Revisiting Covert Multiparty Computation

Geoffroy Couteau

ENS, Paris, France *

Abstract. Is it feasible for parties to securely evaluate a function on their joint inputs, while hiding not only their private input, but even the very fact that they are taking part to the protocol? This intriguing question was given a positive answer in the two-party case at STOC'05, and in the general case at FOCS'07, under the name of *covert multiparty computation* (CMPC). A CMPC protocol allows n players with inputs $(x_1 \cdots x_n)$ to compute a function f with the following guarantees:

- If all the parties are taking part to the protocol, and if the result of the computation is favorable to all the parties, then they get to learn $f(x_1, \cdots, x_n)$ (and nothing more)
- Else, when the result is not favorable to all the parties, or if some player does not participate to the computation, no one gets to learn anything (and in particular, no player can learn whether any of the other parties was indeed participating to the protocol)

While previous works proved the existence of CMPC under standard assumptions, their candidate CMPC protocols were exclusively of theoretical interest. In this work, we revisit the design of CMPC protocols and show that, perhaps surprisingly, this very strong security notion can be achieved essentially for free. More specifically, we show how to build a CMPC protocol out of a standard, state-of-the-art MPC protocol, where both the communication and the computation are the same than the original protocol, up to an additive factor independent of the size of the circuit. Along the way, we prove two variants of the UC theorem which greatly simplify the design and the security analysis of CMPC protocols.

Keywords. Covert multiparty computation, Multiparty computation, Universal composability.

1 Introduction

Suppose that your company has suffered from an attack from a group of hacker; as they have been very careful in the attack, it seems difficult to trace them on your own. However, you suspect that a set of other companies might have been attacked too. A joint computational effort between the companies might allow to trace the hackers, but you are not willing to reveal to concurrent companies that your company was hacked, unless they have been attacked too and can indeed help you tracing the hackers. What can you do?

The above example is a situation where standard secure computation fails to provide a satisfying answer. Indeed, while secure multiparty computation allows players to evaluate any function while hiding their private inputs, it still leaks the information that the players have decided to participate to this secure computation protocol – and in some cases, revealing this information can be undesirable. It might seem infeasible to overcome this issue at first sight: after all, how could players be possibly performing a secure protocol if they did not even agree to execute it? Nevertheless, an elegant solution was suggested at STOC'05 by Ahn *et al.* [vAHL05], who introduced the fascinating notion of *covert two-party computation* (C2PC), later extended to covert multiparty computation (CMPC) by Chandran *et al* at FOCS'07 [CGOS07].[1] The idea is to hide the execution of a multiparty computation protocol inside innocent-looking conversations, so that no one (not even the players themselves) can determine whether the other players are indeed executing the protocol, or just carrying normal conversations. When to start the protocol is made part of its specifications. At the end of the protocol, if all the players participated to the computation and if the result is *favorable* to all the players (in our above example, this would correspond to a result that indeed allows to trace the hackers),[2] they get the output (hence discovering that everyone took part to the computation). In our above example, the companies can try to identify the hackers using a CMPC protocol: if it does not succeed in tracing the hackers, or if some companies have not suffered from an attack (and are therefore not taking part to

* CNRS – UMR 8548 and INRIA – EPI Cascade

[1] To avoid confusion, we emphasize that this notion is not related to the notion of *covert adversaries*, a relaxation of the malicious security model which was introduced in 2007. The similarity of the names is an unfortunate coincidence.

[2] The term *favorable* might be a bit misleading, but it is the one used in previous works. It will be defined formally later on.

the protocol), then no one will know that your company was hacked. Other applications were suggested in [vAHL05, CGOS07]. We point out that in addition to having some applications, covert multiparty computation offers a very interesting theoretical challenge – this statement will be made more precise afterward.

Model for Covert Multiparty Computation. The first requirement to execute a CMPC protocol is to hide the communication between the parties inside ordinary-looking conversations. Think for example of images posted on some social network, out of which anyone can extract what appears to be random bit-strings: for images posted by innocent users, this random bit-string would be nothing more than that, but it could also be that some secret data has been encoded in this string. This is exactly the purpose of steganography, which was formally defined in [HLv02]: steganography allows players to communicate while hiding their secret communication to external observers inside innocent-looking communications. However, it is important to note that steganography together with standard MPC are not sufficient to allow for CMPC. Indeed, even when all the players are interested in executing a protocol, the CMPC model requires that their participation must be revealed to the other players *only if the result is favorable to all the parties*. Therefore, steganography does not give a solution by itself. However, as formally proven in [vAHL05], steganography reduces the task of building a CMPC protocol to the task of building a secure computation protocol for a target function f on some joint input $\boldsymbol{x} = (x_1, \cdots, x_n)$ such that:

- Some of the parties might be *non-participating*, in which case they do only broadcast uniformly random bit-string of the appropriate size at each round (which, in practice, would be a random string extracted from some innocent message they sent).
- There is a public *admissibility function* $g : \{0,1\}^* \mapsto \{0,1\}$ which determines whether the output of the protocol is favorable to all the parties.
- When all the players are participating and $g(\boldsymbol{x}) = 1$, the output of the protocol is set to $f(\boldsymbol{x})$.
- When at least one of the parties is non-participating, or when $g(\boldsymbol{x}) = 0$, the output of the protocol is set to $\perp$. This implies in particular that no one can distinguish between an execution of the protocol with non-favorable inputs, and an execution of the protocol with non-participating players.

The adversarial model considered in [vAHL05, CGOS07] is that of *static, malicious corruption*: the adversary corrupts a subset of the parties before the start of the protocol; the corrupted players might deviate arbitrarily from the specifications of the protocol. As the players cannot explicitely agree on performing the joint computation, the specifications of the protocol must include some synchronisation data, such as, for example, "the protocol starts with the first message sent after 12pm". The parties interested in computing f will start hiding protocol messages in their conversations from that time, while parties that are not interested will just keep on carrying normal conversations.

