Leakage-Tolerant Computation with Input-Independent Preprocessing

Nir Bitansky[1], Dana Dachman-Soled[2], and Huijia Lin[3]

[1] Tel Aviv University
nirbitan@tau.ac.il
[2] University of Maryland
danadach@ece.umd.edu
[3] University of California, Santa Barbara
rachel.lin@cs.ucsb.edu

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. So far, only a handful of specific two-party functionalities have been successfully realized under the notion. General functionalities were only realized under weaker security notions [Boyle, Garg, Jain, Kalai, and Sahai, Crypto13], or relying on leakage-immune input-processing, which needs to be repeated for each and every execution [Boyle, Goldwasser, Jain, Kalai, STOC12].
We construct leakage-tolerant multi-party computation protocols for *general functions*, relying on *input-independent preprocessing* that is performed once and for-all. The protocols tolerate continual leakage, throughout an unbounded number of executions, provided that leakage is bounded within any particular execution. 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 two-party leakage-tolerant protocols with input-independent preprocessing are essentially *equivalent* to two-component OCL compilers satisfying certain strong properties. We then show how to construct such *strong OCL compilers* in the plain model, with the help of $O(1)$ auxliary components.

1 Introduction

Secure Multiparty Computation (MPC) [Yao82, GMW87, BGW88, CCD88] is a central facet of modern cryptography. MPC protocols allows m mutually distrustful parties to securely compute any function $f(\bar{x})$ of their private inputs $\bar{x} = (x_1, \ldots, x_m)$. The security of such a protocol π, which guarantees privacy and correctness to honest parties, is 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*, often referred as an *ideal functionality*, that performs the computation for the parties.

Protocols in the traditional MPC model crucially rely on the assumption that *the internal computation state of honest parties is kept completely secret from the attackers,* and the sole way of affecting honest parties and gaining information regarding their secret state is through the communication interface. However, in reality, the attackers may design their own interfaces, via a myriads of side-channels (e.g. timing, radiation, etc., see [Sta09]), and learn information—termed *leakage*—about the secret state of honest parties. This growing threat has spurred a large body of research devoted to the development of *leakage-resilient cryptography* (see [ADW09] for a survey).

Following this line of research, the works of [GJS11, BCH12] initiated the study of secure interactive protocols in the presence of leakage.In this setting, the adversary can obtain leakage on honest parties' secret states (in addition to controlling the corrupted parties), modelled as the outputs of arbitrary leakage functions, chosen adaptively by the adversary during protocol execution.

A fundamental question concerns the level of security that can be achieved in this model. The common aim in leakage-resilient cryptography is to achieve the same security properties as in the traditional attack model, where there is no leakage. In the context of protocols, such a guarantee means that leakage on the state of honest parties causes no degradation of security; namely, 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 strong guarantee is inherently *impossible* if the real world adversary can directly leak on parties' inputs.

The model in focus of this work: leakage-tolerance. Acknowledging that direct leakage on parties' inputs is often unavoidable, [GJS11, BCH12] put forth the model of *leakage-tolerance* (rather than resilience) that aims to achieve the "best-possible" security guarantee in this scenario. Intuitively, leakage-tolerance means that ℓ-bits of leakage on an honest party's internal state, including inputs, messages, and randomness, "translate" to at most ℓ-bits of leakage only on its private input and output. More precisely, under the real-ideal paradigm, it is required that a real-world executions subject to ℓ bits of leakage on honest parties' secret states, can be simulated by an ideal-world execution, subject to ℓ bits of leakage on the honest parties' ideal states. [4] Here, the ideal state of a party is specified as a part of the description of the ideal functionality; in the most natural (and best possible) setting, it contains only the party's private input and output.

[4] [GJS11] proposes a weaker notion of leakage-tolerance for zero-knowledge protocols that allows a bigger leakage budget, $(1+\varepsilon)\ell$ bits, in the ideal-world execution. In this work, we follow the more stringent notion, with $\varepsilon = 0$, proposed in [BCH12].

