Multiparty Computation Secure Against Continual Memory Leakage

Elette Boyle
MIT
eboyle@mit.edu

Shafi Goldwasser*†
MIT and Weizmann
shafi@mit.edu

Abhishek Jain
UCLA
abhishek@cs.ucla.edu

Yael Tauman Kalai
Microsoft Research
yael@microsoft.com

ABSTRACT

We construct a multiparty computation (MPC) protocol that is secure even if a malicious adversary, in addition to corrupting 1-ϵ fraction of all parties for an arbitrarily small constant $\epsilon > 0$, can *leak* information about the secret state of each honest party. This leakage can be *continuous* for an unbounded number of executions of the MPC protocol, computing different functions on the same or different set of inputs. We assume a (necessary) "leak-free" preprocessing stage.

We emphasize that we achieve leakage resilience *without weakening the security guarantee* of classical MPC. Namely, an adversary who is given leakage on honest parties' states, is guaranteed to learn nothing beyond the input and output values of corrupted parties. This is in contrast with previous works on leakage in the multi-party protocol setting, which weaken the security notion, and only guarantee that a protocol which leaks ℓ bits about the parties' secret states, yields at most ℓ bits of leakage on the parties' private *inputs*. For some functions, such as voting, such leakage can be detrimental.

Our result relies on standard cryptographic assumptions, and our security parameter is polynomially related to the number of parties.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General; F.0 [**Theory**]: General

*Supported under NSF Contract CCF-1018064.

†This material is based on research sponsored by the Air Force Research Laboratory under agreement number FA8750-11-2-0225. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory or the U.S. Government.

STOC'12, May 19–22, 2012, New York, New York, USA.

Keywords

Cryptography, secure multiparty computation, leakage resilience.

1. INTRODUCTION

The notion of *secure multiparty computation* (MPC), introduced in the works of Yao [Yao82] and Goldreich, Micali and Wigderson [GMW87], is one of the cornerstones in cryptography. Very briefly, an MPC protocol for computing a function f allows a group of parties to jointly evaluate f over their private inputs, with the property that an adversary who corrupts a subset of the parties does not learn anything beyond the inputs of the corrupted parties and the output of the function f. Over the years, MPC protocols have found numerous applications, such as in protocols for auctions, electronic voting, private information retrieval, and threshold and proactive cryptography.

The definition of security for MPC assumes that an adversary sees the messages sent and received by honest parties, but their internal state is perfectly secret. However, over the last two decades, it has become increasingly evident that in the real world, attackers can gain various additional information about the secret states of the honest parties via various *side-channel attacks* (see [Koc96, AK96, QS01, GMO01, OST06, HSH+08] and references therein).

In this work, we study MPC in the setting where an adversary, who corrupts an arbitrary subset of parties in the protocol, can also *leak* information about the *entire* secret state of each honest party throughout the protocol execution (except during a designated leak-free preprocessing stage). Leakage is modeled by allowing the adversary to query leakage functions, as follows. Each leakage function is computed by an arbitrary poly-size circuit, with bounded output-length, which is applied to the secret state of an honest processor. The adversary may choose the leakage functions adaptively, based on the entire history of communication, previous leakage, and internal state of corrupted processors.

The security guarantee we aim for and will achieve, is that any adversary in the above leakage model, *does not learn anything beyond the inputs of the corrupted parties and output values of the functions* computed by the MPC protocol. This is formalized via the standard real/ideal world paradigm. In the ideal world, parties do not interact directly, but rather send their inputs to an "ideal functionality", who computes the function for them, and sends them the output. There is no leakage in the ideal world. An MPC protocol is said to be secure, if for every "real world" leakage adver-

sary $\mathcal{A}$ (as above) there exists an "ideal world" simulator S, such that the output of all the parties (including the adversary) in the real world, is computationally indistinguishable from the output of all the parties (including the simulator) in the ideal world.

Weakly *Leakage-Resilient MPC.*

We note that recently there have been several results that consider the problem of constructing leakage-resilient protocols [GJS11, BCH11, DHP11, BGG$^+$11]. However, in contrast to the security guarantee we consider here, all these results give a *weak* security guarantee (though, they do not rely on a leak-free preprocessing stage). They guarantee that an adversary that runs the protocol and leaks ℓ bits about the honest parties' secret state, does not learn more than the output of the function being computed, *and an additional* ℓ *bits* about the private inputs of the honest parties. We note that leakage of ℓ bits on the private inputs of the honest parties could be detrimental to the security of the entire MPC protocol. For example, say the function to be computed by the MPC protocol is to tally up the binary votes of the parties. Then, the ℓ bits can be exactly the complete ℓ votes of any ℓ honest parties, rendering the protocol useless.

Moreover, this weak security notion allows the adversary to learn ℓ bits about the *joint* view of all the honest parties. Thus, another instructive example is to think of the function being computed as a threshold decryption function, where each party has a secret-share of the decryption key. In this case, the weak security guarantee allows the leakage of ℓ bits from the decryption key, which for some decryption algorithms could entirely compromise security.

Interestingly, we use the result in [BGG$^+$11], which constructs an MPC protocol with the weak security guarantee, as a building block to construct a leakage-resilient MPC protocol with the classical (strong) security guarantee.

Security Against Continual Leakage.

We further remark that the weaker security notion previously achieved cannot be extended meaningfully to continual leakage in the MPC setting. That is, it cannot address the setting where the n users do not just perform a one-shot MPC protocol, but rather engage in an unbounded number of MPC protocols for many functions, and during each MPC invocation the adversary leaks ℓ bits from each of the honest party's internal state. This is obvious, as allowing the repeated leakage of new ℓ bits of information on the honest parties' inputs would eventually leak the honest parties' inputs in their entirety. For example, in the setting where a set of parties jointly compute a threshold decryption function (as described above), they may want to carry out many decryption computations, where leakage happens repeatedly. Since each ℓ bits of leakage corresponds to ℓ bits of leakage on the decryption key, the decryption key may eventually be completely leaked! Nonetheless, we use the result of [BGG$^+$11] as a building block to achieve our stronger *continual leakage* security guarantee.

1.1 Our Result: Continual Leakage-Resilient MPC

In this work, we construct a leakage-resilient MPC protocol for any function f, *without weakening the security guarantee*. We consider a continual setting, where parties over time compute many functions on their inputs. Our security guarantee is that an adversary does not learn *anything* beyond the inputs of the corrupted parties and the output of the functions computed, even if he continually leaks information about the honest parties' secret states throughout the protocol executions. Parties' secret states are periodically updated via an update procedure, during which the adversary can continue to leak information. We allow each of the adversary's leakage functions to be an *arbitrary* (shrinking) polynomial time computable function of the *entire* secret state of each honest party (separately), and these leakage functions can be chosen adaptively on all information the adversary has seen thus far.

Theorem (Informal).

Under (standard) intractability assumptions, for every constant $\epsilon > 0$ there exists an MPC protocol for computing an unbounded number of functions among n parties of which at least ϵ fraction are honest. The protocol is secure against *continual* leakage, assuming a *one-time* leak-free preprocessing stage in which the inputs are shared, and where the security parameter is polynomially related to the number of parties n.

A few remarks about our result statement are in order.

"Leak-free" Preprocessing.

We assume the existence of a leak-free preprocessing stage. We stress that this is a *necessary* assumption to obtain our strong security guarantee, since otherwise an adversary can simply leak ℓ bits about an honest party's secret input, before the MPC even commences. More generally, we note that such a leak-free preprocessing stage is a necessary step in the construction of any leakage resilient cryptographic primitive which receives a secret input, and where the security guarantee is that the secret input does not leak. This is the case, for example, in the compilers of [ISW03, FRR$^+$10, JV10, GR10, GR12], which transform algorithms with a secret state into a functionally equivalent leakage-resilient variant of the same algorithm.

We remark that our preprocessing stage in fact has the nice property that it can be decomposed into two parts, namely, (a) an interactive preprocessing phase that is *independent* of the parties' inputs and the functions to be computed, and (b) a *non-interactive* input dispersal phase. We stress that the first phase is run only *once* in the beginning of time, before the parties know what their inputs are or what functions they wish to compute. The second (non-interactive) phase is run whenever the parties choose a set of inputs.

While both of these parts are assumed to be "leak-free", we do allow leakage between them. We refer the reader to Section 3 for a formal description of our model.

Multi-function MPC and Continual Leakage.

We note that in the standard (leak-free) MPC literature, one typically considers a one-shot MPC protocol, as opposed to considering the setting where the parties compute an unbounded (polynomial) number of functions. The reason we focus on the latter setting, is to emphasize that we need to run the leak-free preprocessing stage only *once*, and then the parties can compute any unbounded number of functions $f_1, \ldots, f_\ell$ in a leaky environment.

We further emphasize that we allow the adversary to leak

continuously on the secret states of the parties during the unbounded computations; the only (necessary) requirement is that the secret states of the parties are periodically updated (since otherwise they will eventually be completely leaked). However, the adversary is allowed to leak even *during* each update procedure. We do not bound the total number of bits that the adversary leaks, but rather only bound the leakage rate: i.e., the number of bits leaked *between updates*.

Extending to Multi-Input MPC.

We stated our theorem for the case of computing many functions on a single set of inputs. However, our construction is easily extended to the many-input case. Whenever a party chooses a new input, the (leak-free) non-interactive input phase described above can be repeated. Namely, party P_i on new input x_i performs a local computation on x_i, sends a message to the other parties, and erases x_i. One may think of this model as a "hot potato" model, where the parties never store their inputs for very long (since they are concerned with leakage), but rather immediately share their input (as if it were a "hot potato").

Number of Parties vs. Security Parameter.

Notice that in our theorem, the security parameter is polynomially related to the number of parties. Namely, the security increases with the number of parties. Therefore, this theorem is meaningful only when the number of parties in the MPC protocol is large. One may ask whether this restriction on the number of parties being large, or the restriction that an ϵ-fraction is honest, is inherent, or whether it is simply an artifact of our techniques. Unfortunately, it turns out that this restriction cannot be removed altogether. In particular, one can prove that there does not exist a secure leakage-resilient *two-party* computation protocol in our model.[1] Similarly, one can show that there does not exist a secure leakage-resilient MPC protocol if all the parties except one are malicious. Moreover, jumping ahead, in Section 1.4 we show that proving this theorem for constant number of parties, implies an "only computation leaks (OCL) compiler" (without leak-free hardware) that has only a constant number of sub-computations (or "modules"), which is an interesting open problem on its own. We refer the reader to Section 1.4 for details.

Assumptions.

In our construction, we rely on several underlying cryptographic primitives, including a fully homomorphic encryption (FHE) scheme [Gen09, BGV11], a non-interactive zero-knowledge (NIZK) proof-of-knowledge system [FLS90], a standard MPC protocol [GMW87], an equivocal commitment scheme [FS89], a weakly leakage-resilient MPC protocol [BGG$^+$11], and an LDS compiler [BCG$^+$11] (which can be thought of as a stronger version of an OCL compiler as in [JV10, GR10, GR12]). These primitives have been shown to exist under various standard computational intractability assumptions, and we refer the reader to Section 2 for details on these primitives, and the corresponding assumptions. We note however that all these primitives, excluding FHE, can be based on the DDH assumption.

The use of FHE in our construction is in order to ensure the number of parties required will be independent of the complexity of the functions computed by the MPC protocol.[2]

Applications.

We demonstrate the application of our result to the problem of delegating multi-party computation to outside servers. Generally, the setting is of a large set of parties who need to perform a joint computation, and they would like a service (such as Amazon) to do the computation for them. However, they do not trust any one server, and further believe that any server can be leaked upon.

Usually, MPC provides a solution around the trust problem by using several servers, as follows: Each party secret shares her input, and gives one share to each server; then the servers carry out the desired computation by running an MPC protocol; finally, one argues that if there are sufficiently many honest parties, then security is guaranteed. However, if an adversary can obtain leakage information from the honest servers, then this is no longer true. To argue security in the leaky setting, the servers will need to run a *leakage-resilient* MPC protocol. Moreover, if the servers compute many functions on the secret inputs, then they will need to run an MPC protocol that is secure against *continual leakage*. Let us demonstrate three examples of this setting.

- *Electronic election*: Say an electronic election among many voters is to be held. Clearly running an MPC protocol among all voters is prohibitive, since it requires interaction between every two voters. Instead, the MPC protocol is run by a proxy of n servers. Since these servers compute on very sensitive information, attackers may try to employ various side-channel attacks to learn this information. Thus, to ensure the secrecy of the individual votes, the servers should run a *leakage-resilient* MPC protocol.
- *Medical Data*: One may envision a huge database which contains the medical data of every patient in the US. To compute any global statistic on this data, one would not want to put complete trust in any single database.

[1] The reason is the following: Assume the adversary controls party P_1. In this case, he knows the entire secret state s_1 of P_1, and can choose his leakage function L to depend on s_1: i.e., $L{=}L_{s_1}$. Note that L takes as input the secret state s_2 of P_2, and thus the adversary can leak any (shrinking) function $g(s_1, s_2)$ by setting $L_{s_1}(s_2) \triangleq g(s_1, s_2)$. But, recall that from the secret states (s_1, s_2) the parties can compute any function of the original inputs (x_1, x_2). Therefore, the function leaked can be an arbitrary function of the original inputs. Clearly, such leakage cannot be simulated in the ideal world.

[2] We emphasize that, while FHE immediately solves the related problem of computing on encrypted data, FHE does *not* suffice for our purposes. To illustrate, suppose the parties collectively generate a public key pk for the FHE scheme, so that they each hold a secret share of the corresponding secret key, and then each publish an encryption of their input x_i. Then for any efficiently computable function f, they can easily produce *an encryption* of the desired output, $\mathsf{Enc}_{\mathsf{pk}}(f(\vec{x}))$. However, the challenge is (even for a one-shot function computation) how to enable the parties to collectively *decrypt* this ciphertext and reveal $f(\vec{x})$ itself, while simultaneously ensuring that the adversary (who can corrupt nearly all of the parties, and leak on all the rest) is not able to learn any information on the x_i's.

Instead, it is distributed to n different databases. Each time they need to compute statistics on this data, they engage in an MPC protocol. As in the voting example, since these databases contain very sensitive information, an adversary may try to obtain this information via a leakage attack. Thus, to ensure security, the databases must run an MPC protocol that is secure against *continual leakage*.

- *Differential Privacy*: In the area of differential privacy, great care is taken to ensure that the data of individuals is protected. However, usually it is assumed that there is an *honest* curator, and that the people in the database hand their secret data to this curator. However, it seems likely that people may not trust any single curator with highly sensitive information (such as whether they do or do not have a disease which may scare off life insurance providers). Thus, as in the previous examples, this trusted curator can be replaced by a multitude of parties of which only a small fraction is assumed to be honest. Moreover, if these parties compute on the database using a leakage-resilient MPC protocol, then security is guaranteed even if all the honest parties are leaked upon (as long as some ϵ-fraction of the honest parties are not *fully* leaked upon).

1.2 Related Work

Leakage-Resilient Non-Interactive Primitives.