1.1 State of the Art

In STOC'05, Ahn *et al.* [vAHL05] defined covertness as a collection of security notions (strong internal covertness, strong fairness and final covertness), formally proved that CMPC on steganographic channels can be reduced to CMPC over ordinary (uniform) channels with non-participating players sending uniformly random strings on the channel, and proposed an instantiation secure in the random oracle model under standard assumptions. In FOCS'07, Chandran *et al.* [CGOS07] improved upon the construction of Ahn *et al.* on several aspects. First, they identified some flaws in the protocol of [vAHL05] (in particular, the protocol of Ahn *et al.* does not guarantee the correctness of the output). Then, they proposed a better, simulation-based, unified notion of covert *multiparty* computation, which in particular separates the notion of covertness from the notion of fairness. Eventually, they constructed a generic CMPC protocol in the standard model, assuming a broadcast channel. Later, in STOC'10, Goyal *et al.* [GJ10] gave a constant round CMPC protocol, relying on non-black-box simulation techniques, and proved the impossibility of constructing such a protocol with respect to black-box simulation.

Unfortunately, all of the aforementioned results are purely of theoretical interest (in particular, they heavily rely on expensive Karp reductions to NP-complete problems). While they settle the question of the existence of CMPC, the question of constructing *efficient* protocols satisfying the very strong

security notion of covertness remains wide open. Regarding standard MPC, decades of efforts have been invested into building modular frameworks (such as the UC model [Can01]) and increasingly efficient protocols (see [DPSZ12, DZ13, KOS16] among many others). The state of the art for CMPC is not as satisfying: to our knowledge, the question of building efficient CMPC protocols was only answered for very specific functionalities in the two-party setting, namely string equality and set intersection in [CDSJ16], and a weak form of authentication in [Jar14].

On the Difficulty of CMPC. One of the core reasons why the design of CMPC protocols has proven so complex is that most techniques used in standard MPC break down when one attempts to construct a covert protocol. Indeed, tasks such as opening a commitment, performing a zero-knowledge proof, or verifying whether a ciphertext is valid, do all involve an explicit checking procedure which would immediatly reveal, when it succeeds, that some player is taking part to the protocol. Therefore, designing an efficient CMPC protocol seems inherently harder than building a standard MPC protocol: one must construct a protocol with strictly stronger security properties than standard MPC without using most of the tools that allow for efficient MPC protocols.

Another difficulty regarding covert multiparty computation is that, unlike standard MPC, there is no framework comparable to the UC framework to prove the security of protocols in a composable way, or to design protocols as *reactive* functionalities. As a consequence, CMPC protocols lack the modularity of standard MPC protocols, both in terms of functionality (they are not stateful, reactive functionalities that perform the desired computation in an adaptive way), and of security (it is unclear whether one can rely on previously proven subprotocols to assess the security of some protocol, or even whether one can import existing results from standard MPC).

1.2 Our Contributions

In this paper, we revisit the fascinating notion of covert multiparty computation and address all of the previously outlined issues. More precisely, our contributions in this paper are twofold.

- We construct an analogous of the UC model tailored to covert multiparty computation, in order to simplify the design of CMPC schemes and to extend it to reactive, stateful functionalities. We derive two composition theorems, which allow us in particular to later easily reuse various primitives from standard MPC.
- We propose a new generic CMPC protocol and prove its security in the model of Chandran et al. Our CMPC protocol securely evaluates an arithmetic circuit on the private inputs of n players over a large order field, and is based on the recent protocol of [KOS16], called MASCOT, which is to our knowledge the most efficient protocol for standard MPC on arithmetic circuits. We show that the complexity of our protocol is the same as that of MASCOT up to a constant additive term, suggesting that, perhaps surprisingly, the very strong security notion of covertness can be ensured at essentially no cost.

Let $C(c, d)$ be the communication complexity (in bits) of the MASCOT protocol when evaluating a circuit with c input gates and d multiplication gates. As shown in [KOS16], for a protocol between n players with security parameter κ, $C(c, d) = O(\kappa^2(nc + n^2 d))$. Our protocol communicates $C(c, d) + O(\kappa n^3)$ bits. The security of our protocol can be based on various well-studied assumptions, such as the decisional Diffie-Hellman (DDH) assumption.

1.3 Our Method

Uniform UC Model. We start by designing a variant of the UC model, called the uniform UC model. This captures informally the protocols in which only random-looking strings are exchanged. We show a natural composition theorem for the uniform UC model. Then, we prove a second composition theorem: if a protocol is indeed covert (as defined by Chandra *et al.*) when given access to some ideal primitives, it remains covert if the primitives are replaced by uniform UC implementations of the primitives. This allows us to greatly simplify the design of our CMPC protocol, and to reuse existing protocols without having to prove the full security from scratch.

Functionality $\mathscr{F}_{\mathsf{CMPC}}(f, g)$

$\mathscr{F}_{\mathsf{CMPC}}$ runs with an adversary Adv and a set of n parties $(P_i)_{i\leq n}$, some of whome might not be actually taking part to the protocol, and knows the description of two n-entry functions f and g (the latter is the admissibility function). Before the protocol, Adv can *corrupt* a subset of the parties; the remaining parties are denoted *honest parties*. Honest non-participating parties are assumed to automatically send $\perp$ to the trusted party. The trusted party ignores all messages that are not correctly formatted.

Input. The trusted party waits until it received an input x_i from each party P_i.
Send output to adversary. Let $\boldsymbol{x} = (x_i)_{i\leq n}$. If any entry of $\boldsymbol{x}$ is $\perp$, $\mathscr{F}_{\mathsf{CMPC}}$ sets the output y to $\perp$. Else, it checks that $g(\boldsymbol{x}) = 1$, and sets y to $\perp$ if this check fails. If the check passes, it sets $y \leftarrow f(\boldsymbol{x})$. The trusted party sends y to Adv.
Send output to honest parties. The trusted party waits until it receives from Adv a list of honest parties that should get the output. It sends y to those parties, and $\perp$ to the remaining honest parties.

The honest participating parties output whatever output they received from the functionality. Non-participating and corrupted parties output $\perp$ by convention. The adversary outputs its entire view.