Prior Work. Boyle, Goldwasser, Jain and Kalai [BGJK12] circumvent the impossibility of leakage-resilient protocols, by relying on *leakage-free input-processing*. Informally, in every execution, a leakage-free input processing phase is executed first to encrypt each party's input, and only the ciphertexts are delivered to the parties; thus, the inputs themselves are never exposed to direct leakage. In addition, they rely on a leakage-free input-*independent* preprocessing phase that is *performed once for all executions*. In this setting, they construct MPC protocols for general m-party functions, which are resilient under continual leakage, if a constant fraction of parties is uncorrupted, the number of parties m is polynomial in the security parameter λ, and the leakage on any party within any single execution is a-priori bounded.

While leakage-free input processing leads to a strong guarantee of leakage-resilience; however, it significantly deviates from the regime of leakage-tolerance. Indeed, the main narrative behind leakage-tolerance is that leakage on inputs may be unavoidable; in particular, leakage-free input processing in each and every execution may be impossible or expensive to impose.

Boyle, Goldwasser, Jain, Kalai and Sahai [BGJ$^+$13] construct MPC protocols for general deterministic functions that achieve *joint-state* leakage-tolerance (they do not rely on any leakage-free phase and do not require an a-priori bound on the amount of leakage). Specifically, they consider ideal functionalities where the ideal leaky state of each individual party includes the *joint inputs and outputs of all parties*. Roughly, 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 leakage function of the *joint* inputs and outputs of all parties.

While certainly meaningful, the joint-state model does not seem to capture the best-possible tolerance in the face of leakage. Indeed, in the real world, parties maybe physically separated; thus allowing the real-world adversary only separate leakage on the isolated state of each parties. Ideally, we would expect that such separate leakage on the real state of a given party would translate to leakage on the inputs and outputs corresponding to this party alone. Joint-state leakage-tolerance, however, effectively means that, by leakage on any single party, the adversary may vicariously obtain leakage on the joint inputs and outputs of all the parties together. In this work, we shall aim at obtaining (separate-state) leakage-tolerance, and will refer to protocols achieving joint-state leakage tolerance as *weak leakage-tolerant protocols*.

Leakage-tolerant protocols with direct leakage on inputs (i.e., without leakage-free input-processing), in the separate-state model, are only known for specific two-party tasks, such as secure message transmission, commitment, oblivious transfer and zero knowledge [BCG$^+$11, GJS11, BCH12, Pan14]. Determining the feasibility of such leakage-tolerant protocols for general tasks remains open.

1.1 Contributions

Our primary contribution is constructing multi-party leakage-tolerant protocols for general functions, relying only on an *input-independent* leakage-free preprocessing phase. The input-independent processing is done once and for all, and *continual* leakage-tolerance is maintained throughout any number of executions, provided that the leakage within each execution is bounded.

In more detail, in the model of input-independent processing, each party obtains an initial state to be used later in the computation. The initial states are sampled, without leakage, from a fixed (joint) distribution that is independent of the inputs (or function), which are determined online. The online phase proceeds in an arbitrary number of executions, where in each execution a multiparty function is computed on a new set of inputs. The entire state of each party, including its current inputs, randomness, and initial state, are subject to leakage at any point in the protocol's execution, with the restriction that between every two executions, the leakage on each party's private state is a-priori bounded (in length). The initial state is updated between the executions, under leakage, and previous states and inputs are erased (which is, in fact, necessary in the continual setting). The continual leakage-tolerance achieved in this model means that in every execution, the leakage from each honest party can be emulated by a simulator obtaining the same amount of leakage from its ideal state, consisting only of its own input and output in the current execution.

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 the main results.

OCL vs. LTC. The OCL model [MR04] considers a setting where computation is performed with leaky memory, under the assumption that only the parts "touched" by the computation can leak information. The memory is 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 intuitive property that C' protects the secret k is formalized by the requirement that the adversary's view can be simulated given only the input and output of the computation.

To see the connection with the LTC setting, 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, and they communicate with each other via secret and authenticated secure channels; 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 any point it is only possible to leak from the active party.

Thus, the differences between the models of OCL and LTC are the following: First, the secure communication assumption. This difference can be bridged using existing constructions of leakage-tolerant communication [BCG$^+$11] to replace the secure channels. Second, 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 private 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, the OCL model assumes that all parties are honest and only subject to bounded leakage, whereas in LTC, we must also deal with corruption of parties.

