a sub-component $\tilde{\mathcal{S}}_\rho$ that handles all leakage queries from $\mathbf{Z}$.

Towards constructing an oblivious simulator $\mathcal{S}_{\rho'}$ for ρ', we first observe that an honest execution of ρ' with inputs (x_0^j, x_1^j) outputs (y_0^j, y_1^j) proceeds identically to an honest execution of ρ with inputs $((x_0^j, r_1^j), (x_1^j, r_1^j))$ and outputs $(z_0^j = y_0^j \oplus r_0^j, z_1^j = y_1^j \oplus r_1^j)$ for randomly chosen r_0^j, r_1^j. Furthermore, since (r_0^j, r_1^j) are truly random, (z_0^j, z_1^j) are truly random as well, and the distribution of the inputs and outputs $\{((x_0^j, r_1^j), (x_1^j, r_1^j)), (z_0^j, z_1^j)\}$ remains the same even if (z_0^j, z_1^j) are sampled at random first and (r_0^j, r_1^j) computed accordingly second as $r_b^j = y_b^j \oplus z_b^j$.

Simulator $\mathcal{S}_{\rho'}$ interacts externally with $\mathcal{F}^{g,\infty}_{\mathsf{2LTC\text{-}AUX}}$ and $\mathbf{Z}$; to simulate an honest execution of ρ' with inputs (x_0^j, x_1^j) chosen by $\mathbf{Z}$, it samples two strings $(z_0^j, z_1^j) \leftarrow U_{2m}$ and internally simulates an execution of ρ with inputs $(x_0^j, r_0^j), (x_1^j, r_1^j)$ for $r_b^j = y_b^j \oplus z_b^j$ using $\mathcal{S}_\rho$. More specifically, it forwards all simulated protocol messages by $\mathcal{S}_\rho$ to $\mathbf{Z}$. Whenever $\mathbf{Z}$ submits a leakage query (L, P_b'), it invokes a separate module $\tilde{\mathcal{S}}_{\rho'}$ with its current state containing the current state state of $\mathcal{S}_\rho$ and (z_0^j, z_1^j) as input; $\tilde{\mathcal{S}}_{\rho'}$ internally runs $\tilde{\mathcal{S}}_\rho(\mathsf{state})$ to simulate any leakage answer. Recall that, by definition, $\tilde{\mathcal{S}}_\rho(\mathsf{state})$ produces a state translation function T_b or T_A. If the leakage was on P_b, $\mathcal{S}_\rho(\mathsf{state})$ then expects to receive the leakage on the ideal state of P_b from $\mathcal{F}^{f,\infty}_{\mathsf{2LTC\text{-}AUX}}$ with respect to $L \circ T_b$, which is then forwarded to $\mathbf{Z}$ as the leakage answer. On the other hand, if the leakage was on party P_i^{aux} for some $i \in a$, $\mathcal{S}_\rho(\mathsf{state})$ uses T_A to reconstruct the real states of *all* auxiliary parties and then responds to the leakage query with $L \circ T_A[i]$, where $T_A[i]$ denotes the function outputting the state of the i-th auxiliary party reconstructed by T_A. $\tilde{\mathcal{S}}_{\rho'}$ emulates $\mathcal{F}^{f,\infty}_{\mathsf{2LTC\text{-}AUX}}$ for $\tilde{\mathcal{S}}_\rho$ as follows: For parties P'_b, $b \in \{0,1\}$, it translates the state translation function T_b with respect to ρ into a state translation function T'_b with respect to ρ'.

$$T'_b(x_b^j, y_b^j) = T_b((x_b^j, y_b^j \oplus z_b^j), z_0^j \| z_1^j) \ .$$

It then obtains leakage $L \circ T'_b(x_b^j, y_b^j)$ from $\mathcal{F}^{g,\infty}_{\mathsf{2LTC\text{-}AUX}}$ and forwards it back to $\tilde{\mathcal{S}}_\rho$ and $\mathbf{Z}$. Similarly, for the auxiliary parties P_i^{aux}, $i \in [a]$, $\tilde{\mathcal{S}}_{\rho'}$ sets $T'_A = T_A$. It then computes the leakage $L \circ T'_A[i]$ and forwards it back to $\tilde{\mathcal{S}}_\rho$ and $\mathbf{Z}$.

By construction $\mathcal{S}_{\rho'}$ has the structure of an oblivious simulator; furthermore, it internally emulates an execution of $\mathcal{S}_\rho$ interacting with functionality $\mathcal{F}^{f,\infty}_{\mathsf{2LTC\text{-}AUX}}$ parameterized with inputs $((x_0^j, r_0^j = y_0^j \oplus z_0^j), (x_1^j, r_1^j = y_1^j \oplus z_1^j))$ for randomly sampled $(z_0^j, z_1^j) \leftarrow U_{2m}$ perfectly. Then, the view of the environment $\mathbf{Z}$ interacting with $\mathcal{S}_{\rho'}$ and $\mathcal{F}^{g,\infty}_{\mathsf{2LTC\text{-}AUX}}$ with inputs (x_0^j, x_1^j), is identical to its view with $\mathcal{S}_\rho$ and $\mathcal{F}^{f,\infty}_{\mathsf{2LTC\text{-}AUX}}$ with inputs $(x_0^j, r_0^j), (x_1^j, r_1^j)$ for randomly chosen (r_0^j, r_1^j). By the security of ρ, the latter is indistinguishable from the view of $\mathbf{Z}$ when interacting with an honest execution of ρ with inputs $(x_0^j, r_0^j), (x_1^j, r_1^j)$ for randomly chosen r_0^j, r_1^j, which is identical to an honest execution of ρ with inputs x_0^j, x_1^j. Thus $\mathbf{Z}$ cannot distinguish whether it interacts with an honest execution of ρ' or with $\mathcal{S}_{\rho'}$ and $\mathcal{F}^{g,\infty}_{\mathsf{2LTC\text{-}AUX}}$. This concludes the proof of the Theorem.

□

D.1 The Protocol ρ

Let λ be the security parameter, and let f be an efficiently computable deterministic two-input function. Below we present a two-party leakage-tolerant protocol ρ that strongly emulates the functionality $\mathcal{F}^{f,\infty}_{\mathsf{2LTC\text{-}AUX}}$ in the $(\mathcal{F}_{\mathsf{LSC}}, \mathcal{F}_{\mathsf{LFS}})$-hybrid model, where $\mathcal{F}_{\mathsf{LSC}}$ is the secure communication functionality and $\mathcal{F}_{\mathsf{LFS}}$ captures the leakage-free preprocessing functionality. The protocol assumes a ℓ-continual leakage-resilient strong 2-component OCL scheme with a auxiliary parties. $\Lambda = (\mathsf{Comp}, \Pi = (P_L, P_R, P_{A_1}, \ldots, P_{A_a}))$ with an oblivious simulator $\mathcal{S} = (\mathcal{S}_L, \mathcal{S}_R, \mathcal{S}_A)$.