Fig. 1: Ideal Functionality for Covert Multiparty Computation

CMPC with Preprocessing. We follow the standard paradigm for constructing efficient MPC protocols: in a *preprocessing phase*, some material is computed by the players, independently of their inputs. This material will allows the parties to authenticate inputs and to perform operations on authenticated values in the *online phase*. We construct our protocol by closely following the design of [KOS16], but we carefully remove any explicit checking procedure in the preprocessing functionality it relies on. We instead allow the adversary to introduce errors in the functionality.

Implicit Checking. The core difficulty in the construction is to remove any explicit checking procedure while still preventing malicious players to deviate. In [CGOS07, GJ10], this was done using a primitive called *zero-knowledge proofs to garbled circuit*, whose instantiation involves general NP reductions (and is therefore inefficient). In contrast, our output protocol only relies on a constant number of calls to an *implicit checking functionality*. Intuitively, this functionality allows the players to implicitely check the validity of some statement, by returning random shares of the result of the check to the players. We design an efficient implementation of this implicit checking functionality using a primitive called homomorphic smooth projective hash function. Smooth projective hash functions were introduced in [CS02].

1.4 Notations

For a set S, $x \xleftarrow{\$} S$ denotes the affectation to x of a variable sampled uniformly at random from S. We say that two probability distributions $\mathscr{D}, \mathscr{D}'$ are computationally indistinguishable, and write $\mathscr{D} \overset{\mathsf{c}}{\equiv} \mathscr{D}'$, if no probabilistic polynomial time adversary, given a sample of one of the two distributions picked at random, can guess the correct distribution with probability significantly greater than $1/2$.

2 Universal Composability Theorems for Covert Multiparty Computation

2.1 The Covert Model

The notion of covertness was first introduced in [vAHL05] as a collection of desirable properties, such as strong internal covertness, strong fairness and final covertness. Later in [CGOS07], Chandran et al. proposed a more unified, simulation-based definition. Let us start by recalling the model of [CGOS07].

Ideal Model. We consider n parties, each party P_i holding an input x_i. Let $\boldsymbol{x}$ be the vector $(x_1, \cdots, x_n)$. All the participating parties send their input to the functionality, while the non-participating parties are assumed to send $\perp$. Then, if any of the parties had input a $\perp$, the functionality sets $\perp$ to be the output of the protocol. Else, let $g : (\{0,1\}^*)^n \mapsto \{0,1\}$ be the function which determines whether on input $\boldsymbol{x}$, the output is *favorable* ($g(\boldsymbol{x}) = 1$) or *non-favorable* ($g(\boldsymbol{x}) = 0$). In the latter case, the functionality sets $\perp$ to be the output of the protocol. In the former case, the functionality computes

the output $f(\boldsymbol{x})$. The output is then sent to any subset of players, chosen by the adversary. The functionality is given Figure 1. For an adversary Adv, an execution of $\mathscr{F}_{\mathsf{CMPC}}(f,g)$ with participation data $\boldsymbol{p}$ (indicating which players are taking part to the protocol) and input $\boldsymbol{x}$ (where the input of non-participating parties is $\perp$) is defined as the output of the parties together with the output of the adversary. It is denoted $\mathsf{IDEAL}_{f,g,\mathsf{Adv}}(\boldsymbol{p},\boldsymbol{x})$.

Real Model. Honest participating parties follow the specifications of the protocol. Honest non-participating parties are assumed to send uniformly random messages. We consider *static malicious corruption* of players, in which the corrupted players are chosen once-for-all by the adversary before the start of the protocol, and are entirely controlled by the adversary. Honest participating parties compute their output as specified, non-participating and corrupted parties output $\perp$ by convention, and the adversary outputs its entire view of the execution of the protocol. For an adversary Adv, an execution of a protocol $\Pi(f,g)$ in the real model with participation data $\boldsymbol{p}$ and input $\boldsymbol{x}$ is defined as the output of the parties together with the output of the adversary. It is denoted $\mathsf{REAL}_{\Pi(f,g),\mathsf{Adv}}(\boldsymbol{p},\boldsymbol{x})$.

Definition 1 (Covert security [CGOS07]). *A protocol $\Pi(f,g)$ securely implements $\mathscr{F}_{\mathsf{CMPC}}(f,g)$ if for every probabilistic polynomial-time adversary Adv statically corrupting up to $n-1$ players in the real model, there is an expected polynomial-time adversary S corrupting at most $n-1$ players in the ideal model, such that for any $(\boldsymbol{p},\boldsymbol{x}) \in \{0,1\}^n \times (\{0,1\}^*)^n$,*

$$\{\mathsf{IDEAL}_{f,g,S}(\boldsymbol{p},\boldsymbol{x})\} \stackrel{\mathrm{c}}{\equiv} \{\mathsf{REAL}_{\Pi(f,g),\mathsf{Adv}}(\boldsymbol{p},\boldsymbol{x})\}$$

On the Security Model for CMPC. One can remark that this way of defining security is analogeous to the so-called stand-alone model for standard MPC, which allows to argue the security of a single run of a protocol, but does not tell anything as soon as this protocol is ran in a concurrent setting and does not allow for the design of reactive functionalities. In standard MPC, this issue was solved by introducing a stronger model for multiparty computation: the UC model. The core feature of this model is that if a large-scale protocol is proven secure given access to an ideal functionality F, then it remains secure if F is replaced by a real protocol that securely implements F in the UC model. The UC model allows to greatly simplify security proofs, by studying the security of smaller functionalities in order to argue the security of a more complex functionality.

However, no analogeous to the UC model is known for covert computation. The UC model has proven extremely useful and fruitful for standard MPC. In this paper, we propose a variant of the UC framework to import the versatility of the UC model in CMPC. We will prove two composition theorems, which will be useful for the design of a CMPC protocol. Note, however, that although we will strongly rely on these composition theorems in the design of an efficient CMPC protocol, the protocol itself will still be eventually proven secure in the model of [CGOS07] (which is a stand-alone type of security notion). Before we describe our method, we introduce preliminaries on the universal composability framework.