There has been an extensive amount of research on leakage-resilient cryptography in the past few years. Most prior works construct specific leakage-resilient *non-interactive* primitives, such as leakage-resilient encryption schemes and leakage-resilient signature schemes [DP08, AGV09, Pie09, DKL09, ADW09, NS09, KV09, DGK$^+$10, FKPR10, ADN$^+$10, KP10, GR10, JV10, BG10, BKKV10, DP10, DHLW10a, DHLW10b, LRW11, MTVY11, BSW11, LLW11, DLWW11, BCG$^+$11].

Weakly *Leakage-Resilient Interactive Protocols.*

There has also been prior work on the problem of constructing leakage-resilient interactive protocols [GJS11, BCH11, BGK11, DHP11, BGG$^+$11]. Garg *et. al.* [GJS11] present a leakage-resilient zero-knowledge proof system. Bitansky *et. al.* [BCH11] present leakage-resilient protocols for various functionalities (such as secure message transmission, oblivious transfer, and commitments) which are secure against semi-honest adversaries, and also zero knowledge, in the UC framework. Boyle *et. al.* [BGK11] present a leakage-resilient multi-party coin tossing protocol. Dåmgard, Hazay, and Patra [DHP11] present a general leakage-resilient two-party secure function evaluation protocol for NC^1 functions in the *semi-honest* setting. In their model, they further place a restriction that the adversary must leak on the input and randomness of an honest party's secret state *independently*. Finally, very recently Boyle *et. al.* [BGG$^+$11] constructed a general leakage-resilient MPC protocol that is secure in the UC setting.

However, all the results in the interactive setting mentioned above offer a *weak* security guarantee, that an adversary that leaks ℓ bits in the real world, gains at most ℓ bits of secret information about the secret inputs of the parties. (An exception is the work of [BGK11] that considered the specific coin-tossing functionality, where the parties do not have any secret inputs.) Moreover, the ℓ bits of secret information gained is an arbitrary (poly-size) function of the *joint* inputs $x_1, \ldots, x_n$.

Only Computation Leaks Model.

Finally, we mention that various leakage models have been considered in the literature that restrict the leakage functions in different ways. Most notable is the *only computation leaks* (OCL) model of Micali and Reyzin [MR04]. The axiom of this model is that secret information that is merely stored in memory does not leak, but any information that is used during a computation may leak.

Several results prove security for specific cryptographic primitives in the OCL leakage model [DP08, Pie09, FKPR10]. More generally, it is known how to convert *any* circuit into one that is secure in the OCL model [GR10, JV10, GR12]. In particular, a recent work of Goldwasser and Rothblum [GR12] shows how to do this unconditionally, making no intractability assumptions, and without resorting to secure leak-free hardware, unlike the previous works. Specifically, Goldwasser and Rothblum construct an efficient compiler that takes any circuit (with some secret values hard-wired) and converts it into a leakage-resilient one, consisting of several *modules*, each of which performs a specific sub-computation. The security guarantee is that an adversary, who at any point of time throughout the computation obtains bounded leakage from the "currently active" module, does not learn any more information than having black-box access to the circuit. We will use a variant of this result (namely, an LDS compiler; see Section 2.5) to construct our leakage-resilient MPC scheme. In particular, we use [GR12] as a building block in our construction. See Section 1.3 for details.

We stress that our result does *not* use the OCL assumption, and we allow the adversary to compute leakage functions on *everything* held in the memory of each party (except during the preprocessing phase and during the input phase).

1.3 Overview of Our Construction

Starting point – OCL Compiler.

As discussed earlier, it is known how to convert *any* circuit into one that is secure in the *only computation leaks* (OCL) model (without assuming secure hardware) [GR12]. In light of this result, a natural first idea toward realizing our goal of constructing leakage-resilient MPC protocols, is the following. Let $P_1, \ldots, P_n$ denote the set of all parties, and let $U_{\vec{x}}$ be a universal circuit that has the secret input vector $\vec{x}$ of all the parties hard-wired into it and on input a circuit f outputs $U_{\vec{x}}(f) = f(x)$. Then, very roughly, the candidate MPC protocol works as follows. First, in the "leak free" preprocessing phase, apply the OCL compiler of [GR12] on circuit $U_{\vec{x}}$ to obtain a set of modules $\mathsf{Sub}_1, \ldots, \mathsf{Sub}_n$ such that on any input f, the "compiled" circuit (consisting of $\mathsf{Sub}_1, \ldots, \mathsf{Sub}_n$) outputs $U_{\vec{x}}(f) = f(\vec{x})$. Next, in the computation phase, in order to securely compute a function f, each party P_i *emulates* the module Sub_i (such that the computation of Sub_i is performed by party P_i), where the input of Sub_1 is f, and the output of Sub_n is the protocol output $f(\vec{x})$. Finally, in the update phase, the parties update their respective modules by running the update algorithm of the OCL compiler.

Now, assuming that we can reduce (independent) leakage

on each party to (independent) leakage on its corresponding module, one may hope that the above MPC protocol achieves the desired security properties: in particular, privacy of the inputs that were "encoded" in the preprocessing phase. Unfortunately, as we explain below, this is not the case. Nevertheless, as will be evident from the forthcoming discussion, the above approach serves as a good starting point toward realizing our goal.

OCL Compiler vs. LR-MPC.

There are two main differences between the setting of leakage-resilient MPC (LR-MPC) and an OCL compiler.

1. The first difference is perhaps best illustrated by the fact that an OCL compiler only guarantees security against an *external* adversary who can obtain leakage from the modules. In contrast, in the setting of LR-MPC, we wish to guarantee security against an *internal* adversary, who may also corrupt a subset of the parties.

   More concretely, recall that the security of the OCL compiler crucially relies on the assumption that an external adversary can only obtain *bounded*, *independent* leakage on each module. Further, in order for the correctness of the compiled circuit to hold, each module must perform its computation as specified. As a result, the above approach, at best, yields an MPC protocol that is secure when *all* the parties are honest (not even semi-honest) but can be leaked upon by an external adversary. Specifically, note that if an internal adversary can corrupt some of the parties, then we can no longer guarantee correctness of computation, and even worse, an adversary may be able to obtain *joint* leakage on multiple modules, and learn the *entire* secret state of modules corresponding to corrupted parties, thus violating both of the above stated requirements.

2. The second difference between the OCL compiler and the leakage-resilient MPC setting is that in the OCL setting, the communication between the modules is assumed to be private (but may be leaked), and leakage is assumed to happen "in order"; i.e., only a module which is currently computing can be leaked upon. On the other hand, in the leakage-resilient MPC setting, the entire communication is to be known to the adversary, and moreover, leakage on any party can happen at any time.

Emulating Modules via **Weakly** *LR-MPC.*

Our key idea to circumvent the first problem stated above is to emulate each Sub_i by a designated *set* of parties $S_i = \{P_{i_1}, \ldots, P_{i_\ell}\}$, instead of a single party P_i. More concretely, we secret share Sub_i between $P_{i_1}, \ldots, P_{i_\ell}$, who then run a specific MPC protocol Π to jointly emulate the (functionality of) module Sub_i. Now, note that as long as at least one of the parties in the designated set S_i is honest, the emulation of Sub_i will be "correct", and if leakage on each honest party is bounded, then we can expect the leakage on the module Sub_i to be bounded as well. Furthermore, if all of the designated sets S_i for the modules Sub_i are disjoint (i.e., no party is contained within two different sets), then the leakage on each module will be independent, as required. However, note that since we are in the setting of leakage, in order for the above idea to work, we need the MPC protocol Π to satisfy some form of leakage-resilience. Thus, a priori, it seems that we haven't made any progress at all.

Our next crucial observation is that protocol Π in fact only needs to a satisfy a *weaker* form of leakage-resilience. Specifically, we only require that leakage on the secret state of each party P_{i_ℓ} executing protocol Π (to emulate Sub_i) can be *"reduced"* to leakage on the module Sub_i. (This suffices since the OCL compiler allows bounded leakage on each module.) More generally, this translates to constructing an MPC protocol such that the leakage on the secret states of the honest parties in the real world can be reduced to leakage on the inputs of the honest parties in the ideal world. Fortunately, an MPC protocol (for any poly-size function f) satisfying the above (weak) form of leakage-resilience was recently constructed by Boyle *et al.* [BGG$^+$11]. Thus, we are able to employ their construction here.[3]

However, the result of Boyle *et al.* is only for *deterministic* functions, whereas the modules in the OCL construction compute *randomized* functions. Thus, we need to extend the weakly leakage-resilient MPC to hold for randomized computations. See Section 2.6 (and Section 2.6.1 in particular) for further details.

Using an LDS Compiler Instead of an OCL Compiler.

Our key idea to circumvent the second problem stated above is to use an LDS compiler instead of an OCL compiler. The LDS (leaky distributed system) model was introduced in [BCG$^+$11], and it strengthens the OCL model in two ways (which are exactly the strengthenings we need). First, in the OCL model, leakage occurs in a certain ordering (based on the order of computation). The LDS model strengthens the power of the adversary, by allowing him to leak from the sub-computations in any order he wishes. Moreover, he can leak a bit from Sub_i, then leak a bit from Sub_j, and based on the leakage values, leak again on Sub_i. So, the adversary controls which Sub_i he wishes to leak from. In addition, in the LDS model, the adversary can view and control the entire communication between the modules. We refer the reader to Section 2.5 for details on the LDS compiler.

By using an LDS compiler, as opposed to an OCL compiler, we get around the second problem mentioned above.

Reducing Number of Parties via FHE.

An important issue that was overlooked in the previous discussion is the following. The only known OCL compiler that does not rely on leak-free hardware [GR12], and thus the only known LDS compiler without leak-free hardware, suffers from the drawback that the number of modules in the "compiled" circuit is linear in the size of the original circuit. As a result, when we apply the LDS compiler on $U_{\vec{x}}$, whose size grows with $|\vec{x}|$, the number of resultant modules is more than the number of parties! Thus, a priori, it is not even clear how to realize the above approach.

[3] At this point, an advanced reader may question whether the result of Boyle et al. [BGG$^+$11], in conjunction with a leakage-resilient secret sharing scheme, *directly* yields a leakage-resilient MPC protocol in our model. Unfortunately, this is not the case since the simulator of Boyle et al. requires *joint* leakage on the honest party inputs, even when the real world adversary makes *disjoint* leakage queries on the secret states of honest parties. We refer the reader to Section 1.4 for more details.

In order to resolve this above problem, we make crucial use of fully homomorphic encryption (FHE) in the following manner. Instead of simply applying the LDS compiler to $U_{\vec{x}}$, we now first compute a key pair $(\mathsf{pk}, \mathsf{sk})$ for an FHE scheme, and then apply the LDS compiler to the decryption circuit $\mathsf{Dec}_{\mathsf{sk}}(\cdot)$ with the secret key sk hardwired. Note that the number of resultant modules is now *independent* of the number of parties. Now, in a non-interactive input phase (that is also "leak-free"), the parties P_i each encrypt their respective inputs x_i under the public key pk, and publish the resulting ciphertexts $\hat{x}_i$. Then, whenever the parties wish to compute a functionality f over their inputs, they homomorphically evaluate $\hat{y}_f \triangleq \mathsf{Eval}_{\mathsf{pk}}((\hat{x}_1, ..., \hat{x}_n), f)$, and collectively evaluate the compiled decryption circuit on the value $\hat{y}_f$ in the manner described above.

We note that the use of FHE allows us to obtain the desired property that the preprocessing phase is *independent* of the inputs and functions to be computed, since in this phase a key pair $(\mathsf{pk}, \mathsf{sk})$ is generated and the LDS compiler is applied to the corresponding decryption circuit $\mathsf{Dec}_{\mathsf{sk}}(\cdot)$. In addition, the input phase is non-interactive, since in this phase the parties simply compute and send an encryption of their inputs.

Missing Pieces.

A few technical issues still remain undiscussed. For example, it is not immediately clear how to choose the designated sets of parties S_i such that at least one of the parties in each set S_i is honest, and each set S_i is independent. Very roughly, to address this problem, we employ (an adapted version of) the committee election protocol of Feige [Fei99] to divide the parties into several committees, one for each module. Then, by a careful choice of parameters, we are able to obtain the desired guarantees. We refer the reader to the technical sections for more details.

1.4 Future Directions

LR-MPC for Constant Number of Parties.

Perhaps the most interesting open question left from this work is to construct a leakage-resilient MPC protocol for *constant* number of parties. We note that such a result (even if we only consider adversaries that leak, but do not corrupt any party) will imply the following interesting corollary: The existence of an efficient compiler that converts any circuit into a leakage-resilient circuit that is secure in the "only computation leaks" (OCL) model with *constant* number of modules (and without assuming leak-free hardware). We refer the reader to Section 2.5 for details.

To see this implication, consider such a leakage-resilient MPC protocol. Let (an arbitrary) party P_1 take as his secret input the secret circuit C to be compiled, and the other parties take no inputs. After the leak-free preprocessing stage (and the leak-free input stage), each party P_i holds a secret state s_i. We think of each party P_i as being a module Sub_i in the compiled circuit. To evaluate the circuit C on (public) input x, the modules carry out a leakage-resilient MPC computation of the universal function U_x, that on inputs $\{s_i\}$, which form some sort of secret-sharing of C, outputs $C(x)$. Since the OCL model allows leakage on each module separately, this corresponds to allowing leakage on each party separately, which according to our definition of security gives no information about the secret C beyond the output value $C(x)$.

Weakly *Leakage-Resilient MPC with Disjoint Leakage.*

Another interesting open question is to construct a leakage-resilient MPC protocol without assuming any leak-free stages, and requiring the following weakened security definition: For each "real world" adversary that makes ℓ leakage queries, where each leakage query is applied to the secret state of a *single* honest party, there exists a simulator in the "ideal world" that makes at most ℓ leakage queries, where each leakage query is applied to the input of a *single* honest party.

We note that the recent result of [BGG+11] allowed the adversary in the "real world" to make leakage queries on the *joint* secret state of all the parties, and allowed the simulator in the "ideal world" to make leakage queries that are a function of all the inputs of the honest parties. Unfortunately, their simulator requires joint leakage on the honest party's inputs even in the case where the adversary only makes disjoint leakage queries.

We next show that such a leakage-resilient MPC protocol, where the leakage in the real world and in the ideal world is made on each party separately, would imply a result similar to ours, which allows a leak-free preprocessing stage, but considers a strong security guarantee. Intuitively, in the leak-free preprocessing stage, the parties will secret share their inputs via a secret sharing scheme that is resilient to continual leakage. Such a scheme was recently presented by Dodis *et. al.* [DLWW11]. Then, any time the parties wish to compute a function f of their secret inputs, they will run the weak leakage-resilient MPC protocol. Security follows from the fact that the adversary only gains leakage from the secret share of each party *separately*, and from the fact that the secret-sharing scheme is resilient to continual leakage on each of its shares.

LR-MPC with Non-Interactive Preprocessing.

