Leakage Tolerant Interactive Protocols[*]

Nir Bitansky[†] Ran Canetti[†] Shai Halevi[‡]

April 28, 2011

Abstract

We put forth a framework for expressing security requirements from interactive protocols in the presence of arbitrary leakage. This allows capturing different levels of leakage tolerance of protocols, namely the preservation (or degradation) of security, under coordinated attacks that include various forms of leakage from the secret states of participating components. The framework extends the universally composable (UC) security framework. We also prove a variant of the UC theorem, that enables modular design and analysis of protocols even in face of general, non-modular leakage.

We then construct leakage tolerant protocols for basic tasks, such as, secure message transmission, message authentication, commitment, oblivious transfer and zero knowledge. A central component in several of our constructions is the observation that resilience to adaptive party corruptions (in some strong sense) implies leakage-tolerance in an essentially optimal way.

[*]Research supported by the Check Point Institute for Information Security.

[†]School of Computer Science, Tel Aviv University. Supported by a Marie Curie reintegration grant and an ISF grant.

[‡]IBM T.J. Watson Research Center.

Contents

1 Introduction

Traditionally, cryptographic protocols were studied in a model where participants have a secret state that is assumed to be completely inaccessible by the adversary. In this model, the adversary can only influence the system via anticipated interfaces (such as, the communication among parties). These interfaces are crossed only when the adversary manages to fully corrupt a party, thus gaining access to its entire inner state.

In reality, an intermediate setting often emerges, when the adversary manages to gain some partial information on the secret state of uncorrupted parties. This information, termed *leakage*, can be obtained by a variety of side channels attacks that also bypass the usual interfaces and are often undetectable. Some known examples include timing, power, EM-emission and cache attacks (see, e.g., [23] for a survey).

The threat of leakage gained much attention in the past few years, giving rise to an impressive array of *leakage resilient* schemes for basic cryptographic tasks such as encryption and signatures, as well as general non-interactive circuits (see e.g. [12, 1, 2, 18, 20]). Most existing work so far concentrates on preserving, in the presence of leakage, the same functionality and security guarantees that the original primitives guarantee in a leak-free setting. Such strong leakage resilience guarantees are typically made only for leakage that is restricted in a number of ways. Examples include: putting a bound on the number of leaked information, assuming that leakage only occurs in specific times (e.g. prior to encryption), and assuming that leakage is limited to specific parts of the state (e.g. the active parts in the *only computation leaks* model [21]).

However, in many cases maintaining the same level of security as in a leak-free setting may be too costly, or even outright impossible. To exemplify this, consider the task of *secure message transmission* (SMT), where a sender wishes to transmit a (secret) message m to a receiver, so that the contents of m remains completely hidden from any adversary witnessing the communication. While in the leak-free setting the problem is easily solved using standard semantically secure encryption, in the face of leakage, this is no longer the case. In fact, semantic security is not achievable at all: An adversary that can get even one bit of arbitrary leakage, from either party, can certainly learn ,say, the first bit of the message. This is so since this bit must reside in the party's leaky memory at some point. Consequently, to address such situations in a meaningful way, we must somehow relax the security requirements from SMT protocols in face of leakage. Indeed, the recent work of Halevi and Lin [19] follows this approach in the related context of encryption. Also, a concurent work by Garg et al. [14] takes a similar approach in the context of zero knowledge protocols.

1.1 Our Contribution

We propose a new approach for defining leakage resilience (or rather, *leakage tolerance*) properties of cryptographic protocols. The approach is based on the ideal model paradigm, and in particular on the UC framework. The approach allows formulating relaxed (or weakened) security properties of protocols in face of leakage, and in particular allows specifying how the security of a protocol can degrade with leakage. It also allows specifying leakage-tolerant variants of interactive, multi-party protocols for general cryptographic tasks. In this context, the new modeling also captures attacks that combine leakage with other "network based" attacks such as controlling the communication and corrupting parties. In addition:

- We prove a general security-preserving composition theorem with respect to the proposed notion. This allows constructing and analyzing protocols in a modular way *while preserving leakage tolerance properties.* This is a powerful tool, given the inherently non-modular nature of leakage attacks.

- We describe a methodology for constructing leakage tolerant protocols in this framework. Essentially, we show that any protocol that is secure against adaptive party corruptions (where adaptive security is proven in a certain way) is already leakage tolerant in a strong sense.
- Using the above methodology and other techniques, we construct leakage tolerant protocols for secure channels, commitment, zero knowledge, and honest-but-curious oblivious transfer. (commitment and zero knowledge are realized in the common reference string model.)

Below we describe these contributions in more detail.

Leakage tolerant security within the ideal model paradigm. Following the ideal model paradigm, we define security by requiring that the protocol π at hand provides the same security properties as in an "ideal world" where processing is done by a trusted party running some functionality $\mathcal{F}$. Specifically, in the UC framework, a protocol π UC-realizes a functionality $\mathcal{F}$ if for any adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that no environment $\mathcal{Z}$ can tell whether it is interacting with $\mathcal{A}$ and π or with $\mathcal{S}$ and $\mathcal{F}$.

We consider a "real world" where the adversary can get leakage on the state of any party at any time. As we argued above, such attacks may degrade the security properties of the protocols at hand, and to account for this degradation we also allow leakage from the trusted party in the ideal world. Specifically, the functionality $\mathcal{F}$ defines an "ideal local state" for each party. (Typically this state contains the party's inputs and outputs.) When $\mathcal{A}$ performs leakage measurement L on the state of some party in the real protocol π, the simulator $\mathcal{S}$ is entitled to a leakage measurement L' on the "ideal local state" of that party in the ideal protocol. We allow the simulator to choose any function L', so long that its output length is the same as that of L. For example, we allow our leaky SMT functionality to leak bits from message that it sends, and require that a real world attacker that gets ℓ bits of leakage from the state of the implementation can be simulated by a simulator that learns only ℓ bits about the message. Our model also allows the functionality to react to leakage, in order to handle situations where security is only maintained as long as not too much leakage occurred. (For example, an authenticated channels functionality may allow forgeries once the attacker gets more bits of leakage than the security parameter, but not before that.)

Additional complications arise when attempting to generalize this simplistic model to the case of protocols that comprise of multiple components (or subroutines), and in particular when trying to guarantee security-preserving protocol composition. We describe these below. Before that, however, let us present an observation that underscores the usefulness of the present modeling.

Leakage vs. adaptive corruptions for secure channels. Consider trying to realize leaky SMT in our model using standard encryption; namely, the receiver sends its public key to the sender, who sends back an encryption of m. In the ideal world, the simulator does not witness any communication (and has no information about the message), so it can simulate the cipher by encrypting say the all-zero string, which should be indistinguishable from an encryption of m. However, after seeing the ciphertext the adversary $\mathcal{A}$ can ask for a leakage query specifying (say) the entire secret decryption key and the first bit of m. Although the simulator can now ask for many bits of leakage on m, it can no longer modify the ciphertext that it sent before, and therefore cannot maintain a consistent simulation. More generally, even for a one-bit leakage it is not clear how the simulator can reply to an arbitrary ciphertext-dependent leakage query $L(m, \mathrm{r}andomness)$ from $\mathcal{A}$, while making only an allowed query $L'(m)$ to the leaky SMT functionality.

The same problem arises in the well studied setting of *adaptive corruption*, where the adversary can adaptively corrupt parties throughout the protocol and learn their entire state. Also there, the simulator needs to first generate some messages (e.g., the ciphertext) without knowing the inputs of the parties (e.g. the message m), and later it learns the inputs and has to come up with an internal state that explains the previously-generated messages in terms of these inputs. Indeed, it turns out that techniques for handling adaptive corruptions can be used to get leakage tolerance. In fact, the problem of secure leaky channels can be solved simply by plugging in *non-committing encryption* (NCE) [8], which was developed for adaptively secure communication. A NCE scheme for bit-by-bit encryption allows the generation of a "fake equivocal ciphertext c" that can later be "opened" as an encryption of either 1 or 0, including the corresponding randomness from both sides. Namely, c can be generated together with two sets of randomness pairs $(r_S^0, r_R^0), (r_S^1, r_R^1)$, such that (r_S^0, r_R^0) is the randomness for sender and receiver that "explains" c as an encryption of zero, and (r_S^1, r_R^1) in the randomness that "explains" c as an encryption of one.

Indeed, to obtain secure message transmission in the presence of leakage one can simply encrypt the message using an NCE scheme. The simulator can now generate the fake ciphertext c with the associated randomness pairs, and then can translate any c-dependent leakage function on the entire state (plaintext and randomness) into a leakage function on the plaintext only, which can be queried to the leaky SMT functionality. When leakage on $P \in \{S, R\}$ occurs, the simulator $\mathcal{S}$ just hard-wires the two pairs r_P^0 and r_P^1 into the query, translating the leakage function function $L(b, r_P)$ into $L'(b) = L(b, r_P^b)$. Hence we get:

Theorem 1.1 (informal)**.** *Any Non-Committing Encryption scheme realizes secure leaky channels in the presence of arbitrary leakage (assuming ideally authenticated communication).*

The general case. The above example can be made general. Specifically we show that, with some limitations, any protocol that realizes a functionality $\mathcal{F}$ under adaptive corruptions also realizes a leaky variant $\mathcal{F}^{+\mathsf{lk}}$ under leakage. The "leaky variant" is a natural adaptation that allows leakage on the state of $\mathcal{F}$, just like the leaky SMT allow leakage on the transmitted message. This variant, denoted $\mathcal{F}^{+\mathsf{lk}}$, is identical to $\mathcal{F}$ except that: (a) $\mathcal{F}^{+\mathsf{lk}}$ allows the simulator to apply arbitrary leakage functions to the ideal local state of the party, but notifies the external world of the number of bits leaked. (This notification makes sure that the simulator only asks to leak the same number of bits as in the protocol execution.) (b) Once a party has leaked even one bit, $\mathcal{F}^{+\mathsf{lk}}$ behaves in the same way that $\mathcal{F}$ behaves after a passive adaptive corruption of that party. This means that the obtained leakage tolerance guarantee is meaningful only for functionalities that maintain some security properties even after passive (semi-honest) corruption.

A limitation of this result is that it only works when the given proof of security uses a restricted type of simulators, namely ones that work "obliviously" to the state that they learn when corrupting a player. (We call such simulators *corruption-oblivious.*) We have:

Theorem 1.2 (informal)**.** *If protocol π realizes $\mathcal{F}$ under adaptive corruptions (either semi-honest or Byzantine) with an oblivious simulator, then it also realizes $\mathcal{F}^{+\mathsf{lk}}$ with the same type of corruptions, plus arbitrary leakage.*

Composable leakage tolerance. A useful property of ideal model based notions of security is that they are typically preserved under modular (or universal) composition of protocols. That is, if a protocol π realizes an ideal functionality $\mathcal{F}$, the security properties of $\mathcal{F}$ carry over to any environment where π is used. This allows designing protocols and analyzing their security in a modular way. Extending this "modular security" paradigm to the leaky world seems hopeless: Real world leakage is inherently non-modular, in that the adversary can obtain leakage from the joint state an entire physical device, and is not bound by our modular separation to logical modules of the software running on the device. In contrast, the common models of composable security rely crucially on viewing different sub-modules of a large system as autonomous small systems, each with its own local state. In such models, it is not even clear how to *express* joint leakage from the state of different modules, let alone how to argue about preservation of security properties.

We extend the UC security framework [6] to allow expressing leakage attacks from physical devices that span multiple logical modules (ITM instances). We first allow the protocol analyzer to delineate sets of "jointly leakable ITM instances". Now, we equip each such set P with a new entity, called an aggregator, that has access to the internal states of all the modules in P. To get leakage from the joint state of the modules in a set P, the adversary sends the leakage function L to the relevant aggregator, who applies L to the combined state and returns the result to the adversary. The same mechanism is used to obtain leakage from ideal functionalities, except that here the ideal functionality $\mathcal{F}$ hands the aggregator some "ideal local state" that $\mathcal{F}$ associated with the set P.

Having extended the model of protocol execution to capture leakage attacks, we would like to re-assert the composability property described above, i.e., to re-prove the UC composition theorem from [6] in our setting. However, that theorem was only proved for systems that behave in a "modular way," and the proof no longer holds in the presence of our modularity-breaking aggregator.

Still, we manage to salvage much of the spirit of the UC theorem, as follows. We formulate a more stringent variant of UC security by putting some technical restrictions on the simulator, and then re-assert the UC theorem with respect to the more stringent notion. Similarly to the case of corruption-oblivious simulators, here too we require that the simulator $\mathcal{S}$ handles leakage queries "obliviously". Roughly, $\mathcal{S}$ has a "query-independent" way of translating, via a state-translation function, real world leakage queries $L(\mathsf{state}_\pi)$ to ideal world leakage queries $L'(\mathsf{state}_\mathcal{F})$. Furthermore, it ignores the leakage results in the rest of the simulation. We call such simulators *leakage-oblivious,* and show:

Theorem 1.3 (UC-composition with leakage (informal)). *Let $\rho^\mathcal{F}$ be a protocol that invokes $\mathcal{F}$ as a subroutine. Let π be a protocol that* UC*-emulates $\mathcal{F}$ with a leakage-oblivious simulator. Then the composed protocol $\rho^{\pi/\mathcal{F}}$ (were each call to $\mathcal{F}$ is replaced with a call for π)* UC*-emulates $\rho^\mathcal{F}$ in face of leakage. Furthermore, it does so with a leakage-oblivious simulator.*

Leakage tolerant protocols. We construct leakage tolerant protocols for a number of basic cryptographic tasks. We first observe that the general result regarding the leakage tolerance of adaptively secure protocols (Theorem 1.2) in fact guarantees UC security *with leakage-oblivious simulators*. We then observe that existing adaptively secure protocols for secure channels, UC commitment and UC semi-honest oblivious transfer already have corruption-oblivious simulators, hence we immediately get:

- Assume authenticated communication. Then, any non-committing encryption scheme UC-realizes $\mathcal{F}^{+\mathsf{lk}}_{\mathsf{SMT}}$ in the presence of arbitrary leakage using a leakage-oblivious simulator.

- In the CRS model, the UC commitment protocols of Canetti and Fischlin [9] and Canetti, Lindell, Ostrovsky and Sahai [11], UC realize $\mathcal{F}^{+\mathsf{lk}}_{\mathsf{MCOM}}$ (the leaky version of the multi-instance commitment functionality) in the presence of arbitrary leakage. Furthermore, they do so with leakage-oblivious simulators.

- The semi-honest oblivious transfer protocol of [11] for adaptive corruptions UC realizes $\mathcal{F}^{+\mathsf{lk}}_{\mathsf{OT}}$ (the leaky version of the ideal oblivious transfer functionality in the presence of arbitrary leakage). Furthermore, it does so with leakage-oblivious simulators.

Finally, we address two tasks where adaptive security does not suffice. The first such task is message authentication. Indeed, here $\mathcal{F}^{+\mathsf{lk}}_{\mathsf{AUTH}}$, the leaky variant of the ideal authentication functionality, gives essentially no security guarantees: As soon as even a single bit of information is leaked from the sender, $\mathcal{F}^{+\mathsf{lk}}_{\mathsf{AUTH}}$ behaves as if the sender is fully corrupted, in which case forgery of messages is allowed. We thus first formulate a variant of $\mathcal{F}_{\mathsf{AUTH}}$ that guarantees authenticity as long as the number of bits leaked is less than some threshold. We then realize this functionality, denoted $\mathcal{F}^{+B}_{\mathsf{AUTH}}$, assuming an initial k-bit shared secret key between the parties, and as long as at most $B = O(k)$ bits leak *between each two consecutive transmissions of authenticated messages.* Furthermore, we do this with a leakage-oblivious simulator. The techniques used to realize $\mathcal{F}_{\mathsf{AUTH}}^{+B}$ include information-theoretic leakage resilient message authentication codes, as well as NCE schemes. We note that similar techniques are used for a related goal in [4]. However, the security analysis there is different than the one here.

We observe that, using the above UC theorem with leakage, we can combine the above authentication protocol with any one of the previous protocols (that assume ideally authenticated communication) to obtain composite, leakage tolerant protocols that withstand unauthenticated communication.

The last task we address is zero knowledge. At first it may seem that, as in the case of commitment, existing protocols for UC-realizing the ideal zero knowledge functionality, $\mathcal{F}_{\mathsf{ZK}}$, would work also in the case of leakage. However, this turns out not to be the case. In particular, while the protocol of [9] for UC-realizing $\mathcal{F}_{\mathsf{ZK}:R}$, for some relation R, given $\mathcal{F}_{\mathsf{MCOM}}$ is indeed secure against adaptive corruptions, the simulator turns out not to be corruption-oblivious. In fact, at present we do not know of any protocol that UC-realizes $\mathcal{F}_{\mathsf{ZK}:R}^{+\mathsf{lk}}$ for a non-trivial relation R in the presence of leakage (even with simulators that are not leakage-oblivious).

Instead, we settle for UC-realizing, in the presence of leakage, a weaker variant of $\mathcal{F}_{\mathsf{ZK}:R}^{+\mathsf{lk}}$. This weaker variant permits violation of the soundness requirements if too many bits were leaked from the verifier. We denote this weaker version by $\mathcal{F}_{\mathsf{ZK}:R}^{+B}$, where B is the leakage threshold for the verifier. We have:

Theorem 1.4 (Leaky zero knowledge, informal)**.** *Let R be an NP relation, and let $B = k - \omega(\log k)$ where k is the security parameter. Then there exists a protocol that* UC*-realizes $\mathcal{F}_{\mathsf{ZK}:R}^{+B}$ given access to $\mathcal{F}_{\mathsf{MCOM}}^{+\mathsf{lk}}$.*

Using the (leaky) universal composition theorem and the above protocol for realizing $\mathcal{F}_{\mathsf{MCOM}}$, we obtain a protocol for UC-realizing $\mathcal{F}_{\mathsf{ZK}:R}^{+B}$ in the CRS model.

We have very recently heard of a concurrent work by Garg, Jain and Sahai on zero knowledge in the presence of leakage [14]. In prticular, the idea of using the Goldreich-Kahan approach in this context was communicated to us verbally by Amit Sahai.

2 Modeling Leakage in the UC Framework

This section defines the new model of UC security with leakage. We start with an informal overview in Section 2.1. For the reader who is not interested in full details, this overview should suffice for understanding the rest of the paper. Section 2.2 then summarizes the UC framework, and Section 2.2.1 provides the actual model. Section 2.3 presents the model of UC with leakage in detail.

2.1 Overview

We start with a brief review of the basic concepts of the universally composable (UC) security framework, explain why it needs to be modified to account for leakage, and describe the proposed modifications.

Basic UC. Recall that the basic UC framework considers realization of an "ideal specification" $\mathcal{F}$ by a "real implementation" π. (Formally both $\mathcal{F}$ and π are just protocols, we call them by different names to guide the intuition.) The realization requirement is that for any "real world attacker" $\mathcal{A}$ against the implementation π there exists another adversary $\mathcal{S}$ (called a simulator) against the specification $\mathcal{F}$, such that an "environment" $\mathcal{Z}$ that interacts with $\mathcal{S}, \mathcal{F}$ has essentially the same view as in an interaction with $\mathcal{A}, \pi$. In other words, the adversarial effect on any (potentially large) system $\mathcal{Z}$ executed with π can not be worse then the adversarial effect on the same system where π is replaced with ϕ.

The basic UC execution model lets the environment $\mathcal{Z}$ determine the inputs to the parties running the protocol and see the outputs generated by these parties, and also allow free communication between the environment and the adversary. The adversary typically has full control over the communication between parties, and the ability to "corrupt" parties in various ways. Corruption is modeled as just another interface available to the adversary, where it can send a message "you are corrupted" to any party. (In the case of standard passive corruption, the party responds to this message by handing its entire internal state to the adversary. To model Byzantine corruption, the party also changes the program that it is running from then on.)