2.2 The Universal Composability Framework

The universal composability framework (UC) has been introduced by Canetti in [Can01]. It defines protocols by the mean of systems of interactive Turing machines. The expected behavior of the protocol is captured by an ideal functionality F. This functionality is a very simple interactive machine, which is connected to a set of dummy parties, some of whom might be corrupted by an ideal adversary $\mathscr{S}im$, through perfectly secure authenticated channels. In the real execution of a protocol π, probabilistic polynomial time players, some of whom might be corrupted by a real adversary Adv, interact with each other through some channels. The *environment* refers to an interactive machine Z that oversees the execution of the protocol in one of the two worlds (the ideal world with the functionality F, or the real world with the protocol π). We refer to [Can01], or alternatively to [CDN15], for the definitions of the real world ensembles $\mathsf{EXEC}_{\pi,\mathsf{Adv},Z}$ and the ideal world ensemble $\mathsf{EXEC}_{F,\mathscr{S}im,Z}$. The task of *securely emulating* a functionality F for a protocol π is defined as follows:

Definition 2. *A protocol π securely implements an ideal functionality F in the* UC *model if for any real adversary* Adv*, there is an ideal adversary $\mathscr{S}im$ (called the simulator) such that for any environment Z, $\mathsf{EXEC}_{\pi,\mathsf{Adv},Z} \stackrel{c}{\equiv} \mathsf{EXEC}_{F,\mathscr{S}im,Z}$.*

One of the core features of the UC model is its *composition theorem*. The latters states that any protocol secure in the UC model remains secure under general composition, a setting in which an arbitrary number of protocols (which can be other instances of the same protocol as well as other protocols) are executed concurrently. More formally, the UC model defines the composition $\diamond$ of interactive machines, to prove the following theorem:

Theorem 3 (UC Theorem - Informal). *Let I_F, I_G be two ideal functionalities. Let π_F be a protocol, so that $\pi_F \diamond I_G$ securely implements I_F in the* UC *model (that is, π_F emulates I_F when given access to I_G). Let π_G be a protocol that securely implements I_G in the* UC *model. Then $\pi_F \diamond \pi_G$ securely implements I_F in the* UC *model.*

Our purpose in this section will be to define a restriction of the UC model, and to derive two new composition theorems, tailored to our goal of covert multiparty computation (which involves the participation data of the players, and cannot therefore be captured by the UC framework). However, our goal is ultimately to construct an efficient CMPC protocol. The fully detailed presentation of our framework and our composition theorems involves a quite heavy formalism. To avoid drowning the reader into unnecessary details, we provide a more informal and more reader-friendly introduction to our framework in this section. The interested reader will find the full construction of our framework, and the formal proof of security of our composition theorems, in Appendix A.

2.3 The Uniform UC Framework

At an informal level, it is safe to use a cryptographic primitive inside a covert protocol if and only if only uniformly random-looking messages (from the view of the players themselves) are exchanged in the implementation of the primitive – otherwise, any non-random-looking message would identify its sender as an active participant to the protocol. This intuition was implicitly acknowledged in every previous works on CMPC. Indeed, all known constructions of CMPC protocols rely on standard cryptographic primitives (e.g., commitment schemes, oblivious transfers, garbled circuits), that are enhanced with this "random-lookingness" feature. In previous works, primitives with this feature were denoted *covert* (e.g., covert commitment, covert oblivious transfer). However, although clearly related to the covert security model, these "covert primitives" are not primitives secure in the covert security model. In this section, we aim at formalizing this intuitive random-lookingness notion; to avoid any possible confusion with the covert security model, we choose to call it *uniformity*.

Let us proceed with the definition of our uniformity notion. We say that a simulator $\mathscr{S}im$ of a protocol π is a *uniform simulator* if the transcript of its interaction with any adversary Adv (controlled by the environment Z) who corrupts a strict subset of the parties contains only uniformly random values. Intuitively, a protocol will satisfy the uniformity property if it can be simulated by a uniform simulator. In other words,

Definition 4 (Informal). *A protocol π securely implements an ideal functionality F in the uniform* UC *model if for any real adversary* Adv*, there is a* uniform *simulator $\mathscr{S}im$ such that for any environment Z, $\mathsf{EXEC}_{\pi,\mathsf{Adv},Z} \stackrel{c}{\equiv} \mathsf{EXEC}_{F,\mathscr{S}im,Z}$.*

The uniform UC model is a simple restriction of the standard UC model, and does not involve at all the participation data of the players. A protocol secure in the uniform UC model is in particular secure in the standard UC model. More interestingly, we prove in Appendix A that uniformity is preserved under general composition:

Theorem 5 (Uniform UC Theorem - Informal). *Let I_F, I_G be two ideal functionalities. Let π_F be a protocol, so that $\pi_F \diamond I_G$ securely implements I_F in the uniform* UC *model. Let π_G be a protocol that securely implements I_G in the uniform* UC *model. Then $\pi_F \diamond \pi_G$ securely implements I_F in the uniform* UC *model.*

2.4 A Composition Theorem for CMPC

We now explain how this restriction of the standard UC model will help us in the design of covert multiparty computation protocols. In Appendix A, we restate the specifications of the covert model of [CGOS07], using the formalism of [CDN15]. We extend the composition operator $\diamond$, so as to be able to compose covert protocols (which involve covert players instead of standard players) with standard protocols and functionalities. Eventually, we will prove the following composition theorem:

Theorem 6 (Covert UC Theorem - Informal). *Let I be an ideal functionality, and let (f, g) be two functions. Let π be a protocol, so that $\pi \diamond I$ securely implements $\mathcal{F}_{\mathsf{CMPC}}(f, g)$ in the covert model of Chandran et al. Let π' be a protocol that securely implements I in the uniform UC model. Then $\pi \diamond \pi'$ securely implements $\mathcal{F}_{\mathsf{CMPC}}(f, g)$ in the covert model of Chandran et al.*

Now, to prove the security of a CMPC protocol, we can proceed as follows: we prove its security in a hybrid world, in which it is given access to ideal functionalities for various necessary tasks. Then, we implement these ideal functionalities assuming ideal access to standard primitives, in the uniform UC model. We then describe implementations of the primitives in the uniform UC model. The uniform UC theorem allows us to argue the security (in the uniform UC model) of our implementation of the ideal functionalities. We conclude using the covert UC theorem to argue the security of our full protocol in the covert model.