Bridging the gap: LTC with input-independent preprocessing and strong OCL. As discussed above, the LTC model is meant to model settings where inputs are unavoidably subject to leakage. Our first contribution is a generic transformation from a strengthened form of two-party OCL, referred to as *strong OCL*, to two-party LTC with input-independent preprocessing. Informally, the main feature of strong OCL schemes is that they allow simulating the internal states of the two parties without knowledge of the adversary's leakage functions, and moreover, simulation of the party that produces the output depends only on the output of the computation, *obliviously of the input.* In addition, strong OCL security is guaranteed even under *unordered* leakage.

The transformation yields (continual) LTC protocols for the case of two-party LTC with *no corruptions*, and is a crucial step towards achieving stronger forms of security. Furthermore, we show that 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 satisfies the requirements of strong OCL as is.The OCL schemes of [JV10, GR10, DF12] can be rather directly augmented to satisfy the required strong properties; however, all of these schemes rely on a leakage-free hardware component. Thus far, the only scheme in the literature that does not rely on hardware is the Goldwasser and Rothblum scheme [GR12] (referred to as the GR scheme henthforth), which requires more than two components. [5]

To avoid any reliance on leakage-free hardware in our end result, we relax the requirement of strong 2-component OCL to *strong 2-component OCL with auxiliary parties*. Here the computation is carried out by two main components with the assistance of several auxiliary parties, where we require that the states of the auxiliary parties can be simulated obliviously of both the input and output (the simulation guarantee for the two main parties remains unchanged). We construct such an OCL scheme, without any reliance on hardware. This, in particular, yields a multi-component OCL scheme without hardware.

[5] In most part of [GR12], the scheme is described with polynomially many components. This same scheme can be reduced, however, to $O(1) > 2$ components at the cost of a worst leakage rate.

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) 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 any number of parties m that is polynomial in the security parameter.

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. (The first transformation applies assuming that m is a large enough constant. The second requires that m is polynomial in the security paramter λ, which is inherited from [BGJK12].)

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 (without additional auxiliary parties or hardware). Moreover, the [BGJK12] protocol, in the common reference string model, and any two-party LTC with $O(1)$ auxiliary parties as above, imply security under $(1-\epsilon)m$ corruptions, for any constant ϵ, $m = \lambda^{\Omega(1)}$. 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 parties 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 (leaky) universal composition theorem [Can01, BCH12].

We note that [NVZ13] show how leakage-tolerant protocols with oblivious simulation imply protocols with a relaxed form adaptive security. Their result, however, does not address the input-independent processing model. Applying the same ideas to our leakage-tolerant protocols would naturally result in (relaxed) adaptively-secure protocols in the preprocessing model.

Randomized functionalities. We can further extend the constructions of m-party LTC protocols from Theorem 3 to also support randomized functionalities. For this purpose we design a new leakage-tolerant m-party coin-tossing protocol (in the input-independent processing model). The protocol requires that the number of honest parties is as large as the number of parties in the two-party LTC protocol with auxiliary parties, for the no-corruption setting.

1.2 Techniques

We now present the main ideas and techniques behind out results. We begin by giving some intuition regarding the difficulty of constructing LTC protocols.

Why classical protocols are not leakage-tolerant. A common paradigm for 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*.

The additive secret sharing commonly used, however 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. In contrast, 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. This renders classical protocols entirely insecure. [6]

A plausible route towards circumventing this problem would be to use a *leakage-resilient secret sharing scheme* [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. In the OCL setting, however, all secret inputs are preprocessed offline, while online inputs are public. Thus, a natural question is whether we can import the OCL techniques to the setting of LTC.

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

[6] For example, the value of an intermediate 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.

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 and 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 obtain leakage from any one of the parties $P_L, P_R, P_{A_1}, \ldots, P_{A_a}$. Any leakage functions is evaluated only on the individual state of the leaking party; the number of bits leaked from any given party during a single iteration is bounded by some prefixed length function ℓ.