Let n be the length of the inputs $x_0^j, x_1^j \in \{0,1\}^n$ to be evaluated in the j-th iteration, which is polynomially related with the security parameter. (The reason that we separate the security parameter from the length of the input is that the leakage-bound of the protocol only depends on the security parameter, but not the input length. Thus, by scaling up the security parameter, the absolute number of leakage bits that the protocol tolerates grows.) Our leakage-tolerant protocol below essentially makes use of the OCL scheme to perform the evaluation of $f(x_0^j, x_1^j)$. However, to ensure input privacy, any party should avoid sharing its input in the clear with the other party. Instead, in the j-th iteration, the parties first use the OCL scheme to enable P_1 to obtain an encrypted version $x_0^j \oplus \mathsf{PRF}(r, j)$ of P_0's input, where PRF is a pseudorandom function and the PRF key r is encoded as the OCL secret. Then, instead of directly evaluating f, the OCL scheme is used again to evaluate the following function $g((c^j = (x_0^j \oplus \mathsf{PRF}(r,j)), x_1), \mathsf{PRF}(r,j)) = f(c^j \oplus \mathsf{PRF}(r,j), x_1^j)$.

We now present the leakage-tolerant protocol ρ in detail: It consists of an input-independent pre-processing stage, which is independent of the function and input to be computed, and an online stage that depends on the function and input.

In the following, we simplify notation by denoting by $\mathsf{init}_A^{j,b} = \mathsf{init}_{A,1}^{j,b}, \ldots, \mathsf{init}_{A,a}^{j,b}$, for $j \in [a]$ and $b \in \{1,2\}$, the initial states of *all* auxiliary components of the b-th OCL in the j-th iteration. We similarly define $\mathsf{evl}_A^{j,b}$. Moreover, by $\mathbf{x}_0^j = x_0^1, \ldots, x_0^j$ we denote the sequence of inputs of P_0 in the first j iterations. Similarly, by $\mathbf{x}_1^j = x_1^1, \ldots, x_1^j$ we denote the sequence of inputs of P_1 in the first j iterations and by $\mathbf{y}^j = y_1^1, \ldots, y_1^j$ the sequence of outputs in the first j iterations.

The input-independent preprocessing stage: The pre-processing stage, or formally the leakage-free sampling (LFS) functionality $\mathcal{F}_{\mathsf{LFS}}^{\mathsf{Comp}_\rho}$, on input $(1^\lambda, T)$, where T will be specified later, invokes a compilation algorithm Comp_ρ on $(1^\lambda, T)$, proceeding as follows:

1. Sample $r \leftarrow U_\lambda$ uniformly at random.
2. Sample two pairs of initial states of the OCL scheme Λ w.r.t. secret r independently and randomly: $(\mathsf{init}_L^1, \mathsf{init}_R^1, \mathsf{init}_A^1) \leftarrow \mathsf{Comp}(1^\lambda, U_T, r)$ and $(\mathsf{init}_L^2, \mathsf{init}_R^2, \mathsf{init}_A^2) \leftarrow \mathsf{Comp}(1^\lambda, U_T, r)$.
3. Distribute $\Phi_0 = (\mathsf{init}_L^1, \mathsf{init}_R^2)$ to P_0, $\Phi_1 = (\mathsf{init}_R^1, \mathsf{init}_L^2)$ to P_1 and $\Phi_A = (\mathsf{init}_A^1, \mathsf{init}_A^2)$ to the auxiliary parties.

(Note that the distribution of the initial states are "crossed"—For the first pair, P_0 receives the left initial state, whereas for the second pair P_1 receives the left initial state. In the evaluation procedure below, two sequential OCL evaluations take place, where P_0 plays the left component in the first evaluation and P_1 parties the left component in the second.)

The online stage: For each iteration j, given the initial states Φ_0, Φ_1 and Φ_A sampled in the preprocessing stage, the two parties P_0 and P_1 and a axuiliary parties $P_1^{\mathsf{aux}}, \ldots, P_a^{\mathsf{aux}}$ on common input $(1^\lambda, f, T)$, and private inputs $x_0^j \in \{0,1\}^n$ and $x_1^j \in \{0,1\}^n$, respectively, proceed in the following steps, where all messages are sent through the secure channel functionality $\mathcal{F}_{\mathsf{LSC}}$:

1. *The first OCL evaluation—Compute an encryption $c^j = x_0^j \oplus \mathsf{PRF}(r, j)$ of x_0^j:*
   P_0, P_1 and the auxiliary parties $P_1^{\mathsf{aux}}, \ldots, P_a^{\mathsf{aux}}$ compute $x_0^j \oplus \mathsf{PRF}(r,j)$ using the OCL protocol Π. More precisely, P_0 acts as the left component using initial state $\mathsf{init}_L^{j,1}$, P_1 acts as the right component using initial state $\mathsf{init}_R^{j,1}$ and parties $P_1^{\mathsf{aux}}, \ldots, P_a^{\mathsf{aux}}$ act as the auxiliary components using initial states $\mathsf{init}_{A,i}^{j,1}, \ldots, \mathsf{init}_{A,a}^{j,1}$, respectively. P_0 feeds the following function $g_1^j(r) = g_1^{(j,x_0^j)}(r) = x_0^j \oplus \mathsf{PRF}(r,j)$ to the left component as input. At the end of the evaluation P_1 obtains $\tilde{c}^j$.
2. *The second OCL evaluation—Compute the output $f(x_0^j, x_1^j)$:*
   P_0 and P_1 and the auxiliary parties $P_1^{\mathsf{aux}}, \ldots, P_a^{\mathsf{aux}}$ compute $y^j = f(x_0^j, x_1^j)$ by evaluating the function $g((\tilde{c}^j, x_1^j), \mathsf{PRF}(r,j))$ using Π again. More precisely, P_0 acts as the right component using initial state $\mathsf{init}_R^{j,2}$, P_0 acts as the left component using initial state $\mathsf{init}_L^{j,2}$ and parties $P_1^{\mathsf{aux}}, \ldots, P_a^{\mathsf{aux}}$ act as the auxiliary components using initial states $\mathsf{init}_{A,1}^{j,2}, \ldots, \mathsf{init}_{A,a}^{j,2}$, respectively. P_1 feeds the function $g_2^j(r) = g_2^{(j,\tilde{c}^j,x_1^j)}(r) = f(\tilde{c}^j \oplus \mathsf{PRF}(r,j), x_1^j)$ to the left component as input. At the end of the evaluation P_0 obtains $\tilde{y}^j$.
3. P_0 sends $\tilde{y}^j$ to P_1. They both output $\tilde{y}^j$.

$T = T(n)$ is thus set to bound on the time for computing the functions (g_1^j, g_2^j) on two n-bit inputs.

D.2 Analysis of Security

The following lemma implies Proposition 9.

Lemma 10. *Assume that Λ is a ℓ-continual-leakage-resilient strong OCL scheme. Then the above protocol ρ strongly UC-emulates the functionality $\mathcal{F}_{\mathsf{2LTC\text{-}AUX}}^{f,\infty}$ under ℓ-bounded continual leakage, when no party is corrupted. Furthermore, if Λ has perfect correctness, ρ also has perfect correctness.*

Proof. It follows from the correctness (or perfect correctness) of Λ that with overwhelming probability (or with probability 1, respectively) for each iteration j, the two OCL evaluations computes correctly $\tilde{c}^j = c^j = x_0^j \oplus \mathsf{PRF}(r, j)$ and $\tilde{y}^j = g((\tilde{c}^j, x_1^j), \mathsf{PRF}(r, j)) = f(x_0^j, x_1^j)$. Thus the protocol $\langle P_0, P_1, P_1^{\mathsf{aux}}, \ldots, P_a^{\mathsf{aux}}\rangle$ has correctness (or perfect correctness, respectively).