3 Standard MPC Protocols

Our purpose will be to construct an efficient CMPC protocol, out of a standard MPC protocol. Therefore, in order to detail our construction, it is necessary to first recall some preliminaries on how MPC protocols are built. Here, we will focus on a specific paradigm to design efficient MPC protocols, generally called the SPDZ paradigm, which was the one used in a large number of recent papers (see [DPSZ12, DZ13, KOS16] among many others). It allows to UC securely evaluate an arithmetic circuit between n players, while tolerating static, malicious corruption of any strict subset of the players. More precisely, our goal will be to construct a CMPC protocol out of the recent MASCOT protocol [KOS16], which is currently (to our knowledge) the most efficient MPC protocol is this setting.

The SPDZ paradigm assumes the *preprocessing model*, where most of the burden of the computation is confined to a preprocessing phase which is independent of the input of the parties and the function to be computed. In this preprocessing phase, the parties interact to generate some material that will be used during the online phase. In the online phase, the parties receive their input to the computation.This phase computes an arithmetic circuit on the inputs of the players. It is information-theoretically secure and relies on the material generated during the preprocessing phase.

Notations. Let p be a prime of the form $2^a + b$, where a, b are polynomial in κ. Let $\mathbb{F}$ be the field of prime order p; note that the distribution of random elements x of $\mathbb{F}$, represented as $x = \sum_i x_i 2^i \mod p$ where the x_i are bits, is statistically indistinguishable from the uniform distribution over $\{0, 1\}^a$. We use bold letters to denote vectors.

3.1 Ideal MPC Functionalities

The expected behavior of an MPC protocol is defined using an *arithmetic black box* (ABB), which is a reactive functionality on which the player can load inputs into variables, add or multiply the content of two variables, and output the content of a variable to the players. The ideal functionality is given Figure 2; it is taken verbatim from [KOS16].

The protocol is constructed assuming access to an ideal preprocessing functionality. For the sake of completeness, we recall the ideal preprocessing functionality of the MASCOT protocol Figure 3. The functionality allows to authenticate, compute linear combinations of, and open additively shared values, as well as to compute multiplication triples.

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

Initialize($\mathbb{F}$): On input $(\mathsf{init}, \mathbb{F})$ from all the parties, store $\mathbb{F}$.

Input: On input $(\mathsf{input}, P_i, \mathsf{id}, x)$ from P_i and $(\mathsf{input}, P_i, \mathsf{id})$ from all other parties, with id a fresh identifier and $x \in \mathbb{F}$, store (id, x).

Add: On command $(\mathsf{add}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$ from all parties (where $\mathsf{id}_1, \mathsf{id}_2$ are present in memory), retrieve $(\mathsf{id}_1, x), (\mathsf{id}_2, y)$ and store $(\mathsf{id}_3, x + y)$.

Mult: On command $(\mathsf{mult}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$ from all parties (where $\mathsf{id}_1, \mathsf{id}_2$ are present in memory), retrieve $(\mathsf{id}_1, x), (\mathsf{id}_2, y)$ and store $(\mathsf{id}_3, x \cdot y)$.

Output: On input $(\mathsf{output}, \mathsf{id})$ from all honest parties (where id is present in memory), retrieve (id, y) and output it to the adversary. Wait for an input from the adversary; if this is $\mathsf{deliver}$ then output y to all parties, otherwise output $\perp$.

Fig. 2: Ideal Functionality for MPC Protocols

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

The functionality maintains a dictionary, Val, to keep track of the authenticated values. Entries of Val lie in the (fixed) finite field $\mathbb{F}$. $\mathcal{F}_{\mathsf{prep}}$ also maintains the sets open and Cheat to record all openings and those where the adversary tried to cheat. The functionality is separated into two functionalities, $\mathcal{F}_{\mathsf{auth}}$ and $\mathcal{F}_{\mathsf{triple}}$, as the latter is implemented assuming access to the ideal functionality $\mathcal{F}_{\mathsf{auth}}$.

Functionality $\mathcal{F}_{\mathsf{auth}}$:

Input: On input $(\mathsf{input}, \mathsf{id}_1, \cdots, \mathsf{id}_m, x_1, \cdots, x_m)$ from P_j and $(\mathsf{input}, \mathsf{id}_1, \cdots, \mathsf{id}_m, P_j)$ from all other parties, where $(x_1, \cdots, x_m) \in \mathbb{F}^m$, sets $\mathsf{Val}[\mathsf{id}_i] \leftarrow x_i$ for $i = 1$ to m.

Linear Combination: On command $(\mathsf{lin\text{-}comb}, \overline{\mathsf{id}}, \mathsf{id}_1, \cdots, \mathsf{id}_t, c_1, \cdots, c_t, c)$ from all parties, where $\mathsf{id}_1, \cdots, \mathsf{id}_t) \in \mathsf{Val.keys}()$ and $(c_1, \cdots, c_t, c) \in \mathbb{F}^{t+1}$, set $\mathsf{Val}[\overline{\mathsf{id}}] \leftarrow \sum_{i=1}^{t} c_i \cdot \mathsf{Val}[\mathsf{id}_i] + c$.

Open: On command $(\mathsf{open}, \mathsf{id})$ from all parties, where $\mathsf{id} \in \mathsf{Val.keys}()$, send $\mathsf{Val}[\mathsf{id}]$, wait for x from the adversary, and output x to all the parties.

Check: On command $(\mathsf{check}, x_1, \cdots, x_t, \mathsf{id}_1, \cdots, \mathsf{id}_t)$ from every party, wait for an input from the adversary. If this input is $\mathsf{deliver}$, and if $\mathsf{Val}[\mathsf{id}_i] = x_i$ for $i = 1$ to t, output OK to all parties, otherwise output $\perp$ and terminate.

Abort: On command $\perp$ from the adversary, send $\perp$ to all parties and terminate.

Functionality $\mathcal{F}_{\mathsf{triple}}$:

Input Tuple: On input $(\mathsf{input\text{-}tuple}, P_j, \mathsf{id})$ from all parties, sample $\mathsf{Val}[\mathsf{id}] \xleftarrow{\$} \mathbb{F}$ and output it to P_j.