Finally, an interesting open question that is left by this work, is to construct a leakage-resilient MPC protocol without the initial leak-free preprocessing stage, but only with the leak-free *non-interactive* input stages.

2. PRELIMINARIES

2.1 Non-Interactive Zero Knowledge

DEFINITION 2.1. *[FLS90, BFM88, BSMP91]:* $\Pi = (\mathsf{Gen}, \mathsf{P}, \mathsf{V}, \mathcal{S} = (\mathcal{S}^{\mathsf{crs}}, \mathcal{S}^{\mathsf{proof}}))$ *is an* efficient adaptive NIZK proof system *for a language* $L \in \mathsf{NP}$ *with witness relation* $\mathcal{R}$ *if* $\mathsf{Gen}, \mathsf{P}, \mathsf{V}, \mathcal{S}^{\mathsf{crs}}, \mathcal{S}^{\mathsf{proof}}$ *are all* ppt *algorithms, and there exists a negligible function* μ *such that for all* k *the following three requirements hold.*

- **Completeness***: For all* x, w *such that* $\mathcal{R}(x, w) = 1$*, and for all strings* $\mathsf{crs} \leftarrow \mathsf{Gen}(1^k)$*,*

$$\mathsf{V}(\mathsf{crs}, x, \mathsf{P}(x, w, \mathsf{crs})) = 1.$$

- **Adaptive Soundness***: For all adversaries* $\mathcal{A}$*, if* $\mathsf{crs} \leftarrow \mathsf{Gen}(1^k)$ *is sampled uniformly at random, then the probability that* $\mathcal{A}(\mathsf{crs})$ *will output a pair* (x, π) *such that* $x \notin L$ *and yet* $\mathsf{V}(\mathsf{crs}, x, \pi) = 1$*, is at most* $\mu(k)$*.*

- **Adaptive Zero-Knowledge**: *For all* ppt *adversaries* $\mathcal{A}$,

$$\left| \Pr[\mathsf{Exp}_{\mathcal{A}}(k) = 1] - \Pr[\mathsf{Exp}^{S}_{\mathcal{A}}(k) = 1] \right| \leq \mu(k),$$

where the experiment $\mathsf{Exp}_{\mathcal{A}}(k)$ *is defined by:*

$$\mathsf{crs} \leftarrow \mathsf{Gen}(1^k)$$
$$\textit{Return } \mathcal{A}^{\mathsf{P}(\mathsf{crs},\cdot,\cdot)}(\mathsf{crs})$$

and the experiment $\mathsf{Exp}^{S}_{\mathcal{A}}(k)$ *is defined by:*

$$(\mathsf{crs}, \mathsf{trap}) \leftarrow \mathcal{S}^{\mathsf{crs}}(1^k)$$
$$\textit{Return } \mathcal{A}^{S'(\mathsf{crs},\mathsf{trap},\cdot,\cdot)}(\mathsf{crs}),$$

where $S'(\mathsf{crs}, \mathsf{trap}, x, w) = \mathcal{S}^{\mathsf{proof}}(\mathsf{crs}, \mathsf{trap}, x)$.

We next define the notion of a NIZK proof of knowledge.

DEFINITION 2.2. *Let* $\Pi = (\mathsf{Gen}, \mathsf{P}, \mathsf{V}, \mathcal{S} = (\mathcal{S}^{\mathsf{crs}}, \mathcal{S}^{\mathsf{proof}}))$ *be an efficient adaptive NIZK proof system for an* NP *language* $L \in \mathsf{NP}$ *with a corresponding NP relation* $\mathcal{R}$. *We say that* Π *is a* proof-of-knowledge *if there exists a* ppt *algorithm* E *such that for every* ppt *adversary* $\mathcal{A}$,

$$\Pr[\mathcal{A}(\mathsf{crs}) = (x, \pi) \textit{ and } E(\mathsf{crs}, \mathsf{trap}, x, \pi) = w^* \\ \textit{s.t. } \mathsf{V}(\mathsf{crs}, x, \pi) = 1 \textit{ and } (x, w^*) \notin \mathcal{R}] = \mathrm{negl}(k),$$

where the probabilities are over $(\mathsf{crs}, \mathsf{trap}) \leftarrow \mathcal{S}^{\mathsf{crs}}(1^k)$, *and over the random coin tosses of the extractor algorithm* E.

LEMMA 2.3 ([FLS90]). *Assuming the existence of enhanced trapdoor permutations, there exists an efficient adaptive NIZK proof of knowledge for all languages in NP.*

2.2 Equivocal Commitments

Informally speaking, a bit-commitment scheme is *equivocal* if it satisfies the following additional requirement. There exists an efficient simulator that outputs a fake commitment such that: (a) the commitment can be decommitted to both 0 and 1, and (b) the simulated commitment and decommitment pair is indistinguishable from a real pair. We now formally define the equivocability property for bit-commitment schemes in the CRS model.

The following definition is adapted from [FS89, CIO98].

DEFINITION 2.4. *A non-interactive bit-commitment scheme* $(\mathsf{Gen}, \mathsf{Com}, \mathsf{Rec})$ *in the CRS model is said to be an* equivocal bit-commitment scheme in the CRS model *if there exists a* PPT *simulator algorithm* $\mathcal{S} = (\mathcal{S}^{\mathsf{crs}}, \mathcal{S}^{\mathsf{com}})$ *such that* $\mathcal{S}^{\mathsf{crs}}$ *takes as input the security parameter* 1^k *and outputs a CRS and trapdoor pair,* $(\mathsf{crs}, \mathsf{trap})$*; and* $\mathcal{S}^{\mathsf{com}}$ *takes as input such a pair* $(\mathsf{crs}, \mathsf{trap})$ *and generates a tuple* (c, d^0, d^1) *of a commitment string* c *and two decommitments* d^0 *and* d^1 *(for* 0 *and* 1*), such that the following holds.*

1. *For every* $b \in \{0, 1\}$ *and every* $(c, d^0, d^1) \leftarrow \mathcal{S}^{\mathsf{com}}(\mathsf{crs}, \mathsf{trap})$, *it holds that*

$$\mathsf{Rec}(\mathsf{crs}, c, d^b) = b.$$

2. *For every* $b \in \{0, 1\}$, *the random variables*

$$\{(\mathsf{crs}, c, d) : \mathsf{crs} \leftarrow \mathsf{Gen}(1^k), (c, d) \leftarrow \mathsf{Com}(\mathsf{crs}, b)\}$$

and

$$\{(\mathsf{crs}, c, d^b) : (\mathsf{crs}, \mathsf{trap}) \leftarrow \mathcal{S}^{\mathsf{crs}}(1^k), \\ (c, d^0, d^1) \leftarrow \mathcal{S}^{\mathsf{com}}(\mathsf{crs}, \mathsf{trap})\}$$

are computationally indistinguishable.

Reusable CRS.

Note that the simulator algorithms $\mathcal{S}^{\mathsf{crs}}$ and $\mathcal{S}^{\mathsf{com}}$ are described as separate algorithms in the Definition 2.4 to highlight that it is not necessary to create a separate CRS for every equivocal commitment, i.e., the CRS is *reusable*. In this case, Definition 2.4 can be extended in a straightforward manner to consider indistinguishability of an honestly generated tuple consisting of a crs and polynomially many commitment-decommitment pairs, from a simulated tuple.

LEMMA 2.5 ([CLOS02]). *Assuming the existence of one-way functions, there exists an equivocal bit commitment in the (reusable) CRS model.*

String Equivocal Commitments.

For our purposes, we actually use *string* equivocal commitment schemes. Note that such a scheme can be easily constructed by simply repeating the above bit commitment scheme in *parallel*. More specifically, a commitment to a string of length n is a vector $(c_1, ..., c_n)$, with corresponding decommitment vector $(d_1, ..., d_n)$. The simulator algorithm $\mathcal{S}^{\mathsf{com}}$ produces a commitment vector and a *pair* of decommitment vectors $d^0 = (d^0_1, ..., d^0_n)$, $d^1 = (d^1_1, ..., d^1_n)$. A decommitment to any particular bit string $a = (a, ..., a_n)$ can be formed by selecting the appropriate decommitment values $(d^{a_1}_1, \ldots, d^{a_n}_n)$. We denote this vector as d^a.

2.3 The Elect Protocol

As part of our protocol, we elect disjoint committees, and need the guarantee that (with overwhelming probability in k) the number of parties in each committee is of the correct approximate size, and that a constant fraction of each committee is honest. Such a protocol can be obtained using the technique of Feige's lightest bin committee election protocol [Fei99].

Feige's protocol selects a single committee of approximate size k out of n parties by having each party choose and broadcast a random bin in $\left[\frac{n}{k}\right]$.[4] The elected committee $\mathcal{E}$ consists of the parties in the lightest bin. Feige demonstrated that no set of malicious parties $M \subset [n]$ of size $(1 - \epsilon)n$ can force a committee $\mathcal{E}$ to be elected for which $|\mathcal{E} \cap M|$ is significantly greater than $(1-\epsilon)k$, by using a Chernoff bound to argue that each bin contains nearly ϵk *honest* parties.

Suppose we wish to elect m disjoint committees, each of size approximately k, where k is the security parameter, and where the number of parties n is at least $n \geq mk^2$. We consider the following protocol, Elect. Each party samples a random value $x_i \leftarrow \left[\frac{n}{k}\right]$. The resulting committees are precisely the m lightest bins. Namely, suppose the lightest bin is ℓ_1, the second lightest bin is ℓ_2, etc. Then $\mathcal{E}_j = \{P_i : x_i = \ell_j\}$, for $j = 1, ..., m$.

[4] In Feige's original work [Fei99], he considered the specific case of $k = \log n$. For our purpose, we need to elect committees whose size depends on the security parameter (to achieve negligible error), and thus we consider general k.

LEMMA 2.6. *Let $n \geq mk^2$, and let $M \subset [n]$ be any subset of corrupted parties of size $(1-\epsilon)n$. Then the protocol* Elect *yields a collection of m disjoint committees $\{\mathcal{E}_j\}_{j=1}^m$ such that the following properties simultaneously hold with probability $\geq 1 - e^{-\Theta(k)}$:*

1. $\forall j,\ \frac{\epsilon}{2}k \leq |\mathcal{E}_i| \leq (1+o(1))k$,
2. $\forall j,\ \frac{|\mathcal{E}_j \cap M|}{|\mathcal{E}_j|} < 1 - \frac{\epsilon}{3}$.

The proof of Lemma 2.6 is very similar to that of the disjoint committee election protocol of [BGK11]. We refer the reader to the full version for a complete analysis.

We remark that a constant fraction of honest parties in the elected committees will be needed for the weakly leakage-resilient MPC for *randomized functionalities* (see discussion in Section 2.6.1).

2.4 Fully Homomorphic Encryption

A fully homomorphic public-key encryption scheme (FHE) consists of algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$. The first three are the standard key generation, encryption and decryption algorithms of a public key scheme. The additional algorithm Eval is a deterministic polynomial-time algorithm that takes as input a public key pk, a ciphertext $\hat{x} \leftarrow \mathsf{Enc}_{\mathsf{pk}}(x)$ and a circuit C, and outputs a new ciphertext $c = \mathsf{Eval}_{\mathsf{pk}}(\hat{x}, C)$ such that $\mathsf{Dec}_{\mathsf{sk}}(c) = C(x)$, where sk is the secret key corresponding to the public key pk. It is required that the size of c depends polynomially on the security parameter and the length of the output $C(x)$, but is otherwise independent of the size of the circuit C.

Several such FHE schemes have been constructed, starting with the seminal work of Gentry [Gen09]. Recently, new schemes were presented by Brakerski, Gentry and Vaikuntanathan [BV11, BGV11] that achieve greater efficiency and are based on the LWE assumption. We note that in these schemes, the size of the public key depends linearly on the depth of the functions being evaluated. As a result, the complexity of our preprocessing phase depends on the maximum depth of functions that we would like to compute. This issue can be avoided altogether if we assume that the schemes of [BV11, BGV11] are circular secure.

For our construction, we need an FHE scheme with the following additional property, which we refer to as *certifiability*. Loosely speaking, an FHE scheme is said to be certifiable, if there is an efficient algorithm that takes as input a random string r and tests whether it is "good" to use r as randomness in the encryption algorithm Enc. More precisely, a certifiable FHE scheme is associated with a set R, which consists of all the "good" random strings, such that (1) a random string is in R with overwhelming probability; and (2) The Eval algorithm and the decryption algorithm Dec are correct on ciphertexts that use randomness from R to encrypt. A formal definition follows.

DEFINITION 2.7. *A FHE scheme is said to be* certifiable *if there exists a subset $R \subseteq \{0,1\}^{\mathrm{poly}(k)}$ of possible randomness values for which the following hold.*

1. $\Pr[r \in R] = 1 - \mathrm{negl}(k)$*, where the probability is over uniformly sampled $r \leftarrow \{0,1\}^{\mathrm{poly}(k)}$.*
2. *There exists an efficient algorithm $\mathcal{A}_R$ such that $\mathcal{A}_R(r) = 1$ for $r \in R$ and 0 otherwise.*
3. *We have*

$$\Pr_{\mathsf{pk},\mathsf{sk}}\left[\begin{array}{c}\forall b_1,...,b_n \in \{0,1\},\ \forall r_1,...,r_n \in R,\\ \forall \text{ poly-size circuits } f:\{0,1\}^n \to \{0,1\}\\ \mathsf{Dec}_{\mathsf{sk}}(\mathsf{Eval}_{\mathsf{pk}}(f, c_1,...,c_n)) = f(b_1,...,b_n),\\ \text{where } c_i = \mathsf{Enc}_{\mathsf{pk}}(b_i; r_i)\end{array}\right] = 1 - \mathrm{negl}(k).$$

We note that this property holds, for example, for the schemes of [BV11, BGV11]. For the readers who are familiar with these constructions, the set of "good" randomness R corresponds to encrypting with sufficiently "small noise."

2.5 Leaky Distributed Systems

One of the tools in our construction is a compiler that converts any circuit C (with secrets) into a collection of sub-computations (or "modules") $\mathsf{Sub}_1, ..., \mathsf{Sub}_m$, whose sequential evaluation evaluates the circuit C, and which is secure in the *leaky distributed systems* (LDS) model, a model recently introduced by Bitansky *et. al.* [BCG+11].

Before we describe this model and compiler, let us recall prerequisite prior works [JV10, GR10, GR12], which construct such a compiler in the *"only computation leaks"* (OCL) model. In particular, these works demonstrate a compiler that takes a circuit C and converts it into a circuit C' consisting of m disjoint, ordered sub-computations $\mathsf{Sub}_1, \ldots, \mathsf{Sub}_m$, where the input to sub-computation Sub_i depends only on the output of earlier sub-computations. Each of these sub-computations Sub_i is modeled as a non-uniform randomized poly-size circuit, with a "secret state." It was proven that no information about the circuit C is leaked, even if each of these sub-computations is leaky. More specifically, the adversary can request to see a bounded-length function of each Sub_i (separately), and these leakage functions may be adaptively chosen.