A crucial aspect of the UC framework is its modularity, where programs can call subroutines, and these subroutines are treated as separate entities that can be analyzed separately for security properties. Importantly, local randomness and secrets that are used by a subroutine should typically not be visible to the calling routine or to other components in the system.

A useful technicality in the UC framework, is that it is sufficient to prove security with respect only to the dummy real world adversary $\mathcal{D}$. This is the adversary 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. Relying on the fact that any adversary can be emulated by the environment itself, it is easy to show that the existence of a simulator for the dummy adversary $\mathcal{D}$ implies the existence of a simulator for any adversary.

Leaky UC. A natural approach to modeling leakage within the UC framework is to view it as a weak form of corruption, where the adversary gets some information about the internal state of the leaky party but perhaps not all of it. Also, leakage may resemble "semi-honest" corruption more than "malicious", in that leaky parties keep following the same protocol and do not change their behavior following a leakage event.[1] Thus we could provide yet another interface to the adversary, where it can send a "leak L" message to a party (with L some function), and have that party reply with $L(s)$ where s is its internal state.

The "leakage resilience" of a protocol in this model is determined by the type of leakage functions that the adversary can make, and by the leakage-related properties of the functionality that it realizes. For example, realizing a functionality that does not change its behavior after leakage implies high (or perfect) leakage resilience. On the other hand, realizing a functionality that treats a single bit of leakage the same as malicious corruption of that party implies very low leakage tolerance.

The leakage aggregator. A serious shortcoming of the modeling approach from above is it only lets the adversary obtain leakage on individual processes (or subroutines). In contrast, real life leakage usually provides information that depends on the entire state of a physical device, including all the processes that are currently running on it. To account for this inherently non-modular property of real life leakage, we introduce to the model a new "global entity" that we call the leakage aggregator. The aggregator $\mathcal{G}$ can access the entire internal state of all the components in the system. We allow the adversary to make leakage queries for sets of processes, not just individuals. A leakage query specifies a leakage function L and a set of processes $P = \{p_1, \ldots, p_t\}$. This query is forwarded to the aggregator, who evaluates $L(s_1, \ldots, s_t)$ and returns the result to the adversary. Some important technicalities regarding the working of $\mathcal{G}$ are the following:

- A convention should be set for how to specify the sets of processes and ensure that this is a "legitimate set" for joint leakage. We assume that processes are tagged with "party identifiers" pid, and joint leakage is allowed from all the processes that have the same pid.

[1]In fact, this is not completely true: when a component is an ideal functionality that represent idealized behavior of an entire multiparty protocol, some changes to the behavior may be needed. For instance an idealized signature functionality may allow forgeries once the amount of leakage exceeds the secret-key length in the implementation.

- As done for corruptions, here too the identity of the leaky processes and the amount of leakage needs to be reported to the environment. This forces the simulator, in the ideal world, to use the same amount of leakage from the same processes as in the real world.

- Since ideal functionalities represent idealized constructs that do not necessarily run on physical devices, they are often associated with more than one pid. Thus care should be taken when deciding how an ideal functionality reacts to leakage queries w.r.t. one of its pid's.(For example, the secure-channels functionality runs on behalf of both the sender and the receiver, and would typically react differently to sender-leakage than to receiver-leakage queries.) We let the ideal functionality itself decide how to reply when $\mathcal{G}$ asks it for the state corresponding to any of its pid's. (This is the same convention as used for corruption, where the functionality gets to decide what to reveal to the adversary when one of its pid's is corrupted.) Typically, the "state" associated with a certain pid will be just the inputs that were received from that pid and the outputs it receives.

- To allow security specifications (i.e., functionalities) to react to leakage situations, we have $\mathcal{G}$, upon accessing the state of a module, reports to that module the output size of the leakage function L. Typically, "real world implementations" just ignore this report (since we assume that real world leakage is undetectable), but "ideal functionalities" may use it to change their behavior (e.g., give-up if too much leakage occurred).

With these conventions in place, a leakage operation is handled as follows: First the adversary sends a query $(\mathsf{leak}, L, \mathsf{pid})$ to $\mathcal{G}$, where L is the leakage function and pid is the leaking party ID. Then $\mathcal{G}$ obtains $\mathsf{state}_{\mathsf{pid}}$, the total state of party pid, applies L to $\mathsf{state}_{\mathsf{pid}}$ and returns the result to $\mathcal{A}$. Finally, $\mathcal{G}$ reports the output length of the function L to all the processes whose state is included in $\mathsf{state}_{\mathsf{pid}}$, and reports pid and the output length to the environment (see Figure 1.)

We note that the security guarantee provided by this model is somewhat weaker than one may have desired, as the aggregator reports the *overall* number of leaked bits to *each one* of the processes (or functionalities). This means than when a domain leaks l bits, each one of its components behaves as if the l bits leaked entirely from this component. While this is a relatively weak leakage resilience guarantee, it seems unavoidable in the context of a general model for representing non-modular leakage across multiple logical components.

Leakage-oblivious emulation. Following the approach of basic UC security, the definition of protocol emulation requires that for any adversary $\mathcal{A}$ that attacks the implementation π there exists a simulator $\mathcal{S}$ that attacks the specification $\mathcal{F}$ so that no environment can distinguish between an interaction with $\mathcal{A}$ and π, and an interaction with $\mathcal{S}$ and $\mathcal{F}$. In particular, $\mathcal{S}$ must provide an overall transformation from one interaction scenario to the other, including among others the leakage queries made by $\mathcal{A}$ to the parties (via the aggregator). (As noted above, an equivalent requirement considers, instead of any adversary $\mathcal{A}$, only the dummy adversary, $\mathcal{D}$, that merely passes messages between the environment and the protocol parties.)

This natural requirement, however, has (seemingly inherent) difficulties when considering composition of protocols. In particular, we were not able to prove a general composition theorem in this model (see details in Section 3). Consequently, we consider a more restricted notion of protocol emulation, which we term emulation with leakage-oblivious simulators.

To simplify the exposition, we describe here leakage-oblivious emulation only with respect to the dummy adversary $\mathcal{D}$. A leakage-oblivious simulator $\mathcal{S}$ for the dummy adversary has a special form: Specifically, $\mathcal{S}$ has a separate subroutine $\tilde{S}$ for handling leakage. When $\mathcal{S}$ receives from the environment a request to apply a leakage function L to a set P of processes, $\tilde{S}$ is invoked to produce a "state translation" function T. This function is meant to transform the internal state of P in the specification

$\mathcal{F}$ into "the actual state" in the implementation π. Once T is produced, the aggregator is given the composed leakage function $L \circ T$. Finally, when the leakage result is returned, it is forwarded directly to the environment and $\mathcal{S}$ returns to its state prior to the leakage event.

We stress that the subroutine $\tilde{S}$ operates **independently of the leakage circuit** L, its only input is the current state of $\mathcal{S}$ and a party identifier pid. Also, the leakage operation has **no side effect on** $\mathcal{S}$. That is, following the leakage event $\mathcal{S}$ return to the state that it had before that event. (See Figure 3.)

2.2 UC security: A brief review

We summarize the UC security framework [6]. For brevity and simplicity, we describe a somewhat restricted variant; Still, the summary is intended to provide sufficient detail for verifying the treatment in this work. The description below is rather terse (and is taken almost verbatim from the appendix of [10]). Further elaboration and justification of definitional choices appears in [6, 7].

The basic model of computation. The first step in formulating the UC framework is to specify the underlying model of distributed computation. While in this work we do not modify this model, we briefly review it.

The basic computing unit is an algorithm, or a program, to be run on some physical device that is connected to a network. Such programs are represented as interactive Turing machines (ITMs). An ITM has an input tape, where inputs from a calling program are written, an incoming communication tape for incoming messages from the network, and a subroutine output tape where outputs from subroutines of the program are written. In addition an ITM has a special *identity tape* which contains its code (in some standard representation) and a unique identifier. The identifier has two fields: a session identifier (sid) and a party identifier (pid). An *ITM instance* (ITI) represents an instance of an ITM in an execution of a system. Formally, an ITI is specified via the contents of its identity tape, namely the code, sid and pid. The sid represents the "protocol instance" that the ITI is part of; that is, an instance of a protocol in an execution of a system is the set of all ITIs that have the same code and sid. The pid can represent different things; in this work we will use the interpretation that all ITIs with the same pid are running on the same physical device (and thus may leak jointly).

A *system of ITMs* is a pair (I, C) where I is an ITM, called the initial ITM, and C is a *control function.* An execution of a system starts by running the initial ITM from some initial configuration (which includes its identity and some input). Once this ITM specifies a message to be written to one of the tapes of a new ITI, this ITI is added to the system and activated. The computation proceeds by having each ITI proceed according to its code until it writes a message to another ITI. Once it does so, the message is written to the recipient ITI, subject to potential modification by the control function. The output of the computation is the output of the initial ITM.

we consider polynomial time (PT) ITMs. An ITM M is PT if at any point during the execution of an ITI with program M the overall runtime is bounded by a polynomial in its *residual input length,* where the residual input length is the number of bits written on the input tape of this ITI, minus the number of bits written by this ITI to the input tapes of other ITIs.

2.2.1 The basic model

We first present the underlying model of computation, which provides the basic mechanics on top of which the notion of protocol security is defined.

Interactive Turing Machines (ITMs). The basic computing element is an Interactive Turing Machine (ITM), which represents a program written for a distributed system. The UC framework uses a formalism of an ITM that augments the original formalism of [17, 15] with some additional structure, for the purpose of capturing protocols in multi-party, multi-instance systems. Specifically, an ITM is a Turing machine with the following additional constructs. It has three special tapes that represent three different types of information coming from external sources: The input tape represents information coming from the "calling protocol"; the communication tape represents information coming from other parties over untrusted communication links; the subroutine output tape represents information coming from "subroutine protocols" in a trusted way. In addition, an ITM has a special identity tape which cannot be written on by the ITM transition function, or program. The contents of the identity tape is interpreted as three values: The program of the ITM, represented in some canonical form, A session-identifier (sid), representing a specific protocol session, and a party identifier (pid), representing an identity of a party within that session. In the context of this work we will use the pid as an indicator of the physical device on which the ITI runs. That is, all the ITIs that represent processes that run on the same physical device (and can thus leak in a correlated way) have the same pid. Finally, to the standard ITM syntax we add the ability to perform an external write instruction. The semantics of this instruction are defined below.

Systems of ITMs. Running programs in a distributed system is captured as follows. An ITM instance (ITI) $\mu = (M,\ \mathrm{ID})$ is an ITM M (namely, a program) along with a string ID=(sid,pid), called the identity of μ. An ITI represents a running instance of the program M where the identity ID is written on its identity tape. A system of ITMs is a pair (M, C), where M is an ITM and $C : \{0,1\}^* \rightarrow \{0,1\}^*$ is a control function that determines the effect of the external write commands.

An execution of a system (M, C) of ITMs, on input x, consists of a sequence of activations of ITIs. Initially, the system consists of a single ITI with program M, some fixed identity (say, $\mathrm{ID} = (0, 0)$), and x written on the input tape. This ITI, called the initial ITI, is then activated.

In each activation of an ITI, the active ITI runs its program. The execution ends when the initial ITI halts. The output of an execution is the output of the initial ITI.

It remains to specify the effect of the external-write operation. This operation specifies a target ITI (namely, program and identity), a tape out of $\{$input, communication, subroutine output$\}$, and data to be written. When an external-write operation is carried out, the control function C is applied to the sequence of external write requests in the execution so far. Then:[2]

1. If C returns 1 then:
   (a) If an ITI with the same identity as the target ITI does not exist in the system then a new ITI with the specified program and identity is added to the system.
   (b) The specified data is written to the specified tape of the (unique) ITI whose identity agrees with the identity in the external write command, along with the identity of the writing ITI.

   In either case, the active ITI becomes inactive and the target ITI is activated.
2. If C returns 0, or the active ITI halts, then the initial ITI is activated.
3. If C returns another value, then this value is interpreted as a description of an ITM M. The effect is the same as in Case 1, except that the program of the target ITI is taken to be M rather than the value specified in the external write command.

[2]A more formal description in terms of sequences of configurations can be extracted from this description. See [6]. Also, in [6] the semantics of an external write operation are somewhat more complex, for the purpose of obtaining additional expressive power. (The main difference is that in [6] the adversary is allowed to create new ITIs by delivering messages to them. This allows capturing those natural situations where new parties are "prompted to join" from within the protocol instance, rather than being invoked by other protocols. However, it also allows a situation where ITIs receive inputs and subroutine outputs from ITIs whose code is unknown and untrusted; this requires special model provisions such as letting the target ITI know the code of the writing ITI in some cases.)

Subroutines. An ITI μ is a subroutine of ITI μ' in an execution if μ wrote to the input tape of μ or μ' wrote to the subroutine output tape of μ'.

Protocols and protocol instances. A protocol is formalized as a single ITM, that represents the programs to be run by all the intended participants. (When the protocol specifies several different roles, the ITM describes the programs for all the roles. The role is then given as part of the input.) An instance (or session) of a protocol π with session identifier sid, within a system of ITMs, is the set of ITIs that run the program π and whose session identifier is sid.

Polynomial Time ITMs. We consider ITMs that run in probabilistic polynomial time (PPT), where PPT is defined as follows: An ITM M is PPT if there exists a constant $c > 0$ such that, for any ITI μ with program M, at any point during its run, and for any contents of the random tape, the overall number of steps taken is at most n^c, where n is the overall number of bits written to the input tape of μ minus the overall number of bits written by μ to input tapes of other ITIs. (The purpose of this definition is to have syntactically verifiable conditions which guarantee that running a system of ITMs does not consume "super-polynomial resources". In particular, it can be seen that an execution of a system of ITMs, where the initial ITM is PPT, and the control function is polytime computable, can be simulated on a standard PPT Turing machine.)

2.2.2 Defining security of protocols

Recall that protocols that securely carry out a given task are defined via comparison with an ideal process for carrying out the task. Formalizing this notion is done in several steps, as follows. First, we define the process of executing a protocol in the presence of an adversarial environment. We then define what it means for one protocol to "emulate" another protocol. Next, we define the "ideal process" for carrying out the task in terms of a special idealized protocol. A protocol is said to securely carry out the task if it emulates the idealized protocol for that task.

The model for protocol execution. The model for executing a protocol π is parametrized by a security parameter $k \in \mathbf{N}$, and three ITMs: the ITM π, an ITM $\mathcal{A}$ called the adversary, which represents the adversarial activity against a single instance of π, and an ITM $\mathcal{Z}$, called the environment, which represents the rest of the system. Specifically, to run protocol π on input x, execute the system of ITMs $(\mathcal{Z}, C_{\mathcal{A},\pi})$. It remains to describe the control function $C_{\mathcal{A},\pi}$, namely the external write capabilities of each ITI.

In essence, the definition of C captures a model where a *single instance* of π interacts with $\mathcal{Z}$ and $\mathcal{A}$. $\mathcal{Z}$ controls the inputs to parties and reads the outputs. All communication (via the communication tapes) must pass through $\mathcal{A}$. In addition, the parties of π can create subroutine ITIs, can write to the input tapes of the subroutines, and receive outputs from the subroutine on the subroutine output tapes of the calling parties. More precisely:

External writes by the environment: The environment can write only to of other ITIs. The program of the first ITI invoked by the environment is set (by the control function) to be the program of the adversary $\mathcal{A}$. The programs of all the other ITIs that the environment writes to are set to be the protocol π.[3] In addition, the session IDs of all the ITIs invoked by the environment (other than the adversary) must be the same. That is, let s denote the sid of the first ITI to be invoked with program

[3]The reason for having the control function (rather than the environment) determine the program π is to allow a situation where the program π is changed into another program π' without having the environment being necessarily aware of the change. This detail is necessary for the definition of protocol emulation to make sense.

π. Then all the remaining ITIs invoked by the environment must have sid s. Consequently, all the ITIs invoked by the environment, except for the adversary, belong to the same instance of π.

External writes by the adversary: The adversary can write only to the communication tapes of ITIs. In addition, it is not allowed to create new ITIs; namely, if the adversary performs an external write request with non-existing target ITI, the control function returns 0. (As mentioned above, this restriction is not imposed in [6], resulting in a more expressive but somewhat more complex model.)

External writes by other ITIs: An ITI μ other than the environment and the adversary can write only to the subroutine output tapes of ITIs that have previously written to the input tape of μ, to the input tapes of ITIs that μ has invoked, and to the input tapes of ITIs with the same session ID as μ. In addition, it can write to the communication tape of the adversary. (Writing to input tapes of ITIs is the same session ID will become useful when defining ideal protocols.)

We also use the convention that creation of a new ITI must be done by writing to the input tape of that ITI; the data written in this activation must start with 1^k, where k is the security parameter.

Modeling party corruption. Since the modeling of party corruption will be central to the modeling of leakage, we describe it in more detail. Formally, party corruption is modeled as a special message sent by the adversary to the corrupted party (ITI). Different types of corruptions (e.g., passive, Byzantine) are modeled as parameters in the corruption message. The response of a party (ITI) to an incoming corruption message is formally treated as part of the protocol specification. This modeling has the advantage that general notions and theorems such as UC emulation, the UC theorem, and the universality of the dummy adversary apply regardless of the specific corruption model. However, some additional formalism is necessary in order to make sure that the formal corruption operation corresponds to the generally accepted intuitive notion of party corruption. Specifically, we that an ITM is *corruption compliant* if its program consists of a main program σ (which can be thought of as an "operating system" of sorts), and a subroutine π which represents the actual program run by the ITM. The main program relays all inputs, incoming messages, and subroutine outputs to π, with the exception of the corruption messages sent by the adversary. The behavior of σ upon receipt of the corruption message essentially determine the corruption model.

Let us specify the behavior of σ for two salient types of corruption. In the case of passive party corruption, σ behaves as follows. When an ITI μ receives the first corruption message, the σ part of the code of μ reports that μ has been corrupted to all the ITIs that have written on μ's input tape. Upon receipt of all other corruption messages, σ returns to the adversary the entire current state of π. Note that π is never notified of the corruption message; this captures the intuitive concept that a party is generally not aware of being passively corrupted. Also, note that here the adversary has to explicitly ask for each new report of internal state; however this formalism is chosen for convenience only and is of no real consequence.

In the case of Byzantine corruptions, it is assumed that the corruption message from the adversary includes in it a description of an ITM M. Here σ behaves the same as before, except that it immediately replaces the code M instead of π.