Next, we show that ρ strongly UC-emulates the functionality $\mathcal{F}^{f,\infty}_{\mathsf{2LTC\text{-}AUX}}$ under ℓ-bounded continual leakage, when no party is corrupted. Recall that to show this, it suffices to consider a dummy adversary $\mathcal{D}$ that does not corrupt any party, and simply follows all instructions of the environment $\mathbf{Z}$. In particular, $\mathcal{D}$ forwards all the protocol messages between the honest parties P_0, P_1, $P_1^{\mathsf{aux}}, \ldots, P_a^{\mathsf{aux}}$ to $\mathbf{Z}$, executes all the leakage queries issued by $\mathbf{Z}$ and forwards the answers to $\mathbf{Z}$. We want to show that there is an oblivious simulator $\mathcal{S}_\rho$, such that, for every environment $\mathbf{Z}$, every large enough $\lambda \in \mathbb{N}$ and auxiliary input $z \in \{0,1\}^{\mathrm{poly}(\lambda)}$, the outputs of $\mathbf{Z}(1^\lambda, z)$, when interacting with $\mathcal{S}_\rho$ and functionality $\mathcal{F}^{f,\infty}_{\mathsf{2LTC\text{-}AUX}}$ or interacting with $\mathcal{D}$ and ρ in the hybrid model with functionalities $\mathcal{F}_{\mathsf{LSC}}$ and $\mathcal{F}_{\mathsf{LFS}}^{\mathsf{Comp}_\rho}$, are indistinguishable. Below we construct such a simulator $\mathcal{S}_\rho$ and show its correctness.

The simulator $\mathcal{S}_\rho$. The leakage simulator $\mathcal{S}_\rho$ simulates the protocol messages and leakage answers for $\mathbf{Z}$ as follows:

SIMULATE PROTOCOL MESSAGES: Recall that P_0, P_1 and parties $P_1^{\mathsf{aux}}, \ldots, P_a^{\mathsf{aux}}$ communicate via the secure channel functionality $\mathcal{F}_{\mathsf{LSC}}$, which only leaks the length of the messages to the adversary. Since every message in the protocol belongs to an OCL evaluation, let $B = B(\lambda, T)$ be a bound on the lengths of the messages exchanged between the left and right components when evaluating T-time functions using Λ and security parameter λ. Then, for every message in the protocol, $\mathcal{S}_\rho$ sends message $(\mathsf{send}, S, R, |B|)$ with the appropriate senders and receivers $S, R \in \{P_0, P_1, P_1^{\mathsf{aux}}, \ldots, P_a^{\mathsf{aux}}\}$ to $\mathbf{Z}$.

SIMULATE LEAKAGE ANSWERS: Whenever $\mathbf{Z}$ sends a leakage query (P_b, L) for $b \in \{0,1\}$ or (P_i^{aux}, L) for $i \in [a]$, to simulate the answer to this query, $\mathcal{S}_\rho$ invokes a subroutine $\widetilde{\mathcal{S}}$. $\widetilde{\mathcal{S}}$ proceeds in three steps: (1) It produces a state translation function T_b or T_A, for leakage queries on P_b or P_i^{aux}, respectively; (2) If the leakage is on P_b, $b \in \{0,1\}$, $\widetilde{\mathcal{S}}$ obtains the leakage on the ideal state of P_b (which consists of its input and output pair (x_b^j, y^j) in the j-th iteration) with respect to the composed leakage function $L \circ T_b$ and obtains output ans; (3) If the leakage is on P_i^{aux}, $i \in [a]$, $\widetilde{\mathcal{S}}$ computes $\mathsf{ans} = L \circ T_A[i]$ (where $T_A[i]$ denotes the function outputting the state of the i-th auxiliary party reconstructed by T_A) without leaking on the ideal state of any party, and (4) forwards the answer ans to $\mathbf{Z}$. After the subroutine call completes, the execution returns to $\mathcal{S}_\rho$ at the state before it receives the leakage query.

The most important step in $\widetilde{\mathcal{S}}$ is constructing, independently of the leakage function L, the state translation function T_b, which translates the ideal-state (x_b^j, y^j) of P_b in the j-th iteration to a simulated real-state of P_b or the state translation function T_A which reconstructs the simulated real-states of all auxiliary parties $P_1^{\mathsf{aux}}, \ldots, P_a^{\mathsf{aux}}$. Below we describe how for any polynomial $p(\cdot)$, to simulate the real-states of P_0 and P_1 in each iteration $j \in [p = p(\lambda)]$ using (x_0^j, x_1^j) and y^j and the real-states of $P_1^{\mathsf{aux}}, \ldots, P_a^{\mathsf{aux}}$ without obtaining ideal inputs; we then observe that the simulation of each party's state depends only on their own ideal view, which leads to the desired construction of T_b, T_A. More precisely, for each iteration, $\mathcal{S}_\rho$ keeps in its state $\mathsf{state}_{\mathcal{S}}$ a sufficiently long random string $\eta^j = \eta^1, \ldots, \eta^j$ such that $\eta^j = \tilde{c}^j \| \eta_1^j \| \eta_2^j$. η^j is used in every invocation of $\widetilde{\mathcal{S}}$ to ensure consistency. Given (x_0^j, x_1^j, y^j) and η^j the real-states of P_0 and P_1 and parties $P_1^{\mathsf{aux}}, \ldots, P_a^{\mathsf{aux}}$, respectively can be simulated by the following procedure:

PROCEDURE $\mathsf{Ideal}^{(f,\rho,p)}(1^\lambda, \mathbf{x}_0^p, \mathbf{x}_1^p \ ; \ \eta^j)$: In the j-th iteration, the real states of P_0, P_1 consist of (x_0^j, x_1^j, y^j) and the states of the two OCL evaluations. The real states of $P_1^{\mathsf{aux}}, \ldots, P_a^{\mathsf{aux}}$ consist of the states of the two OCL evaluations but no input or output.

- *Simulate the states of the first OCL evaluation:*
  For each iteration $1 \le j \le p(\lambda)$, the first OCL evaluation computes ciphertext $c^j = g_1^{(j,x_0^j)}(r) = x_0^j + \mathsf{PRF}(r,j)$. Since the secret $\mathsf{PRF}(r,j)$ is pseudorandom, the output of the evaluation is simulated using a truly random simulated ciphertext $\tilde{c}^j$. Then, the initial states $(\mathsf{init}_L^{j,1}, \mathsf{init}_R^{j,1}, \mathsf{init}_A^{j,1})$ and evaluation states $(\mathsf{evl}_L^{j,1}, \mathsf{evl}_R^{j,1}, \mathsf{evl}_A^{j,1})$ are simulated as:

$$(\widetilde{\mathsf{intl}}_L^{j,1}, \widetilde{\mathsf{evl}}_L^{j,1}) = \mathcal{S}_L(1^\lambda, T, g_1^{j,(x_0^j)}, \tilde{c}^j; \ \eta_1^j)$$
$$(\widetilde{\mathsf{intl}}_R^{j,1}, \widetilde{\mathsf{evl}}_R^{j,1}) = \mathcal{S}_R(1^\lambda, T, \tilde{c}^j; \ \eta_1^j)$$
$$(\widetilde{\mathsf{intl}}_A^{j,1}, \widetilde{\mathsf{evl}}_A^{j,1}) = \mathcal{S}_A(1^\lambda, T, \ \eta_1^j)$$