These works also consider the continual leakage setting, where leakage occurs over and over again in time. In this setting, the secret state of each Sub_i must be continually updated or refreshed. To this end, after each computation, all the Sub_i's update their secret state by running a randomized protocol Update. We stress that leakage may occur during each of these update protocols, and that such leakage may be a function of both the current secret state and the randomness used by the Update procedure.

In this work, we use such a compiler which is secure in the LDS model [BCG+11]. The LDS model strengthens the OCL model in two ways. First, in the LDS model, the adversary is allowed to *view and control the entire communication between modules*; in contrast, the OCL model assumes the communication between modules is kept secret from the adversary, and that the messages are generated honestly. Second, in the LDS model, the adversary may leak adaptively on each module *in any order*. For instance, the adversary may leak a bit from Sub_i, then a bit from Sub_j, and based on the results, leak again on Sub_i. In contrast, the OCL model only allows the adversary to request leakage information from the module that is currently computing. In particular, this restricts the adversary to leak on modules in order (i.e., first leak from Sub_1, then from Sub_2, etc.).

REMARK 2.8. *For the sake of simplicity of notation, we assume (without loss of generality) that the module Sub_i only sends messages to Sub_{i+1} (where we define $\mathsf{Sub}_{m+1} \triangleq \mathsf{Sub}_1$).*

Moreover, we assume for simplicity that during each computation, where C is evaluated on some input v, each module Sub_i sends a single *message to Sub_{i+1}, and that Sub_m does not send a message to any module, and simply outputs $C(v)$. This assumption indeed holds for the LDS compiler of [BCG+11] which is based on [GR12]. We note that this assumption is not needed for our result to be correct, but it simplifies the notation.*

At the end of each time period, the modules "refresh" their inner state by applying a (possibly distributed) Update procedure, after which they erase their previous state. As with the rest of the computation, the Update procedure is also exposed to leakage, and the adversary controls the exchange of messages during the update.

Definition 2.9 (Leaky Distributed Systems (LDS)). *In a λ-bounded* LDS *attack, a* PPT *adversary $\mathcal{A}$ interacts with modules $(\mathsf{Sub}_1, ..., \mathsf{Sub}_m)$ by adaptively performing any sequence of the following actions:*

- $\mathsf{Interact}(j, \mathsf{msg})$**:** *For $j \in [m]$, send the message* msg *to the j'th submodule, Sub_j, and receive the corresponding reply. Note that the modules are message-driven: they become activated when they receive a message from the attacker, at which point they compute and send the result, and then wait for additional messages.*
- $\mathsf{Leak}(j, L)$**:** *For $j \in [m]$ and a poly-size leakage function $L : \{0,1\}^* \to \{0,1\}$, if strictly fewer than λ queries of the form $\mathsf{Leak}(j, \cdot)$ have been made so far, $\mathcal{A}$ receives the evaluation of L on the secret state of the j'th submodule, Sub_j. Otherwise, $\mathcal{A}$ receives $\perp$.*

In a continual λ-LDS attack*, the adversary $\mathcal{A}$ repeats a λ-bounded* LDS *attack polynomially many times, where between every two consecutive attacks the secret states of the modules are updated. The update is done by running a distributed* Update *protocol among all the modules. We also allow $\mathcal{A}$ to leak during the* Update *procedure, where the leakage function takes as input both the current secret state of Sub_j and the randomness it uses during the* Update *procedure.*

We denote by time period t of submodule Sub_j the time period between the beginning of the $(t-1)$'st Update *procedure and the end of the t'th* Update *procedure in that submodule (note that these time periods are overlapping).*[5] *We allow the adversary $\mathcal{A}$ to leak at most λ bits from each Sub_j during each (local) time period.*

*We refer to such an adversary $\mathcal{A}$ as an λ-*LDS *adversary, and denote the output of $\mathcal{A}$ in such an attack by $\mathcal{A}[\lambda : \mathsf{Sub}_1, ..., \mathsf{Sub}_m : \mathsf{Update}]$.*

We say that the collection of modules $(\mathsf{Sub}_1, ..., \mathsf{Sub}_m)$ is *λ-secure* in the LDS model if for any λ-LDS adversary $\mathcal{A}$ interacting with the modules as described above, there exists a PPT simulator who simulates the output of $\mathcal{A}$.

Definition 2.10 (LDS-Secure Circuit Compiler). *We say that $(\mathcal{C}, \mathsf{Update})$ is a λ-*LDS *secure circuit compiler if for any circuit C and $(\mathsf{Sub}_1, ..., \mathsf{Sub}_m) \leftarrow \mathcal{C}(C)$, the following two properties hold:*

1. **Correctness:** *The collection of modules $(\mathsf{Sub}_1, ..., \mathsf{Sub}_m)$ maintain the functionality of C when all the messages between them are delivered intact.*
2. **Secrecy:** *For every* PPT *λ-*LDS *adversary $\mathcal{A}$ there exists a* PPT *simulator $\mathcal{S}$, such that for any ensemble of poly-size circuits $\{C_n\}$ and any auxiliary input $z \in \{0,1\}^{\mathrm{poly}(n)}$:*

$$\left\{\mathcal{A}(z)[\lambda : \mathsf{Sub}_1, ..., \mathsf{Sub}_m : \mathsf{Update}]\right\}_{n\in\mathbb{N}, C\in C_n} \approx_c \left\{\mathcal{S}^C(z, 1^{|C|})\right\}_{n\in\mathbb{N}, C\in C_n},$$

where $\mathcal{S}$ only queries C on the inputs $\mathcal{A}$ sends to the first module, Sub_1.

Theorem 2.11 ([BCG+11]). *Assuming the existence of a non-committing encryption scheme and a λ-*OCL *circuit compiler which compiles a circuit C to $m(|C|)$ modules, there exists a λ-*LDS *secure circuit compiler $(\mathcal{C}, \mathsf{Update})$ for which $\mathcal{C}(C)$ has the same number of modules, $m(|C|)$.*

We note that there are known constructions of non-committing encryption schemes based on standard cryptographic assumptions, such as the DDH assumption and the RSA assumption. Moreover, a very recent work of Goldwasser and Rothblum [GR12] constructs a λ-OCL circuit compiler (unconditionally) with the following properties.

Theorem 2.12 ([GR12]). *For any security parameter k, there (unconditionally) exists a λ-*OCL *secure circuit compiler for $\lambda = \tilde{\Omega}(k)$, that takes any circuit C into a collection of $O(|C|)$ modules, each of size $O(k^3)$.*

Remark 2.13 (Folklore). *If one additionally assumes the existence of a fully homomorphic encryption (FHE) scheme, then there exists a λ-*LDS *secure circuit compiler $(\mathcal{C}, \mathsf{Update})$ such that for every poly-size circuit C, the number of output sub-computations $\mathsf{Sub}_1, ..., \mathsf{Sub}_m$ generated by $\mathcal{C}$ is polynomial in the security parameter of the FHE scheme and* independent *of the size of C.*

2.6 Weakly Leakage-Resilient MPC

Our construction of a leakage-resilient MPC protocol in the preprocessing model (defined in Section 3.2), uses as a building block an MPC protocol that is leakage-resilient with respect to a *weaker* notion of secrecy (where the ideal world is weakened), as was recently constructed in [BGG+11]. For lack of a better name, we call it *weakly* leakage-resilient MPC. Below, we recall the security model from [BGG+11].

Very briefly, the security definition in [BGG+11] follows the ideal/real world paradigm. They consider a real-world execution without a leak-free preprocessing stage, though they do assume the existence of an honestly generated CRS.[6] The adversary, in addition to corrupting a number of parties, can obtain leakage information on the joint secret states of the honest parties at any point during the protocol execution. Leakage queries may be adaptively chosen based on all information received up to that point (including responses to previous leakage queries), and are computed on the joint secret states of all the honest parties.

[5] Intuitively, time period t is the entire time period where the t'th updated secret states can be leaked. Note that during the t'th Update procedure, both the $(t-1)$'st and the t'th secret state may leak, which is why the time periods are overlapping.

[6] The CRS is simply a truly random string, and thus, could be generated in a leaky environment.

Note that one cannot hope to realize the standard ideal world security in the presence of such leakage attacks.To this end, [BGG+11] consider an ideal world experiment where in addition to learning the output of the function evaluation, the simulator is also allowed to request leakage on the inputs of all the honest parties jointly. Below, we describe the ideal and real world experiments and give the formal security definition from [BGG+11].

Ideal World.

We first describe the ideal world experiment, where n parties $P_1, \ldots, P_n$ interact with an ideal functionality for computing a function f. An adversary may corrupt any subset $M \subset \mathcal{P}$ of the parties. As in the standard MPC ideal world experiment, the parties send their inputs to the ideal functionality and receive the output of f evaluated on all inputs. The main difference from the standard ideal world experiment is that the adversary is also allowed to make *leakage queries* on the inputs of the honest parties. Such queries are evaluated on the *joint* collection of all parties' inputs. The ideal world execution proceeds as follows.

Inputs: Each party P_i obtains an input x_i. The adversary is given auxiliary input z and selects a subset of parties $M \subset \mathcal{P}$ to corrupt.

Sending inputs to trusted party: Each honest party P_i sends its input x_i to the ideal functionality. For each corrupted party $P_i \in M$, the adversary may select any value x_i' and send it to the ideal functionality.

Trusted party computes output: Let $x_1', \ldots, x_n'$ be the inputs that were sent to the ideal functionality. The ideal functionality computes $f(x_1', \ldots, x_n')$.

Adversary learns output: The ideal functionality first sends the evaluation $f(x_1', ..., x_n')$ to the adversary. The adversary replies with either $\mathsf{continue}$ or abort.

Honest parties learn output: If the message is abort, the ideal functionality sends $\perp$ to all honest parties. If the adversary's message was $\mathsf{continue}$, then the ideal functionality sends the function evaluation $f(x_1', \ldots, x_n')$ to all honest parties.

Leakage queries on inputs: The adversary may send (adaptively chosen) leakage queries in the form of efficiently computable functions L_j (described as a circuit). On receiving such a query, the ideal functionality computes $L_j(x_1', \ldots, x_n')$ and returns the output to the adversary.

Outputs: Honest parties output their inputs and the messages they obtained from the ideal functionality. Malicious parties may output an arbitrary PPT function of their initial input (auxiliary input and random-tape) and the message it has obtained from the ideal functionality.

An ideal world adversary $\mathcal{S}$ who obtains a total of λ bits of leakage is referred to as a λ-*leakage ideal adversary*. The overall output of the ideal-world experiment consists of all the inputs and values received by honest parties from the ideal functionality, together with the output of the adversary, and is denoted by $\mathsf{W\text{-}IDEAL}^{f}_{\mathcal{S},M}(1^k, \vec{x}, z)$.

Real World.

The real-world experiment begins by first choosing a common random string crs. Then, each party P_i receives an input x_i and the adversary $\mathcal{A}$ receives auxiliary input z. These values can depend arbitrarily on the crs, but need to be efficiently computable given the crs. However, for the sake of simplicity of notation, throughout this section we assume that these values are independent of the crs.

The adversary $\mathcal{A}$ selects any arbitrary subset $M \subset \mathcal{P}$ of the parties to corrupt. Each corrupted party $P_i \in M$ hands over its input to $\mathcal{A}$. The parties $P_1, \ldots, P_n$ now engage in an execution of a real n-party protocol Π. The adversary $\mathcal{A}$ sends all messages on behalf of the corrupted parties, and may follow an arbitrary polynomial-time strategy. In contrast, the honest parties follow the instructions of Π. Furthermore, at any point during the protocol execution, the adversary may make leakage queries of the form L and learn $L(\mathsf{state}_{\mathcal{P}\setminus M})$, where $\mathsf{state}_{\mathcal{P}\setminus M}$ denotes the concatenation of the protocol states state_i of each honest party P_i. We allow the adversary to choose the leakage queries adaptively.

Honest parties have the ability to toss fresh coins at any point in the protocol, and at that point these coins are added to the state of that party. At the conclusion of the protocol execution, each honest party P_i generates an output according to Π. Malicious parties may output an arbitrary PPT function of the view of $\mathcal{A}$.

An adversary $\mathcal{A}$ who obtains at most λ bits of leakage is referred to as a λ-*leakage real adversary*. Let $\mathsf{Gen_w}$ denote the CRS generation algorithm. Further, let $\mathsf{W\text{-}REAL}^{\Pi}_{\mathcal{A}}(1^k, \mathsf{crs}, \vec{x}, z)$ be the random variable that denotes the values output by the parties at the end of the protocol Π (using $\mathsf{crs} \leftarrow \mathsf{Gen_w}(1^k)$ as the CRS). Then, the overall output of the real-world experiment is defined as the tuple $(\mathsf{crs}, \mathsf{W\text{-}REAL}^{\Pi}_{\mathcal{A},M}(1^k, \mathsf{crs}, \vec{x}, z))$.

We now state the formal security definition.

Definition 2.14 (λ-Weakly Leakage-Resilient MPC). *A protocol Π evaluating a functionality f is a λ-*weakly leakage-resilient *MPC protocol if for every PPT λ-leakage real adversary $\mathcal{A}$, there exists a λ-leakage ideal adversary $\mathcal{S} = (\mathcal{S}^{\mathsf{crs}}, \mathcal{S}^{\mathsf{exec}})$, corrupting the same parties as $\mathcal{A}$, such that for every input vector $\vec{x}$, every auxiliary input $z \in \{0,1\}^*$, and every subset $M \subset \mathcal{P}$, it holds that the distribution*

$$\left\{\mathsf{crs}, \mathsf{W\text{-}IDEAL}^{f}_{\mathcal{S}^{\mathsf{exec}}(\mathsf{crs},\mathsf{trap}),M}(1^k, \vec{x}, z)\right\}_{k\in N}$$

is computationally indistinguishable from the distribution

$$\left\{\mathsf{crs}', \mathsf{W\text{-}REAL}^{\Pi}_{\mathcal{A},M}(1^k, \mathsf{crs}', \vec{x}, z)\right\}_{k\in N},$$

where $(\mathsf{crs}, \mathsf{trap}) \leftarrow \mathcal{S}^{\mathsf{crs}}(1^k)$, and $\mathsf{crs}' \leftarrow \mathsf{Gen_w}(1^k)$.

Theorem 2.15 ([BGG+11]). *Based on the DDH assumption, for every poly-size function f, for every leakage bound $\lambda \in \mathbb{N}$, and any number of parties and corrupted parties, there exists a protocol Π in the common random string model for computing f that is λ-weakly leakage resilient as per Definition 2.14.*

Remark 2.16. *We note that Theorem 2.15 holds even if we allow the input vector $\vec{x}$ and the auxiliary input z to be arbitrary poly-time computable functions of the crs. We eliminated this dependency from Definition 2.14 only for the sake of simplicity of notation.*

REMARK 2.17 (STANDALONE VS. UC SECURITY). *The main result in [BGG+11] actually achieves a stronger notion of* universally composable (UC) *security, at the cost of additionally relying on the decisional linear assumption over bilinear groups. Indeed, their UC-secure WLR-MPC construction relies on a leakage-resilient UC-NIZK system, whose only known construction [GJS11, GOS06] is based on the decisional linear assumption in the bilinear groups setting.*

However, for the present paper, it suffices to obtain a "standalone" secure construction of WLR-MPC. Thus, it is possible to replace the UC-NIZK system with a standalone secure interactive *weakly leakage-resilient ZKPoK system. This, in turn, can be based on the DDH assumption. The resulting WLR-MPC achieves standalone security based on only the DDH assumption in the CRS model.*

Security against disjoint leakage.