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

Protocol emulation. Informally, we say that a protocol π UC-emulates protocol π' if for any adversary $\mathcal{A}$ there exists an adversary $\mathcal{A}'$ such that no environment $\mathcal{Z}$, on any input, can tell with non-negligible probability whether it is interacting with $\mathcal{S}$ and parties running π, or it is interacting with $\mathcal{A}'$

and parties running π'. This means that, from the point of view of the environment, running protocol π is 'just as good' as interacting with π'.[4] This notion is formalized as follows. A distribution ensemble is called binary if it consists of distributions over $\{0, 1\}$. We have:

Definition 2.1. *Two binary distribution ensembles $\{X(k, a)\}_{k\in\mathbf{N},a\in\{0,1\}^*}$ and $\{Y(k, a)\}_{k\in\mathbf{N},a\in\{0,1\}^*}$ are called indistinguishable (written $X \approx Y$) if for any $c, d \in \mathbf{N}$ there exists $k_0 \in \mathbf{N}$ such that for all $k > k_0$ and for all $a \in \{0, 1\}^{k^d}$ we have*

$$|\Pr(X(k, a) = 1) - \Pr(Y(k, a) = 1)| < k^{-c}.$$

Definition 2.2 (Protocol emulation)**.** *Let π and π' be protocols. We say that π UC-emulates π' if for any adversary $\mathcal{A}$ there exists an adversary $\mathcal{A}'$ such that for any environment $\mathcal{Z}$ that outputs a value in $\{0, 1\}$ we have*

$$\mathsf{EXEC}_{\pi',\mathcal{A}',\mathcal{Z}} \approx \mathsf{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}.$$

This work makes use of the following simplified formulation of UC emulation. Let the dummy adversary $\mathcal{D}$ be the adversary that merely reports to the environment all the messages sent by the parties, and follows the instructions of the environment regarding which messages to deliver to parties. Then, it is enough to prove security with respect to the dummy adversary. That is:

Definition 2.3 (Protocol emulation with the dummy adversary)**.** *Let π and π' be protocols. We say that π UC emulates π' with the dummy adversary if there exists an adversary $\mathcal{A}'$ such that for any environment $\mathcal{Z}$ that outputs a value in $\{0, 1\}$ we have*

$$\mathsf{EXEC}_{\pi',\mathcal{A}',\mathcal{Z}} \approx \mathsf{EXEC}_{\pi,\mathcal{D},\mathcal{Z}}.$$

Claim 2.1 ([6])**.** *Protocol π UC-emulates protocol π' iff π UC-emulates π' with respect to the dummy adversary.*

Ideal functionalities and ideal protocols. A key ingredient in the ideal process for a given task is the ideal functionality that captures the desired behavior, or in other words, the specification of that task. The ideal functionality is modeled as an ITM (representing a "trusted party") that interacts with the parties and the adversary.

For convenience, the process for realizing an ideal functionality is represented as a special type of protocol, called an ideal protocol. In the ideal protocol $I_\mathcal{F}$ for ideal functionality $\mathcal{F}$ all parties simply hand their inputs to an ITI with program $\mathcal{F}$, session ID that is equal to the local session ID, and party ID set to some fixed value, say $\perp$. Whenever a party in $I_\mathcal{F}$ receives a value from $\mathcal{F}$ on its subroutine output tape, it immediately copies this value to the subroutine output tape of the ITI that invoked it. We call the parties of the ideal protocol dummy parties. The adversary interacting with the ideal protocols is called the simulator and denoted $\mathcal{S}$.

Definition 2.4 (Realizing functionalities)**.** *Let π be a protocol, and let $\mathcal{F}$ be an ideal functionality. We say that π UC-realizes $\mathcal{F}$ if π UC-emulates $I_\mathcal{F}$, the ideal protocol for $\mathcal{F}$.*

[4] To be precise, the definition of protocol emulation only quantifies over *balanced* environments. An environment is balanced if at any point in time the overall length of input to the adversary is at least some polynomial in the overall length the of inputs given to the rest of the ITIs in the system. As explained in [6], failing to make this restriction makes the definition unreasonably strong, and also causes technical problems with the composition theorem.

Ideal functionalities and party corruption. An ideal functionality represents an ideal specification, rather than an actual program that runs on an actual, physical device. Thus, party corruption messages sent to an ideal functionality do not directly represent physical corruption. Instead, the behavior of an ideal functionality upon receipt of corruption messages from the adversary specifies the security requirements from the realizing protocols upon party corruption.

In general, ideal functionalities can modify their behavior in arbitrary ways as a function of the corruption requests received from the adversary so far. (For instance, an ideal functionality may allow the adversary to modify sensitive information as soon as more than some number of parties have been corrupted.) Still, we define some "standard" behavior of an ideal functionality in face of corruption. Specifically, we say that an ideal functionality $\mathcal{F}$ is *standard corruption* if:

1. An instance of $\mathcal{F}$ with sid s keeps some "ideal local state" S_p for each dummy party (s, p) that interacts with this instance of $\mathcal{F}$. (Here p is the pid of this dummy party.)

2. Upon receipt of the first "corrupt p" message from the adversary, $\mathcal{F}$ first notifies the dummy party (s, p) that it has been corrupted. Next, in each future "corrupt p" message, $\mathcal{F}$ returns to the adversary the contents of the ideal state S_p.

It is stressed that a standard corruption functionality can specify additional instructions to be performed upon receipt of a corruption message; it can also alter its overall behavior as exemplified above.

Universal Composition. Let ρ be a protocol that uses one or more instances of some protocol ϕ as a subroutine, and let π be a protocol that UC-emulates ϕ. The composed protocol $\rho^{\pi/\phi}$ is constructed by modifying the program of ρ so that calls to ϕ are replaced by calls to π. Similarly, subroutine outputs coming from π are treated as subroutine outputs coming from ϕ. The universal composition theorem says that protocol $\rho^{\pi/\phi}$ behaves essentially the same as the original protocol ρ. That is:

Theorem 2.1 (universal composition). *Let π, ϕ, ρ be protocols, such that π UC-emulates ϕ. Then the protocol $\rho^{\pi/\phi}$ UC-emulates ρ. In particular, if ρ UC-realizes an ideal functionality $\mathcal{F}$ then so does $\rho^{\pi/\phi}$.*

We note that the universal composition theorem hold only in the case where protocols π and ϕ are *modular.* Essentially, a protocol is modular if in no instance s of this protocol there is a subroutine ITI I of some ITI which is part of the instance (or a subroutine thereof), where I receives input from or sends outputs to and ITI that is not a descendant of a member of instance s.[5]

2.3 UC security with leakage

The model of executing protocol π with environment $\mathcal{Z}$ and adversary $\mathcal{A}$ remains the same as in the basic UC framework, with the exception that now we allow the adversary to perform an additional *leakage* instruction. Towards formalizing this operation, we first recall that in the leaky UC model we assume that ideal functionalities (i.e., ITIs whose pid is $\perp$) may have ideal local states, where each ideal local state is associated with a pid. Now, we define the $\mathsf{total\ state}$ of a given pid P at a certain global configuration of a system of ITMs to be the concatenation of the local states of all the ITIs with pid P, together with the ideal local states that are associated with pid P within all the ideal functionalities in the system. (Intuitively, the total state of pid P corresponds to all the state that is physically available for joint leakage.)

Next, a leakage request (by $\mathcal{A}$) is assumed to specify a function L, represented by a circuit, and a pid P. As a consequence a new ITI called an aggregator $\mathcal{G}$ is invoked. The aggregator obtains the total state of P (say, by querying all the relevant ITIs), applies L to this global state, and returns the result to $\mathcal{A}$. In addition, all the ITIs with pid P and all the ideal functionalities that had ideal state associated with P

[5] In [6] modular protocols are called *subroutine respecting.*

are notified as to the number of bits in the description of the value given to $\mathcal{A}$. The leakage operation is described in Figures 1 and 2.

As in the case of passive party corruption, leakage messages are modeled as messages that are intercepted and processed by the "operating system" part, σ, of the program (σ, π) running each affected ITI μ. Here σ forwards the current state of π to the aggregator $\mathcal{G}$. When $\mathcal{G}$ returns the number of bits leaked, σ reports this number to all the "parent ITIs", namely to all the ITIs that either wrote on μ's input tape or that μ wrote on their subroutine output tapes. (Similarly to the case of party corruption, this reporting operation is central to the notion of UC security with leakage, in that it provides a formal guarantee as to the number of bits leaked in this leakage operation.) The state of the program π is not affected by the leakage operation, nor by the reporting of the number of bits leaked. (This accounts for the fact that real world leakage is undetectable.)

A leakage operation consists the following stages:

1. $\mathcal{A}$ sends $(\mathsf{leak}, L, \mathsf{pid})$ to $\mathcal{G}$, where L is the leakage function and pid is the leaking party ID.
2. $\mathcal{G}$ obtains $\mathsf{state}_{\mathsf{pid}}$, the total state of party pid, and returns $L\left(\mathsf{state}_{\mathsf{pid}}\right)$ to $\mathcal{A}$.
3. $\mathcal{G}$ reports the output length of the function L to all the subroutines of party pid.

Figure 1: The Leakage Operation

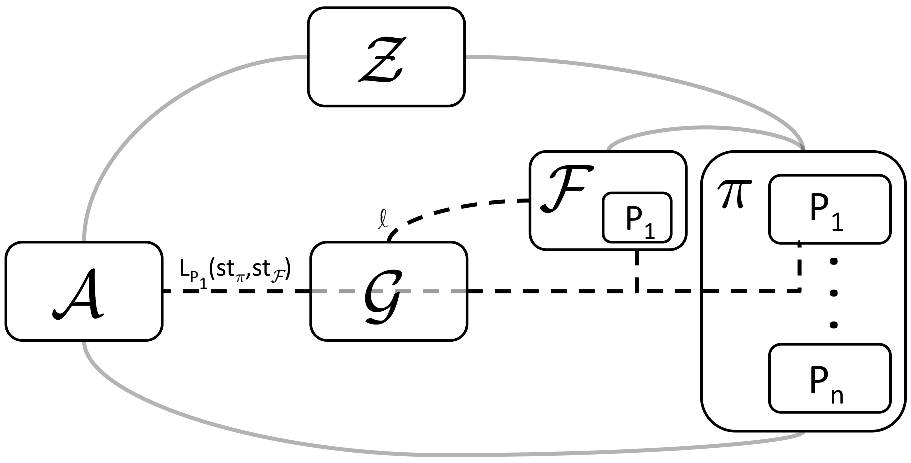


Figure 2: A leakage operation on party P_1 participating in protocol $\pi^{\mathcal{F}}$ (which makes calls to the ideal functionality $\mathcal{F}$). The leakage function L is sent to the aggregator which obtains $\mathsf{state}_{P_1} = (\mathsf{state}_\pi, \mathsf{state}_\mathcal{F})$ and returns $L(\mathsf{state}_{P_1})$. At the end of the operation the aggregator reports to $\mathcal{F}$ the number ℓ of leaked bits.

We remark that the aggregator can be thought of as a protocol component that aggregates multiple processes, or ITIs. Thus, formally speaking, the model of protocol execution did not change. Instead, we only specified a special class of protocols. We call such protocols leaky protocols. We note however that leaky protocols are not "modular", since they allow ITIs in some protocol instance to obtain input and provide output to the aggregator, which is not part of the calling protocol instance. This property will indeed be the main obstacle in formulating and proving a composition theorem for such protocols.

Standard leaky ideal functionalities. The standard behavior of ideal functionalities under leakage is a natural extension of the standard behavior of ideal functionalities under party corruption. That is, an instance of a *standard leakage* ideal functionality $\mathcal{F}$ keeps some "ideal local state" S_p for each dummy party (s, p) that interacts with this instance of $\mathcal{F}$. Next, upon receipt of each "leak p" message from the adversary, $\mathcal{F}$ forwards the state S_p to the aggregator. When the aggregator reports the number l of bits leaked in this operation, $\mathcal{F}$ reports this number l to the dummy party (s, p).

UC protocol 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 the implementation protocol or with $\mathcal{S}$ and the ideal specification protocol.

Definition 2.5 (Weak emulation for leaky protocols)**.** *Let π and ϕ be leaky protocols. Then π weakly* UC-*emulates ϕ for any PPT adversary $\mathcal{A}$ there exists a PPT simulator $\mathcal{S}$ such that for any PPT environment $\mathcal{Z}$ it holds* $\mathsf{EXEC}_{\mathcal{A},\pi,\mathcal{Z}} \approx_c \mathsf{EXEC}_{\mathcal{S},\phi,\mathcal{Z}}$.

However, the above definition of emulation turns out too weak in order to provide general composability guarantees. We thus define a stronger notion of emulation. For this notion, called *leakage-oblivious emulation,* we only consider the simulator for the dummy adversary. We require that there exists a simulator $\mathcal{S}$ for the dummy adversary that has the following additional property. We require that $\mathcal{S}$ has a special subroutine $\tilde{S}$ for handling leakage. Whenever $\mathcal{S}$ receives from the environment a request to apply a leakage function L to the state of pid P, $\tilde{S}$ is invoked to produce a "state translation circuit" T. The circuit T is meant to transform the internal state of P in the specification protocol into its state in the implementation protocol. Once T is produced, the aggregator is given the composed leakage circuit $L \circ T$. When the leakage result is returned, it is forwarded directly to the environment and $\mathcal{S}$ returns to its state prior to the leakage event. The operation of $\mathcal{S}$ during leakage is detailed in Figure 3.

We require that the subroutine $\tilde{S}$ operates **independently of the leakage circuit** L (its only input is the current state of $\mathcal{S}$ and pid). We stress that the leakage operation has **no side effect on** $\mathcal{S}$. That is, the state of $\mathcal{S}$ prior to the leakage event and its state after are the same. (It will be convenient to treat $\tilde{S}$ as a deterministic subroutine, namely all the necessary random choices are made in advance by $\mathcal{S}$.) We call such a simulator a *leakage-oblivious simulator*.

Given a message $(\mathsf{leak}, L, \mathsf{pid})$ from the environment, $\mathcal{S}$ acts as follows:

1. Record state_0, the state prior to the leakage query.
2. Invoke $\tilde{S}(\mathsf{state}_0, \mathsf{pid})$ to obtain a translation circuit T.
3. Send to aggregator $(\mathsf{leak}, L \circ T, \mathsf{pid})$, where $L \circ T$ is the composition of L on T.
4. Given the leakage result, forward it to the environment.
5. Return to state_0.

Figure 3: Handling Leakage with a Leakage Oblivious Simulator

Definition 2.6 (Leakage-oblivious emulation)**.** *Let π and ϕ be leaky protocols. Then π strongly* UC-*emulates ϕ if there exists a* **leakage-oblivious simulator** *$\mathcal{S}$ such that for any environment $\mathcal{Z}$:*

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

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

We remark that, since we have not changed the model of execution, the dummy adversary theorem from the basic model still holds here. Therefore we have that leakage-oblivious emulation implies standard emulation (w.r.t all adversaries).

3 Universal Composition of leaky protocols

We now state the universal composition theorem for leaky protocols and leakage-oblivious simulators (as defined in Section 2). Let π be an implementation and $\mathcal{F}$ be a specification. (As mentioned earlier, formally these are just two protocols, and the different names are meant only to help the intuition.) Also let $\rho = \rho[\pi]$ be a protocol that includes subroutine calls to π. Below we denote by ρ^{π} the system where the subroutine calls to π are actually processed by π and by $\rho^{\mathcal{F}}$ the system where these subroutine calls are processed by $\mathcal{F}$.[6]

The UC theorem [6] states that if π UC-realizes $\mathcal{F}$, then ρ^{π} UC-realizes $\rho^{\mathcal{F}}$; however, that theorem does not hold in the presence of the modularity-breaking aggregator $\mathcal{G}$. The proof of the UC-theorem in [6] relies on all the processes (ITIs) being "modular", namely a process can only interacts with its caller and its subroutines (and the adversary).[7] As we have seen, modularity is incompatible with the definition of leaky protocols; indeed, all processes are required to interact with the aggregator, which is neither their caller nor their subroutine (nor an adversarial entity). Still, if π realizes $\mathcal{F}$ with a leakage-oblivious simulator, we can recover the same result. Below we call a protocol "modular up to leakage" if it only interacts with its caller, its subroutines, the adversary and the aggregator.

Theorem 3.1 (UC-composition with leakage)**.** *Let $\rho, \pi, \mathcal{F}$ be protocols as above, all modular up to leakage, such that π UC-emulates $\mathcal{F}$ with a leakage-oblivious simulator. Then ρ^{π} UC-emulates $\rho^{\mathcal{F}}$. Furthermore, it does so with a leakage-oblivious simulator.*

Proof overview. The proof follows the outline of the proof of the basic UC theorem, here we focus on the required adjustments due the leakage. For sake of simplicity, in this overview we assume that ρ invokes only a single instance of the sub-protocol π. Recall that we need to construct a leakage-oblivious simulator $\mathcal{S}_\rho$ such that no environment can tell whether it is interacting with ρ^{π} and the $\mathsf{dummy\ adversary}$ $\mathcal{D}$, or with $\rho^{\mathcal{F}}$ and $\mathcal{S}_\rho$. The construction of $\mathcal{S}_\rho$ is naturally based on the leakage-oblivious simulator $\mathcal{S}_\pi$ as guaranteed by the premise. That is, $\mathcal{S}_\rho$ runs a copy of $\mathcal{S}_\pi$; As in the basic UC theorem, the interaction between $\mathcal{Z}$ and the parties is separated into two parts. The interaction with π is dealt with by $\mathcal{S}_\pi$, which generates messages for the corresponding sub-parties and handles incoming messages from these parties. The effect of the environment on rest of the system, is handled by direct interaction with the external parties running ρ.

Leakage queries are handled by way of a subroutine $\tilde{\mathcal{S}}_\rho$ that generates a state translation T_ρ, as needed for leakage-oblivious simulation. Recall that the leakage function L that $\mathcal{S}_\rho$ receives from the environment was designed to be applied to a "real protocol state" in ρ^{π} (and since ρ runs a single copy of π then this state is of the form $(\mathsf{state}_\rho, \mathsf{state}_\pi)$). The simulator $\mathcal{S}_\rho$, on the other hand, can only ask the aggregator for leakage on the state of $\rho^{\mathcal{F}}$, which is of the form $(\mathsf{state}_\rho, \mathsf{state}_{\mathcal{F}})$. To bridge this gap, $\tilde{\mathcal{S}}_\rho$ runs the "state translation subroutine" $\tilde{\mathcal{S}}_\pi$. (This can be done since $\mathcal{S}_\rho$ has the entire current state of $\mathcal{S}_\pi$.) Once $\tilde{\mathcal{S}}_\pi$ produces a state translation function T_π, $\tilde{\mathcal{S}}_\rho$ generates its own state translation function $T_\rho\left(\mathsf{state}_\rho, \mathsf{state}_{\mathcal{F}}\right) = \left(\mathsf{state}_\rho, T_\pi\left(\mathsf{state}_{\mathcal{F}}\right)\right)$ and sends to the aggregator the leakage function

$$L'\left(\mathsf{state}_\rho, \mathsf{state}_{\mathcal{F}}\right) \;=\; L\left(T_\rho\left(\mathsf{state}_\rho, \mathsf{state}_{\mathcal{F}}\right)\right) \;=\; L\left(\mathsf{state}_\rho, T_\pi\left(\mathsf{state}_{\mathcal{F}}\right)\right).$$

Observe that already at this stage we rely crucially on $\mathcal{S}_\pi$ being leakage-oblivious: If $\mathcal{S}_\pi$ was expecting to see a leakage function $L_\pi\left(\mathsf{state}_\pi\right)$ before producing the translation, then we could not use it (since

[6]In Section 2.2, $\rho^{\mathcal{F}}$ is denoted $\rho^{\mathcal{F}/\pi}$.

[7]This is also called "subroutine respecting".

$\mathcal{S}_\rho$ does not know the state state_ρ, and therefore cannot write the description of the induced function $L_{\mathsf{state}_\rho}(\mathsf{state}_\pi) = L(\mathsf{state}_\rho, \mathsf{state}_\pi)$). Once the aggregator returns an answer, $\mathcal{S}_\rho$ passes it to the environment and returns to its previous state (including the previous state of the sub-simulator $\mathcal{S}_\pi$).