We require an oblivious simulator $\mathcal{S}$ that simulates the states of all parties $P_L, P_R, P_{A_1}, \ldots, P_{A_a}$ without knowledge of the leakage functions specified by the adversary; the leakage to the adversary is computed 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, we focus in this technical overview, 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. Furthermore, we 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 straightforward. 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.

To remedy this, 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 reduces 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 key 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 $(\tilde{c}, \tilde{r})$ is distributed 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. We design 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 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 not hard to see that by starting from an LTC with a auxiliary parties, we 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 $\kappa' = \text{poly}(\kappa)$. 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. Intuitively, 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] (henceforth, 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 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 proven secure in a weaker model of OCL, where the leakage adversary can only obtain leakage from the component that is *currently activated*, implying that leakage occurs in the same order as the sequence of sub-computations. As a first step towards our construction, we argue that the GR-scheme is also secure against "unordered leakage". Second, we provide new simulation procedures for the DF-scheme and the GR ciphertext banks as required by strong OCL. We defer a more detailed description of the joint scheme to the full version.

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

The high-level idea behind our transformations is as follows: The input processing of the multi-party LTC can be performed online, and under leakage, jointly by two parties. 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, no longer requiring leakage-free preprocessing. Intuitively, the two-source extraction guarantees—provided that there is only bounded separate leakage on each of the random strings—that 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 assuming that the party P_{i_1} assisting P_{i_0}, as well as the other assisting parties $P_{i'_1}, \dots, P_{i'_a}$, are all honest. In particular, we achieve a protocol in the no corruption 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 aggregate the processed inputs into a single input that is safe to use. We observe that the input-processing in the [BGJK12] protocol possesses additional properties, which give rise to such an 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 [BGJK12], there is no leakage on the encryption's randomness).

We implement the above idea as 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 an 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. Also, 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 final encryption of x_i is semantically-secure provided that any one of the zero encryptions c_C, which is the case as long as there exists some non-corrupted coalition C of parties. Moreover, the NIZKs guarantee that malicious parties cannot bias the result of the computation.

The above transformation withstands the same number of corruptions as the [BGJK12] protocol: it allows $(1-\epsilon)m$ corruptions, for $m > \lambda^{-\Omega(1)}$.

A note on universal composability and randomized functionalities. In the stand-alone setting, the security of our LTC protocols follows directly from the leakage resilience of [BGJK12] protocols in the stand-alone setting, which is shown in [BGJK12]. To show the full leakage tolerance defined in the UC setting [BCH12], we need to rely on protocols that are leakage-resilient in the UC setting, which is outside the scope of [BGJK12]. To bridge this gap, we modify the original [BGJK12] protocols to a UC variant, by replacing all building blocks in [BGJK12] with their corresponding UC counterparts. While most building blocks have standard UC version, there is no known leakage-tolerant m-party coin-tossing protocol in the UC setting. We construct such a protocol, in the input-independent pre-processing model, relying on our two-party LTC protocols with auxliary parties. This yields a coin-tossing protocol that us secure as long as sufficiently many parties are honest (the same as the number of parties in the LTC protocol). This coin-tossing protocol not only facilitates a UC variant of [BGJK12], but also allows implementing randomized functionalities.

1.3 Organization

In Section 2, we provide the formal definition of strong OCL compilers, and in Section 3 we construct two party LTC protocols secure with no corruption and only leakage from two component strong OCL compilers. Due to the lack of space, we leave the security proof of the two party LTC protocols, as well as the construction of strong OCL compilers and the final multiparty LTC protocols secure with corruptions to the full version.

2 Strong Only-Computation-Leaks Compilation

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 $\Pi = (P_1^{\text{OCL}}, P_2^{\text{OCL}}, \cdots, P_N^{\text{OCL}})$. 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* Since our end goal is constructing composable leakage tolerant protocols, where the simulator is oblivious of the leakage queries from the adversary, we consider strengthened OCL schemes which have obvious simulators.

OCL schemes with oblivious simulation: Let $\{U_T(k, f)\}_{T\in\mathbb{N}}$ denote the family of *universal circuits* where 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 1 (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,*

$P_1^{\scriptscriptstyle OCL}$ receives an input $f_i \in \{0,1\}^T$ and $P_2^{\scriptscriptstyle OCL}$ 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 $P_j^{\scriptscriptstyle OCL}$ during the i^{th} iteration, including its state during the update phase.