Triple: On command $(\mathsf{triple}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$ from all parties, sample two random values $(x, y) \xleftarrow{\$} \mathbb{F}^2$ and sets $(\mathsf{Val}[\mathsf{id}_1], \mathsf{Val}[\mathsf{id}_2], \mathsf{Val}[\mathsf{id}_3]) \leftarrow (x, y, x \cdot y)$.

Fig. 3: Functionality for Performing Computation on Additively Shared Values

3.2 Arithmetic on Authenticated Shares

The input function is constructed by *sharing* and *authenticating with a MAC* the inputs of the players. The sharing scheme and the MAC are linear; as a consequence, linear combinations can be directly and locally evaluated on authenticated shares (which correspond to loaded inputs). Unlike the linear combination, evaluating products requires interactions. It uses some material created during the pre-processing phase and relies again on the linear property of the authenticated sharing scheme. When calling an input or a product command, all the messages exchanged by the players consist in *partial openings* of authenticated shares. A partial opening of a share reveals the value it contains, but does not allow to verify the authenticity of the share. After a number of values have been partially opened, a checking procedure is executed. This procedure uses the MAC value (without revealing it) to verify that the partially opened shares are indeed authentic. This is sufficient to ensure that all the players behaved honestly. Once the checking protocol succeeds, the players partially open the output, and call the checking procedure again to verify that it is valid.

We will require an all-or-nothing secret sharing scheme over a field $\mathbb{F}$, whose additive law is denoted $+$: for a value $x \in \mathbb{F}$, we denote by $\langle x \rangle$ a secret sharing of x over a field $\mathbb{F}$, which means that each player P_j gets a value $s_j(x) \in \mathbb{F}$ subject to $\sum_{j \in [n]} s_j(x) = x$ (when x must be unknown to the players,

the values $s_j(x)$ will be taken uniformly at random so that their sum is x). To authenticate shares, the players will first generate a share of a uniformly random MAC $\Delta \in \mathbb{F}$ (each player generates its share of Δ at random over $\mathbb{F}$). An *authenticated share* of $x \in \mathbb{F}$, denoted $[\![x]\!]$, is a pair $(\langle x \rangle, \langle \Delta \cdot x \rangle)$. We denote by $\delta_j(x)$ the share of the player P_j of $\Delta \cdot x$. Note that standard arithmetic operations can easily be performed over authenticated shares:

- Given $[\![x]\!] = (\langle x \rangle, \langle \Delta \cdot x \rangle)$ and $[\![y]\!] = (\langle y \rangle, \langle \Delta \cdot y \rangle)$, $[\![x + y]\!]$ is computed as $(\langle x \rangle + \langle y \rangle, \langle \Delta \cdot x \rangle + \langle \Delta \cdot y \rangle)$, where $\langle x \rangle + \langle y \rangle$ indicates that each player locally compute the sum of his shares of x and y.
- Given $[\![x]\!]$ and a public value λ, $[\![\lambda x]\!]$ is computed as $(\lambda \cdot \langle x \rangle, \lambda \cdot \langle \Delta \cdot x \rangle)$, where $\lambda \cdot \langle x \rangle$ indicates that each player locally multiplies his share of x by λ.
- As the players have shares of the MAC Δ, they already have an authenticated share of 1; therefore addition by a constant is simply multiplication of the authenticated share of 1 by this constant, followed by an addition.

We will say that the players *partially open* an authenticated share $[\![x]\!] = (\langle x \rangle, \langle \Delta \cdot x \rangle)$ when they open the first part $\langle x \rangle$ only. The second part will be used in the checking procedure to verify that the partial openings are correct. We will call *multiplication triple* a triple of the form $([\![a]\!], [\![b]\!], [\![a \cdot b]\!])$.

Correlated Oblivious Product Evaluation with Errors. The design of the MASCOT protocol crucially relies on a new primitive called *correlated oblivious product evaluation with errors* (COPEe). This protocol is an arithmetic generalization of oblivious transfer extensions, which were introduced by Ishai *et al.* [IKNP03]. It involves two players, P_1 with input $(x_1, \cdots, x_t) \in \mathbb{F}^t$, and P_2 with input $\Delta \in \mathbb{F}$, called the correlation. The protocol outputs shares over $\mathbb{F}$ of Δx. The correlation Δ cannot change during successive executions of the COPEe protocol: in the *initialization* phase, Δ is stored, and in the *extend* phase, only P_1 provides an input. The functionality includes errors in the sense that the sender is allowed to behave maliciously, in a way formally defined in the functionality. The COPEe protocol is used by the players to authenticate additive shares of a value x, using a MAC Δ.

4 An Efficient Covert Multiparty Computation Protocol

Equipped with the covert UC model, we are now ready to describe our construction of an efficient generic CMPC protocol. Our protocol assumes broadcast channels and a common reference string. The latter is unavoidable if we want to use our model: several impossibility results strongly narrow the range of protocols that can UC implement a functionality in the plain model [Can01, CKL03, DDM+06], or in a variety of other alternative models [KL11]. As a uniform UC implementation of a functionality is in particular a UC implementation of this functionality, these impossibility results apply to our setting. Our basis will be the MASCOT protocol of [KOS16].

4.1 Covert Arithmetic Black Box

The first step toward constructing a generic reactive CMPC protocol is to design the ideal functionality we are going to implement. We will do so by modifying the ideal ABB functionality (Figure 2) so as to deal with both the participation data of the players and the admissibility function. The functionality is represented Figure 4.

Following the standard method for building MPC protocols, we would like to design the protocol in an hybrid world, in which it is given access to the ideal preprocessing functionality $\mathscr{F}_{\mathsf{prep}}$ (Figure 3). However, in our setting, this approach is inherently flawed. Recall that our covert UC model allows us to argue the security of our protocol in the covert model when given access to *uniform UC* implementations of the ideal functionalities it relies on. The uniform UC security notion is a very strong security notion – so strong, in fact, that many standard ideal functionalities cannot be possibly implemented in this model. This is the case of $\mathscr{F}_{\mathsf{prep}}$: it involves in particular the check command, which cannot be simulated by a simulator that sends only uniformly random messages. There are other issues: for example, the open command cannot be simulated by a uniform simulator when called on a variable whose content is known to a player (it would allow that player to "check" the value of this variable). We will tackle

Functionality $\mathscr{F}_{\mathsf{CABB}}$

$\mathscr{F}_{\mathsf{CABB}}$ runs with an adversary Adv and a set of n parties $(P_i)_{i\leq n}$, some of whome might not be actually taking part to the protocol. Before the protocol, Adv corrupt a subset of the parties. The remaining parties are denoted *honest parties*. Honest non-participating parties are assumed to automatically send $\perp$ to the trusted party. The trusted party ignores all messages that are not correctly formatted (in particular, as non-participating parties send $\perp$ automatically, the commands $\mathsf{input}, \mathsf{add}, \mathsf{mult}$ are never triggered if there is a non-participating party).