It is clear from the description that $\mathcal{S}_\rho$ is leakage-oblivious. The validity of $\mathcal{S}_\rho$ is shown by reduction to the validity of $\mathcal{S}_\pi$. That is, given an environment $\mathcal{Z}_\rho$ that distinguishes an execution of $\rho^\pi, \mathcal{D}$ from an execution of $\rho^{\mathcal{F}}, \mathcal{S}_\rho$, we construct an environment $\mathcal{Z}_\pi$ that distinguishes an execution of $\pi, \mathcal{D}$ from an execution of $\mathcal{F}, \mathcal{S}_\pi$. The environment $\mathcal{Z}_\rho$ simulates an execution of $\mathcal{Z}_\rho, \rho$ "in its head", except that all messages corresponding to π are forwarded to the external execution. Indeed, leakage queries aside, we have: (a) If the external execution consists of $\mathcal{S}_\pi$ and $\mathcal{F}$ then the entire (composed) execution amounts to running $\mathcal{Z}_\rho$ with $\mathcal{S}_\rho$ and $\rho^{\mathcal{F}}$; (b) If the external execution consists of $\mathcal{D}$ and π then the entire (composed) execution amounts to running $\mathcal{Z}_\rho$ with $\mathcal{D}$ and ρ^π.

Extending this argument to include leakage, the environment $\mathcal{Z}_\pi$ acts as follows. When $\mathcal{Z}_\rho$ produces a leakage query L to be evaluated on $\mathsf{state}_\rho, \mathsf{state}_\pi$, $\mathcal{Z}_\pi$ computes the simulated state state_ρ, and computes the restricted leakage function $L_{\mathsf{state}_\rho}(\mathsf{state}_\pi) = L(\mathsf{state}_\rho, \mathsf{state}_\pi)$ which should be evaluated only on state_π. Note again that the $\mathcal{S}_\pi$ subroutine of $\mathcal{S}_\rho$ never sees this leakage function; however, since $\mathcal{S}_\pi$ is leakage-oblivious, then the state-translation function that it outputs when run as a subroutine of $\mathcal{S}_\rho$ is nonetheless the same as the state-translation function that it outputs when run with the environment $\mathcal{Z}_\pi$. The rest of the argument remains unchanged.

3.1 The Actual Proof

We now present the full proof, which deals with the case where multiple instances of the subroutine π are potentially invoked. For the reader who is not interested in the full details, the above overview should suffice for understanding the rest of the paper, and the following proof can be skipped.

proof. Let $\rho, \pi, \mathcal{F}$ be protocols as above, all modular up to leakage, such that π UC-emulates $\mathcal{F}$ with a leakage-oblivious simulator. We construct a leakage-oblivious simulator $\mathcal{S}_\rho$, so that no environment $\mathcal{Z}$ can tell whether it is interacting with the dummy adversary $\mathcal{D}$ and ρ^π, or with $\mathcal{S}_\rho$ and $\rho^{\mathcal{F}}$. We know that π emulates $\mathcal{F}$ with a leakage-oblivious simulator, and let $\mathcal{S}_\pi$ be this leakage-oblivious simulator. Then for any environment $\mathcal{Z}_\pi$: $\mathsf{EXEC}_{\mathcal{D},\pi,\mathcal{Z}_\pi} \approx_c \mathsf{EXEC}_{\mathcal{S}_\pi,\mathcal{F},\mathcal{Z}_\pi}$.

Simulator $\mathcal{S}_\rho$ proceeds as follows. Recall that an environment $\mathcal{Z}$ expects to interact with parties running ρ^π. We separate the interaction between $\mathcal{Z}$ and the parties into two parts. To mimic the invocation of each instance of π (and its descendants), $\mathcal{S}_\rho$ runs a corresponding instance of the simulator S_π. The copies of $\mathcal{S}_\pi$ will generate messages for the corresponding sub-parties as well as handle incoming messages from these parties. In addition, they will be utilized in leakage operations for constructing the required state transformation circuits. The rest of the interaction between $\mathcal{S}_\rho$ and the environment is forwarded directly to the external parties of $\rho^{\mathcal{F}}$.

A bit more precisely, let k be the number of instances of $\mathcal{F}$ used by $\rho^{\mathcal{F}}$ in the current execution. For each instance $\mathcal{F}^{(i)}$, $i \in [k]$ of $\mathcal{F}$ invoked by a party running $\rho^{\mathcal{F}}$, $\mathcal{S}_\rho$ runs a copy $\mathcal{S}_\pi^{(i)}$ of $\mathcal{S}_\pi$. Messages sent by $\mathcal{Z}$ to a party running $\pi^{(i)}$, are forwarded to the corresponding instance $\mathcal{S}_\pi^{(i)}$. Messages sent by $\mathcal{Z}$ to be delivered to other parties (that run either $\rho^{\mathcal{F}}$ or some other subroutine of $\rho^{\mathcal{F}}$), are delivered directly to theses parties. Incoming messages from parties running an instance $\mathcal{F}^{(i)}$ are forwarded to $\mathcal{S}_\pi^{(i)}$. Incoming messages from other parties are forwarded to $\mathcal{Z}$. Messages from $S_\pi^{(i)}$ to parties running $\mathcal{F}^{(i)}$ are forwarded to these parties. Messages from $\mathcal{S}_\pi^{(i)}$ to its environment are forwarded to $\mathcal{Z}$. Messages from other parties running $\rho^{\mathcal{F}}$ (or any descendant other than $\mathcal{F}$) are forwarded to $\mathcal{Z}$.

Handling leakage. When $\mathcal{S}_\rho$ receives a leakage message $(\mathsf{leak}, L, \mathsf{pid})$, it invokes the following subroutine $\tilde{\mathcal{S}}_\rho$. First, for $i = 1, ..., k$ $\tilde{\mathcal{S}}_\rho$ invokes $\tilde{\mathcal{S}}_\pi^{(i)}$, where $\tilde{\mathcal{S}}_\pi^{(i)}$ is the special leakage handling subroutine of $\mathcal{S}_\pi^{(i)}$. Then, $\tilde{\mathcal{S}}_\rho$ obtains the state translation functions $T_\pi^{(1)}, ..., T_\pi^{(k)}$ and produces the state translation function

$$T_\rho\left(\mathsf{state}_\rho, \mathsf{state}_\pi^{(1)}, \ldots, \mathsf{state}_\pi^{(k)}\right) = \left(\mathsf{state}_\rho, T_\pi^{(1)}\left(\mathsf{state}_\pi^{(1)}\right), \ldots, T_\pi^{(k)}\left(\mathsf{state}_\pi^{(k)}\right)\right).$$

Here $\mathsf{state}_\pi^{(i)}$ represents the state of the party corresponding to the i'th instance of π, and state_ρ represents the party's state with respect to ρ and any potential other subroutines thereof (other than π). Next $\mathcal{S}_\rho$ sends to the aggregator the leakage function $L'(\cdot) = L\left(T_\rho(\cdot)\right)$. The aggregator evaluates it on the appropriate state and returns

$$L\left(T_\rho\left(\mathsf{state}_\rho, \mathsf{state}_\mathcal{F}^{(1)}, \ldots, \mathsf{state}_\mathcal{F}^{(k)}\right)\right),$$

which $\mathcal{S}_\rho$ passes to the environment, before returning to its state prior to the leakage event. Before proving the validity of $\mathcal{S}_\rho$, we note that $\mathcal{S}_\rho$ is indeed a leakage-oblivious simulator.

Validity. The validity of $\mathcal{S}_\rho$ is shown based on the validity of $\mathcal{S}_\pi$, using a hybrid argument. Similarly to the proof of the UC theorem, for $i \in [k]$ we let ρ_i denote a hybrid protocol where the interaction with the first i instances of $\mathcal{F}$ remains unchanged, whereas the rest of the instances of $\mathcal{F}$ are replaced with instances of π.

For this purpose, ρ_i is executed with a hybrid adversary $\mathcal{S}_\rho^{(i)}$ which acts as $\mathcal{S}_\rho$ for messages corresponding to $\mathcal{F}^{(1)}, \ldots, \mathcal{F}^{(i)}$ (i.e. it runs copies of $\mathcal{S}_\pi$) and acts as the dummy $\mathcal{D}$ for all messages corresponding to the instances $\pi^{(i+1)}, \ldots, \pi^{(k)}$ (as well as for messages which do not correspond to π).

To translate a leakage query L during a leakage operation, $\mathcal{S}_\rho^{(i)}$ retrieves the translation circuits $T_1, \ldots, T_i$ using the leakage sub-routines $\tilde{\mathcal{S}}_\pi^{(1)}, \ldots, \tilde{\mathcal{S}}_\pi^{(i)}$, and sets

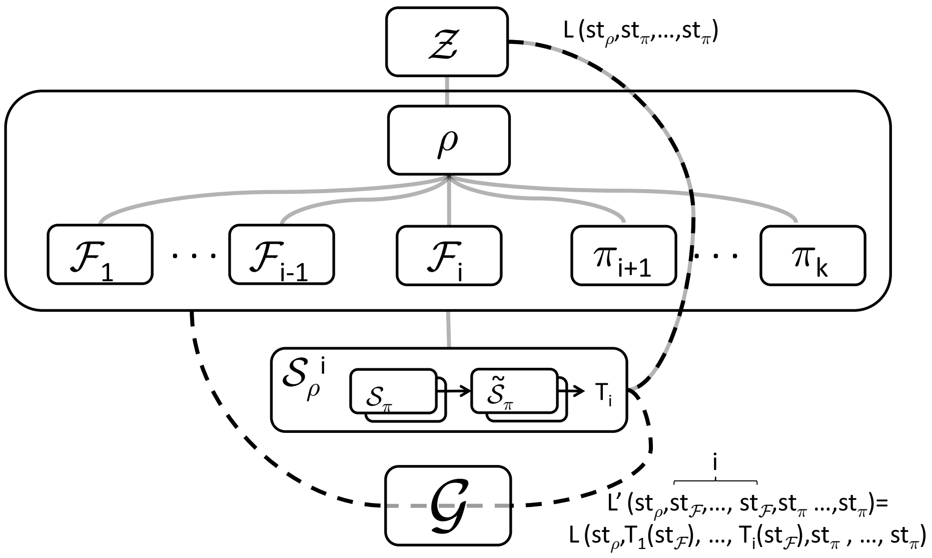
$$L'\left(\mathsf{state}_\rho, \mathsf{state}_\mathcal{F}^{(1)}, \ldots, \mathsf{state}_\mathcal{F}^{(i)}, \mathsf{state}_\pi^{(i+1)}, \ldots, \mathsf{state}_\pi^{(k)}\right) =$$
$$L'\left(\mathsf{state}_\rho, T_1\left(\mathsf{state}_\mathcal{F}^{(1)}\right), \ldots, T_i\left(\mathsf{state}_\mathcal{F}^{(i)}\right), \mathsf{state}_\pi^{(i+1)}, \ldots, \mathsf{state}_\pi^{(k)}\right)$$

Note that $\mathsf{EXEC}_{\rho_0, \mathcal{S}_\rho^{(0)}, \mathcal{Z}}$ is the same distribution ensemble as $\mathsf{EXEC}_{\rho^\pi, \mathcal{D}, \mathcal{Z}}$, and $\mathsf{EXEC}_{\rho_k, \mathcal{S}_\rho^{(k)}, \mathcal{Z}}$ is the same distribution ensemble as $\mathsf{EXEC}_{\rho, \mathcal{S}_\rho, \mathcal{Z}}$. Now, assume that there exists an environment $\mathcal{Z}$ that can distinguish an execution with $\mathcal{D}$ and ρ^π from an execution with $\mathcal{S}_\rho$ and $\rho^\mathcal{F}$. We construct an environment $\mathcal{Z}_\pi$ which distinguishes a stand alone execution π with $\mathcal{D}$ from an execution of $\mathcal{F}$ with $\mathcal{S}_\pi$.

$\mathcal{Z}_\pi$ chooses a random $i \in [k]$. It then runs a simulated execution of $\mathcal{Z}$ and ρ_i with the following exception. $\mathcal{Z}_\pi$ uses its external interaction (which is either with $\mathcal{F}$ and $\mathcal{S}_\pi$ or with π and $\mathcal{D}$) to replace the parts of the simulated execution corresponding to the interaction with the i'th instance of $\mathcal{F}$. More specifically, whenever some simulated party running ρ_i, passes an input to a party running $\mathcal{F}^{(i)}$, $\mathcal{Z}_\pi$ passes the input to the external party. Outputs generated by the external party are treated as outputs from $\mathcal{F}^{(i)}$ to the simulated party. Messages from the simulated $\mathcal{S}_\rho^{(i)}$ to $\mathcal{S}_\pi^{(i)}$ are forwarded to the external adversary and messages from the external adversary back to $\mathcal{Z}_\pi$ are treated as messages from the simulated $\mathcal{S}_\pi^{(i)}$ to the simulated $\mathcal{S}_\rho^{(i)}$. When the simulated $\mathcal{Z}$ halts, $\mathcal{Z}_\pi$ also halts and outputs the same. The hybrid executions of $\mathcal{Z}_\pi$ with ρ_i/ρ_{i+1}, and $\mathcal{Z}$ with ρ_i are depicted in Figure 4.

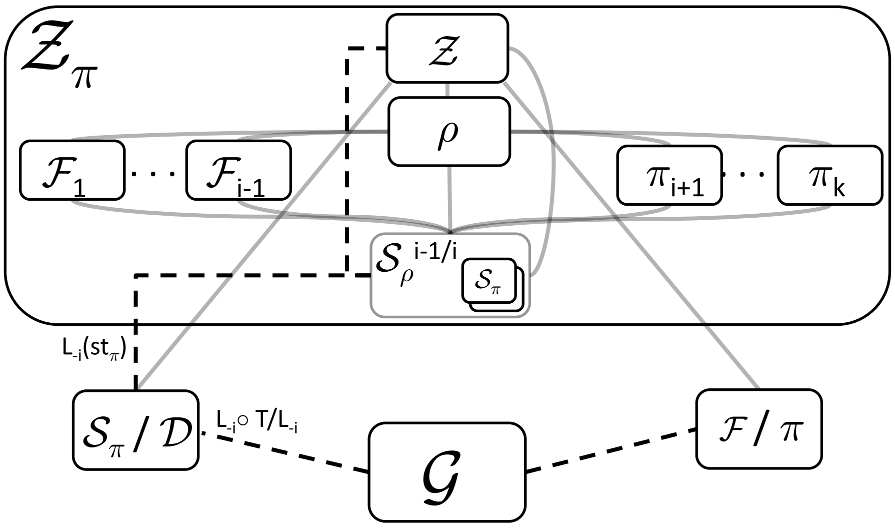
Leakage. To simulate the leakage operation which occurs when the simulated $\mathcal{Z}$ outputs a message $(\mathsf{leak}, L, \mathsf{pid})$, $\mathcal{Z}_\pi$ sends a message $(\mathsf{leak}, L', \mathsf{pid})$ to the external adversary, where L' is generated as follows. $\mathcal{Z}_\pi$ retrieves the translation circuits $T_1, \ldots, T_{i-1}$ from $\tilde{\mathcal{S}}_\pi^{(1)}, \ldots, \tilde{\mathcal{S}}_\pi^{(i-1)}$ and computes the simulated partial state

$$\mathsf{state}_{-i} = \left(\mathsf{state}_\rho, T_1\left(\mathsf{state}_{\mathcal{F}^{(1)}}\right), \ldots, T_{i-1}\left(\mathsf{state}_{\mathcal{F}^{(i-1)}}\right), \mathsf{state}_{\pi^{(i+1)}}, \ldots, \mathsf{state}_{\pi^{(k)}}\right)$$



$\mathcal{Z}$
L (st$_\rho$,st$_\pi$,...,st$_\pi$)
ρ
$\mathcal{F}_1$
$\mathcal{F}_{i-1}$
$\mathcal{F}_i$
π_{i+1}
π_k
$\mathcal{S}_\rho^{\,i}$
$\mathcal{S}_\pi$
$\tilde{\mathcal{S}}_\pi$
T$_i$
$\mathcal{G}$
i
L' (st$_\rho$,st$_\mathcal{F}$,..., st$_\mathcal{F}$,st$_\pi$ ...,st$_\pi$)=
L (st$_\rho$,T$_1$(st$_\mathcal{F}$), ..., T$_i$(st$_\mathcal{F}$),st$_\pi$, ..., st$_\pi$)

(a) Hybrid execution of $\mathcal{Z}$ with ρ_i $\mathcal{S}_\rho^{(i)}$



(b) Hybrid execution of $\mathcal{Z}_\pi$ with ρ_i, in which the i'th instance is either π or $\mathcal{F}$.

Figure 4: The analysis of $\mathcal{S}_\pi$. Since the simulator $\mathcal{S}_\rho$ is leakage-oblivious, the view of $\mathcal{Z}$ is the same in both executions. The dotted lines denote the leakage operation, while the gray solid lines denote the standard communication and input/output lines.

It then sets $L' = L_{-i}$, which is the restriction of L to state_{-i}. When the leakage result is returned to $\mathcal{Z}_\pi$, it is forwarded to $\mathcal{Z}$ as the result of its leakage query. In addition, in case Z_π is simulating any ideal functionalities, they are notified of the output length of L, simulating the notification they would get from the leakage aggregator in an actual execution.

The proof is completed by observing that if $\mathcal{Z}_\pi$ interacts with $\mathcal{S}_\pi$ and $\mathcal{F}$ (as the external protocol), the simulated $\mathcal{Z}$ has the same view as in an execution of ρ_i, while in case $\mathcal{Z}_\pi$ interacts with $\mathcal{D}$ and π, the simulated $\mathcal{Z}$ has the same view as in an execution of ρ_{i-1}.

It is stressed that the validity of the analysis relies in a crucial way on the fact that $\mathcal{S}_\pi$ is a leakage-oblivious simulator. In particular, the leakage queries in the two executions are structured differently. Still, this difference does not affect the the states of the instances of $\mathcal{S}_\pi$. Furthermore, in the two executions the environment obtains consistent leakage results. □

4 From Adaptive Security to Leakage Tolerance

Recall that the adversary in the UC framework can adaptively corrupt parties during the protocol execution, thereby learning their entire internal state. If the corruption is passive (semi-honest) then the party keeps following the same program as it did before the corruption, and if it is byzantine (malicious), then the adversary also gets to control the program that the corrupted party run from now on.

As already pointed out, leakage can be thought of as a form of corruption, where the adversary gains partial information on the inner state of a party. The opposite is also true, passive corruption can be viewed simply as leaking the entire internal state. The challenges in simulation are also similar: for both corruption and leakage the simulator must translate some "ideal state" that it gets from the functionality into a "real state" that it can show the environment, and do it in a way that is consistent with the transcript so far. Below we formalize this similarity, showing that "in principle" a protocol that realizes some functionality $\mathcal{F}$ in the presence of passive adaptive corruptions also realizes it in the presence of leakage. There are considerable restrictions, however. Most importantly, the implication holds only for oblivious simulators. (We defined leakage-oblivious simulator in Section 2, and will define corruption-oblivious simulator below.) Also, $\mathcal{F}$ must be adapted to handle leakage queries, and the implication holds only for a particular (natural) way of doing this adaptation, as we describe below.

Adapting functionalities to leakage. Let $\mathcal{F}$ be functionality that was designed for a leakage-free model with corruptions. This means that $\mathcal{F}$ already has some mechanism to reply to messages from the adversary about corruptions of players. We now need to adapt it by explaining how it reacts to leakage queries from the aggregator $\mathcal{G}$. The adaptation is natural: Any time that $\mathcal{G}$ asks for the state of party pid for the purpose of evaluating leakage, the functionality replies with exactly the same thing that it would have given the adversary if pid was corrupted at this time. Then, once $\mathcal{G}$ reports the number of leakage bits, the functionality forwards this number on the I/O lines of party pid. [8] Thereafter, the functionality behaves just as if party pid was corrupted. We denote the resulting functionality by $\mathcal{F}^{+\mathsf{lk}}$. We stress that if $\mathcal{F}$ was designed to react differently to passive and byzantine corruptions, then it uses the passive corruption mode to handle leakage.