In Definition 2.14, the real-world adversary $\mathcal{A}$ is allowed to obtain *joint* leakage on the secret states of the honest parties. In the present work, we consider a weaker adversarial model, in which the leakage on each honest party in the real world is *disjoint* (i.e., $\mathcal{A}$ is not allowed to leak on the joint secret states of the honest parties). Theorem 2.15 clearly still applies to this setting. However, we note that the *ideal world* guarantee does not become stronger when we consider this set of restricted adversaries: that is, even to simulate such adversaries, the simulator $\mathcal{S}$ needs *joint* leakage on the inputs of all the honest parties.[7]

2.6.1 Security for randomized functions

We note that Theorem 2.15 holds for *deterministic* functions. In this work, we need to use a weak leakage resilient protocol for *randomized* functions (since the modules in the OCL leakage resilient circuit compute randomized functions). We show that in our setting, where leakage in the real world is disjoint, the number of parties is polynomially related to the security parameter, and a constant fraction of the parties are honest, then we can construct weak leakage resilient protocols for randomized functions.

THEOREM 2.18 (INFORMAL). *Theorem 2.15 holds also for randomized functions if we restrict the adversaries to leak on the honest parties* disjointly*, when the number of parties is polynomially related to the security parameter, and ϵ-fraction of them are honest for some constant $\epsilon > 0$.*

Due to lack of space we defer the proof of this theorem to the final version.

3. OUR MODEL

In this section, we present the MPC model and the security definition considered in this paper. We start by giving a brief overview of our model and then proceed with a formal description.

Overview. We consider the setting of n parties $\mathcal{P} = \{P_1, ..., P_n\}$ within a synchronous point-to-point network with authenticated broadcast channel [DS83] who wish to jointly compute any ppt function over their private inputs. Specifically, we consider the case where the parties wish to perform *arbitrarily many* evaluations of functions of their choice. We refer to a protocol that allows computation of multiple functions (over a given set of inputs) as a *multi-function MPC protocol*. Unlike the standard MPC setting, we consider security of a multi-function MPC protocol against "leaky" adversaries that may (continuously) leak on the secret state of each honest party during the protocol execution.

To formally define security, we turn to the real/ideal paradigm. Very briefly, we consider a real-world execution where an adversary, who corrupts any arbitrary number of parties in the system, may additionally obtain arbitrary bounded, independent leakage on the secret state of each honest party. However, unlike the recent works on leakage-resilient interactive protocols [GJS11, BCH11, BGK11, DHP11, BGG+11], we consider the *standard* ideal world model, where the adversary does not learn *any* information on the honest party inputs.

Note that if we do not put any restriction on the real-world adversary, and in particular, if he is allowed to obtain leakage *throughout* the protocol execution, then it is impossible to realize the standard ideal world model, since the adversary may simply leak on the inputs of the honest parties, while this information cannot be simulated in the ideal world. With this in mind, we (necessarily) allow for a "leak-free" one-time *preprocessing stage* that happens at the beginning of the real-world execution. Furthermore, to withstand *continual* leakage attacks, we (necessarily) allow for periodic *updates* of the secret values of the parties. We allow leakage to occur during this update procedure as usual.

We now proceed to give a formal description of our model in the remainder of this section. In Section 3.1, we describe the ideal world experiment. In Section 3.2, we describe the real world experiment. Finally, in Section 3.3, we present our security definition.

Throughout this work, we assume that the functions to be evaluated give the same output to all parties. This is for simplicity of exposition, since otherwise, if the output itself is a secret value (given to an honest party) then this value can be leaked. This can be handled by complicating our security guarantees, and, indeed, one can tweak our construction to ensure that the adversary learns *only* leakage information on such outputs. However, for the sake of simplicity, we choose to avoid this issue in this manuscript.

3.1 Ideal World

In the ideal world, each party P_i sends her input x_i to a trusted third party. Whenever the adversary $\mathcal{A}$ sends a poly-size circuit f to the trusted party, it sends back $f(x_1, \ldots, x_n)$. Since we consider the case of dishonest majority, we can only obtain security with abort: i.e., the adversary first receives the function output $f(x_1, \ldots, x_n)$, and then chooses whether the honest parties also learn the output, or to prematurely abort. The adversary can query the trusted party many times with various functions f_j. Moreover, these functions can be adaptively chosen, based on the outputs of previous functions. The ideal world model is formally described below.

Inputs: Each party P_i obtains an input x_i. The adversary is given auxiliary input z. He selects a subset of the

[7] As mentioned in Section 1.4, if we could simulate real-world adversaries that obtain only *disjoint* leakage queries, with a simulator that obtains only *disjoint* leakage queries, then this would almost immediately give us a result similar to ours: An MPC protocol with preprocessing that is secure against continual leakage.

parties $M \subset \mathcal{P}$ to corrupt, and is given the inputs x_ℓ of each party $P_\ell \in M$.

Sending inputs to trusted party: Each honest party P_i sends its input x_i to the ideal functionality. For each corrupted party $P_i \in M$, the adversary may select any value x'_i and send it to the ideal functionality.

Trusted party computes output: Let $x'_1, ..., x'_n$ be the inputs that were sent to the trusted party. Then, the following is repeated for any (unbounded) polynomial number of times:

- **Function selection:** The adversary chooses a poly-size circuit f_j, and sends it to the ideal functionality.
- **Adversary learns output:** The ideal functionality sends the evaluation $f_j(x'_1, ..., x'_n)$ to the adversary. The adversary replies with either continue or abort.
- **Honest parties learn output:** If the adversary's message was abort, then the trusted party sends $\perp$ to all honest parties and the experiment concludes. Otherwise, if the adversary's message was continue, then it sends the function output $f_j(x'_1, ..., x'_n)$ to all honest parties.

Outputs: Honest parties output all the messages they obtained from the ideal functionality. Malicious parties may output an arbitrary PPT function of the adversary's view.

The overall output of the ideal-world experiment consists of the outputs of all parties. For any ideal-world adversary $\mathcal{S}$ with auxiliary input $z \in \{0,1\}^*$, any input vector $\vec{x}$, any set of functions $\{f_j\}_{j=1}^p$ chosen by the adversary, and security parameter k, we denote the output of the corresponding ideal-world experiment by

$$\mathsf{IDEAL}_{\mathcal{S},M}\left(1^k, \vec{x}, z, \{f_j\}_{j=1}^p\right).$$

Note that this is a slight abuse of notation since the functions $\{f_j\}_{j=1}^p$ may be chosen adaptively.

3.2 Real World

The real world execution begins by an adversary $\mathcal{A}$ selecting any arbitrary subset of parties $M \subset \mathcal{P}$ to corrupt. The parties then engage in an execution of a real n-party multi-function MPC protocol $\Pi = (\Pi_{\mathsf{Pre}}, \Pi_{\mathsf{input}}, \Pi_{\mathsf{Online}})$ that consists of three stages, namely, (a) a *preprocessing phase*, (b) an *input phase*, and (c) an *online phase*, as described below. We assume that honest parties have the ability to toss fresh coins at any point. Throughout the execution of Π, the adversary $\mathcal{A}$ sends all messages on behalf of the corrupted parties, and may follow an arbitrary polynomial-time strategy. In contrast, the honest parties follow the instructions of Π. Furthermore, at any point during the protocol execution (except during the preprocessing and the input phases), the adversary may leak on the *entire* secret state of each honest parties, via an *MPC leakage query*, defined as follows.

DEFINITION 3.1. *An* MPC leakage query *is defined by* $\mathsf{Leak}(i, L)$*, where* $i \in [n]$ *and* $L : \{0,1\}^* \to \{0,1\}$ *is a poly-size circuit. When an adversary sends a leakage query* $\mathsf{Leak}(i, L)$*, he receives the evaluation of* L *on the* entire *secret state of party* P_i*.*

We now formally describe the different phases in the protocol.

Preprocessing phase: This phase is *interactive* and *leak-free*, and is run only once. It is independent of the inputs of the parties, and is independent of the functions that will later be evaluated. Thus, this phase can be run in the beginning of time, before the parties even know what their inputs are, or what functions they would like to evaluate.

We assume that no leakage occurs during the run of this preprocessing phase, but we do allow leakage to occur as soon as the preprocessing phase ends. At the end of this phase each party P_i has an (initial) secret state $\mathsf{state}_1^{P_i}$.

Input phase: This phase is *non-interactive* and *leak-free*, and depends only on the inputs $x_1, ..., x_n$ (independent of the functions to be computed). Whenever a party P_i gets (or chooses) a secret input x_i, she does some local computation which may depend on her secret input x_i and on her secret state $\mathsf{state}_1^{P_i}$. She then sends a message to all parties, and erases her secret input x_i. One may think of this as a "hot potato" model, where the parties never store their inputs for very long (since they are concerned with leakage), but rather immediately share their input as if it were a "hot potato".

We assume that the party P_i is not leaked upon during the execution of this phase. However, leakage may occur between the preprocessing phase and the input phase, and leakage may occur immediately after the input phase.

We emphasize that each party can change her input as often as she wants by simply re-running the input phase with the new input.[8]

Online phase: This phase takes place in a *leaky* environment. During this phase, the parties carry out an unbounded number of function evaluations on their inputs, and update their respective secret states. At any point during this phase, $\mathcal{A}$ may make adaptively-chosen leakage queries, as per Definition 3.1, in the manner as described below.

Whenever $\mathcal{A}$ wishes to compute a function f_j (represented as a poly-size circuit), all parties execute the function evaluation protocol Π_{Comp}, described below. Whenever $\mathcal{A}$ wants the honest parties to update their secret states, all parties execute the update protocol Π_{Update}, described below. We let $\Pi_{\mathsf{Online}} = (\Pi_{\mathsf{Comp}}, \Pi_{\mathsf{Update}})$. We begin at leakage time period $\ell = 1$; after each update procedure, ℓ is incremented.

- **Computation procedure:**
  1. All parties execute protocol $\Pi_{\mathsf{Comp}}(f_j)$, where honest parties P_i act in accordance with input $\mathsf{state}_\ell^{P_i}$. Note that the secret state of parties may change during the execution of this protocol, as dictated by Π_{Comp}.

[8]For simplicity, in the security proof in Section 5, we assume that the parties run the input phase only once, however the proof extends readily to the case that the parties rerun the input phase many times with different inputs.

2. At the conclusion of the computation phase, each honest party P_i outputs his final message of the protocol (which should correspond to the evaluation of f_j). Malicious parties may output an arbitrary PPT function of the view of $\mathcal{A}$.

- ℓ^{th} **Update procedure:**
  1. All parties execute protocol Π_{Update}, where honest parties P_i act in accordance with input $\mathsf{state}_\ell^{P_i}$.
  2. At the conclusion of the update phase, each honest party P_i sets $\mathsf{state}_{\ell+1}^{P_i}$ to be P_i's output from Π_{Update}. Each honest P_i erases $\mathsf{state}_\ell^{P_i}$.
  3. Increment $\ell \leftarrow \ell + 1$.

Leakage: Initialize each leaked_ℓ to 0. Each leakage query (i, L) made by $\mathcal{A}$ during the ℓ^{th} time period is answered as follows.

- During the **computation** phase: if $\mathsf{leaked}_\ell \geq \lambda$, then $\mathcal{A}$ receives $\emptyset$. Otherwise, $\mathcal{A}$ receives the evaluation of L on the current secret state of party P_i, and $\mathsf{leaked}_\ell \leftarrow \mathsf{leaked}_\ell + 1$.
- In ℓth **update** phase: if *either* $\mathsf{leaked}_\ell \geq \lambda$ *or* $\mathsf{leaked}_{\ell+1} \geq \lambda$, then $\mathcal{A}$ receives $\emptyset$. Otherwise, $\mathcal{A}$ receives the evaluation of L on the current secret state of party P_i, and both $\mathsf{leaked}_\ell \leftarrow \mathsf{leaked}_\ell + 1$ and $\mathsf{leaked}_{\ell+1} \leftarrow \mathsf{leaked}_{\ell+1} + 1$.

We emphasize that the $\mathcal{A}$'s leakage queries may be made on any party, adaptively chosen based on all information received up to that point (including responses to previous leakage queries). The only restriction is that the number of bits leaked between the execution of any two consecutive update protocols is bounded. Note that the leakage queries made during the ℓ'th update phase (where parties transition between their ℓ'th and $(\ell+1)$'st secret states) are counted against *both* the ℓ'th and $(\ell+1)$'st time period, where the ℓ'th time period is the time period where the party stores her ℓ'th secret state. The reason for this "double counting" is that during the ℓ'th update phase, the adversary can leak both on the ℓ'th secret state and on the $\ell+1$'st secret state of the party.

We refer to an adversary who corrupts t parties $M \subset \mathcal{P}$ and makes up to λ leakage queries in each time period as a (t, λ)-*continual leakage adversary.*

For any adversary $\mathcal{A}$ with auxiliary input $z \in \{0,1\}^*$, any inputs $\{x_i\}_{i=1}^n$, any set of functions $\{f_j\}_{j=1}^p$ chosen (adaptively) by the adversary, and any security parameter k, we denote the output of the multi-function MPC protocol $\Pi = (\Pi_{\mathsf{Pre}}, \Pi_{\mathsf{input}}, \Pi_{\mathsf{Online}})$ by

$$\mathsf{REAL}_{\mathcal{A},M}^{\Pi}\left(1^k, \vec{x}, z, \{f_j\}_{j=1}^p\right).$$

Loosely speaking, we say that a protocol Π is a *leakage-resilient* multi-function MPC protocol if any adversary, who corrupts a subset of parties, receives leakage information as described above, and runs the protocol with honest parties on any (unbounded) sequence of functions $f_1, ..., f_p$, gains *no information* about the inputs of the honest parties beyond the output of the functions $f_j(x_1, ..., x_n)$ for $j = 1, ..., p$. We formalize this in the next subsection.

3.3 Security Definition

In what follows, we formally define our model of security; i.e., what it means for a real-world protocol to emulate the desired ideal world.