- *Simulate the states of the second OCL evaluation:*
  For each iteration $1 \le j \le p(\lambda)$, the second OCL evaluation computes $g_2^{(j,c^j,x_1^j)}(r) = f(c^j \oplus \mathsf{PRF}(r,j), x_1^j) = y^j$. Since the simulated ciphertext is $\tilde{c}^j$, the simulated input function for the second evaluation is set to $g_2^{(j,\tilde{c}^j,x_1^j)}$. Then, the initial states $(\mathsf{init}_L^{j,2}, \mathsf{init}_R^{j,2}, \mathsf{init}_A^{j,2})$ and evaluation states $(\mathsf{evl}_L^2, \mathsf{evl}_R^2, \mathsf{evl}_A^{j,2})$ are simulated as:

$$(\widetilde{\mathsf{intl}}_L^{j,2}, \widetilde{\mathsf{evl}}_L^{j,2}) = \mathcal{S}_L(1^\lambda, T, g_2^{j,(\tilde{c}^j,x_1^j)}, y^j; \ \eta_2^j)$$
$$(\widetilde{\mathsf{intl}}_R^{j,2}, \widetilde{\mathsf{evl}}_R^{j,2}) = \mathcal{S}_R(1^\lambda, T, y^j; \ \eta_2^j)$$
$$(\widetilde{\mathsf{intl}}_A^{j,2}, \widetilde{\mathsf{evl}}_A^{j,2}) = \mathcal{S}_A(1^\lambda, T; \ \eta_2^j)$$

- Finally, the procedure $\mathsf{Ideal}^{(f,\rho,p)}(1^\lambda, x_0, x_1 \ ; \ \eta^p)$ outputs the simulated state of P_0 in all p iterations:

$$\widetilde{\mathsf{state}}_0 = \left[(x_0^j, y^j, g_1^{(j,x_0^j)}, \widetilde{\mathsf{intl}}_L^{j,1}, \widetilde{\mathsf{evl}}_L^{j,1}, \widetilde{\mathsf{intl}}_R^{j,2}, \widetilde{\mathsf{evl}}_R^{j,2}) \right]_{j \in [p]},$$

  , the simulated state of P_1,

$$\widetilde{\mathsf{state}}_1 = \left[(x_1^j, y^j, \tilde{c}^j, g_2^{(j,\tilde{c},x_1)}, \widetilde{\mathsf{intl}}_R^{j,1}, \widetilde{\mathsf{evl}}_R^{j,1}, \widetilde{\mathsf{intl}}_L^{j,2}, \widetilde{\mathsf{evl}}_L^{j,2}) \right]_{j \in [p]}$$

  and the simulated states of the auxiliary parties $P_1^{\mathsf{aux}}, \ldots, P_a^{\mathsf{aux}}$

$$\widetilde{\mathsf{state}}_A = \left[(\widetilde{\mathsf{intl}}_A^{j,1}, \widetilde{\mathsf{evl}}_A^{j,1}, \widetilde{\mathsf{intl}}_A^{j,2}, \widetilde{\mathsf{evl}}_A^{j,2}) \right]_{j \in [p]}.$$

We observe that, indeed, the j-th simulated state, $\widetilde{\mathsf{state}}_0[j]$, of P_0 only depends on (x_0^j, y^j) and the random strings $\eta^j = \tilde{\mathbf{c}}^j \| \eta_1^j \| \eta_2^j$, while the j-th simulated state, $\widetilde{\mathsf{state}}_1[j]$, of P_1 only depends on (x_1^j, y^j) and the random strings η^j and that the simulated state $\widetilde{\mathsf{state}}_A$ of the auxiliary parties $P_1^{\mathsf{aux}}, \ldots, P_a^{\mathsf{aux}}$ does not depend on any ideal state and depends only on the random strings η^j. Therefore, in the j-th iteration when $b = 0$, the subroutine $\widetilde{\mathcal{S}}$ outputs the function $T_0 = T_0^{\eta^j}$ (with the random string $\eta^{\mathbf{j}}$ hardwired in) that on inputs (x_0^j, y^j) computes $\widetilde{\mathsf{state}}_0[j]$ (the j-th tuple contained in $\widetilde{\mathsf{state}}_0$) as procedure Ideal does; in the j-th iteration when $b = 1$, $\widetilde{\mathcal{S}}$ outputs $T_1 = T_1^{\eta^j}$ that on inputs (x_1^j, y^j) computes $\widetilde{\mathsf{state}}_1[j]$ (the j-th tuple contained in $\widetilde{\mathsf{state}}_1$) as procedure Ideal does. Finally, in the j-th iteration, the subroutine $\widetilde{\mathcal{S}}$ outputs the function $T_A = T_A^{\eta^j}$ that does not take input and outputs $\widetilde{\mathsf{state}}_A[j]$ (the j-th tuple contained in $\widetilde{\mathsf{state}}_A$) as procedure Ideal does. This completes the description of the simulator.

Validity of simulator $\mathcal{S}_\rho$. By construction, the simulator $\mathcal{S}_\rho$ simulates the protocol messages to $\mathbf{Z}$ perfectly. Therefore, to show that the environment $\mathbf{Z}(1^\lambda, z)$ cannot distinguish its interaction with $\mathcal{S}_\rho$ using $\mathcal{F}^{f,\infty}_{\mathsf{2LTC\text{-}AUX}}$ from the interaction with $\mathcal{D}$ using ρ, it suffices to show that all the leakage answers obtained in the ideal world are indistinguishable from those obtained in the real world, as long as the total amount of leakage from each party in each time period is $\ell(\lambda)$-bounded.

As described above in the construction of $\widetilde{\mathcal{S}}$, the procedure for simulating the leakage answers can be viewed as first simulating the entire states $(\widetilde{\mathsf{state}}_0, \widetilde{\mathsf{state}}_1) \leftarrow \mathsf{Ideal}^{(f,\rho,p)}(1^\lambda, \mathbf{x}_0^p, \mathbf{x}_1^p)$ of P_0 and P_1 in $p(\lambda)$ iterations using consistent randomness, and then applying an ℓ-leakage attack on the simulated states. Thus, it suffices to prove that the simulated states $(\widetilde{\mathsf{state}}_0, \widetilde{\mathsf{state}}_1)$ are indistinguishable from the real states $(\mathsf{state}_1, \mathsf{state}_1)$ to all environments that choose the inputs $(\mathbf{x}_0^p, \mathbf{x}_1^p)$ of P_0 and P_1 and obtain at most $\ell(\lambda)$ bits of leakage, separately on the state of each party in each time period—we refer to such an environment as an *ℓ-bounded continual leakage adversary.*

Formally, let $(\mathsf{state}_0, \mathsf{state}_1) \leftarrow \mathsf{Real}^{(f,\rho,p)}(1^\lambda, \mathbf{x}_0, \mathbf{x}_1)$ denote the random process that samples the states of P_0 and P_1 in p honest executions of ρ with inputs $\mathbf{x}_0^p, \mathbf{x}_1^p$. To show the correctness of $\mathcal{S}_\rho$, it suffices to prove the following claim.

Claim 11. *Assume that Λ is a ℓ-continual-leakage-resilient strong OCL scheme. Then, for every polynomial $p(\cdot)$, the following two ensembles are indistinguishable to all ℓ-bounded continual leakage adversaries.*

- $\{\mathsf{Ideal}^{(f,\rho,p(\lambda))}(1^\lambda, \mathbf{x}_0^p, \mathbf{x}_1^p)\}_{\lambda,n\in\mathbb{N},\mathbf{x}_0^p,\mathbf{x}_1^p\in\{0,1\}^{n\cdot p}}$,
- $\{\mathsf{Real}^{(f,\rho,p(\lambda))}(1^\lambda, \mathbf{x}_0^p, \mathbf{x}_1^p)\}_{\lambda,n\in\mathbb{N},\mathbf{x}_0^p,\mathbf{x}_1^p\in\{0,1\}^{n\cdot p}}$.