Note the implication of viewing leakage as corruption: in principle, reaction to leakage could be gradual - a functionality $\mathcal{F}$ can change its behavior proportionally to the amount of leakage, or to have a leakage threshold up to which it does one thing and after which it does another. However, the reaction of $\mathcal{F}$ to (passive) corruption is typically "all or nothing", it is either not affected or it completely "gives up". Using our convention from above, this "all or nothing" reaction is carried over to $\mathcal{F}^{+\mathsf{lk}}$. For example, if $\mathcal{F}$ is an authenticated channels functionality, then $\mathcal{F}^{+\mathsf{lk}}$ will permit forgery as soon as even a single bit is leaked. On the other hand, if $\mathcal{F}$ is a commitment functionality then leakage events have no effect

[8]This number-reporting action is meant only to allow the environment to do its leakage bookkeeping.

on the subsequent behavior of $\mathcal{F}^{+\mathsf{lk}}$. Although the transformation that we prove below works for every functionality $\mathcal{F}$, its "meaningfulness" depends crucially on the way $\mathcal{F}$ handles passive corruptions.

Corruption-oblivious simulators. The intuition for why adaptive corruption implies leakage tolerance is that if we can simulate the **entire** state of an adaptively corrupted party, then we should also be able to simulate only parts of its state (according to a particular leakage function). The problem with this intuition, however, is that future behavior of the simulator may depend on the entire state that the simulator learns during corruption, which is not available to the leakage simulator.

We thus restrict our attention to special simulators that are oblivious to the state learned during corruption (similarly to the leakage-oblivious simulators from Section 2). As for leakage, we only define corruption-oblivious simulators for the dummy adversary $\mathcal{D}$ (which is sufficient). The simulator $\mathcal{S}$ for $\mathcal{D}$ should have a special subroutine $\tilde{S}$ for handling passive corruptions. When $\mathcal{S}$ receives from the environment a request $(\mathsf{passive\ corrupt}, \mathsf{pid})$ to passively corrupt a party pid, $\mathcal{S}$ invokes $\tilde{S}$ to produce a state translation function T. T is used to transform the "internal state" that $\mathcal{F}$ (or the hybrid-world protocol) returns for party pid, into a state of the "real world" implementation protocol π for this party. Then $\mathcal{S}$ sends a passive corrupt message to pid, obtains the corresponding state (from $\mathcal{F}$ or the hybrid-world instance), applies to it the transformation T and returns the result $\mathsf{state}_\pi = T(\mathsf{state})$ to the environment. After the result is forwarded to the environment, $\mathcal{S}$ returns to its state prior to the time it invoked $\tilde{S}$.

Note that since this is passive corruption, then party pid can keep evolving its state after the initial corruption, and the environment can ask to see the updated state from time to time. $\mathcal{S}$ handles each such update request as a new passive corruption query, invoking $\tilde{S}$ again to get state-translation function, calling the functionality again, etc. (We note that there is no restriction on the way that $\mathcal{S}$ handles Byzantine corruptions.) The details of handling passive corruptions are described in Figure 5.

Given a message $(\mathsf{passive\ corrupt}, \mathsf{pid})$ from the environment:

1. Record the current state, state_0.
2. Invoke $\tilde{S}(\mathsf{passive\ corrupt}, \mathsf{state}_0, \mathsf{pid})$ to obtain a translation circuit T.
3. Send pid a message $(\mathsf{passive\ corrupt})$ (when pid is a dummy party in an ideal protocol, the message is given to the ideal functionality).
4. Given the (updated) inner state of pid, apply T and forward the result to the environment.
5. Return to state_0.

Figure 5: Handling Corruptions with a Strong Simulator $\mathcal{S}$

We stress that $\mathcal{S}$ does not make any direct use of the state of the corrupted parties. In particular, the future operation of $\mathcal{S}$ with respect to simulating the messages generated by corrupted parties is done independently of their secret local states. As seen in subsequent sections, in some cases this turns out to be a strong restriction. We are now ready to state and prove the main result of this section.

Theorem 4.1. *Let π be a protocol that* UC*-realizes an ideal functionality $\mathcal{F}$ in the presence of passive adaptive corruptions (but no leakage), with a corruption-oblivious simulator. Then π also* UC*-realizes $\mathcal{F}^{+\mathsf{lk}}$ with a leakage-oblivious simulator in the* UC *model with leakage.*

Proof. Let $\mathcal{S}$ be the corruption-oblivious simulator for the dummy adversary against π (in the non-leaky model). We construct a leakage-oblivious simulator $\mathcal{S}_L$ for the dummy adversary against π in model of UC with leakage. $\mathcal{S}_L$ runs an emulated copy of $\mathcal{S}$, letting it deal with the entire interaction with the environment and the functionality $\mathcal{F}^{+\mathsf{lk}}$, with the exception of leakage queries. Whenever the environment $\mathcal{Z}$ sends a leakage message $(\mathsf{leak}, L, \mathsf{pid})$, $\mathcal{S}_L$ invokes a special leakage subroutine $\tilde{\mathcal{S}}_L$ as follows: $\tilde{\mathcal{S}}_L$ just runs the passive corruption subroutine $\tilde{S}$ with pid and the state of the emulated $\mathcal{S}$ as input, and outputs whatever translation function T that $\tilde{\mathcal{S}}_L$ outputs. Then $\mathcal{S}_L$ send to the aggregator the composed leakage circuit $L \circ T$, forwards the reply to the environment, and returns to its state prior to the leakage event. Clearly $\mathcal{S}_L$ is leakage-oblivious, it remains to argue that it is valid.

The validity of $\mathcal{S}_L$ is shown based on the validity of $\mathcal{S}$. Assume towards contradiction there exist an environment $\mathcal{Z}_L$ which distinguishes an execution of π with $\mathcal{D}$ from an execution of the ideal protocol $I_{\mathcal{F}^{+\mathsf{lk}}}$ with $\mathcal{S}_L$. We construct an environment $\mathcal{Z}$ which distinguishes an execution of π with $\mathcal{D}$ from an execution of $I_{\mathcal{F}}$ with $\mathcal{S}$ in the non-leaky model. $\mathcal{Z}$ runs an emulated copy of $\mathcal{Z}_L$, every exchange of messages between $\mathcal{Z}$ and the external adversary is done according to $\mathcal{Z}_L$, with the exception of leakage messages which are dealt with as follows. Given a message $(\mathsf{leak}, L, \mathsf{pid})$, $\mathcal{Z}$ provides an input$(\mathsf{passive\ corrupt}, \mathsf{pid})$ to the external adversary. When the inner state of pid is returned, $\mathcal{Z}$ applies the leakage function L to it and gives it to the emulated $\mathcal{Z}_L$. In addition, $\mathcal{Z}$ notifies $\mathcal{Z}_L$ regarding the number of leakage bits that it obtained (emulating the corresponding notification in the leakage model). It is left to observe that, by our construction of $\mathcal{Z}_L$: (a) When the external adversary is $\mathcal{D}$, and the protocol is π, the view of the emulated $\mathcal{Z}_L$ is distributed exactly as its view in a leaky execution with $\mathcal{D}$ and π; (b) When the external adversary is $\mathcal{S}$, and the protocol is $I_{\mathcal{F}}$, the ideal protocol for $\mathcal{F}$, the view of the emulated $\mathcal{Z}_L$ is distributed exactly as its view in a leaky execution with $\mathcal{S}_L$ and the ideal protocol for $\mathcal{F}^{+\mathsf{lk}}$. We stress that the last implication strongly relies on the fact that $\mathcal{S}$ does not depend on the result of a corruption, and that $\mathcal{F}^{+\mathsf{lk}}$ under (even a single bit of) leakage from a PID acts as $\mathcal{F}$ under adaptive corruption of that PID. □

On the necessity of oblivious simulation. The following example highlights the need for corruption-oblivious simulation, and the importance of the exact definition of what state belongs to what party within ideal functionalities.

Example 4.1. Let $\mathcal{F}$ be the standard corruption functionality that takes n-bit inputs from two parties, P_0 and P_1, and outputs nothing. As soon as party P_i provides input x_i, the virtual local state of P_i is set to x_i. Now, consider the following protocol π: First, the parties give their inputs to some trusted party that returns a random b_i to P_i such that $b_0 + b_1 = \langle x_0, x_1 \rangle$ where $\langle , \rangle$ denotes inner-product in $\mathbb{F}_2$. (The inner product can be replaced by any two-source extractor.) Next, the parties output nothing and halt.

It can be seen that π securely realizes $\mathcal{F}$ with respect to adaptive corruptions. This is so since, once the first party P_i is corrupted, the simulator learns x_i and can give x_i to the adversary, plus a random bit instead of b_i. Now, when P_{1-i} is corrupted, the simulator learns x_{1-i} and can determine the bit b_{1-i} so that $b_0 + b_1 = \langle x_0, x_1 \rangle$. However, notice that here the simulator is not oblivious: The handling of the second corruption depends on the input value x_i of the first corrupted party. Indeed, π does not seem to realize $\mathcal{F}^{+\mathsf{lk}}$ with even one bit of leakage from each party, and even with respect to the weak notion of leakage resilience (Definition 2.5). The adversary can ask to leak b_i from P_i and thus learn $\langle x_0, x_1 \rangle$. However, in the ideal model for $\mathcal{F}^{+\mathsf{lk}}$, assuming x_0, x_1 are long random strings, the simulator has no hope of learning $\langle x_0, x_1 \rangle$. This is so since in the ideal model, the simulator can only perform one-bit leakage on x_0 and x_1 separately, and hence it can not guess $\langle x_0, x_1 \rangle$ with non-negligible advantage.

Finally notice that, had we modified $\mathcal{F}$ so that the virtual local state of each party includes both inputs, the adaptive simulation would become oblivious, and the protocol would UC-realize $\mathcal{F}^{+\mathsf{lk}}$ with leakage. But then, of course, $\mathcal{F}^{+\mathsf{lk}}$ would provide only a weak level of security.

Composition of corruption-oblivious simulators. We note that since corruption-oblivious simulators can be viewed as a special case of leakage-oblivious simulators (for leaking the identity function), the proof of the leaky UC Theorem 3.1 implies that corruption-oblivious simulation is preserved under universal composition:

Corollary 4.1 (of Theorem 3.1)**.** *Let $\rho, \pi, \mathcal{F}$ be protocols that are modular upto leakage, such that π UC-emulates $\mathcal{F}$ with a corruption-oblivious simulator. Then ρ^{π} UC-emulates $\rho^{\mathcal{F}}$ with a corruption-oblivious simulator.*

5 Realizing Leaky Adaptations of Basic Interactive Tasks

This section describes the construction of leakage tolerant protocols for for several interactive tasks. We describe constructions for secure message transmission, (semi-honest) oblivious transfer, commitment and zero knowledge. The latter constructions all assume ideal authenticated channels. We then present a (composable) construction for leaky authenticated channels.

The leakage tolerant variants of the first three primitives are formulated and realized using the approach described in Section 4. That is, given an ideal functionality $\mathcal{F}$ we demonstrate existing protocols that UC-realize $\mathcal{F}$ with adaptive (semi-honest) corruptions, and where there is a corruption-oblivious simulator. We then conclude that the same protocol UC-realizes $\mathcal{F}^{+\mathsf{lk}}$ in the presence of arbitrary leakage. We note that for all three functionalities, given a passive corruption request, the functionality does not modify its behavior other than handing the private inputs and outputs of the corrupted party to the adversary. This implies that $\mathcal{F}^{+\mathsf{lk}}$actually guarantees optimal leakage tolerance.

The leakage tolerant variants of zero knowledge and authentication are formulated and realized in other ways. In the case of authentication, the reason to depart from the paradigm of Section 4 is that the leaky version of the ideal authentication functionality, $\mathcal{F}_{\mathsf{AUTH}}^{+\mathsf{lk}}$ provides essentially no security in face of leakage of even one bit from the sender. Thus, a stronger functionality is needed. In contrast, in the case of zero knowledge the leaky version of $\mathcal{F}_{\mathsf{ZK}}$ provides a *strong* guarantee, which we could not achieve. We thus relax it to allow for a realization in the presence of leakage.

Non-Committing Encryption. We first recall the definition of non-committing encryption (NCE) [8], which is a basic ingredient in our construction of leakage tolerant secure communication and oblivious transfer protocols. Informally, non-committing encryption schemes are semantically secure (bit) encryption schemes, with the additional property that a simulator can generate special public keys and ciphertexts that can be "opened" to (i.e. demonstrated to be the encryption of) both a 0 and a 1.

Definition 5.1. *[8] A non-committing (bit) encryption (NCE) scheme consists of a tuple* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Sim})$ *where* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is a semantically secure encryption scheme (with negligible decryption error), and* Sim *is a* PPT *simulation algorithm that on input* 1^k *outputs a tuple* $(e, c, r_G^0, r_E^0, r_G^1, r_E^1)$ *such that for every* $b \in \{0,1\}$ *the following distributions are computationally indistinguishable:*

1. *The joint view of an honest sender and an honest receiver in a normal encryption of b:*
$$\{(e, c, r_G, r_E) : (e, d) = \mathsf{Gen}(1^k; r_G), c = \mathsf{Enc}_e(b; r_E)\}$$
2. *A simulate view of an encryption of b:*
$$\{(e, c, r_G^b, r_E^b) : (e, c, r_G^0, r_E^0, r_G^1, r_E^1) \leftarrow \mathsf{Sim}(1^k)\}$$

The scheme is said to have oblivious key sampling if in addition to the above:

- *There is an oblivious public key sampling algorithm* $\widehat{\mathsf{Gen}}$*, which on input* $1^k, r_{\hat{G}}$ *samples a public key* $\hat{e}$ *which is indistinguishable from the public keys generated by* Gen*.*
- *There is a reverse sampling algorithm which given a public key* e *generated by* Gen*, outputs a random* $r_{\hat{G}}$ *such that* $\widehat{\mathsf{Gen}}(1^k, r_{\hat{G}}) = e$*.*

5.1 Secure Message Transmission

This section shows that the standard NCE-based protocol for adaptively secure message transmission over ideally authenticated channels has natural leakage tolerance properties. For simplicity, we concentrate on the model of passive corruptions.

We first recall the standard secure message transmission functionality, $\mathcal{F}_{\mathsf{SMT}}$ (see e.g. [6]). The functionality is invoked by the sender with a message to be sent and the identity (pid) of the intended recipient. $\mathcal{F}_{\mathsf{SMT}}$ first reports the length of the message and the identities of the sender and recipient to the adversary. Once the adversary approves, the message is delivered to the intended recipient. When the adversary asks to corrupt either the sender or the recipient, $\mathcal{F}_{\mathsf{SMT}}$ outputs to the relevant party that it has been corrupted, and discloses the message to the adversary.

In spite of the fact that the standard realization of secure message transmission is done in a bit by bit manner, we define $\mathcal{F}_{\mathsf{SMT}}$ for messages of arbitrary length. This allows presenting the security requirements in the presence of leakage in a clearer way.

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

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

- Given input $(\mathsf{send}, S, R, \mathsf{sid}, m)$, send a message $(\mathsf{send}, S, R, |m|, \mathsf{sid})$ to the adversary $\mathcal{S}$. Once $\mathcal{S}$ allows forwarding the message, send $(\mathsf{sent}, S, \mathsf{sid}, m)$ to R.
- Given a message $(\mathsf{corrupt}, P)$, where $P \in \{S, R\}$, disclose m to the adversary.

Figure 6: The Message Transmission Functionality

The leaky adaptation. Consider $\mathcal{F}^{+\mathsf{lk}}_{\mathsf{SMT}}$, the leaky adaptation of $\mathcal{F}_{\mathsf{SMT}}$. Recall that $\mathcal{F}^{+\mathsf{lk}}_{\mathsf{SMT}}$ behaves the same as $\mathcal{F}_{\mathsf{SMT}}$ under passive adaptive corruptions; namely, the leaky state of parties is the message m, and the functionality continues to operate as usual when leakage occurs. In addition, $\mathcal{F}^{+\mathsf{lk}}_{\mathsf{SMT}}$ reports the number of bits leaked to the affected party (which just forwards it to the environment).

We now recall the standard use of NCE to realize $\mathcal{F}_{\mathsf{SMT}}$ in the face of adaptive corruption, assuming authenticated channels. Furthermore, we observe that the resulting protocol has a corruption-oblivious simulator. This will enable us to deduce leakage resilience properties of this protocol.

Protocol SMT

1. Given input $(\mathsf{send}, S, R, \mathsf{sid}, m)$, S notifies R of its intention to send an $|m|$-bit message.
2. When R is informed of the transmission intention, it samples secret and public keys $(d_i, e_i) \leftarrow \mathsf{Gen}(1^k, r_G)$, for all $1 \leq i \leq |m|$, and sends $\{e_i\}$ to S.
3. Upon receiving $\{e_i\}$ from R, S encrypts $c_i \leftarrow \mathsf{Enc}_{e_i}(m; r_E)$ and sends $\{c_i\}$ to R.
4. When R receives $\{c_i\}$ it decrypts using Dec_{d_i} and outputs the result.

Figure 7: Message Transmission Protocol based on NCE

Proposition 5.1. *Protocol* SMT UC*-realizes* $\mathcal{F}_{\mathsf{SMT}}$ *in the face of adaptive semi-honest adversaries, assuming authenticated channels. Moreover, emulation is done with a corruption-oblivious simulator.*

Corollary 5.1 (of Theorem 4.1 and Proposition 5.1)**.** *Protocol* SMT UC*-realizes* $\mathcal{F}^{+\mathsf{lk}}_{\mathsf{SMT}}$ *with a leakage-oblivious simulator, assuming authenticated channels.*

Proof sketch. We construct a corruption-oblivious simulator $\mathcal{S}$ for the dummy adversary. Recall that $\mathcal{S}$ should have a special subroutine $\tilde{S}$ for dealing with corruptions by state translation, and that $\mathcal{S}$ itself should not be directly affected by the corruption. The simulator $\mathcal{S}$ operates as follows. When $\mathcal{S}$ gets a message $(\mathsf{send}, S, R, |m|, \mathsf{sid})$ from the functionality $\mathcal{F}_{\mathsf{SMT}}$, it samples $|m|$ equivocal tuples

$$\mathsf{equiv_i} = \left(e_i, c_i, r^0_{G,i}, r^0_{E,i}, r^1_{G,i}, r^1_{E,i}\right) \leftarrow \mathsf{Sim}(1^k)$$

It then simulates the messages between S, R using the equivocal keys $\{e_i\}$ and ciphers $\{c_i\}$. We now describe how passive corruptions are handled by the special subroutine $\tilde{S}$, according to their timing.

1. **Prior to message exchange:** Here, we are guaranteed that in a real execution, the receiver R still did not sample any keys; indeed, the sampling of keys is only done after a notification message from sender S. Hence, translating the ideal state of R to a real one is straightforward. The same also holds for S, whose state may include m (but still not the randomness for future encryptions). Hence, state translation is also trivial. The state of the corrupted party in both cases is unnecessary for the simulation to go on, as required for corruption-oblivious simulation.

2. **Subsequent corruptions:** Here, we are guaranteed that the functionality $\mathcal{F}_{\mathsf{SMT}}$ already contains the message m, and corruption can be dealt with as follows. The subroutine $\tilde{S}$ is invoked with the $\{\mathsf{equiv_i}\}$ tuples, the identity $P \in \{S, R\}$ of the corrupted party, and the transcript simulated so far. It produces a translation circuit T which given the message m (which is the ideal state) returns randomness (a real state) for P which is consistent with m (and the timing of the corruption in the protocol). Specifically:

   - In case the sender is corrupted (i.e. $P = S$), T returns m and if the ciphers $\{c_i\}$ were already sent, it also returns $\left\{r^{m_i}_{E,i}\right\}$.
   - In case the receiver is corrupted (i.e. $P = R$), T returns $\left\{r^{m_i}_{G,i}\right\}$ and if the ciphers $\{c_i\}$ were already sent it also returns m.