Definition 3.2 (Leakage-Resilient MPC). *A multi-function evaluation protocol* $\Pi = (\Pi_{\mathsf{Pre}}, \Pi_{\mathsf{input}}, \Pi_{\mathsf{Online}})$ *is said to be* λ-leakage-resilient against t malicious parties *if for every PPT* (t, λ)*-continual leakage MPC adversary* $\mathcal{A}$ *in the real world, there exists a PPT adversary* $\mathcal{S}$ *corrupting the same parties in the ideal world such that for every input vector* $\vec{x}$*, every auxiliary input* z*, and any (adaptively chosen) set of functions* $\{f_j\}_{j=1}^p$ *where* $p = \text{poly}(k)$*, it holds that*

$$\mathsf{IDEAL}_{\mathcal{S},M}\left(1^k, \vec{x}, z, \{f_j\}_{j=1}^p\right) \approx_c \mathsf{REAL}_{\mathcal{A},M}^{\Pi}\left(1^k, \vec{x}, z, \{f_j\}_{j=1}^p\right).$$

Note that we do *not* allow the simulator to request leakage on honest parties' inputs in the ideal world, as was done in [BCH11, DHP11, BGG+11], and thus model a stronger notion of secrecy than what was achieved in prior works.[9]

4. OUR CONSTRUCTION

In this section, we construct a leakage-resilient multi-function MPC protocol, as defined in Section 3. Our construction uses the following ingredients:

1. $(\mathcal{C}, \mathsf{Update})$: a λ-LDS secure circuit compiler, as in Theorem 2.11. Recall for a circuit C, the compiler $\mathcal{C} : C \mapsto (\mathsf{Sub}_1, ..., \mathsf{Sub}_m)$ yields a collection of modules whose sequential execution evaluates C, and which are secure in the LDS model (see Section 2.5 for details).
2. Elect: a public-coin protocol for electing m disjoint committees (where m is the number of modules from above), each of size approximately k, as in Lemma 2.6.
3. $(\mathsf{Gen}_{\mathsf{eq}}, \mathsf{Com}, \mathsf{Rec}, \mathcal{S}_{\mathsf{eq}} = (\mathcal{S}_{\mathsf{eq}}^{\mathsf{crs}}, \mathcal{S}_{\mathsf{eq}}^{\mathsf{com}}))$: a crs-based equivocal commitment scheme, as in Lemma 2.5.
4. $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$: a fully homomorphic public-key encryption (FHE) scheme that is certifiable with respect to an efficiently testable set $R \subseteq \{0,1\}^{\text{poly}(k)}$, as described in Section 2.4.
5. $(\mathsf{Gen}_{\mathsf{nizk}}, \mathsf{P}, \mathsf{V}, \mathcal{S}_{\mathsf{nizk}} = (\mathcal{S}_{\mathsf{nizk}}^{\mathsf{crs}}, \mathcal{S}_{\mathsf{nizk}}^{\mathsf{proof}}))$: a non-interactive zero-knowledge (NIZK) proof of knowledge (as in Lemma 2.3) for the NP language
$$L = \{(\mathsf{pk}, \hat{x}) : \exists\ (x, r) \text{ s.t. } r \in R,\ \hat{x} = \mathsf{Enc}_{\mathsf{pk}}(x; r)\}, \quad (1)$$
where $R \subseteq \{0,1\}^{\text{poly}(k)}$ is the set for which the FHE scheme is certifiable.
6. $\mathsf{MPC}(F)$: a standard multiparty computation protocol for evaluating a function F, with no leakage resilience guarantees, such as [GMW87].
7. $(\mathsf{Gen}_{\mathsf{w}}, \mathsf{MPC}_{\mathsf{w}}(F))$: a λ-*weakly* leakage-resilient multiparty computation (WLR-MPC) protocol for evaluating a function F in the common random string model, as given by Theorem 2.18.

[9] With the (necessary) addition of a one-time leak-free preprocessing phase.

THEOREM 4.1. *Fix any constants $\epsilon, \delta > 0$. Then, assuming the existence of the ingredients 1 - 7 listed above (where the* LDS *circuit compiler and WLR-MPC protocol are secure with leakage parameter λ), there exists a λ-leakage-resilient multi-function evaluation MPC protocol $\Pi = (\Pi_{\mathsf{Pre}}, \Pi_{\mathsf{input}}, \Pi_{\mathsf{Update}})$ for $n \geq k^{\delta}$ parties, tolerating $t = (1-\epsilon)n$ corrupted parties.*

Remark.

The reason we need the number of parties to be polynomially related to the security parameter is two-fold. First, in the preprocessing phase, the protocol Π_{Pre} elects committees $\mathcal{E}_1, \ldots, \mathcal{E}_m$, and security of the protocol relies on the fact that these committees are disjoint and each committee contains a constant fraction of honest parties. Thus, if n is a constant, then the resulting security guarantee is that the advantage of any PPT distinguisher in the security game is bounded (from below) by a constant. More generally, the advantage is $\geq 2^{-\epsilon n}$ (see Lemma 2.6).

The second reason we the number of parties must be large is that the number m of disjoint committees $\mathcal{E}_1, \ldots, \mathcal{E}_m$ we need to elect is large. This is because the number of committees is exactly the number of modules generated by the LDS compiler, when applied to the decryption circuit $\mathsf{Dec}_{\mathsf{sk}}$ of the underlying FHE scheme. Since the only LDS compiler we know (that does not use secure hardware) requires $m = O(|\mathsf{Dec}_{\mathsf{sk}}|)$, the number of modules must be at least the security parameter of the underlying FHE scheme (which we can set to be k^{δ}).

We now present the protocol $\Pi = (\Pi_{\mathsf{Pre}}, \Pi_{\mathsf{input}}, \Pi_{\mathsf{Online}})$, where $\Pi_{\mathsf{Online}} = (\Pi_{\mathsf{Comp}}, \Pi_{\mathsf{Update}})$. At a high level, Π is defined as follows:

Preprocessing phase Π_{Pre}: In the preprocessing phase, the parties run a (standard) MPC to collectively generate a key pair $(\mathsf{pk}, \mathsf{sk})$ for the FHE scheme, and to secret share sk in such a way that (a) learning the shares of corrupted parties, and leakage on each remaining share, does not damage the security of the FHE, but (b) collectively, the shares can be used to evaluate the decryption circuit in a leaky environment. More specifically, shares are generated by running the LDS compiler on the decryption circuit $\mathsf{Dec}_{\mathsf{sk}}(\cdot)$ (with sk hardwired) to obtain a sequence of modules $\mathsf{Sub}_1, ..., \mathsf{Sub}_m$; the parties elect corresponding (disjoint) committees $\mathcal{E}_1, ..., \mathcal{E}_m$, and secret share each Sub_j among parties in $\mathcal{E}_j$, using a standard secret sharing scheme (e.g., the simple xor scheme). To ensure that parties provide the correct secret shares of the Sub_j's in future computations, within the MPC the parties collectively generate and publish commitments to each correct share.

In addition, the preprocessing phase is used to generate crs setup information for subsidiary tools used throughout the protocol. This is also done via a (standard) MPC.

(Note that the preprocessing procedure is independent of parties' secret inputs and functions to be evaluated.)

Input phase Π_{input}: Each time a party P_i wishes to submit a new secret input x_i, she computes and publishes an encryption $\hat{x}_i$ of x_i under the FHE scheme (specifically, under the public key pk for the FHE that was generated during the preprocessing phase). To ensure that malicious parties do not send malformed ciphertexts, which could ruin the correctness of homomorphic evaluation later down the line (and potentially damage security), each party accompanies her published ciphertext $\hat{x}_i$ with a NIZK proof of knowledge that the ciphertext is properly formed.

Online phase Π_{Online}: The online phase consists of two parts: the *computation phase*, in which parties collectively evaluate a queried function f on all inputs, and the *update phase*, in which parties collectively refresh their secret states.

Computation phase Π_{Comp}: Each time the adversary requests the evaluation of a function f on all parties' inputs, two steps take place. First, each party (individually) homomorphically evaluates the function f on the *encrypted* vector of inputs $\hat{x} = (\hat{x}_1, ..., \hat{x}_n)$. Note that the result, $\hat{y}_f$, is an encryption of the desired value $f(\vec{x})$. Next, the parties jointly decrypt, using their shares of sk from the preprocessing phase. Namely, the parties execute the sequence of modules $\mathsf{Sub}_1, ..., \mathsf{Sub}_m$ obtained by the LDS compiler applied to $\mathsf{Dec}_{\mathsf{sk}}(\cdot)$, where the input to the first module Sub_1 is $\hat{y}_f$. To emulate the execution of each module Sub_j, the parties of committee $\mathcal{E}_j$ run a WLR-MPC protocol among themselves. Within the WLR-MPC, the parties of $\mathcal{E}_j$ combine their secret shares $\mathsf{Sub}_{j,i}$ (checking first to make sure each party's share agrees with the corresponding published commitment) and execute the computation dictated by Sub_j. Communication between modules is performed by having *all* parties of committee $\mathcal{E}_j$ send the appropriate message to *all* parties of the next committee, $\mathcal{E}_{j+1}$. The output of the final module, Sub_m, is the evaluation $f(\vec{x})$.

Update phase Π_{Update}: Each time the adversary requests that parties update their secret states, the parties execute the update procedure of the LDS compiler, where each module computation is performed via a WLR-MPC among the parties of the corresponding committee, as above. The only difference here is that the secret state Sub_j of each module is also changing. Thus, during each execution of a module Sub_j, the corresponding committee must also generate fresh secret shares for its parties, and new commitment and decommitment information for each share. To provide the required correctness and secrecy guarantees, this process takes place as part of the committee's WLR-MPC execution.

The formal descriptions of $\Pi_{\mathsf{Pre}}, \Pi_{\mathsf{input}}, \Pi_{\mathsf{Comp}}$, and Π_{Update} appear in Figures 1, 2, 3, and 4, respectively.

REMARK 4.2. *Throughout the protocol description (as well as throughout the proof), we define* abort *to be the action of broadcasting the message "abort" to all parties. At any point in which a party receives an "abort" message, he runs* abort *and exits the protocol.*

5. PROOF OF SECURITY

PROOF OF THEOREM 4.1. Let $\mathcal{A}$ be any real-world PPT adversary for Π. Denote by $M \subset \mathcal{P}$ the set of parties corrupted by $\mathcal{A}$.

Preprocessing Phase:
Input: 1^k. No leakage allowed.

1. The parties elect m disjoint committees $\mathcal{E}_j$ of size approximately k' by running Elect. Here, k' is the security parameter for the FHE scheme and $m = \text{poly}(k')$ is the number of modules produced by the LDS compiler when run on the decryption circuit $\mathsf{Dec}_{\mathsf{sk}}$ for this security parameter. We take $k' = k^{\Theta(1)}$ as large as possible while maintaining $m \cdot k'^2 \leq n$.

2. All parties engage in an execution of the (standard) MPC protocol $\mathsf{MPC}(F_{\mathsf{crs}})$ to compute the (randomized) functionality F_{crs} described as follows. Functionality F_{crs} does not take any inputs and computes the following: (a) a CRS $\mathsf{crs}_{\mathsf{w}} \leftarrow \mathsf{Gen}_{\mathsf{w}}(1^k)$ for the weakly leakage-resilient MPC protocol $(\mathsf{Gen}_{\mathsf{w}}, \mathsf{MPC}_{\mathsf{w}}(F))$, (b) a CRS $\mathsf{crs}^i_{\mathsf{eq}} \leftarrow \mathsf{Gen}_{\mathsf{eq}}(1^k)$ for each party P_i for the equivocal commitment scheme $(\mathsf{Gen}_{\mathsf{eq}}, \mathsf{Com}, \mathsf{Rec}, \mathcal{S}_{\mathsf{eq}})$, and (c) a CRS $\mathsf{crs}^i_{\mathsf{nizk}} \leftarrow \mathsf{Gen}_{\mathsf{nizk}}(1^k)$ for each party P_i for the NIZK proof of knowledge system $(\mathsf{Gen}_{\mathsf{nizk}}, \mathsf{P}, \mathsf{V}, \mathcal{S}_{\mathsf{nizk}})$. Denote by crs the tuple $(\{\mathsf{crs}^i_{\mathsf{eq}}, \mathsf{crs}^i_{\mathsf{nizk}}\}_{i=1}^n, \mathsf{crs}_{\mathsf{w}})$.

3. All parties engage in an execution of the (standard) MPC protocol $\mathsf{MPC}(F_{\mathcal{E}_1,\ldots,\mathcal{E}_m,\mathsf{crs}})$ to collectively compute the randomized functionality $F_{\mathcal{E}_1,\ldots,\mathcal{E}_m,\mathsf{crs}}$ (that does not take any inputs) defined as follows:

   The (randomized) function:

   Generate a key pair $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^{k'})$ for the FHE scheme.

   Evaluate the LDS circuit transformation on the decryption circuit for sk:

   $$(\mathsf{Sub}_1, ..., \mathsf{Sub}_m) \leftarrow \mathcal{C}(\mathsf{Dec}_{\mathsf{sk}}).$$

   (We abuse notation and denote by Sub_j both the computation of the submodule and the secret state corresponding to the submodule.)

   For each $j \in [m]$, secret share $\mathsf{Sub}_j = \mathsf{Sub}_{j,1} \oplus \cdots \oplus \mathsf{Sub}_{j,|\mathcal{E}_j|}$ among the parties in the j'th committee, $\mathcal{E}_j$.

   For each share $\mathsf{Sub}_{j,i}$ generated in the previous step, compute a commitment $(c_{j,i}, d_{j,i}) \leftarrow \mathsf{Com}(\mathsf{crs}^{\alpha}_{\mathsf{eq}}, \mathsf{Sub}_{j,i})$, where P_α is the i'th party in $\mathcal{E}_j$ (i.e., the party that receives the share $\mathsf{Sub}_{j,i}$).

   Output: The outputs are as follows.

   $$\begin{array}{rl} \text{All parties:} & \mathsf{pk},\ \{c_{j,i}\}_{j\in[m], i\in[|\mathcal{E}_j|]} \\ \text{Party } i \text{ of } \mathcal{E}_j\text{:} & \mathsf{Sub}_{j,i},\ d_{j,i} \end{array}$$

4. Each party erases all intermediate values of the MPC executions.

(Note that Steps 2 and 3 can be combined into a single multi-party computation execution, but have been split into two separate executions for ease of explanation and proof).

Figure 1: Protocol Π_{Pre}: Preprocessing phase.

Input Phase: Party P_i wishes to submit a new private input, x_i. No leakage allowed.
Public inputs: pk, $\{\mathsf{crs}^i_{\mathsf{nizk}}\}_{i=1}^n$.
Private input: x_i, held by party P_i.

Party P_i performs the following steps:

1. Sample a value $r_i \leftarrow R \subseteq \{0,1\}^{\text{poly}(k)}$ via rejection sampling. Recall the FHE scheme is certifiable with respect to the set $R \subseteq \{0,1\}^{\text{poly}(k)}$ (see Definition 2.7).

2. Encrypt $\hat{x}_i = \mathsf{Enc}_{\mathsf{pk}}(x_i; r_i)$.

3. Compute a NIZK proof of knowledge that $(\mathsf{pk}, \hat{x}_i) \in L$ using witness (x_i, r_i) and CRS $\mathsf{crs}^i_{\mathsf{nizk}}$. (See Equation (1) above for the definition of L). That is, $\pi_i \leftarrow \mathsf{P}(\mathsf{crs}^i_{\mathsf{nizk}}, (\mathsf{pk}, \hat{x}_i), (x_i, r_i))$.