Initialize($\mathbb{F}$): On input $(\mathsf{init}, \mathbb{F})$ or $\perp$ from all the parties, label all honest players with input $\perp$ as *non-participating*, and the remaining players (honest or corrupted) as *participating*. If there is no non-participating party, store $\mathbb{F}$.

Input: On input $(\mathsf{input}, P_i, \mathsf{id}, x)$ from P_i and $(\mathsf{input}, P_i, \mathsf{id})$ from all other parties, with id a fresh identifier and $x \in \mathbb{F}$, store (id, x).

Add: On command $(\mathsf{add}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$ from all parties (where $\mathsf{id}_1, \mathsf{id}_2$ are present in memory), retrieve $(\mathsf{id}_1, x), (\mathsf{id}_2, y)$ and store $(\mathsf{id}_3, x + y)$.

Mult: On command $(\mathsf{mult}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$ from all parties (where $\mathsf{id}_1, \mathsf{id}_2$ are present in memory), retrieve $(\mathsf{id}_1, x), (\mathsf{id}_2, y)$ and store $(\mathsf{id}_3, x \cdot y)$.

Output: On input $(\mathsf{output}, \mathsf{id}_f, \mathsf{id}_g)$ from all *participating* parties (where $(\mathsf{id}_f, \mathsf{id}_g)$ are present in memory), if there is at least one non-participating party, output $\perp$ to all the players. Else, retrieve (id_g, x). If $x = 0$, output $\perp$ to all the parties. If $x = 1$, retrieve (id_f, y) and output it to the adversary. Wait for an input from the adversary. If this is $(\mathsf{deliver}, L)$, where L is a subset of the honest parties, output y to all parties in L and $\perp$ to the remaining parties, and terminate. Otherwise, output $\perp$ and terminate.

Fig. 4: Covert Ideal Functionality for CMPC Protocols

these problems by designing a variant of the preprocessing functionality, called *implicit preprocessing functionality*, that does never perform any explicit check. Instead, the adversary is allowed to introduce errors during the computation; these errors will be detected during the output phase.

4.2 Implicit Preprocessing Functionality

The preprocessing functionality allows to authenticate and perform operations on shares. In this section, we outline our modifications to the preprocessing functionality of [KOS16]. Our purpose is to remove every explicit check of the functionality, which would prevent us from implementing it in the uniform UC model. Instead of using a checking procedure as in [KOS16], the functionality allows the adversary to introduce errors at any step of the computation. In addition, it produces some material that the players will later use in a separate *implicit checking procedure* which will perform all the necessary checks. This procedure is separated from $\mathscr{F}_{\mathsf{i\text{-}prep}}$ as its design will strongly depend on the design of the output protocol.

To deal with the open command, we need to distinguish between identifiers that can point to a random value (from the view of all the players) from the identifies pointing to sensible values. Therefore, we add a command rand-input to the functionality, that allows the players to input random values, and we let the functionality maintain a label on identifiers that point to a random value. We then explicitly forbid the functionality to open non-random values.

Eventually, we must deal with the generation of multiplication triples, which are necessary for the multiplication command. The implementation of this step in [KOS16] involves explicit checking procedure as well as a sacrificing procedure, both of which cannot be allowed in our setting. Therefore, we consider instead a simplified multiplication triple procedure, which generates *unauthenticated* shares of a multiplication triple (a, b, ab). We will let the players implicitely authenticate these shares, and perform an implicit sacrificing procedure.

Handling Non-Participating Players. There is one more technical requirement that we have to handle. Suppose that some functionality sends an output to all the players when all the parties call a specific command, and does nothing otherwise. Then this functionality would immediatly allow to detect non-participating players: indeed, players calling the functionality would never get any output, as soon as a non-participating player sends a random (hence not correctly formatted) input to the functionality.

Functionality $\mathscr{F}_{\mathsf{i\text{-}prep}}$

The functionality maintains a dictionary, Val, to keep track of the authenticated values. Entries of Val lie in the (fixed) finite field $\mathbb{F}$. $\mathscr{F}_{\mathsf{i\text{-}prep}}$ maintains the sets Open, Cheat, and Label to record all openings and those where the adversary tried to cheat, and to label random inputs.

Functionality $\mathscr{F}_{\mathsf{i\text{-}auth}}$:

Initialize: On input $(\mathsf{init}, \Delta_i)$ from each party P_i, sets the MAC $\Delta \leftarrow \sum_i \Delta_i$. Let $\boldsymbol{\Delta} \in \{0,1\}^{\log p}$ be the bit decomposition of Δ.

Input: On input $(\mathsf{input}, \mathsf{id}_1, \cdots, \mathsf{id}_m, x_1, \cdots, x_m)$ from P_j and $(\mathsf{input}, \mathsf{id}_1, \cdots, \mathsf{id}_m, P_j)$ from all other parties, where $(x_1, \cdots, x_m) \in \mathbb{F}^m$, sets $\mathsf{Val}[\mathsf{id}_i] \leftarrow x_i$ for $i = 1$ to m. Wait for m inputs from the adversary. For each index i such that the input of the adversary is the token random, pick $y_i \xleftarrow{\$} \mathbb{F}$ and output uniformly random shares of (x_i, y_i) to all the players; for the remaining indices, output uniformly random shares of $(x_i, \Delta \cdot x_i)$ to all the parties.