Proof. We consider a sequence of hybrid experiments hyb_0 to hyb_4; we denote by $\mathsf{hyb}_i(1^\lambda, \mathbf{x}_0^p, \mathbf{x}_1^p)$ the output of the hybrid hyb_i.

Hybrid hyb_0: This hybrid runs $p = p(\lambda)$ honest executions of the protocol ρ with inputs $\mathbf{x}_0^p, \mathbf{x}_1^p$ and outputs the honest states of $(P_0, P_1, P_1^{\mathsf{aux}}, \ldots, P_a^{\mathsf{aux}})$ in each iteration. Thus, the output distributions of hyb_0 and Real are identical. The states of $(P_0, P_1, P_1^{\mathsf{aux}}, \ldots, P_a^{\mathsf{aux}})$ are

$$\mathsf{state}_0 = \left[(x_0^j, y^j, g_1^{(j,x_0^j)}, \mathsf{init}_L^{j,1}, \mathsf{evl}_L^{j,1}, \mathsf{init}_R^{j,2}, \mathsf{evl}_R^{j,2})\right]_{j\in[p]} ,$$

$$\mathsf{state}_1 = \left[(x_1^j, y^j, \tilde{c}^j, g_2^{(j,\tilde{c}^j,x_1^j)}, \mathsf{init}_R^{j,1}, \mathsf{evl}_R^{j,1}, \mathsf{init}_L^{j,2}, \mathsf{evl}_L^{j,2})\right]_{j\in[p]} ,$$

$$\mathsf{state}_A = \left[(\mathsf{init}_A^{j,1}, \mathsf{evl}_A^{j,1}, \mathsf{init}_A^{j,2}, \mathsf{evl}_A^{j,2})\right] .$$

Hybrid hyb_1: This hybrid proceeds identically to hyb_0, except for the following: In the j-th iteration of hyb_0, according to ρ, P_1 uses the output of the first OCL evaluation $\tilde{c}^j$ to construct the input function $g^{(j,\tilde{c}^j,x_1^j)}$ for the second OCL evaluation. In hyb_1, P_1 additionally receives the correct output $c^j = \mathsf{PRF}(r, j) + x_0^j$ of the first OCL evaluation, and uses c^j to construct the input function $g^{(j,c^j,x_1^j)}$ for the second OCL evaluation; in particular, $(\mathsf{evl'}_L^{j,2}, \mathsf{evl'}_R^{j,2}, \mathsf{evl'}_A^{j,2})$ correspond to the input c^j. (The output of the first OCL evaluation is ignored). The states of $(P_0, P_1, P_1^{\mathsf{aux}}, \ldots, P_a^{\mathsf{aux}})$ in this hybrid are

$$\mathsf{state}_0^{\mathsf{hyb}_1} = \left[(x_0^j, y^j, g_1^{(j,x_0^j)}, \mathsf{init}_L^{j,1}, \mathsf{evl}_L^{j,1}, \mathsf{init}_R^{j,2}, \boxed{\mathsf{evl'}_R^{j,2}})\right],$$

$$\mathsf{state}_1^{\mathsf{hyb}_1} = \left[(x_1^j, y^j, \boxed{c^j, g_2^{(j,c^j,x_1^j)}}, \mathsf{init}_R^{j,1}, \mathsf{evl}_R^{j,1}, \mathsf{init}_L^{j,2}, \boxed{\mathsf{evl'}_L^{j,2}})\right],$$

$$\mathsf{state}_A^{\mathsf{hyb}_1} = \left[(\mathsf{init}_A^{j,1}, \mathsf{evl}_A^{j,1}, \mathsf{init}_A^{j,2}, \boxed{\mathsf{evl'}_A^{j,2}})\right].$$

By the correctness of Λ, in each iteration the OCL output $\tilde{c}^j$ equals to the correct output c^j, with overwhelming probability. Therefore, the output distributions of hyb_0 and hyb_1 are statistically close.

Hybrid hyb_2: This hybrid proceeds identically to hyb_1, except that in each iteration, the first OCL evaluation in the execution of ρ is simulated (and only the second OCL evaluation is carried out honestly). More precisely, the preprocessing stage of ρ is executed as in hyb_1, sampling random string r and producing initial states $\Phi_0 = (\mathsf{init}_L^1, \mathsf{init}_R^2)$, $\Phi_1 = (\mathsf{init}_R^1, \mathsf{init}_L^2)$, $\Phi_A = (\mathsf{init}_A^1, \mathsf{init}_A^2)$. Then, in iteration j, instead of performing the first evaluation according to ρ, hyb_2 simulates the states of the first OCL evaluation by invoking the simulator $\mathcal{S}$ of Λ with honest input $g_1^{(j,x_0^j)}$ and output $c^j = \mathsf{PRF}(r,j) + x_0^j$. (The initial states $\mathsf{init}_L^1, \mathsf{init}_R^1, \mathsf{init}_A^1$ sampled by the preprocessing stage are ignored.) For every iteration $j \in [p]$ we have:

$$(\widetilde{\mathsf{init}}_L^{j,1}, \widetilde{\mathsf{evl}}_L^{j,1}) = \mathcal{S}_L(1^\lambda, T, g_1^{(j,x_0^j)}, c^j;\ \eta_1^j)$$
$$(\widetilde{\mathsf{init}}_R^{j,1}, \widetilde{\mathsf{evl}}_R^{j,1}) = \mathcal{S}_R(1^\lambda, T, c^j;\ \eta_1^j)$$
$$(\widetilde{\mathsf{init}}_A^{j,1}, \widetilde{\mathsf{evl}}_A^{j,1}) = \mathcal{S}_A(1^\lambda, T;\ \eta_1^j)$$

Next, in each iteration, the second OCL evaluation is carried out honestly as in ρ, with input function $g_2^{(j,c^j,x_1^j)}$ and initial states $\mathsf{init}_L^{j,2}, \mathsf{init}_R^{j,2}$. Finally, hyb_2 outputs the simulated states of $(P_0, P_1, P_1^{\mathsf{aux}}, \ldots, P_a^{\mathsf{aux}})$:

$$\mathsf{state}_0^{\mathsf{hyb}_2} = \left[(x_0^j, y^j, g_1^{(j,x_0^j)}, \boxed{\widetilde{\mathsf{init}}_L^{j,1}, \widetilde{\mathsf{evl}}_L^{j,1}}, \mathsf{init}_R^{j,2}, \mathsf{evl'}_R^{j,2})\right]_{j\in[p]},$$

$$\mathsf{state}_1^{\mathsf{hyb}_2} = \left[(x_1^j, y^j, c^j, g_2^{(j,c^j,x_1^j)}, \boxed{\widetilde{\mathsf{init}}_R^{j,1}, \widetilde{\mathsf{evl}}_R^{j,1}}, \mathsf{init}_L^{j,2}, \mathsf{evl'}_L^{j,2})\right]_{j\in[p]},$$

$$\mathsf{state}_A^{\mathsf{hyb}_2} = \left[(\boxed{\widetilde{\mathsf{init}}_A^{j,1}, \widetilde{\mathsf{evl}}_A^{j,1}}, \mathsf{init}_A^{j,2}, \mathsf{evl'}_A^{j,2})\right]_{j\in[p]}.$$

The only difference between hyb_1 and hyb_2 is that in the former the state corresponding to the first OCL evaluation is generated honestly, whereas, in the second, it is simulated. Recall that ℓ-bounded cotinual leakage adversaries leak at most ℓ bits from the states of P_0, P_1, $P_1^{\mathsf{aux}}, \ldots, P_a^{\mathsf{aux}}$ separately in each time period; by the construction of ρ, they leak at most ℓ bits from the states of the left component (emulated by P_0) and right component (emulated by P_1) and each of the auxiliary components (emulated by $P_1^{\mathsf{aux}}, \ldots, P_a^{\mathsf{aux}}$) of the first OCL evaluation separately. Since all the other elements in the state of $(P_0, P_1, P_1^{\mathsf{aux}}, \ldots, P_a^{\mathsf{aux}})$ can be simulated, it follows from the ℓ-continual leakage resilience of Λ that the outputs of hyb_1 and hyb_2 are indistinguishable to all ℓ-bounded continual leakage adversaries.