4. Send the pair $(\hat{x}_i, \pi_i)$ to all parties.

   (It suffices to send it to parties in $\mathcal{E}_1$.)

5. Erase initial input x_i, together with all intermediate values of the input phase.

Figure 2: Protocol Π_{input}: Input phase.

We construct an adversary $\mathcal{S}$ in the ideal world who simulates the real-world view of $\mathcal{A}$ by simulating the honest parties in the real world experiment. We do so by a sequence of intermediate steps, where we show how to simulate these values given less and less information, eventually given only the function evaluations $f(x_1, ..., x_n)$, as in the ideal-world experiment. More explicitly, we consider the following sequence of hybrid experiments. We note that all ideal functionalities in the hybrid experiments are implicitly *with abort*: i.e., the ideal functionality first outputs to only the adversary, who decides whether outputs are also delivered to honest parties, or whether the protocol ends in abort.

In what follows we describe all of our hybrid experiments. We defer the construction of the corresponding simulator and the proof of indistinguishability to the full version. For each hybrid, we include (in the parentheses) the primary reason why Hybrid i can be simulated from Hybrid $i-1$.

Hybrid 0. The real world: i.e., the adversary interacts with honest parties in the real-world experiment running Π.

Hybrid 1. (Elect protocol) The same as the real-world experiment, except that if any of the committees $\mathcal{E}_1, ..., \mathcal{E}_m$ elected during the preprocessing phase has *fewer* than $\frac{\epsilon}{2}k$ parties, or if the fraction of honest parties in any committee is less than $\frac{\epsilon}{3}$, the experiment immediately concludes with output fail. We assume for simplicity of notation (later on) that, if the experiment does not fail, the first party of each committee $\mathcal{E}_j$ is honest.

Hybrid 2. (MPC security) The same as Hybrid 1, except instead of collectively generating the CRS values (for the equivocal commitment scheme, the WLR-MPC, and the NIZK proof system) via an MPC protocol during the preprocessing phase, we assume a setup model where these values are (honestly) generated beforehand, and all parties run with these CRS values

Computation Phase:
Public inputs: f, pk, $\hat{x} = (\mathsf{Enc}_{\mathsf{pk}}(x_1), ..., \mathsf{Enc}_{\mathsf{pk}}(x_n))$, $\mathsf{crs} = (\{\mathsf{crs}^i_{\mathsf{eq}}, \mathsf{crs}^i_{\mathsf{nizk}}\}_{i=1}^n, \mathsf{crs}_{\mathsf{w}})$, $\mathcal{E}_1, ..., \mathcal{E}_m$, $\{c_{j,i}\}_{j\in[m], i\in[|\mathcal{E}_j|]}$.
Private inputs: $(\mathsf{Sub}_{j,i}, d_{j,i})$, held by party i of $\mathcal{E}_j$.

1. All parties homomorphically evaluate f on the encrypted input vector: $\hat{y} = \mathsf{Eval}_{\mathsf{pk}}(\hat{x}, f)$.

   (It suffices that only parties in $\mathcal{E}_1$ compute $\hat{y}$.)

2. The parties execute the Decryption Cascade with $\mathsf{input}_1 = \hat{y}$.

Decryption Cascade:

1. For $j = 1, ..., m$:

   The parties in $\mathcal{E}_j$ engage in an execution of the λ-weakly leakage-resilient MPC protocol $\mathsf{MPC}_{\mathsf{w}}(F_j)$ using CRS $\mathsf{crs}_{\mathsf{w}}$ to compute the (randomized) functionality F_j defined as follows:

   Input: $(\mathsf{Sub}_{j,i}, d_{j,i}, \mathsf{input}_j)$, held by party i of $\mathcal{E}_j$.

   The function F_j:

   (a) If any of the input_j's are inconsistent, or $\mathsf{Sub}_{j,i} \neq \mathsf{Rec}(\mathsf{crs}^{\alpha}_{\mathsf{eq}}, c_{j,i}, d_{j,i})$ for any i, where P_α is the i'th party in committee $\mathcal{E}_j$ (i.e., if any party's share does not agree with the corresponding published commitment), then abort.

   (b) Otherwise, let $\mathsf{Sub}_j = \bigoplus_{i=1}^{|\mathcal{E}_j|} \mathsf{Sub}_{j,i}$.

   (c) Evaluate the j'th module on input_j: that is, $\mathsf{input}_{j+1} := \mathsf{Sub}_j(\mathsf{input}_j)$.

   Output: All parties learn input_{j+1}.

   At the conclusion of the WLR-MPC execution, each party in $\mathcal{E}_j$ erases all intermediate values generated during the WLR-MPC, keeping only $(\mathsf{Sub}_{j,i}, d_{j,i})$.

   Each party in $\mathcal{E}_j$ sends the value of input_{j+1} to all parties in $\mathcal{E}_{j+1}$ (where $\mathcal{E}_{m+1} := \mathcal{P}$ the set of *all* parties).

   If any party in $\mathcal{E}_{j+1}$ receives disagreeing values of input_{j+1} from parties in $\mathcal{E}_j$, then abort.

2. Output input_{m+1} as the desired evaluation $f(x)$.

Figure 3: Protocol Π_{Comp}: Compute phase.

Update Phase:
Public inputs: $\mathcal{E}_1, ..., \mathcal{E}_m$, $\mathsf{crs} = (\{\mathsf{crs}^i_{\mathsf{eq}}, \mathsf{crs}^i_{\mathsf{nizk}}\}_{i=1}^n, \mathsf{crs}_{\mathsf{w}})$, $\{c_{j,i}\}_{j\in[m], i\in[|\mathcal{E}_j|]}$.
Private inputs: $(\mathsf{Sub}_{j,i}, d_{j,i})$, held by party i of $\mathcal{E}_j$.

All parties run the Update protocol of the LDS compiler, as follows.

1. Each time the parties in committee $\mathcal{E}_j$, who are simulating submodule Sub_j, receive a message msg_j from the parties in committee $\mathcal{E}_{j-1}$, who are simulating submodule Sub_{j-1}, they compute the function G that would have been computed by Sub_j upon receiving the message input_j when running the Update protocol. (The parties in committee $\mathcal{E}_1$ start with $\mathsf{msg}_1 \triangleq \perp$).

   The computation of G is done by running an execution of the λ-weakly leakage-resilient MPC protocol using $\mathsf{crs}_{\mathsf{w}}$ to collectively execute the following (randomized) function:

   Input: $(\mathsf{Sub}_{j,i}, d_{j,i}, \mathsf{msg}_j)$, held by party i of $\mathcal{E}_j$, crs, $\{c_{j,i}\}_{i\in[|\mathcal{E}_j|]}$, held by all parties.

   The (randomized) function:

   (a) If any of the msg_j's are inconsistent, or $\mathsf{Sub}_{j,i} \neq \mathsf{Rec}(\mathsf{crs}^{\alpha}_{\mathsf{eq}}, c_{j,i}, d_{j,i})$ for any i, where P_α is the i'th party in committee $\mathcal{E}_j$ (i.e., if any party's share does not agree with the corresponding published commitment), then abort. Otherwise, let $\mathsf{Sub}_j = \bigoplus_{i=1}^{|\mathcal{E}_j|} \mathsf{Sub}_{j,i}$.

   (b) Evaluate $(\mathsf{Sub}'_j, \mathsf{msg}_{j+1}) \leftarrow G(\mathsf{Sub}_j, \mathsf{msg}_j)$. Here, Sub'_j denotes an updated version of the submodule information, and msg_{j+1} denotes the message to be sent to submodule $j+1$ as dictated by Update.

   (c) Secret share the new value $\mathsf{Sub}'_j = \mathsf{Sub}'_{j,1} \oplus \cdots \oplus \mathsf{Sub}'_{j,|\mathcal{E}_j|}$ into $|\mathcal{E}_j|$ shares using the xor secret sharing scheme.

   (d) For each share $\mathsf{Sub}'_{j,i}$ generated in the previous step, compute a new commitment $(c'_{j,i}, d'_{j,i}) \leftarrow \mathsf{Com}(\mathsf{crs}^{\alpha}_{\mathsf{eq}}, \mathsf{Sub}'_{j,i})$, where P_α is the i'th party in committee $\mathcal{E}_j$.

   Output: The outputs are as follows.

   All parties: msg_{j+1}, $\{c'_{j,i}\}_{i\in[|\mathcal{E}_j|]}$
   Party i of $\mathcal{E}_j$: $\mathsf{Sub}'_{j,i}$, $d'_{j,i}$

   At the conclusion of the WLR-MPC execution, each party in $\mathcal{E}_j$ erases all intermediate values generated during the WLR-MPC, keeping only $(\mathsf{Sub}_{j,i}, d_{j,i})$.

2. *All* the parties of $\mathcal{E}_j$ send msg_{j+1} to *all* parties in $\mathcal{E}_{j+1}$.

3. Each party in $\mathcal{E}_j$ sends all new commitments $\{c'_{j,i}\}_{i\in[|\mathcal{E}_j|]}$ to every party. If any disagreeing values are sent by parties in $\mathcal{E}_j$, then abort.

At the conclusion of the update phase, each party erases their initial input together with all intermediate values of the update phase.

Figure 4: Protocol Π_{Update}: Update phase.

as shared common knowledge. We denote this ideal functionality by crs.

Ideal functionalities in Hybrid 2: crs.

Hybrid 3. (CRS simulation) The same as Hybrid 2, except that some of the CRS values are generated using the simulation algorithms. More specifically,

- For the first party in each committee, its crs for the equivocal commitment scheme is generated using the simulator; i.e., for each such party P_i, $(\mathsf{crs}^i_{\mathsf{eq}}, \mathsf{trap}^i) \leftarrow \mathcal{S}^{\mathsf{crs}}_{\mathsf{eq}}(1^k)$.
- For each malicious party P_ℓ, we generate its crs for the NIZK proof of knowledge using the simulator, by computing $(\mathsf{crs}^\ell_{\mathsf{nizk}}, \mathsf{trap}^\ell) \leftarrow \mathcal{S}^{\mathsf{crs}}_{\mathsf{nizk}}(1^k)$.
- The crs for the WLR-MPC protocol is simulated by computing $(\mathsf{crs}_{\mathsf{w}}, \mathsf{trap}_{\mathsf{w}}) \leftarrow \mathcal{S}^{\mathsf{crs}}_{\mathsf{w}}(1^k)$.

The remaining crs values are generated honestly, as before. We denote this new ideal functionality by crsSim.

Ideal functionalities in Hybrid 3: crsSim.

Hybrid 4. (MPC security) The same as Hybrid 3, except that the second MPC in the preprocessing phase (which generates a key pair for the FHE scheme, runs the LDS transformation, etc) is replaced by the corresponding ideal (randomized) functionality F_{Pre}. Note that F_{Pre} takes no inputs.

Overall, this hybrid is the same as the real world, except that the preprocessing phase consists only of the execution of Elect and one-time oracle access to crsSim and F_{Pre}.

Ideal functionalities in Hybrid 4: $\mathsf{crsSim}, F_{\mathsf{Pre}}$.

Hybrid 5. (WLR-MPC security) The same as Hybrid 4, except each underlying weakly leakage-resilient MPC execution in the decryption cascade is replaced with the ideal functionality F_j that accepts inputs from all parties in $\mathcal{E}_j$ and replies with the evaluation of F_j on these inputs (as described in Figure 3). Similarly, each WLR-MPC execution in the update phase is replaced with the ideal functionality G_j that accepts inputs from parties and replies with the evaluation of G_j on these inputs (as described in Figure 4).

The adversary no longer makes leakage queries of the form $\mathsf{Leak}(i, L)$, as he did in all previous hybrids. Instead, leakage queries are of the form $\mathsf{Leak}(L)$, and are made directly to the ideal functionalities $\{F_j\}, \{G_j\}$. The corresponding ideal functionality evaluates the queried function L on the collection of received inputs from parties. As before, leakage time periods span from the beginning of one Update procedure to the end of the next, and the adversary may make no more than λ leakage queries in any time period.

Ideal functionalities in Hybrid 5: $\mathsf{crsSim}, F_{\mathsf{Pre}}, \{F_j\}, \{G_j\}$.

Hybrid 6. (Equivocal commitments) Same as Hybrid 5, except that the ideal functionality F_{Pre} is replaced by a slightly modified functionality F'_{Pre}. Loosely speaking, F'_{Pre} is the same as F_{Pre}, except that for the first party in each committee (which is assumed to be honest), F'_{Pre} generates a *simulated* commitment to the party's secret share.

Explicitly, F'_{Pre} has a trapdoor trap^j, for the first party of each committee $\mathcal{E}_j$, hardwired into it. Just as F_{Pre}, the functionality F'_{Pre} takes no inputs; it samples a key pair for the FHE scheme, evaluates the LDS transformation of the circuit $\mathsf{Dec}_{\mathsf{sk}}(\cdot)$, and generates secret shares $\mathsf{Sub}_{j,i}$ for each of the resulting secret modules. Further, F'_{Pre} honestly generates a commitment

$$(c_{j,i}, d_{j,i}) \leftarrow \mathsf{Com}(1^k, \mathsf{Sub}_{j,i})$$

to $\mathsf{Sub}_{j,i}$ as usual for the secret share of all *but the first* party in each committee. For the *first* party in each committee (which is assumed to be honest), F'_{Pre} generates a *simulated* commitment

$$(\tilde{c}_{j,1}, \tilde{d}^0_{j,1}, \tilde{d}^1_{j,1}) \leftarrow \mathcal{S}^{\mathsf{com}}_{\mathsf{eq}}(\mathsf{crs}^j_{\mathsf{eq}}, \mathsf{trap}^j),$$

and sets $d_{j,1} = \tilde{d}^{\mathsf{Sub}_{j,1}}_{j,1}$.

Ideal functionalities in Hybrid 6: $\mathsf{crsSim}, F'_{\mathsf{Pre}}, \{F_j\}, \{G_j\}$.

Hybrid 7. (Equivocal commitments) Same as Hybrid 6, except that the ideal functionalities $\{G_j\}$ are modified in the same fashion as the step above. Namely, we replace each G_j with a new ideal functionality G'_j with the following differences. G'_j has a trapdoor trap^j, for the first party of each committee $\mathcal{E}_j$, hardwired into it. G'_j accepts the same inputs as G_j, and carries out the same computation as G_j, with the following exception: For the first party in each committee, instead of honestly generating a commitment to the secret share $\mathsf{Sub}_{j,1}$, the functionality G'_j generates a *simulated* commitment $(\tilde{c}_{j,1}, \tilde{d}^0_{j,1}, \tilde{d}^1_{j,1}) \leftarrow \mathcal{S}^{\mathsf{com}}_{\mathsf{eq}}(\mathsf{crs}^j_{\mathsf{eq}}, \mathsf{trap}^j)$, and sets $d_{j,1} = \tilde{d}^{\mathsf{Sub}_{j,1}}_{j,1}$. (Note that the ideal functionalities $\{F_j\}$ in the decryption cascade do not generate new secret shares and thus do not need to be modified in this fashion).