Correctness with adaptive input selection: *For every $\lambda \in \mathbb{N}$, $T \in \mathbb{N}$ polynomially related to λ, $k \in \{0,1\}^T$, auxiliary input $z \in \{0,1\}^{\mathrm{poly}(\lambda)}$, and PPT adversary $\mathcal{A}$, in the following real experiment $\mathsf{RealExp}^{\infty}_{\mathcal{A}}(1^\lambda, T, k, z)$ 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 are correct.*

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

We next 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 obtains leakage independently from each component during OCL evaluations (with inputs chosen adaptively by the adversary), whereas in the ideal world, it obtains leakage from states of the components simulated by an oblivious simulator. More formally,

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

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

Definition 2 (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.*

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

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 the output.

Definition 3 (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 2 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 as the left and right components, denoted by 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 $P^{\mathrm{OCL}}_{A_1}, \cdots, P^{\mathrm{OCL}}_{A_N}$, whose states can be simulated independently of the input and output; in this case, 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.

3 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.

Notation. By $\mathcal{F}^{f}_{\mathsf{2LTC\text{-}AUX}}$ we denote the 2-party ideal leaky functionality computing function f with auxiliary parties. By $\mathcal{F}_{\mathsf{LSC}}$ we denote the secure communication functionality and by $\mathcal{F}_{\mathsf{LFS}}$ we denote the input-idependent leakage-free preprocessing functionality which provides the initial states for all parties.

We now state the main theorem of this section:

Theorem 4. *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.

Proposition 1. *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.*

Theorem 4 directly follows from Proposition 1 using standard techniques.

3.1 The Protocol ρ

Let λ be 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 [7]. Our leakage-tolerant protocol below utilizes the OCL scheme to perform the evaluation of $f(x_0^j, x_1^j)$. To ensure input privacy, a party must avoid sharing its input in the clear with another party. Instead, in the j-th iteration, the parties first use the OCL scheme to allow 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 $g((c^j = (x_0^j \oplus \mathsf{PRF}(r,j)), x_1), \mathsf{PRF}(r,j)) = f(c^j \oplus \mathsf{PRF}(r,j), x_1^j)$.

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, 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.

We present the leakage-tolerant protocol ρ in detail in Figure 1:

Acknowledgement We thank Elette Boyle and Abhishek Jain for valuable discussions and the anonymous reviewers for helpful comments and suggestions.

References

ADW09. Joël Alwen, Yevgeniy Dodis, and Daniel Wichs, *Survey: Leakage Resilience and the Bounded Retrieval Model*, Information Theoretic Security - ICITS 2009 (Kaoru Kurosawa, ed.), 2009, pp. 1–18.

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.

[7] 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.

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.

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.

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

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.

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.

NVZ13. Jesper Buus Nielsen, Daniele Venturi, and Angela Zottarel, *On the connection between leakage tolerance and adaptive security*, Public Key Cryptography, 2013, pp. 497–515.

Pan14. Omkant Pandey, *Achieving constant round leakage-resilient zero-knowledge*, TCC, 2014, pp. 146–166.

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.

The leakage tolerant protocol ρ

The input-independent preprocessing stage: 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 [8]:

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.

The online stage: For each iteration j, given the initial states Φ_0, Φ_1 and Φ_A sampled in the preprocessing stage, P_0, P_1 and $P_1^{\mathsf{aux}}, \ldots, P_a^{\mathsf{aux}}$ on common input $(1^\lambda, f, T)$, and private inputs $x_0^j \in \{0,1\}^n$ and $x_1^j \in \{0,1\}^n$, proceed in the following steps, where all messages are sent via 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: 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 $P_1^{\mathsf{aux}}, \ldots, P_a^{\mathsf{aux}}$ act as the auxiliary components using initial states $\mathsf{init}_{A,i}^{j,1}, \ldots, \mathsf{init}_{A,a}^{j,1}$. 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, P_1 and $P_1^{\mathsf{aux}}, \ldots, P_a^{\mathsf{aux}}$ 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: 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.

Fig. 1. The Leakage Tolerant Protocol ρ