Hybrid hyb_3: This hybrid proceeds identically to hyb_2 except that in each iteration, the second OCL evaluation in the execution of ρ is also simulated. More precisely, in iteration j, after running the preprocessing stage and the simulation of the first OCL evaluation as in hyb_2, the state of the second OCL evaluation is simulated by invoking the simulator $\mathcal{S}$ of Λ with honest input $g_2^{(j,c^j,x_1^j)}$ and output y^j. (The initial states $(\mathsf{init}_L^2, \mathsf{init}_R^2, \mathsf{init}_A^2)$ sampled by the preprocessing stage are ignored.) For each iteration $j \in [p]$:

$$(\widetilde{\mathsf{init}}_L^{j,2}, \widetilde{\mathsf{evl}}_L^{j,2}) = \mathcal{S}_L(1^\lambda, T, g_2^{(j,c^j,x_1^j)}, y^j;\ \eta_2^j)$$
$$(\widetilde{\mathsf{init}}_R^{j,2}, \widetilde{\mathsf{evl}}_R^{j,2}) = \mathcal{S}_R(1^\lambda, T, y^j;\ \eta_2^j)$$
$$(\widetilde{\mathsf{init}}_A^{2}, \widetilde{\mathsf{evl}}_A^{2}) = \mathcal{S}_A(1^\lambda;\ \eta_2)$$

Finally, hyb_3 outputs the states of $(P_0, P_1, P_1^{\mathsf{aux}}, \ldots, P_a^{\mathsf{aux}})$ as

$$\mathsf{state}_0^{\mathsf{hyb}_3} = \left[(x_0^j, y^j, g_1^{(j,x_0^j)}, \widetilde{\mathsf{init}}_L^{j,1}, \widetilde{\mathsf{evl}}_L^{j,1}, \boxed{\widetilde{\mathsf{init}}_R^{j,2}, \widetilde{\mathsf{evl}}_R^{j,2}})\right]_{j\in[p]},$$

$$\mathsf{state}_1^{\mathsf{hyb}_3} = \left[(x_1^j, y^j, c^j, g_2^{(j,c^j,x_1^j)}, \widetilde{\mathsf{init}}_R^{j,1}, \widetilde{\mathsf{evl}}_R^{j,1}, \boxed{\widetilde{\mathsf{init}}_L^{j,2}, \widetilde{\mathsf{evl}}_L^{j,2}})\right]_{j\in[p]},$$

$$\mathsf{state}_A^{\mathsf{hyb}_3} = \left[(\widetilde{\mathsf{init}}_A^{j,1}, \widetilde{\mathsf{evl}}_A^{j,1}, \boxed{\widetilde{\mathsf{init}}_A^{j,2}, \widetilde{\mathsf{evl}}_A^{j,2}})\right]_{j\in[p]}.$$

It follows from the same argument for hybrid hyb_2 that by the ℓ-continual-leakage resilience of Λ, the output distributions of hyb_2 and hyb_3 are indistinguishable to all ℓ-bounded continual leakage adversaries.

Hybrid hyb_4: This hybrid proceeds identically to hyb_3 except for the following: Note that in hyb_3, in each iteration, the honest initial states sampled by the preprocessing stages are ignored, and only the pseudorandom string $\mathsf{PRF}(r,j)$ is used (implicitly when using $c^j = x_0^j \oplus \mathsf{PRF}(r,j)$) for later simulation of the OCL evaluations. In hyb_4, in each iteration, instead of using $\mathsf{PRF}(r,j)$ to compute the output of the first OCL evaluation $c^j = x_0^j \oplus \mathsf{PRF}(r,j)$, simply omit the preprocessing stage and sample directly a random string $\tilde{c}^j$ as the output of the first OCL evaluation. Due to the security of the pseudorandom function PRF, we have that $\{c^1, \ldots, c^p\}$ and $\{\tilde{c}^1, \ldots, \tilde{c}^p\}$ are computationally indistinguishable. Thus, the simulated states output by hyb_3 and hyb_4 are also computationally indistinguishable. Furthermore, notice that hyb_4 proceeds identically to the Ideal procedure; we have that the output distributions of hyb_4 and Ideal are identical.

Finally, by a hybrid argument, we conclude that the honest states of $P_0, P_1, P_1^{\mathsf{aux}}, \ldots, P_a^{\mathsf{aux}}$ in ρ are indistinguishable to the simulated states by Ideal to all ℓ-bounded continual leakage adversaries. □

□

D.3 From Leakage-Tolerant Protocols back to Strong OCL Compilation

Our construction of a two-party, a-auxiliary party continual leakage tolerant protocol ρ, in the input-independent pre-processing model, crucially relies on the strong continual leakage-resilience property of the underlying OCL scheme. In this section, we show that this property is, in fact, necessary. In particular, given any two-party, a-auxiliary party continual leakage tolerant protocol secure without corruptions, we can construct a strong continual $2+a$-component OCL scheme.

Theorem 7. *Assume that, for every efficiently computable deterministic two-input function $f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$, there is a 2-party, a-auxiliary party protocol ρ that strongly UC-emulates the functionality $\mathcal{F}^{f,\infty}_{\mathsf{2LTC\text{-}AUX}}$ under ℓ-bounded leakage in the $(\mathcal{F}_{\mathsf{LSC}}, \mathcal{F}_{\mathsf{LFS}})$-hybrid model, when no party is corrupted. Then, there is a $2+a$-component ℓ-continual-leakage-resilient strong OCL scheme Λ. Furthermore, if ρ has perfect correctness, so does Λ.*

The high-level intuition for the theorem was sketched in the introduction. We defer a detailed proof to the full version.

E Multi-Party Leakage-Tolerant Protocols Secure Against Malicious Corruptions

In the previous section, we restricted attention to the case of two-party LTC protocols with auxiliary parties (2PLTC-AUX). In this section, we show how to obtain multi-party LTC protocols (MPLTC) by leveraging our two-party protocols with auxiliary parties and existing MPLTC that are based on input-*dependant* processing (IDP). We first give the relevant definitions, and then state and prove our results.

The MPLTC functionality. The ideal functionality corresponding to MPLTC of a given function f is defined analogously to the two-party case (as in Section A.2). Also, like in the two-party case, we may consider either a one-time version or a many-time version in the model of continual leakage. Below we give the more general functionality for the continual case.