Ideal functionalities in Hybrid 7: $\mathsf{crsSim}, F'_{\mathsf{Pre}}, \{F_j\}, \{G'_j\}$.

Hybrid 8. (Binding of Com) Similar to Hybrid 7, except that all secret shares are eliminated, and committees interact directly with the m modules $(\mathsf{Sub}_1, ..., \mathsf{Sub}_m)$. More specifically, the following changes are made:

- The ideal functionality crsSim is replaced by a slightly modified functionality crsSim', which executes exactly as crsSim, but in addition sends to the adversary all trapdoors for simulated equivocal commitment crs values (for the first party in each committee).
- The ideal functionality F'_{Pre} is replaced by a simple ideal functionality F_{pk} that takes no inputs, generates a key pair $(\mathsf{pk}, \mathsf{sk})$ for the FHE scheme, and publishes pk.
- The sequence of ideal functionalities $\{F_j\}, \{G'_j\}$, as introduced in the previous steps, are replaced by the corresponding (LDS model) interactions with the m modules $(\mathsf{Sub}_1, ..., \mathsf{Sub}_m)$ generated by the LDS compiler:

  $$(\mathsf{Sub}_1, ..., \mathsf{Sub}_m) \leftarrow \mathcal{C}(\mathsf{Dec}_{\mathsf{sk}}(\cdot)).$$

  Namely,

  – The decryption cascade takes place as follows. For each $j = 1, ..., m$, beginning with $\mathcal{E}_1$ and

$\mathsf{input}_1 = \hat{y}$, all parties in committee $\mathcal{E}_j$ send input_j to the corresponding module Sub_j. If all input_j's are consistent, they receive back $\mathsf{input}_{j+1} \leftarrow \mathsf{Sub}_j(\mathsf{input}_j)$. The parties of $\mathcal{E}_j$ then send input_{j+1} to all parties in the next committee $\mathcal{E}_{j+1}$, who (if the received values are consistent) repeat the same process. At the conclusion of the decryption cascade, the parties of the final committee $\mathcal{E}_m$ send the resulting value input_{m+1} (which is supposedly $f(\vec{x})$) to all parties.

– The update procedure is similar to the decryption cascade. The modules execute the LDS update procedure, interacting with each other via the committees $\mathcal{E}_1, ..., \mathcal{E}_m$.

Instead of making leakage queries to the ideal functionalities $\{F_j\}, \{G_j\}$, the adversary now makes queries of the form $\mathsf{Leak}(j, L)$, and receives the evaluation of L on the secret state of the jth module, Sub_j. As before, leakage time periods span from the beginning of one Update procedure to the end of the next, and the adversary may make no more than λ leakage queries in any time period.

Ideal functionalities in Hybrid 8: $\mathsf{crsSim}', F_{\mathsf{pk}}, \{\mathsf{Sub}_j\}$.

Hybrid 9. (LDS security) Same as Hybrid 8, except that all modules Sub_j are removed. Instead, parties interact with an ideal decryption functionality $\mathsf{Dec}_{\mathsf{sk}}$, as described below.

In the preprocessing phase, the parties execute Elect, and are given pk and crs values (where some of the crs values are generated with trapdoors, as described in Hybrid 3). The input phase takes place as usual. In the online phase, for each function f that is queried by the adversary, the parties homomorphically compute the corresponding ciphertext $\hat{y} = \mathsf{Eval}_{\mathsf{pk}}(\hat{x}, f)$. All parties in the first committee, $\mathcal{E}_1$, send $\hat{y}$ to the ideal decryption functionality $\mathsf{Dec}_{\mathsf{sk}}(\cdot)$ with abort. If all received $\hat{y}$'s are consistent, the ideal functionality responds by sending the resulting decryption $\mathsf{Dec}_{\mathsf{sk}}(\hat{y})$ to the adversary, where sk is the decryption key that was generated by F_{pk}. If the adversary allows, $\mathsf{Dec}_{\mathsf{sk}}(\hat{y})$ is also sent to all honest parties; otherwise, the experiment concludes in abort. The update phase no longer takes place. No leakage queries are allowed at any point of the experiment.

Ideal functionalities in Hybrid 9: $\mathsf{crsSim}', F_{\mathsf{pk}}, \mathsf{Dec}_{\mathsf{sk}}$.

Hybrid 10. (Soundness/PoK of NIZK, certifiability of FHE) Differs from Hybrid 9 in the following ways:

- The ideal functionalities $\mathsf{crsSim}', F_{\mathsf{Pre}}$, and the execution of Elect, are removed from the preprocessing phase.
- The input phase no longer takes place.
- The ideal decryption functionality $\mathsf{Dec}_{\mathsf{sk}}$ is replaced by the ideal-world functionality Evaluate, which takes input x_i from each party and evaluates functions f queried by the adversary on the set of all parties' inputs $\vec{x}$, as defined in Section 3.1.
- In addition, the adversary is given as auxiliary input

$$z' := \left(\mathsf{pk}, \{\hat{x}_i, \mathsf{crs}^i_{\mathsf{nizk}}, \pi_i\}_{i \notin M}\right),$$

where $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(1^k)$, and for each honest party P_i, the triple $(\mathsf{crs}^i_{\mathsf{nizk}}, \hat{x}_i, \pi_i)$ is computed using the *real* input x_i of P_i. That is, the values in the triple are computed by $\mathsf{crs}^i_{\mathsf{nizk}} \leftarrow \mathsf{Gen}_{\mathsf{nizk}}(1^k)$; $r_i \leftarrow R$; $\hat{x}_i = \mathsf{Enc}_{\mathsf{pk}}(x_i; r_i)$; $\pi_i \leftarrow \mathsf{P}(\mathsf{crs}^i_{\mathsf{nizk}}, (\hat{x}_i, \mathsf{pk})(x_i, r_i))$.

Overall, Hybrid 10 is the following.

Parties begin by submitting their inputs to the ideal functionality Evaluate. More specifically, each *honest* party P_i submits his input x_i. The adversary is given the corresponding auxiliary input z', computed as a function of the honest parties' inputs $\{x_i\}_{i \notin M}$. Upon receiving z', the adversary submits the inputs of *malicious* parties to Evaluate.

The preprocessing and input phases no longer take place. During the online phase, for each function f that is queried by the adversary, Evaluate responds by sending the adversary the evaluation of f on the set of all submitted inputs $(x_1, ..., x_n)$. If the adversary allows, the evaluation is also sent to all honest parties; otherwise, the experiment concludes in abort.

Note that Hybrid 10 is *nearly* the ideal-world experiment. Indeed, the only difference is that the adversary is given the auxiliary input z'.

Ideal functionalities in Hybrid 10: Evaluate.

Hybrid 11. (Security of FHE, ZK of NIZK) The ideal world: i.e., the adversary *only* receives $f(x_1, ..., x_n)$ for each f selected to be computed. Note that this is the same as Hybrid 10, except that the adversary no longer receives the auxiliary input. (See Section 3.1 for the detailed experiment).

The output of each hybrid experiment consists of the outputs of all parties, where honest parties output in accordance with the dictated protocol, and malicious parties may output any efficiently computable function of the view of the adversary. For every adversary $\mathcal{A}_\ell$ with auxiliary input $z \in \{0,1\}^*$ running in hybrid experiment ℓ with initial inputs $\vec{x}$, we denote the output of the corresponding hybrid ℓ experiment by

$$\mathsf{HYB}_\ell\left(\mathcal{A}_\ell, 1^k, z, \{x_i\}_{i=1}^n\right).$$

It remains to prove that for every $\ell = 0, ..., 10$ and for every adversary $\mathcal{A}_\ell$ running in Hybrid ℓ, there exists an adversary $\mathcal{A}_{\ell+1}$ running in Hybrid $(\ell+1)$ such that

$$\mathsf{HYB}_\ell\left(\mathcal{A}_\ell, 1^k, z, \{x_i\}_{i=1}^n\right) \approx_c \mathsf{HYB}_{\ell+1}\left(\mathcal{A}_{\ell+1}, 1^k, z, \{x_i\}_{i=1}^n\right).$$

Note that once we show this, the theorem will follow, as this will imply that for each adversary $\mathcal{A}$ in the real-world experiment (Hybrid 0), there is an adversary $\mathcal{A}_{11}$ in the ideal-world experiment (Hybrid 11), such that

$$\mathsf{HYB}_0\left(\mathcal{A}, 1^k, z, \{x_i\}_{i=1}^n\right) \approx_c \mathsf{HYB}_{11}\left(\mathcal{A}_{11}, 1^k, z, \{x_i\}_{i=1}^n\right)$$

as desired. We defer these indistinguishability proofs to the full version of this manuscript.

□

6. REFERENCES

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

[ADW09] Joël Alwen, Yevgeniy Dodis, and Daniel Wichs. Leakage-resilient public-key cryptography in the bounded-retrieval model. In *CRYPTO*, pages 36–54, 2009.

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

[AK96] Ross Anderson and Markus Kuhn. Tamper resistance: a cautionary note. In *WOEC'96: Proceedings of the 2nd conference on Proceedings of the Second USENIX Workshop on Electronic Commerce*, pages 1–11, 1996.

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

[BCH11] Nir Bitansky, Ran Canetti, and Shai Halevi. Leakage tolerant interactive protocols. Cryptology ePrint Archive, Report 2011/204, 2011.

[BFM88] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications (extended abstract). In *STOC*, pages 103–112, 1988.

[BG10] Zvika Brakerski and Shafi Goldwasser. Circular and leakage resilient public-key encryption under subgroup indistinguishability - (or: Quadratic residuosity strikes back). In *CRYPTO*, pages 1–20, 2010.

[BGG+11] Elette Boyle, Sanjam Garg, Shafi Goldwasser, Abhishek Jain, Yael Tauman Kalai, and Amit Sahai. Leakage-resilient multiparty computation. Manuscript, 2011.

[BGK11] Elette Boyle, Shafi Goldwasser, and Yael Tauman Kalai. Leakage-resilient coin tossing. In *DISC*, 2011.

[BGV11] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully homomorphic encryption without bootstrapping. ECCC, Report 2011/111, 2011.

[BKKV10] Zvika Brakerski, Yael Tauman Kalai, Jonathan Katz, and Vinod Vaikuntanathan. Overcoming the hole in the bucket: Public-key cryptography resilient to continual memory leakage. In *FOCS*, pages 501–510, 2010.

[BSMP91] Manuel Blum, Alfredo De Santis, Silvio Micali, and Giuseppe Persiano. Noninteractive zero-knowledge. *SIAM J. Comput.*, 20(6):1084–1118, 1991.

[BSW11] Elette Boyle, Gil Segev, and Daniel Wichs. Fully leakage-resilient signatures. In *EUROCRYPT*, 2011.

[BV11] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. In *FOCS*, 2011.

[CIO98] Giovanni Di Crescenzo, Yuval Ishai, and Rafail Ostrovsky. Non-interactive and non-malleable commitment. In *STOC*, pages 141–150, 1998.

[CLOS02] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *STOC*, pages 494–503, 2002.

[DGK+10] Yevgeniy Dodis, Shafi Goldwasser, Yael Tauman Kalai, Chris Peikert, and Vinod Vaikuntanathan. Public-key encryption schemes with auxiliary inputs. In *TCC*, pages 361–381, 2010.

[DHLW10a] Yevgeniy Dodis, Kristiyan Haralambiev, Adriana López-Alt, and Daniel Wichs. Cryptography against continuous memory attacks. In *FOCS*, pages 511–520, 2010.

[DHLW10b] Yevgeniy Dodis, Kristiyan Haralambiev, Adriana López-Alt, and Daniel Wichs. Efficient public-key cryptography in the presence of key leakage. In *ASIACRYPT*, pages 613–631, 2010.

[DHP11] Ivan Damgard, Carmit Hazay, and Arpita Patra. Leakage resilient two-party computation. Cryptology ePrint Archive, Report 2011/256, 2011.

[DKL09] Yevgeniy Dodis, Yael Tauman Kalai, and Shachar Lovett. On cryptography with auxiliary input. In *STOC*, pages 621–630, 2009.

[DLWW11] Yevgeniy Dodis, Allison Lewko, Brent Waters, and Daniel Wichs. Storing secrets on continually leaky devices. In *FOCS*, 2011.

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

[DP10] Yevgeniy Dodis and Krzysztof Pietrzak. Leakage-resilient pseudorandom functions and side-channel attacks on feistel networks. In *CRYPTO*, pages 21–40, 2010.

[DS83] Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM J. Comput.*, 12(4):656–666, 1983.

[Fei99] Uriel Feige. Noncryptographic selection protocols. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, 1999.

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

[FLS90] Uriel Feige, Dror Lapidot, and Adi Shamir. Multiple non-interactive zero knowledge proofs based on a single random string (extended abstract). In *FOCS*, pages 308–317, 1990.

[FRR+10] Sebastian Faust, Tal Rabin, Leonid Reyzin, Eran Tromer, and Vinod Vaikuntanathan. Protecting circuits from leakage: the

computationally-bounded and noisy cases. In *EUROCRYPT*, pages 135–156, 2010.

[FS89] Uriel Feige and Adi Shamir. Zero knowledge proofs of knowledge in two rounds. In *CRYPTO*, pages 526–544, 1989.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, 2009.

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

[GMO01] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In *CHES*, pages 251–261, 2001.

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

[GOS06] Jens Groth, Rafail Ostrovsky, and Amit Sahai. Perfect non-interactive zero knowledge for np. In *EUROCRYPT*, pages 339–358, 2006.

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

[GR12] Shafi Goldwasser and Guy N. Rothblum. How to compute in the presence of leakage. *Electronic Colloquium on Computational Complexity (ECCC)*, 19, 2012.

[HSH+08] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *USENIX Security Symposium*, pages 45–60, 2008.

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

[JV10] Ali Juma and Yevgeniy Vahlis. Protecting cryptographic keys against continual leakage. In *CRYPTO*, pages 41–58, 2010.

[Koc96] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO*, pages 104–113, 1996.

[KP10] Eike Kiltz and Krzysztof Pietrzak. Leakage resilient elgamal encryption. In *ASIACRYPT*, pages 595–612, 2010.

[KV09] Jonathan Katz and Vinod Vaikuntanathan. Signature schemes with bounded leakage resilience. In *ASIACRYPT*, pages 703–720, 2009.

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

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

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

[MTVY11] Tal Malkin, Isamu Teranishi, Yevgeniy Vahlis, and Moti Yung. Signatures resilient to continual leakage on memory and computation. In *EUROCRYPT*, 2011.

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

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of aes. In *CT-RSA*, pages 1–20, 2006.

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

[QS01] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (ema): Measures and counter-measures for smart cards. In *E-smart*, pages 200–210, 2001.

[Yao82] Andrew C. Yao. Theory and applications of trapdoor functions. In *Proc. 23rd FOCS*, pages 80–91, 1982.