   Once T is produced, m is obtained from the functionality $\mathcal{F}_{\mathsf{SMT}}$, $T(m)$ is forwarded to the environment, and $\mathcal{S}$ returns to its state prior to the corruption message.

As defined above $\mathcal{S}$ is indeed a corruption-oblivious simulator. The validity of $\mathcal{S}$ follows directly from the properties of the NCE. Indeed, if both parties are corrupted, the view of the environment consists of $\left\{\left(m_i, e_i, c_i, r^{m_i}_{G,i}, r^{m_i}_{E,i}\right)\right\}$, while in any other case, it will contain some proper part of each of these tuples. In all cases, this view is either simulated by Sim (in the ideal execution) or generated by the NCE algorithms (in the real execution). The NCE assures that the two cases are indistinguishable. □

5.2 Oblivious Transfer

Oblivious transfer [22, 13] is a two-party functionality, involving a sender with ℓ inputs $x_1, ..., x_\ell$, and a receiver with input $i \in [\ell]$. The receiver should learn x_i (and nothing else) and the sender should learn nothing. Again we concentrate on the model of passive corruptions. To simplify notation we consider the case that the x_i's are bits. A detailed definition of the ideal oblivious transfer functionality, denoted $\mathcal{F}_{\mathsf{OT}^\ell}$, appears in Figure 8.

The leaky adaptation. Once again we recall that the leaky variant $\mathcal{F}^{+\mathsf{lk}}_{\mathsf{OT}^\ell}$ (ideal state and response to leakage) is defined according to the behavior of $\mathcal{F}_{\mathsf{OT}^\ell}$ under passive corruptions. That is, the leaky state of the sender consists of $x_1, \ldots, x_\ell$, the state of the receiver consists of i, x_i and the functionality does not change its behavior after leakage.

Functionality $\mathcal{F}_{\mathsf{OT}^\ell}$

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

1. Upon receiving a message $(\mathsf{send}, S, R, \mathsf{sid}, x_1, ..., x_\ell)$ from party S:
   (a) If a record (S, R, i) exists then set $\mathsf{output} \leftarrow x_i$; Else, record $(S, R, x_1, ..., x_\ell)$.
   (b) Send $(\mathsf{send}, S, R, \mathsf{sid})$ to the adversary.
2. Upon receiving a message $(\mathsf{receive}, S, R, \mathsf{sid}, i)$ from party R, do:
   (a) If a record $(S, R, x_1, ..., x_\ell)$ exists then set $\mathsf{output} \leftarrow x_i$. Else, record (S, R, i).
   (b) Send $(\mathsf{receive}, S, R, \mathsf{sid})$ to the adversary.
3. Once output is defined and the adversary allows forwarding the message, output this value to R.
4. Upon receiving $(\mathsf{corrupt}, S)$, disclose $x_1, \ldots, x_\ell$ to the adversary.
5. Upon receiving $(\mathsf{corrupt}, R)$, disclose i to the adversary, and if the output x_i was already set, disclose it as well.

Figure 8: The oblivious transfer functionality, $\mathcal{F}_{\mathsf{OT}^\ell}$

We now describe how NCE with oblivious key sampling is used to realize $\mathcal{F}_{\mathsf{OT}^\ell}$ in the face of adaptive semi-honest corruptions [11]. Once again, to deduce security in the face of leakage we will show that emulation can be done with corruption-oblivious simulation.

Protocol OT

1. Given input $(\mathsf{send}, S, R, \mathsf{sid}, x_1, \ldots, x_\ell)$, S notifies R of its intention to transfer a message.
2. Given input $(\mathsf{receive}, S, R, \mathsf{sid}, i)$, R waits to receive the notification from S. Once such a notification is accepted, R samples one pair of secret and public keys $(d, e) \leftarrow \mathsf{Gen}(1^k, r_G)$ and $\ell - 1$ oblivious keys
   $\hat{e}_j \leftarrow \widehat{\mathsf{Gen}}(1^k, r_{\hat{G},j})$, where $j \in [\ell] - \{i\}$. It then sends $\vec{e} = (\hat{e}_1, \ldots, \hat{e}_{i-1}, e, \hat{e}_{i+1}, \ldots, \hat{e}_\ell)$ to S.
3. Upon receiving $\vec{e}$ from R, S encrypts $c_i \leftarrow \mathsf{Enc}_e(m; r_{E,i})$, and for all $j \neq i$: $c_j \leftarrow \mathsf{Enc}_{\hat{e}_j}(m; r_{E,j})$. It sends $\vec{c}$ to R.
4. When R receives $\vec{c}$ it decrypts c_i using Dec_d and outputs the result.

Figure 9: Oblivious Transfer Protocol based on NCE [11]

Proposition 5.2. *Protocol* OT UC*-realizes* $\mathcal{F}_{\mathsf{OT}^\ell}$ *in the face of adaptive semi-honest adversaries, assuming authenticated channels. Moreover emulation is done with a corruption-oblivious simulator.*

Corollary 5.2. *Protocol* OT UC*-realizes* $\mathcal{F}^{+\mathsf{lk}}_{\mathsf{OT}^\ell}$ *with a leakage-oblivious simulator (under the same terms).*

Proof sketch. We construct a corruption-oblivious simulator $\mathcal{S}$ for the dummy adversary. The simulator $\mathcal{S}$ operates as follows. Once it gets both messages $(\mathsf{send}, S, R, \mathsf{sid})$, $(\mathsf{receive}, S, R, \mathsf{sid})$ from the functionality $\mathcal{F}_{\mathsf{OT}^\ell}$, it samples ℓ equivocal tuples

$$\mathsf{equiv}_j = (e_j, c_j, r^0_{G,j}, r^0_{E,j}, r^1_{G,j}, r^1_{E,j}) \leftarrow \mathsf{Sim}(1^k)$$

In addition, $\mathcal{S}$ applies the reverse sampling algorithm to obtain $r_{\hat{G},j}$, such that $e_j = \widehat{\mathsf{Gen}}(1^k, r_{\hat{G},j})$ for all $j \in [\ell]$. These values are stored and will be used to handle corruptions. $\mathcal{S}$ then simulates the messages between S, R using the equivocal public keys $\{e_j\}$ and ciphers $\{c_j\}$. We next describe how passive corruptions are handled by the special subroutine $\tilde{S}$, according to their timing.

1. **Prior to message exchange:** In an execution of protocol OT, neither party has so far sampled randomness for the encryptions. Hence, translating their ideal state to a real one is simple. Furthermore, the state of the corrupted party in both cases is unnecessary for $\mathcal{S}$ to continue the simulation, as required for corruption-oblivious simulation.

2. **Subsequent corruptions:** Here, we are guaranteed that the functionality $\mathcal{F}_{\mathsf{OT}^\ell}$ already contains $x_1, \ldots, x_\ell$, and corruption can be dealt with as follows. The subroutine $\tilde{S}$ is invoked with the equiv_j tuples, the identity $P \in \{S, R\}$ of the corrupted party, and the transcript simulated so far. It produces a translation circuit T which given the ideal state returns a consistent real state. Specifically:

   - In case the sender is corrupted (i.e. $P = S$), $T(x_1, \ldots, x_\ell)$ returns $(x_1, \ldots, x_\ell)$ and if the cipher $\vec{c}$ was already sent, it also returns $(r_{E,j}^{x_j})_j$.
   - In case the receiver is corrupted (i.e. $P = R$), $T(i, x_i)$ returns:
   $$r_{\hat{G},1}, \ldots, r_{\hat{G},i-1}, r_G^{x_i}, r_{\hat{G},i+1}, \ldots, r_{\hat{G},\ell}$$
   If the cipher $\vec{c}$ was already sent it also returns x_i.

   Once T is produced, the inner state is obtained from the functionality, T is applied and the result is forwarded to the environment. Finally, $\mathcal{S}$ returns to its state prior to the corruption message.

It is evident that $\mathcal{S}$ operates as a corruption-oblivious simulator. The validity of $\mathcal{S}$ follows directly from the properties of the NCE (with oblivious key sampling). Indeed, if both parties are corrupted the view of the environment consists of $(x_j, e_j, c_j, r_{\hat{G},j}, r_{E,j}^{x_j})$, where $j \in [\ell] - \{i\}$, and $(x_i, e_i, c_i, r_{G,i}^{x_i}, r_{E,i}^{x_i})$. In any other case, it will contain a proper part of these elements. In all cases, this view is either simulated by Sim and the reverse sampling algorithm (in the ideal execution) or generated by the NCE algorithms (in the real executions). The NCE guarantees that the two cases are indistinguishable. □

5.3 Commitment

We show how to obtain universally composable commitment protocols with optimal leakage tolerance. That is, informally, when obtaining ℓ bits of arbitrary leakage from the committer, the adversary essentially obtains at most ℓ bits of information on the committed string. Leakage from the receiver gives no advantage whatsoever to the adversary. In particular, it does not increase the ability of the adversary to compromise the binding property of the commitment.

This is done by observing that existing protocols for UC-realizing the multi-commitment functionality, $\mathcal{F}_{\mathsf{MCOM}}$, in the presence of adaptive corruptions, actually do so with corruption-oblivious simulators. This allows us to apply Theorem 4.1 and conclude that these same protocols UC-realize $\mathcal{F}_{\mathsf{MCOM}}^{+\mathsf{lk}}$ with strong leakage simulators.

Let us recall the formulation of $\mathcal{F}_{\mathsf{MCOM}}$ (see e.g. [6, 11]). The functionality allows for multiple concurrent two-party interactions, where each interaction consists of two phases: a commit phase, where the receiver of the commitment obtains some information which amounts to a "commitment" to an unknown value, and a reveal, phase where the receiver obtains an "opening" of the commitment to some value, and verifies whether the opening is valid. The security guarantees are: (a) The committed value remains secret until explicitly opened by the committer. (b) Once the commit phase ends, there is only a single value that the receiver will accept as a valid opening. In addition, of course, a committer and receiver that follow the protocol should be able to commit and open values successfully.[9]

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

1. Upon receiving an input $(\mathsf{commit}, \mathsf{sid}, \mathsf{cid}, C, R, x)$ from C (or from the adversary if (cid, C, R) is marked as corrupted), where $x \in \{0,1\}^*$, do: If no (cid, C, R, x') tuple is recorded then record (cid, C, R, x), and send $|x|$ to the adversary. Once the adversary acknowledges, output $(\mathsf{receipt}, \mathsf{sid}, \mathsf{cid}, C)$ to R.

   In addition, if (cid, C, R) is corrupted, (cid, C, R, x') is already recorded for some x' and $(\mathsf{receipt}, \mathsf{sid}, \mathsf{cid}, C)$ still has not been output to R then replace x for x' in the recorded tuple.
2. Upon receiving an input $(\mathsf{open}, \mathsf{sid}, \mathsf{cid}, R)$ from C (or from the adversary if (cid, C, R) is marked as corrupted) do: If there is a recorded value (cid, C, R, x) then generate a delayed output $(\mathsf{open}, \mathsf{sid}, \mathsf{cid}, x)$ to R. Otherwise, do nothing.
3. Upon receiving a message $(\mathsf{Byzantine-committer-corrupt}, \mathsf{sid}, \mathsf{cid}, C, R)$ from the adversary, output $(\mathsf{Byzantine-corrupted}, \mathsf{cid})$ to C and mark (cid, C, R) as corrupted.

Figure 10: The Ideal Multi-instance Commitment functionality, $\mathcal{F}_{\mathsf{MCOM}}$. C and R are party identities (PIDs), sid is the session identity (SID) of the present instance, and cid is the commitment identifier.

Proposition 5.3. *The protocols of [9] and [11] for realizing $\mathcal{F}_{\mathsf{MCOM}}$ in the presence of adaptive corruptions in the common reference string model do so with corruption-oblivious simulators.*

Corollary 5.3. *The protocols of [9] and [11] UC-realizes $\mathcal{F}^{+\mathsf{lk}}_{\mathsf{OT}^\ell}$ with a leakage-oblivious simulator (under the same terms).*

Proof Sketch. The proof amounts to merely observing that the simulators described in the analyses in [9] and [11] are (almost) already corruption-oblivious simulators. We briefly recall the protocol and simulator of [9], as well as the slight modification needed in order to make it a strong simulator. The case of the [11] commitment protocol is analogous.

In the [9] protocol, the reference string consists of an instance f_0, f_1 of a claw-free pair of trapdoor permutations, plus a public encryption key e of a CCA-secure encryption scheme (G, E, D), which has the additional property that ciphertexts are pseudorandom. To commit to a string x, with parameters $\sigma = (sid, pid, C, R)$, the committer proceeds as follows: For each $i = 1..|x|$, the committer chooses a random r_i, sets $y_i = f_{x_i}$, and sends $\sigma, y_i, c_{i,0}, c_{i,1}$ to the receiver, where $c_{i,x_i} = E(e, \sigma \circ i \circ r_i, \rho_i)$, ρ_i is the randomness used for encryption, and $c_{i,1-x_i}$ is chosen at random from $\{0,1\}^{|c_{i,x_i}|}$. The receiver records the received values and outputs $(\mathsf{receipt}, \sigma)$. To open a commitment σ, the committer sends $x, r_1, .., r_{|x|}, \rho_1, ..., \rho_{|x|}$. The verifier verifies the consistency of $r_1, .., r_{|x|}, \rho_1, ..., \rho_{|x|}$ with respect to x and the commitment values and if all verifications succeed it outputs $(\mathsf{open}, \sigma, x)$.

For the simulation, first note that the verifier has no secret information, thus with respect to passive corruption of the verifier the simulation is strong in a trivial way. Before describing the behavior of the simulator the case of passive corruption of the committer, recall that the simulator chooses a reference string f_0, f_1, e such that it knows the inverse functions f_0^{-1}, f_1^{-1}. Now, as long as the committer is not Byzantinely corrupted at the time of generating the commitment message, the simulator generates a simulated commitment message as follows: Having received $\sigma = (\mathsf{sid}, \mathsf{cid}, C, R), n = |x|$ from $\mathcal{F}_{\mathsf{MCOM}}$, the simulator chooses $y_1, ..., y_n, r_1^0, ..., r_n^0, r_1^1, ..., r_n^1$ such that $y_i = f_0(r_i^0) = f_1(r_i^1)$ for all

[9]We note that $\mathcal{F}_{\mathsf{MCOM}}$ allows the adversary to modify the value committed by a corrupted committer. However, this holds only in the case of Byzantine corruptions and so does not affect the behavior of $\mathcal{F}^{+\mathsf{lk}}_{\mathsf{MCOM}}$ upon leakage.

i, and sends the simulated commitment string that consists of $(\sigma, y_i, c_i^0, c_i^1)$ for each $i = 1..n$, where $c_i^0 = E(e, \sigma \circ i \circ r_i^0, \rho_i^0)$ and $c_i^1 = E(e, \sigma \circ i \circ r_i^1, \rho_i^1)$. To later open the simulated commitment to any given value x, the simulator reveals the appropriate partial randomness $r_1^{x_1}, \rho_1^{x_1}, ..., r_n^{x_n}, \rho_n^{x_n}$.

To deal with passive corruption of the committer in a strong way, the simulator needs to come up with a function T that translates the ideal local state of the committer C within $\mathcal{F}_{\mathsf{MCOM}}$ into its real local state in the protocol. Recall that this state consists of the committed value x. (In fact, it consists of all the committed values in all exchanges in which pid C is a committer.) The function T is constructed as follows. First, the simulator chooses a simulated commitment message as described above. Then, the function T includes in its description both openings of each bit location in the simulated commitment; then, $T(x)$ outputs the corresponding opening of the simulated commitment to the string x. It is stressed that the simulation corruption-oblivious. In particular, the main program of the simulator does not receive any information regarding the information learned during the leakage. $\square$

5.4 Zero Knowledge

We adapt the zero knowledge ideal functionality to tolerate leakage and demonstrate a protocol that realizes the adapted functionality in the presence of leakage. Recall that $\mathcal{F}_{\mathsf{ZK}:R}$, for a relation R, takes from the prover P an input (x, w), and outputs x to the verifier V only if $R(x, w)$ holds. This formulation guarantees that V learns nothing but the validity of x. It also guarantees that if the verifier accepts (gets an output x), then P actually "knows" a witness w such that $R(x, w) = 1$.

Adapting $\mathcal{F}_{\mathsf{ZK}}$ to leakage, we can ideally hope to realize a functionality with optimal tolerance, such as $\mathcal{F}_{\mathsf{ZK}}^{+\mathsf{lk}}$, which can "gracefully" tolerate arbitrary leakage from the prover, and in addition does not give up on soundness even in face of arbitrary leakage on the verifier. However, we could not manage to realize such a functionality. Instead, we consider an adaptation that can tolerate arbitrary leakage from the prover, but only a bounded amount of leakage from the verifier before soundness breaks. Before presenting our eventual adaptation and implementation, we briefly sketch the difficulties which prevent us from achieving optimal leakage tolerance.

How tolerant are the classic protocols? As shown in [9, 11], the parallel repetition of classic 3-round zero knowledge protocols, such as Blum's Hamiltonian cycle [5], and GMW's 3-coloring [16], UC-realize the basic (non-leaky) $\mathcal{F}_{\mathsf{ZK}}$, given access to (non-leaky) ideal commitment. Moreover, they do so even in the presence of adaptive corruptions. However, the proofs of security of these protocols do *not* yield corruption-oblivious simulation. Thus we cannot conclude that these protocols UC-realize $\mathcal{F}_{\mathsf{ZK}}^{+\mathsf{lk}}$ in the presence of leakage.

In fact, without any modifications, these protocols seem inherently impossible to simulate in the face of leakage. To demonstrate this, let us recall GMW's 3-coloring protocol. Here the prover, who possesses a 3-coloring c, chooses a random permutation σ of the three colors, and commits to the permuted coloring $\sigma(c)$. The verifier then requires that the prover opens the colors of a random edge, and checks that its endpoints are indeed colored differently. Now, consider a (Byzantinely) corrupted verifier V^*, which also obtains leakage on the prover's coloring during the protocol. This verifier can leak, for example, the secret permutation σ and then ask the (honest) prover to open the colors $\sigma(c(i)), \sigma(c(j))$ of some random edge (i, j). Finally, it can leak again the true colors $c(i), c(j)$. Simulating such a behavior seems impossible (assuming $\mathsf{3COL} \notin \mathsf{BPP}$). Indeed, once the simulator simulates σ for the first leakage, it essentially becomes committed to it for the rest of the protocol. Then, when it is required to simulate the opening of $\sigma(c(i)), \sigma(c(j))$, it essentially has no information on c, and hence, if it can consistently simulate the second leakage query, then essentially it must "know" a proper coloring of the entire graph[10]. We stress that this inherent difficulty also fails simulators that are not leakage-oblivious

[10]This intuition can be made formal; namely, given such a simulator we can construct an algorithm for 3-coloring arbitrary graphs.

(and are thus allowed to depend on both the leakage function and the leakage result).

Our solution and its price. To overcome the above problem we require that at the beginning of the protocol, the verifier commits to all its challenges. This already allows the simulation to go through; now the simulator can first extract the challenge edge (i, j), choose random colors for it $c'(i), c'(j)$, and than have the leakage return a permutation mapping the real $c(i), c(j)$ to $c'(i), c'(j)$. In fact, we show that this adjustment is enough for simulating any malicious verifier.

This adjustment comes, however, at a price: Unlike the original protocols, where the verifier was of the public coins type (and had no secret state), now the verifier commits to its challenges, and the secrecy of these challenges is crucial for the protocol's soundness. Hence, we can not hope that in such a protocol the verifier will be able to withstand arbitrary amounts of leakage; in particular once the prover leaks all of the verifier's challenge, soundness is doomed.