Functionality $\mathcal{F}^{f,\infty}_{\mathsf{MLTC}}$

Running with parties $(P_1, \ldots, P_m)$ and an adversary $\mathcal{S}$, $\mathcal{F}^{f,\infty}_{\mathsf{MLTC}}$ operates as follows:

- Given inputs $(x^i_1, \ldots, x^i_m)$ from $(P_1, \ldots, P_m)$, it computes $(y^i_1, \ldots, y^i_m) = f(x^i_1, \ldots, x^i_m)$, notifies $\mathcal{S}$ of the computation, and once $\mathcal{S}$ allows, sends y^i_j to P_j.
- The ideal leaky state of any party P_j, after obtaining the i^{th} input and before obtaining $i+1^{st}$, is (x^i_j, y^i_j).

Figure 9: The multi-party continual LTC Functionality

LTC with IDP. We consider LTC protocols where parties are assisted by a leak-free preprocessing functionality $\mathcal{F}_{\mathsf{P}}$ that captures an input-independent preprocessing (IIP) phase (similarly to $\mathcal{F}_{\mathsf{LFS}}$) and a related input-dependant processing (IDP) phase. Specifically, $\mathcal{F}_{\mathsf{P}}$ produces initial states $(\mathsf{init}_1, \ldots, \mathsf{init}_m)$ sampled from a fixed distribution Δ, and hands them to the parties, possibly together with a public sampling state pub_Δ. Then, whenever a party P_i obtains an input $x = x_i^j$ in the j^{th} execution, it locally invokes $\mathcal{F}_\mathsf{P}$, which executes a randomized input processing procedure $\Pi(x, \mathsf{pub}_\Delta)$. The procedure, given the input x and public state pub_Δ of the initial state sampler, returns a processed input $\bar{x}$.

When considering IDP (as in [BGJK12]), it is natural to assume that inputs never reach the leaky states of parties, and go directly through the processing functionality, i.e., the leaky state of P_i with respect to $\mathcal{F}_\mathsf{P}$ only contains the processed input. Since we are interested in constructing leakage-tolerant protocols, and avoid assuming that the input is not leaked on, we will settle for a weaker $\mathcal{F}_\mathsf{P}$ functionality that does include the parties' inputs, as part of the leaky state. It can be verified that leakage-tolerant (or fully leakage-resilient) protocols relying on the strong version of $\mathcal{F}_\mathsf{P}$ (in particular, [BGJK12]) imply leakage-tolerant (and necessarily not fully leakage-resilient) protocols relying on the weaker form of $\mathcal{F}_\mathsf{P}$, which exposes inputs to leakage.

Functionality $\mathcal{F}_\mathsf{P}^{\Delta,\Pi}$

Running with parties $P_1, \ldots, P_m$ and an adversary $\mathcal{A}$, $\mathcal{F}_\mathsf{P}$ operates as follows:

- When activated with security parameter λ, and possibly length parameter n, sample initial states $(\mathsf{init}_1, \ldots, \mathsf{init}_m; s_\Delta) \leftarrow \Delta(1^\lambda, 1^n)$, and a corresponding public state s_Δ.
- **Initial state distribution:** When invoked for the first time, send $(\mathsf{init}_i, s_\Delta)$ to each P_i.
- **Input processing:** When invoked by P_i, during the j^{th} execution, with input x_i^j, return $\bar{x}_i^j \leftarrow \Pi(1^\lambda, x_i^j, s_\Delta)$. In subsequent invocations, during the j^{th} execution, return $\perp$.
- The leaky state of P_i initially includes $(\mathsf{init}_i, s_\Delta)$. At any point, P_i can instruct $\mathcal{F}_\mathsf{P}$ to erase init_i from its state (the public state s_Δ need not be erased). In addition, the leaky state of P_i during the j^{th} execution includes $(x_i^j, \bar{x}_i^j)$.

Figure 10: The preprocessing functionality

The BGJK protocol. Boyle et al. [BGJK12] construct a continual. fully leakage-resilient MPC protocol with input-dependant processing, based on standard intractability assumptions and a common reference string. Their protocol accounts for m-party functions, for any sufficiently large m, and can withstand $(1-\epsilon)m$ malicious corruptions for an arbitrarily small constant ϵ.

Their protocol is constructed under the assumption that inputs never reach the leaky states of parties, and go directly through the input processing procedure. As noted above, this implies a leakage-tolerant protocol in our input-processing model (i.e., the $\mathcal{F}_\mathsf{P}$-hybrid model), where the inputs can be leaked on. Also, their protocol is not formally presented within the leaky UC framework; however, it can be extended to this setting. Finally, we note that, in the setting of no corruptions, one could consider a simple no corruption variant of their protocol (based on honest OCL decryption). Such a variant would only require fully homomorphic encryption and would not require a common reference string.

Overall, casting their protocol into our setting of leakage tolerance, we have:

Theorem 8 (follows from [BGJK12])**.**

1. *Assume the existence of m-component OCL against ℓ-bounded continual leakage and fully homomorphic encryption. Then, for any m-party f, there exists a protocol π that (strongly) UC-emulates $\mathcal{F}_\mathsf{MLTC}^\infty$ in the $(\mathcal{F}_\mathsf{P}, \mathcal{F}_\mathsf{LSC})$-hybrid model, when no party is corrupted.*

2. *In the CRS mode, and assuming also WLTC protocols, non-interactive extractable equivocal commitments, and adaptive NIZKs, there exists a protocol π' that (strongly) UC-emulates $\mathcal{F}^{\infty}_{\mathsf{MLTC}}$ in the $(\mathcal{F}_{\mathsf{P}}, \mathcal{F}_{\mathsf{LSC}})$-hybrid model, when at most $(1-\epsilon)m$ parties are corrupted, where ϵ is an arbitrarily small constant and $m = \lambda^{\Omega(1)}$.*[4]

Removing input-dependent processing using 2PLTC-AUX. To obtain MPLTC protocols, we provide two transformations from MPLTC protocols that rely on IDP to protocols that instead rely on 2PLTC-AUX and input-*independent* processing (IIP). Both transformations rely on 2PLTC-AUX secure in the case of no corruptions. The first is a generic transformation for the case that no party is corrupted. The second transformation is for the case that all but a small constant fraction of the parties are corrupted; this transformation starts from the BGJK protocol (taking advantage of specific properties of its IDP). The transformations apply for both the one-time and the continual settings.

Theorem 9.

1. *Assume the existence of a protocol ρ that strongly UC-emulates the functionality $\mathcal{F}^{f}_{\mathsf{2LTC\text{-}AUX}}$ under ℓ-bounded continual leakage, with a auxiliary parties, when no party is corrupted. Let $m \geq a+2$ and let π be any m-party protocol that (strongly) UC-emulates the m-party functionality $\mathcal{F}^{f}_{\mathsf{MLTC}}$ under ℓ-bounded continual leakage in the $\mathcal{F}_{\mathsf{P}}$-hybrid model, with no corruptions. Then there is a protocol ρ that strongly UC-emulates the m-party functionality $\mathcal{F}^{f,\infty}_{\mathsf{MLTC}}$ under ℓ-bounded continual leakage in the $\mathcal{F}_{\mathsf{LFS}}$-hybrid model, with no corruptions.*

2. *Assume the existence of a protocol ρ that strongly UC-emulates the functionality $\mathcal{F}^{f}_{\mathsf{2LTC\text{-}AUX}}$ under ℓ-bounded continual leakage, with a auxiliary parties, when no party is corrupted. Given the assumptions of Theorem 8, we have that the BGJK Protocol π' from Theorem 8 strongly UC-emulates $\mathcal{F}^{f}_{\mathsf{MLTC}}$ under ℓ-bounded continual leakage in the in the $\mathcal{F}_{\mathsf{P}}$-hybrid model in the presence of N number of malicious corruptions. Under the same assumptions, there exists a protocol ρ that strongly UC-emulates $\mathcal{F}^{f,\infty}_{\mathsf{MLTC}}$ under ℓ-bounded continual leakage in the $\mathcal{F}_{\mathsf{LFS}}$-hybrid model, in the presence of the same number N of malicious corruptions*

By instantiating the 2PLTC-AUX with the protocols constructed in Section D, which require a auxiliary parties, we deduce MPLTC protocols based on IIP in the continual setting.

Our tranformation for the no-corruption setting. The high-level idea is simple: the input processing of the MPLTC can be performed online, and under leakage, jointly by $a+2$ parties by utilizing the protocol ρ (of Section D) that strongly UC-emulates $\mathcal{F}^{f}_{\mathsf{2LTC\text{-}AUX}}$ under ℓ-bounded continual leakage, with a auxiliary parties, when no party is corrupted. Namely, to process the input x_i of a given party P_i, it will use the help of another party $P_{i'}$ who holds input and a auxiliary parties without input. P_i and $P_{i'}$ each sample independently a long enough random string r_i and $r_{i'}$, respectively, and all $a+2$ parties participate in 2PLTC-AUX to compute the two-party function $g((x_i, \mathsf{pub}, r_i), r_{i'})$ that computes the processing function $\bar{x}_i = \Pi(x_i, \mathsf{pub}; \mathsf{Ext}(r_i, r_{i'}))$ where the randomness $r = \mathsf{Ext}(r_i, r_{i'})$ is derived from the two random strings using a two-source extractor (e.g., inner product). Once, each party obtains this processed input, the parties then run the original MPLTC. Intuitively, by the guarantees of two-source extraction, provided that there is only bounded separate leakage on each of the random strings, the randomness $r = \mathsf{Ext}(r_i, r_{i'})$ is statistically independent of the leakage, achieving the same effect as leakage-free input preprocessing.

[4]More accurately, in this case, the fully homomorphic encryption should also have a *certifiable randomness* property (satisfied by most known schemes).

Extending to the semi-honest corruptions setting. The above procedure is only secure assuming that $P_{i'}$ and all auxiliary parties assisting parties P_i and $P_{i'}$ are honest. Indeed, assuming $P_{i'}$ or one of the auxiliary parties is (even semi-honestly corrupted) the adversary may be able to leak jointly on $r_{i'}, r_i$ since the security guarantees for our 2PLTC-AUX protocols hold only when no parties are corrupted.

Our approach towards overcoming this problem in the semi-honest setting is to have each party P_i jointly process its input with all possible subsets of $a+1$ other parties, and then aggregate the processed inputs, some of which were computed with dishonest parties, into one processed input. Since the total number of corrupted parties is bounded, we are guaranteed that at least one subset is entirely honest and thus the aggregated input will be safe to use. While we do not know how to do this in general, we observe that the input-processing in the BGJK protocol possesses some additional properties, which give rise to such an approach. Specifically, in the BGJK protocol IDP function $\Pi(x_i, \mathsf{pk}; r) := \mathsf{Enc}_{\mathsf{pk}}(x_i; r)$ simply samples an encryption of the input x_i under a public key pk for a fully-homomorphic encryption scheme, where pk is determined as part of the input-independent processing (in particular, there is no leakage on the randomness for the encryption).

To implement the above idea, for each subset $\mathcal{S} \subseteq [m] \setminus \{i\}$ (where m is the number of parties participating in the protocol) of size $a+1$ we let each P_i jointly compute with all parties P_j for $j \in \mathcal{S}$, an encryption $\mathsf{c}_{\mathcal{S}}$ of zero, where the randomness is computed by a two-source extractor, as above. Then, P_i aggregates all these ciphers by adding them together to a new zero encryption $\mathsf{c} = \sum_{\mathcal{S}\subseteq[m]\setminus\{i\},|\mathcal{S}|=a+1} \mathsf{c}_{\mathcal{S}}$, and uses them to get a fresh encryption, c_{x_i} of his input x_i, by encrypting x_i under leakage (and thus non-securely) and adding to it the aggregated zero encryption c. It can be shown that, in known fully homomorphic encryption schemes, the eventual encryption of x_i is semantically-secure provided that any one of the zero encryptions $\mathsf{c}_{\mathcal{S}}$ is.

We note that the noise distribution of c_{x_i} resulting from the above implementation not statistically or computationally close to the noise distribution of encryptions of x_i returned by the ideal pre-processing functionality $\mathcal{F}_{\mathsf{P}}$. However, since the total number of corrupted parties is bounded, there must be some set of completely honest parties $\mathcal{S}^*$, and so we have the property that even under semi-honest corruptions, there is some encryption $\mathsf{c}_{\mathcal{S}^*}$ corresponding to the set S^* that can be replaced with a random encryption of a non-zero value during the simulation. This is sufficient for the proofs of BGJK to go through. We elaborate further on this in the full verison.

Achieving security in the malicious corruptions setting. In the malicious setting, the approach outlined above fails due to the following issues: (1) A malicious party can affect the correctness of the subcomputations and in particular, the encryptions $\mathsf{c}_{\mathcal{S}}$ for sets $\mathcal{S}$ which contain corrupted parties may not be valid encryptions of 0. This, in turn, will affect the correctness of the entire computation. (2) For the security analysis of the BGJK protocol π', the preprocessing must output not only an encryption of P_i's input x_i, but also an NIZK of plaintext knowledge, which allows the simulator to extract the party's input. However, P_i cannot output an NIZK of plaintext knowledge for its ciphertext $\mathsf{c}_{x_i} = \mathsf{Enc}_{\mathsf{pk}}(x_i; r) + \mathsf{c}$, since he does not know the randomness used to compute c. In the following we outline how to solve both of these problems.

- **Addressing** (1)**:** Instead of using the 2PLTC-AUX protocol to compute $\mathsf{Enc}_{\mathsf{pk}}(0; r)$, we run the 2PLTC-AUX protocol with each subset $\mathcal{S}$ of parties to compute both $\mathsf{c}_{\mathcal{S}} = \mathsf{Enc}_{\mathsf{pk}}(0; r)$ as well as an NIZK proof that $\mathsf{c}_{\mathcal{S}}$ is a valid encryption of 0.

- **Addressing** (2)**:** P_i computes $\mathsf{c} = \sum_{\mathcal{S}\subseteq[m]\setminus\{i\},|\mathcal{S}|=a+1} \mathsf{c}_{\mathcal{S}}$ and adds to it a fresh encryption, $\mathsf{Enc}_{\mathsf{pk}}(x_i; r_i)$ of its own input x_i, yielding c_{x_i}. In addition, P_i computes an NIZK of knowledge that c_{x_i} is a sum of $X = 1 + \binom{m-1}{a+1}$ encryptions $\mathsf{c}_1, \mathsf{c}_2, \ldots, \mathsf{c}_X$ such that (1) P_i knows the plaintext and randomness for c_1 (2) For each $\mathsf{c}_2, \ldots, \mathsf{c}_X$, P_i knows a correctly verifiying NIZK proof π for the statement that c_i is a valid encryption of 0.

Formalizing and proving the security of the above modifications is fairly straightforward. We defer a full analysis to the full version.