Consequently, we only realize a weaker adaptation, where the verifier can only tolerate a bounded amount of leakage. (The prover can still tolerate arbitrary leakage.) More specifically, we can tolerate arbitrary leakage on the verifier's randomness so long that a super-logarithmic amount of min-entropy is maintained.

We stress that we do not rule out the possibility of realizing the stronger leaky adaptation $\mathcal{F}_{\mathsf{ZK}}^{+\mathsf{lk}}$, which can tolerate arbitrary leakage from the verifier. Coming up with such a protocol is an interesting open question.

5.4.1 The details of our solution.

We consider a leaky adaptation, where the verifier can only tolerate a bounded amount of leakage and the prover can tolerate arbitrary leakage. The adapted functionality $\mathcal{F}_{\mathsf{ZK}:R}^{+B}$ is parameterized by a binary relation R, and expects a single input, $(\mathsf{prove}, P, V, \mathsf{sid}, x, w)$ from a prover P. If $R(x, w)$ holds and the adversary allows, then $\mathcal{F}_{\mathsf{ZK}:R}^{+B}$ outputs $(\mathsf{verified}, P, V, \mathsf{sid}, x)$ to V. If $R(x, w)$ does not hold, then no output is generated. If the adversary instructs to corrupt P then it learns w. Furthermore, if no output was yet written to V then the adversary is allowed to change the values of x and w. The leaky ideal state of P consists of (x, w), while V has no ideal state. Leakage on P does not affect the functionality's behavior. On the other, when leakage on V crosses the bound B, the functionality allows the adversary to send arbitrary (possibly false) statements to V. The details are described in Figure 11.

We now turn to describe a protocol which UC-realizes $\mathcal{F}_{\mathsf{ZK}:R}^{+B}$ in the $\mathcal{F}_{\mathsf{MCOM}}^{+\mathsf{lk}}$-hybrid model, where R is the Hamiltonian cycle relation. The protocol adjusts Blum's 3-round protocol by adding the verifier's commitment to its challenges at the beginning of the protocol. The protocol involves k parallel repetitions (where k is the security parameter) and can tolerate $k - \omega(\log k)$ bits of leakage from the verifier (which is optimal w.r.t bounded leakage).

Proposition 5.4. *Let R be the Hamiltonian cycle relation, and let $B = k - \omega(\log(k))$, where k is the security parameter. Protocol* HAMCYC UC*-realizes $\mathcal{F}_{\mathsf{ZK}:R}^{+B}$ in the $\mathcal{F}_{\mathsf{MCOM}}^{+\mathsf{lk}}$-hybrid model (assuming authenticated channels). Moreover, emulation is done with a strong leakage simulator.*

Corollary 5.4 (of the (leaky) UC theorem and Proposition 5.3)**.** *Let* COM *be the commitment protocol of [9, 11]. Then, Protocol* $\mathsf{HAMCYC}^{\mathsf{COM}}$ UC*-realizes $\mathcal{F}_{\mathsf{ZK}:R}^{+B}$ with leakage-oblivious simulation.*

We now sketch the proof Proposition 5.4.

Proof sketch. We construct a strong leakage simulator $\mathcal{S}$ for the dummy adversary. For this purpose we should describe how does the leakage sub-routine $\tilde{S}$ operates. We first address the main two interesting cases. Then we explain briefly how to deal with all other cases.

Functionality $\mathcal{F}^{+B}_{\mathsf{ZK}:R}$

$\mathcal{F}^{+B}_{\mathsf{ZK}:R}$ is parameterized by a binary relation R and a leakage bound B. It has a leakage counter $\mathsf{leakage}$ for V that is initially set to zero. It proceeds as follows.

1. Upon receiving an input $(\mathsf{prove}, P, V, \mathsf{sid}, x, w)$ from party P, check whether $R(x, w)$. If so send $(\mathsf{verified}, P, V, \mathsf{sid}, x)$ to the adversary, and once it allows send the same message to V.
2. Upon receiving a message $(\mathsf{corrupt}, P, \mathsf{sid})$ from the adversary, send w to the adversary. Furthermore, if the adversary now provides a value (x', w') such that $R(x', w')$ holds, and no output was yet given to V, then output $(\mathsf{verified}, P, V, \mathsf{sid}, x)$ to V.
3. **Prover leakage:** Given a message (leak, P), give the aggregator the ideal state (x, w). Continue behaving as above.
4. **Verifier leakage:** Given a message (leak, V), give the aggregator the (empty) ideal state $\perp$. Once the number of leakage bits ℓ is returned set $\mathsf{leakage} = \mathsf{leakage} + \ell$. If $\mathsf{leakage} > B$ allow cheating. That is, notify the simulator that the bound was crossed, and at the request of the simulator send $(\mathsf{verified}, P, V, \mathsf{sid}, x')$ to V (regardless of the statement's validity).
5. Reporting leakage: In both cases, the functionality gets from the aggregator the number of leaked bits and outputs it to the leaking party (which just forwards it to the environment).

Figure 11: The leaky zero knowledge functionality, $\mathcal{F}_{\mathsf{ZK}}$. Items 1 and 2 are the same as in the non-leaky formulation, $\mathcal{F}_{\mathsf{ZK}}$, in [6]. Items 3-5 are new.

Protocol HAMCYC

1. Given input $(\mathsf{verify}, P, V, \mathsf{sid}, G)$, where G is a graph over nodes $[n]$, V uses $\mathcal{F}^{+\mathsf{lk}}_{\mathsf{MCOM}}$ to commit to k challenge bits $b_1, \ldots, b_k$.
2. Given input $(\mathsf{prove}, P, V, \mathsf{sid}, G, C)$, where C is a cycle in G, the prover P proceeds as follows.
   (a) Wait to receive a notification from $\mathcal{F}^{+\mathsf{lk}}_{\mathsf{MCOM}}$ that V committed to its challenges.
   (b) Once notified, sample random permutations $\{\sigma_i : [n] \to [n]\}_{i\in[k]}$.
   (c) Use $\mathcal{F}^{+\mathsf{lk}}_{\mathsf{MCOM}}$ to commit to $\{\sigma_i\}$ and to the adjacency matrices of the permuted graphs $\{\sigma_i(G)\}$.
3. Given notification from $\mathcal{F}^{+\mathsf{lk}}_{\mathsf{MCOM}}$ that P committed, V asks $\mathcal{F}^{+\mathsf{lk}}_{\mathsf{MCOM}}$ to open all challenges $b_1, \ldots, b_k$.
4. Given V's challenges, P acts as follows. If $b_i = 0$, P asks $\mathcal{F}^{+\mathsf{lk}}_{\mathsf{MCOM}}$ to open all the commitments corresponding to the i'th challenge (including all edges and the permutation itself). If $b_i = 1$, P asks $\mathcal{F}^{+\mathsf{lk}}_{\mathsf{MCOM}}$ to open only the edges of the permuted cycle $\sigma_i(C)$
5. Upon receiving the appropriate open messages from $\mathcal{F}^{+\mathsf{lk}}_{\mathsf{MCOM}}$, V verifies that the opened values are valid. That is, if $b_i = 0$, it checks that the opened graph is indeed a permutation of G under the opened permutation, and if $b_i = 1$, it checks that the opened edges indeed form a cycle. If the validation went through, it outputs $(\mathsf{verified}, P, V, \mathsf{sid}, G)$.

Figure 12: The Protocol for proving Hamiltonian cycle in the $\mathcal{F}^{+\mathsf{lk}}_{\mathsf{MCOM}}$-hybrid model

1. V **is corrupted at the beginning of the execution and** P **is never corrupted**. Let V^* be the new (Byzantine) verifier controlled by the environment. $\mathcal{S}$ first emulates $\mathcal{F}_{\mathsf{MCOM}}$ for V^*, and records the challenges $b_1, \ldots, b_k$ given by $\mathcal{V}^*$ to $\mathcal{F}_{\mathsf{MCOM}}$. (These will be essential for strong simulation of future leakage on P). $\mathcal{S}$ now samples challenge responses as follows. For each i, if $b_i = 0$, $\mathcal{S}$ samples a random permutation $\sigma_i : [n] \to [n]$, and if $b_i = 1$, $\mathcal{S}$ samples a random cycle C_i and a random node $u_i \in [n]$. Then, $\mathcal{S}$ simulates the notification to P of the committed challenges. It also simulates the notification of $\mathcal{F}_{\mathsf{MCOM}}$ regarding P's commitments. Then, once the verifier opens its challenges, $\mathcal{S}$ simulates the opening of the corresponding commitments using the sampled answers.

   Simulation of leakage on P**.** Leakage on P can occur at any time; in particular, it is possible that P leaks before answering the verifier's challenge, after or both. We describe how $\tilde{S}$ performs state translation. Once leakage occurs, $\tilde{S}$ produces a circuit T, which given the ideal state (G, C) produces a real state. In case the verifier still did not commit to its challenges, the real state still does not contain the prover's permutations and commitments, it is identical to the ideal state (G, C). Hence, in this case state translation is straightforward.

   The more interesting case is where the verifier already committed to its challenges. Then, the real state includes the permutations and possibly also the committed values. Since the committed values are determined by the permutations and the pair (G, C), we ignore them from here on and assume the real state is of the form $(G, C, \{\tilde{\sigma}_i\}_{i\in[k]})$. The translation T will have hardwired into it the challenges $\{b_i\}$, the simulated $\{\sigma_i\}_{i:b_i=0}$ and the simulated $\{C_i, u_i\}_{i:b_i=1}$. Given the input (G, C), it outputs $(G, C, \{\tilde{\sigma}_i\}_{i\in[k]})$, which is consistent with its simulated answers and the witness C, as follows. For each $i \in [k]$, if $b_i = 0$, it sets $\tilde{\sigma}_i = \sigma_i$, and if $b_i = 1$, it sets $\tilde{\sigma}_i$ so that $\tilde{\sigma}_i(C) = C_i$ and $\tilde{\sigma}(1) = u_i$. Note that since C_i, u_i were chosen at random, the permutation $\tilde{\sigma}_i$ is just a random permutation of $[n]$.

   Once the translation is produced, $\tilde{S}$ sends the aggregator the leakage function composed with T and forwards the leakage results to the environment. $\mathcal{S}$ then returns to its state prior to the leakage.

   Validity. In this scenario the view of the environment is distributed identically to its view in the real execution. Indeed, the simulated real state of P at all stages of the protocol is consistent with the answers to all the challenges. In addition the simulated $\tilde{\sigma}_i$ are all distributed correctly (as random permutations). We stress that for this to hold it is crucial that the challenges are extracted from V^* prior to the leakage, which allows sampling the answers ahead of time, and performing state translation consistently.

2. P **is corrupted at the beginning of the execution and** V **is never corrupted**. Here, S first tries to extract a Hamiltonian cycle from the corrupted P^*, controlled by the environment. For this purpose, it simply emulates $\mathcal{F}_{\mathsf{MCOM}}$ for P^*, recording the permutations σ_i that P^* produces. It then takes the verifier's part and simulates commitments for random challenges $b_1, \ldots, b_k$. Then, it requires that P^* opens these values. In case, there is a $b_i = 1$ such that σ_i is a valid permutation and the opened cycle C' is valid, then $\mathcal{S}$ finds a Hamiltonian cycle $\sigma_i^{-1}(C')$. It then gives the functionality $\mathcal{F}_{\mathsf{ZK}}$ the cycle it found as a new witness and completes the simulation. Otherwise, it aborts.

   When leakage on V **occurs.** The leaky state of V in the real execution, only includes its state w.r.t $\mathcal{F}_{\mathsf{MCOM}}$, which is the challenge bits $b_1, \ldots, b_k$. Leakage oblivious simulation is straight forward, the translation T simply outputs $\{b_i\}$.

If at any point the simulator is notified by $\mathcal{F}_{\mathsf{ZK}}$ that the leakage bound was crossed, it continues its interaction with P^* to check if it managed to answer the simulated challenges, and if so it instructs $\mathcal{F}_{\mathsf{ZK}}$ to send $(\mathsf{verified}, P, V, \mathsf{sid}, G)$ to V.

Validity. The simulation is invalid only in cases where all three following hold: (a) P^* manages to make V accept; (b) a Hamiltonian cycle could not be produced; (c) the total leakage on V did not exceed the bound $B = k - \omega(\log k)$. Denote this "bad" event by E. We show that:

$$\Pr_{b_1 \ldots b_k}[E] \leq 2^{-\omega(\log k)}$$

Claim 5.1. *If V accepts and a Hamiltonian cycle can not be produced, then the commitments of P^* uniquely determine all values $b_1, \ldots, b_k$.*

Proof. Let σ be the i'th committed permutation for some $i \in [k]$. If σ is a not a valid permutation of G, then b_i is surely 1, or the verifier would not accept. If it is valid then b_i is surely 0 or else, an Hamiltonian cycle can be produced. Indeed, if $b_i = 1$, since the verifier accepts, it must be that a valid cycle is opened and it can be mapped to a Hamiltonian cycle by the valid σ. □

It follows that the probability that E occurs is not larger than the probability of guessing random $b_1, \ldots, b_k$ given at most B bits of leakage, which in turn occurs w.p. at most $2^{-k+B} \leq 2^{-\omega(\log k)}$.

We briefly explain how the simulator treats the rest of the cases. This treatment is essentially based on the two main cases above.

3. **No party is corrupted.** Here, the challenges of V, $b_1, \ldots, b_k$ are sampled by the simulator itself, simulation is done as in the first case, only with the simulated challenges, instead of the extracted challenges. This also includes leakage on P. Leakage on V is treated as in the second case.

4. **Adaptive corruptions** When P is corrupted its state is simulated consistently with the sampled randomness and the witness. This is done similarly to the case of leakage (described in the first item). Indeed, this can be viewed as a special form of leakage, where the entire state leaks. When V is corrupted, the simulator simply gives the challenges $b_1 \ldots b_k$ it sampled on its own.

This completes the proof. □

5.5 Authenticated Channels

In this section we construct a protocol for realizing *leaky authenticated channels* with bounded leakage resilience. More specifically, the protocol UC-realizes an ideal functionality, $\mathcal{F}^{+B}_{\mathsf{Auth}}$, that guarantees authenticated communication *as long as the overall leakage between any two transmissions of some messages does not exceed a pre-specified bound B.*

The protocol we present uses two main building blocks: (a) non-committing encryption (NCE) (b) information theoretic c-time message authentication codes (MACs), which are resilient to a constant leakage rate from the secret key. The idea behind the protocol is simple. The parties initially share a (leaky) secret key K_1. Then the protocol proceeds inductively; at each round, a current authentication key K_i is used to authenticate the i-th message, m_i. In addition, a fresh key K_{i+1} is generated and transmitted using non-committing encryption. These transmitted ciphers are also authenticated using K_i. To allow the authentication to go through, we need our underlying leaky MAC scheme to allow authentication of messages which are polynomially longer than the secret key. This is achieved using *universal hashing*. Concretely, the protocol we present tolerates, between each two transmissions, roughly $k/10$

bits of leakage on the k-long secret key. Similar techniques are used for a related goal in [4]. However, the security analysis there is different than the one here.

The protocol we construct admits leakage-oblivious simulation and is thus composable. We can, therefore, use it as a basic building block supporting any protocol that requires authenticated channels, when ideally authenticated channels are unavailable. We stress, however, that when doing so the leakage tolerance of the higher-level protocol, naturally degrades to that of the authentication protocol. We exemplify by implementing (secret and authenticated) *secure channels* in Section 5.5.2.

5.5.1 The details of our solution.

We now present the details of functionality $\mathcal{F}^{+B}_{\mathsf{Auth}}$. The functionality deals with the authentication of multiple messages. An alternative, simpler formulation of $\mathcal{F}^{B}_{\mathsf{Auth}}$ would handle the authentication of only a single message. The multi-message case could then be handled via composition of multiple protocols that realize the single-message $\mathcal{F}^{+B}_{\mathsf{Auth}}$. However, in our case the realizing protocols require the parties to initially share a secret key. Therefore running many independent single-message authentication protocols would result in a very large initial shared key. Instead, we show that one short initial key suffices for realizing the multi-message functionality.

Functionality $\mathcal{F}^{+B}_{\mathsf{Auth}}$

Running with parties S, R and an adversary $\mathcal{S}$, $\mathcal{F}^{+B}_{\mathsf{Auth}}$ is parameterized by a leakage bound B. It has a leakage counter $\mathsf{leakage}$, initially set to zero. The functionality operates as follows:

1. **Transmission:** Given the i'th input $(\mathsf{send}, S, R, \mathsf{sid}, m_i)$ forward it to the adversary $\mathcal{S}$. Once $\mathcal{S}$ allows forwarding the message, send $(\mathsf{sent}, S, \mathsf{sid}, i, m_i)$ to R. Store (i, m_i). Reset the leakage counter $\mathsf{leakage} \leftarrow 0$.
2. **Corruption:** Given a message $(\mathsf{corrupt}, S)$, if the current message m_i was not yet delivered to R, allow the adversary to change m_i to a new m'_i. In the future, deliver to R any message of the adversary's choice.
3. **Leaky Virtual state:** Given a message (leak, P), where $P \in \{S, R\}$, give the aggregator $\perp$ (empty ideal state).
4. **Reaction to leakage:** Given a notification that ℓ bits leaked from any of the parties), update $\mathsf{leakage} \leftarrow \mathsf{leakage} + \ell$. In case the accumulated leakage crossed the bound, i.e $\mathsf{leakage} > B$, act as after corruption; namely, allow the adversary to change the current m_i to m'_i. Moreover, from this point on, allow the adversary to send messages of its choice.

Figure 13: The multi-message leaky authenticated channels functionality, $\mathcal{F}^{+B}_{\mathsf{Auth}}$

As mentioned above, the protocol we present relies on non-committing encryption (NCE) and (information theoretic) c-time message authentication codes (MACs). Before presneting the actual protocol we briefly review the notion of leakage-resilient MACs.

Leaky c-time authentication. A B-leakage-resilient c-time MAC scheme assures that given authentications of c distinct messages, it is impossible to forge an authentication of another message, so long that the leakage on the secret MAC key does not exceed the bound B. We shall also require that the schemes are able to handle messages of length which is polynomial in the length of the secret key. Such (information theoretic) schemes can be constructed based on universal hashing [24, 3]. For the sake of completeness in Appendix A we present a simple MAC scheme which is resilient to a constant leakage rate. Our scheme is $(c-1)$-time, with keys of length k, messages of length s (typically $s = k^{O(1)}$), and tags of length $t = k/(c+1)$. Given leakage ℓ and at most $c-1$ previous authentications, the forging probability is at most $2^{-t+\ell+O(\log s)}$.

We now present the protocol.

Protocol LAC

Setup: We assume both parties initially share a symmetric MAC key $K_1 \overset{U}{\leftarrow} \{0,1\}^k$ for a leakage resilient Auth scheme.

Invariant: Prior to the transmission of the i'th message m_i, the parties share a symmetric key K_i, which was generated at round $i-1$.

Authenticated transmission i**:** When S is given input $(\mathsf{send}, S, R, \mathsf{sid}, m_i)$, it exchanges messages with R as follows. All outgoing messages are authenticated and all incoming messages are checked for authenticity using K_i. Messages which do not pass authentication are ignored.

1. **Transmission.** S sends R the message m_i and informs of its intention to transmit a new MAC key.
2. **NCE key generation.** R samples secret and public keys $(d_j, e_j) \leftarrow \mathsf{Gen}(1^k, r_G)$, for all $j \in [k]$, and sends $\{e_j\}$ to S.
3. **Encryption and key regeneration.** Upon receiving $\{e_j\}$ from R, S samples a new authentication key $K_{i+1} \overset{U}{\leftarrow} \{0,1\}^k$. It then encrypts $c_j \leftarrow \mathsf{Enc}_{e_j}(K_{i+1,j}; r_E)$ and sends $\{c_j\}$ to R.
4. **Decryption and output.** When R receives $\{c_j\}$ it decrypts the new authentication key K_{i+1} and outputs the message (i, m_i).

Figure 14: Leaky Authenticated Channels Protocol

Proposition 5.5. *Assume* Auth *is a 3-time MAC scheme that is resilient to* $2B$ *bits leakage (where* $B = B(k)$ *is a function of the secret key length). Then protocol* LAC UC*-realizes* $\mathcal{F}^{+B}_{\mathsf{Auth}}$ *in the presence of leakage. Moreover, emulation is leakage-oblivious.*

We note that instantiating Auth with Construction A.1 in Appendix A, allows realizing $\mathcal{F}^{+B}_{\mathsf{Auth}}$ with $B = k/10 - \omega(\log k)$.

Proof sketch. We construct a leakage-oblivious simulator $\mathcal{S}$, with leakage subroutine $\tilde{S}$, for the dummy adversary. The simulator $\mathcal{S}$ has two operation modes: (a) before the functionality $\mathcal{F}^{+B}_{\mathsf{Auth}}$ gives up (due to leakage which exceeds the bound B or due to corruption). (b) after $\mathcal{F}^{+B}_{\mathsf{Auth}}$ gives up. Simulation in the second mode is straightforward, as the simulator can freely ask $\mathcal{F}^{+B}_{\mathsf{Auth}}$ to send messages of its choice and, thus, perfectly emulate a real world execution. From here on we focus on the first operation mode.

Sampling of MAC keys. Prior to the simulation of transmission $i-1$, the simulator already samples a random MAC key $K_i \overset{u}{\leftarrow} \{0,1\}^k$ for the i'th transmission. As detailed later, this key is potentially used by $\mathcal{S}$ during simulation of leakage in transmission $i-1$.

Simulating communication for the transmission i**.** When the simulator is informed by $\mathcal{F}_{\mathsf{Auth}}$ that a message m was received for transmission, it samples k equivocal tuples

$$\mathsf{equiv_i} = (e_i, c_i, r^0_{G,i}, r^0_{E,i}, r^1_{G,i}, r^1_{E,i}) \leftarrow \mathsf{Sim}(1^k)$$

Using the equivocal keys $\{e_i\}$ and ciphers $\{c_i\}$, $\mathcal{S}$ then simulates the messages between S, R corresponding to the transmission of the message and encryption of the new MAC key K_{i+1}. The messages are all appended by authentication tags, which the simulator computes using the MAC key K_i. Fake messages, sent by the environment to be forwarded to the parties, are discarded (as long as $\mathcal{F}_{\mathsf{Auth}}$ did not give up).

Simulating leakage during transmission i. Leakage events are treated by the subroutine $\tilde{S}$ according to their timing. We describe how $\tilde{S}$ constructs the ideal-to-real state translation T, and how it is augmented according to the timing of leakage.

1. **Between the last (simulated) message in transmission $i-1$ and the first (simulated) sender message in transmission i:** Here the (secret) state of both parties in the real world protocol, includes messages keys $K_1, \ldots, K_{i-1}, K_i$, and the NCE randomness from all previous transmissions. The transformation T will have hardwired into it the MAC keys and the simulated NCE randomness, corresponding to the leaking party (the sender will require $\vec{r}_E$ and the receiver $\vec{r}_G$). Given the ideal (empty) state $\perp$, T simply outputs the MAC keys and the NCE randomness, which is taken from the equivocal randomness consistently with the keys.

2. **Between the first sender message and the receiver's message:** Here the real state of the receiver also includes the NCE randomness for the transmission of the new MAC key K_{i+1}. We will hard-wire into T the simulated randomness corresponding to the correct bits of K_{i+1}, which was sampled by the simulator. The corresponding T simply adds to its output the NCE randomness for K_{i+1}.

3. **Between the receiver's message and the last sender message:** Now the real state of the sender also includes the NCE randomness for the transmission of K_{i+1}, as well as the K_{i+1} itself. Therefore, we shall also hard-wire into T the correct NCE randomness for K_{i+1}, and K_{i+1} itself. These will be added to the output of T.

Validity. Informally, we show that the above simulation strategy is valid inductively. That is, assuming validity of transmissions $1, \ldots, i-1$ we show that transmission i is also valid. This is done in two stages: (a) we claim that during transmission i the environment fails to forge authentications based on the leakage resilient MAC; (b) we show that simulating the transmission of key K_{i+1} (for the next transmission) is valid based on the NCE scheme (and similarly to Proposition 5.1 which assumes authenticated channels).

More formally, consider hybrid executions where the first $i-1$ transmissions are dealt with as in the ideal execution with $\mathcal{S}$, and all following $i, i+1, \ldots$ transmissions are dealt with as in the real execution. The state of each party at the outset of the i'th transmission is inherited from the previous ideal executions, this includes all MAC keys $K_1, \ldots, K_i$, and encryption randomness, which is taken from the simulated equivocal tuples consistently with the keys. We then show that no environment $\mathcal{Z}$ can distinguish two adjacent hybrids

$$\begin{aligned} \mathsf{EXEC}_i &= \mathsf{Ideal}_1, \ldots, \mathsf{Ideal}_{i-1}, \mathsf{Real}_i, \mathsf{Real}_{i+1}, \ldots \\ \mathsf{EXEC}_{i+1} &= \mathsf{Ideal}_1, \ldots, \mathsf{Ideal}_{i-1}, \mathsf{Ideal}_i, \mathsf{Real}_{i+1}, \ldots \end{aligned}$$

As the first step we show that during the i'th transmission in EXEC_i, the environment fails to forge authentications (w.r.t the current key K_i); hence, like in EXEC_{i+1}, non-authentic messages are discarded. Indeed, leaving leakage aside, all the simulated ciphers of K_i seen by the environment are generated by the NCE simulator from scratch, and are hence independent of K_i (information theoretically). Moreover, since K_i is part of the leaky state only during transmissions $i-1$ and i and we assume that the leakage bound B was not crossed in either, then the total leakage on K_i is at most $2B$. Hence, by the security of the $2B$-resilient Auth, the environment fails to forge authentications. The second step is showing that the environment can not tell whether the i'th transmission consists of simulated ciphers and leakage or real ones. This now follows from the properties of the NCE scheme exactly as in Proposition 5.1 when used over authenticated channels. $\square$

5.5.2 Secure channels over unauthenticated links.

As previously explained, we can now use $\mathcal{F}^{+B}_{\mathsf{Auth}}$ as a building block for other protocols; namely, construct leaky protocols in the $\mathcal{F}^{+B}_{\mathsf{Auth}}$-hybrid model. When doing so the leakage tolerance of the composed protocol is naturally affected by that of $\mathcal{F}^{+B}_{\mathsf{Auth}}$. We exemplify this by implementing (secret and authentic) *secure channels*. Recall that implementing secure message transmission (SMT) over ideally authenticated channels, we managed to construct a protocol which can tolerate arbitrary leakage. However, once we implement this protocol over leaky authenticated channels we are no longer guaranteed to maintain this tolerance. Instead, we can implement a less tolerant functionality $\mathcal{F}^{+B}_{\mathsf{LSC}}$ which maintains a sound behavior so long that the leakage between messages does not exceed the bound B. Once this bound is crossed, $\mathcal{F}^{+B}_{\mathsf{LSC}}$ completely gives-up, losing both secrecy and authenticity.

Functionality $\mathcal{F}^{+B}_{\mathsf{LSC}}$

Running with parties S, R and an adversary $\mathcal{S}$, $\mathcal{F}^{B}_{\mathsf{LSC}}$ is parameterized by a leakage bound B. It has a leakage counter $\mathsf{leakage}$, initially set to zero. The functionality operates as follows:

1. **Transmission:** given the i'th input $(\mathsf{send}, S, R, \mathsf{sid}, m_i)$. Send a message $(\mathsf{send}, S, R, |m_i|, \mathsf{sid})$ to the adversary $\mathcal{S}$. Once $\mathcal{S}$ allows forwarding the message, send $(\mathsf{sent}, S, \mathsf{sid}, i, m_i)$ to R. Store (i, m_i). Then reset the leakage counter $\mathsf{leakage} \leftarrow 0$.
2. **Corruption:** given a message $(\mathsf{corrupt}, P)$, where $P \in \{S, R\}$, disclose all messages stored so far and all future messages to the adversary. In case S is the corrupted party and the current message m_i was not yet delivered to R, allow the adversary to change m_i to a new m_i'. In addition, allow it in the future to send R messages of its choice.
3. **Leaky Virtual state:** given a message (leak, P), where $P \in \{S, R\}$, give all stored messages to the aggregator.
4. **Reaction to leakage:** given a notification that ℓ bits leaked from any of the parties), update $\mathsf{leakage} \leftarrow \mathsf{leakage} + \ell$. In case the accumulated leakage crossed the bound, i.e $\mathsf{leakage} > B$, allow the adversary to change the current m to m'. Moreover, from this point on give-up. That is, disclose any message m to the adversary and allow it to change the message to a new message of its choice.

Figure 15: The multi-message leaky secure channels functionality, $\mathcal{F}^{+B}_{\mathsf{LSC}}$. Unlike in $\mathcal{F}^{+B}_{\mathsf{Auth}}$, here secrecy is also guaranteed. That is, the adversary only gets the message length $|m_i|$ rather than m_i itself.

The protocol implementing $\mathcal{F}^{+B}_{\mathsf{LSC}}$ is implemented as the SMT protocol over authenticated channels, with the exception that all messages are transmitted using $\mathcal{F}^{+B}_{\mathsf{Auth}}$.

Proposition 5.6. *Protocol* LSC UC*-realizes* $\mathcal{F}^{+B}_{\mathsf{LSC}}$ *in the* $\mathcal{F}^{+B}_{\mathsf{Auth}}$*-hybrid model with leakage-oblivious simulation.*

Corollary 5.5 (of the UC theorem)**.** *Protocol* $\mathsf{LSC}^{\mathsf{LAC}}$ UC*-realizes* $\mathcal{F}^{+B}_{\mathsf{LSC}}$ *with leakage-oblivious simulation.*

Protocol LSC

Parties S and R exchange the following messages using $\mathcal{F}^{+B}_{\mathsf{Auth}}$.

1. **Notification.** S notifies R of its intention to send an $|m_i|$-bit message.
2. **NCE key generation.** When R is informed of the transmission intention, it samples secret and public keys
$(d_j, e_j) \leftarrow \mathsf{Gen}(1^k, r_G)$, for all $1 \leq j \leq |m_i|$, and sends $\{e_j\}$ to S.
3. **Encryption.** Upon receiving $\{e_j\}$ from R, S encrypts $c_j \leftarrow \mathsf{Enc}_{e_j}(m_j; r_E)$ and sends $\{c_j\}$ to R.
4. **Decryption.** When R receives $\{c_j\}$ it decrypts the message m_i, and outputs m_i.

Figure 16: Leaky Secure Channels Protocol in the $\mathcal{F}^{+B}_{\mathsf{Auth}}$-hybrid model

Proof sketch. As in The proof of Proposition 5.5, the simulator leakage-oblivious simulator has to modes of operation: (a) before the leakage bound is crossed (or corruption occurs); (b) after. Again, simulation in the second case is straight forward. In the first case, we are essentially guaranteed to have authenticated channels and hence the proof follows as the one of Proposition 5.1 for the SMT protocol. □

Acknowledgments. We thank Amit Sahai for telling us about the problems with proving leakage tolerance of the standard three round Zero knowledge protocols and about the way this problem is solved in [14].

References

[1] A. Akavia, S. Goldwasser, and V. Vaikuntanathan. Simultaneous hardcore bits and cryptography against memory attacks. In O. Reingold, editor, *Theory of Cryptography - TCC 2009*, volume 5444 of *Lecture Notes in Computer Science*, pages 474–495. Springer, 2009.

[2] J. Alwen, Y. Dodis, and D. Wichs. Survey: Leakage Resilience and the Bounded Retrieval Model. In K. Kurosawa, editor, *Information Theoretic Security - ICITS 2009*, volume 5973 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2009.

[3] M. Atici and D. R. Stinson. Universal hashing and multiple authentication. In *"Advances in Cryptology - CRYPTO 1996"*, volume 1109, pages 16–30, 1996.

[4] N. Bitansky, R. Canetti, S. Goldwasser, S. Halevi, Y. Kalai, and G. Rothblum. Program obfuscation with leaky hardware. Manuscript, 2011.

[5] M. Blum. How to prove a theorem so no one else can claim it, 1986. International Congress of Mathematicians.

[6] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145, 2001.

[7] R. Canetti. Universally composable signature, certification, and authentication. In *"17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004)"*, pages 219–235. IEEE Computer Society, 2004.

[8] R. Canetti, U. Feige, O. Goldreich, and M. Naor. Adaptively secure multi-party computation. In *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing*, pages 639–648, Philadelphia, PA, May 1996. ACM. Longer version available as MIT-LCS-TR 682, 1996.

[9] R. Canetti and M. Fischlin. Universally composable commitments. In *"Advances in Cryptology - CRYPTO 2001"*, volume 2139 of *Lecture Notes in Computer Science*, pages 19–40. Springer, 2001.

[10] R. Canetti and J. Herzog. Universally composable symbolic security analysis. *J. Cryptology*, 24(1):83–147, 2011.

[11] R. Canetti, Y. Lindell, R. Ostrovsky, and A. Sahai. Universally composable two-party and multi-party secure computation. In *STOC*, pages 494–503, 2002.

[12] S. Dziembowski and K. Pietrzak. Leakage-resilient cryptography. In *49th FOCS - 2008*, pages 293–302. IEEE Computer Society, 2008.

[13] S. Even, O. Goldreich, and A. Lempel. A Randomized Protocol for Signing Contracts. *Communications of the ACM*, 28(6):637–647, June 1985.

[14] S. Garg, A. Jain, and A. Sahai. Leakage-resilient zero knowledge and its applications. Personal communications, 2011.

[15] O. Goldreich. *Foundations of Cryptography, Basic Tools*, volume 1. Cambridge University Press, 2001.

[16] O. Goldreich, S. Micali, and A. Wigderson. Proofs that yield nothing but their validity for all languages in np have zero-knowledge proof systems. *J. ACM*, 38(3):691–729, 1991.

[17] S. Goldwasser, S. Micali, and C. Rackoff. The Knowledge Complexity of Interactive Proof Systems. *SIAM J. Comput.*, 18(1):186–208, 1989.

[18] S. Goldwasser and G. N. Rothblum. Securing computation against continuous leakage. In T. Rabin, editor, *Advances in Cryptology - CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 59–79. Springer, 2010.

[19] S. Halevi and H. Lin. After-the-fact leakage in public-key encryption. In Y. Ishai, editor, *Theory of Cryptography - TCC 2011*, volume 6597 of *Lecture Notes in Computer Science*, pages 107–124. Springer, 2011.

[20] A. Juma and Y. Vahlis. Protecting Cryptographic Keys against Continual Leakage. In T. Rabin, editor, *Advances in Cryptology - CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 41–58. Springer, 2010.

[21] S. Micali and L. Reyzin. Physically observable cryptography. In *TCC'04*, volume 2951 of *Lecture Notes in Computer Science*, pages 278–296. Springer, 2004.

[22] M. O. Rabin. "how to exchange secrets by oblivious transfer". Technical Report Technical Memo TR-81, Aiken Computation Laboratory, Harvard University, 1981.

[23] F.-X. Standaert. Introduction to side-channel attacks. In I. M. Verbauwhede, editor, *Secure Integrated Circuits and Systems*, pages 27–44. Springer, 2009.

[24] M. Wegman and L. Carter. New hash functions and their use in authentication and set equality. In *J. of Computer and System Sciences*, volume 22, pages 265–279, 1981.

A A Leaky MAC Scheme

In this section we present a simple MAC scheme which is resilient to a constant leakage rate. Our scheme is $(c-1)$-time, with keys of length k, messages of length s (typically $s = k^{O(1)}$), and tags of length $t = k/(c+1)$. Given leakage ℓ and at most $c-1$ previous authentications, the forging probability is at most $2^{-t+\ell+O(\log s)}$.

Construction A.1. *Let $\mathcal{H} : \{0,1\}^t \to \{0,1\}^t$ be a c-wise independent hashing family, with keys of length ct (e.g. evaluation of degree $c-1$ polynomials over $\mathbb{F}_{2^t}$). We define a new hashing family $\tilde{H} : \{0,1\}^s \to \{0,1\}^t$. A random hash $\tilde{h} \in \tilde{\mathcal{H}}$ consists of a random $h \in \mathcal{H}$ and a random prime $p < 2^t$; in particular, each key is of size $(c+1)t = k$. The authentication algorithm is defined as follows:*

$$\mathsf{Auth}_{\tilde{h}}(m) = \tilde{h}(m) = h(m \bmod p)$$

Where m is interpreted as a number in $[2^s]$.

We next show that any (unbounded) adversary, given $c-1$ authentications on distinct $c-1$ (adaptively chosen) messages $m_1, \ldots, m_{c-1}$, can not produce an authentication for a new message $m_c \notin \{m_i\}_{i<c}$ with probability greater than $2^{-t+O(\log s)}$. Intuitively, this follows from the fact that taking the number m modulo p is an (information theoretic) collision resistant function, so long that p is random and hidden from the attacker (indeed this ingredient can be replaced with any collision resistant function). Moreover, assuming that there are no collisions, the c-wise independent hash evaluated on the short $\bmod\ p$ numbers, maps them to a truly uniform tuple. The precise analysis requires a bit more attention to details, since messages are adaptively chosen. We note that given the above, it follows immediately that the forging probability given ℓ bits of leakage on the key is at most $2^{-t+O(\log s)+\ell}$.

Claim A.1. *Let $(f_1, \ldots, f_c)$ be message choice functions, where for all $1 < i$: $f_{i+1} : \{0,1\}^{t\times i} \to \{0,1\}^t$ and f_1 is a constant in $\{0,1\}^t$. Let $F : \{0,1\}^{t\times(c-1)} \to \{0,1\}^t$ be any (forger) function. Then:*

$$\Pr_{\tilde{h}}\left[F(\tilde{h}(M_1), \ldots, \tilde{h}(M_{c-1})) = \tilde{h}(M_c) \wedge M_c \notin \{M_i\}_{i<c} : M_i = f_i(\tilde{h}(M_1) \ldots \tilde{h}(M_{i-1}))\right] \le 2^{-t+O(\log s)}$$

Proof. We show that the claim holds for any fixed realization $m_1, \ldots, m_c$ such that $m_c \notin \{m_i\}_{i<c}$. For such fixed $\vec{m}$, denote by $S(\vec{m})$ the event given by the claim. Also let (h, p) denote the underlying c-wise independent hash $h \in \mathcal{H}$ and prime $p < 2^t$. Then:

$$\Pr_{h,p}[S(\vec{m})] \le \Pr_p[\exists i < c : m_c = m_i \bmod p] + \Pr_h[S(\vec{m}) : \forall i < c : m_c \ne m_i \bmod p]$$

By the c-wise independence of $\mathcal{H}$ the right term is bounded by 2^{-t}. It is left to bound the collision probability expressed by the left term. We note that a collision occurs only if p divides $|m_c - m_i|$ for some $i < c$. However, this difference is of size at most 2^s, implying there are no more than s distinct primes dividing it. On the other hand, by the prime numbers theorem, the number of primes $p < 2^t$ is $\Theta(2^t/t)$, implying that such an event occurs with probability at most $2^{-t+O\log(s)}$. The result follows. $\square$