Random Input: On input $(\mathsf{rand\text{-}input}, \mathsf{id})$ from all parties, pick $x \xleftarrow{\$} \mathbb{F}$, set $\mathsf{Val}[\mathsf{id}_i] \leftarrow x$, and sets $\mathsf{Label}[\mathsf{id}] \leftarrow \mathsf{random}$. Wait for an input from the adversary. If this input is the token random, pick $y \xleftarrow{\$} \mathbb{F}$ and output uniformly random shares of (x, y) to all the players, otherwise output uniformly random shares of $(x, \Delta \cdot x)$ to all the players.

Linear Combination: On command $(\mathsf{lin\text{-}comb}, \bar{\mathsf{id}}, \mathsf{id}_1, \cdots, \mathsf{id}_t, c_1, \cdots, c_t, c)$ from the honest parties, where $\mathsf{id}_1, \cdots, \mathsf{id}_t) \in \mathsf{Val.keys}()$ and $(c_1, \cdots, c_t, c) \in \mathbb{F}^{t+1}$, set $\mathsf{Val}[\bar{\mathsf{id}}] \leftarrow \sum_{i=1}^{t} c_i \cdot \mathsf{Val}[\mathsf{id}_i] + c$. If there is an index i^* such that $c_{i^*} \neq 0$ and $\mathsf{Label}[\mathsf{id}_{i^*}] = \mathsf{random}$, set $\mathsf{Label}(\bar{\mathsf{id}}) \leftarrow \mathsf{random}$. For all such index i^*, erase the random token from $\mathsf{Label}[\mathsf{id}_{i^*}]$.

Random Open: On command $(\mathsf{open}, \mathsf{id})$ from all parties, where $\mathsf{id} \in \mathsf{Val.keys}()$, and $\mathsf{Label}[\mathsf{id}_{i^*}] = \mathsf{random}$, send $\mathsf{Val}[\mathsf{id}]$, wait for x from the adversary, output x to all the parties, and erase the random symbol from $\mathsf{Label}[\mathsf{id}_{i^*}]$.

Abort: On command $\perp$ from the adversary, send $\perp$ to all parties and terminate.

Functionality $\mathscr{F}_{\mathsf{i\text{-}triple}}$:

Input Tuple: On input $(\mathsf{input\text{-}tuple}, P_j, \mathsf{id})$ from all parties, sample $\mathsf{Val}[\mathsf{id}] \xleftarrow{\$} \mathbb{F}$ and output it to P_j. Wait for an input from the adversary. If this input is the token random, pick $y \xleftarrow{\$} \mathbb{F}$ and output uniformly random shares of (x, y) to all the players, otherwise output uniformly random shares of $(x, \Delta \cdot x)$ to all the players.

Implicit Triple: On command $(\mathsf{implicit\text{-}triple}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3, \mathsf{id}_\rho, \mathsf{id}_\sigma)$ from all parties, pick five uniformly random values $(a, b, c, \hat{a}, \hat{c}) \xleftarrow{\$} \mathbb{F}^5$. wait for an input from the adversary; if this input is the token random, send uniformy random shares of $(a, b, c, \hat{a}, \hat{c})$ to the parties. Otherwise, send uniformy random shares of $(a, b, ab, \hat{a}, \hat{a}b)$ to the parties.

Fig. 5: Ideal Implicit Preprocessing Functionality

To handle this situation, we implicitely suppose the following behavior for all our functionalities (except $\mathscr{F}_{\mathsf{CABB}}$, which takes the participation data into account): the players append κ zeroes to any command they call. The functionality only look at the input of players whose input ended with κ zeroes. If the commands called by all these players are consistent, then the functionality plays its role as if the remaining players had called the correct command with random inputs. This is just a technicality to ensure that the players will still get outputs from the functionality when some players are not participating.

4.3 Implementing the Covert Arithmetic Black Box

We now provide a detailed implementation of the functionality $\mathscr{F}_{\mathsf{CABB}}$, in the $\mathscr{F}_{\mathsf{i\text{-}prep}}$-hybrid model, assuming broadcast channels. We start by presenting the protocol without the output command; we will deal with this command afterward. This protocol is essentially identical to [KOS16]; the only major difference is regarding the triple generation procedure, as the functionality $\mathscr{F}_{\mathsf{auth}}$ produces unauthenticated triples. we describe the protocol Figure 6. It remains to implement the output command. This is the most intricate part of the construction, and requires to use methods that differ significantly from standard MPC protocols. Before we describe it, we introduce some primitives on which it relies.

Covert Protocol Π

Each player P_i maintains the set $\mathsf{AuthValues}_i$ to track all the authenticated values, the set $\mathsf{OpenValues}_i$ to track all the partially opened values, and the set $\mathsf{SacrificeValues}_i$ to track all the multiplication tuple that will require a sacrifice procedure to ensure their correctness.

Initialize: The parties call $\mathscr{F}_{\mathsf{i\text{-}prep}}$ with the command implicit-triple to load random (possibly erroneous) multiplication tuples $(\langle a\rangle, \langle b\rangle, \langle ab\rangle, \langle \hat{a}\rangle, \langle \hat{a}\rangle)$ for each multiplicative gate of both f and g, and with the command input-tuple to load masks $[\![r]\!]$ for each input of the parties. Each player P_i adds his authenticated shares of the masks to $\mathsf{AuthValues}_i$.

Input: To input a value x_i with identifier id_i, party P_i takes a mask $[\![r_i]\!]$, and broadcasts $\varepsilon \leftarrow x_i - r_i$ (recall that he received r_i from $\mathscr{F}_{\mathsf{i\text{-}prep}}$ when creating the mask). The parties locally compute $[\![x_i]\!] \leftarrow [\![r_i]\!] + \varepsilon$.

Add: The parties locally compute $[\![x+y]\!] \leftarrow [\![x]\!] + [\![y]\!]$.

Multiply: On input $([\![x]\!], [\![y]\!])$, the parties do the following:

1. Take a multiplication tuple $(\langle a\rangle, \langle b\rangle, \langle ab\rangle, \langle \hat{a}\rangle, \langle \hat{a}\rangle)$, and execute the following Random Input procedure: the players call the Input procedure defined above on each share of $(a, b, ab, \hat{a}, \hat{a}b)$ to get authenticated shares of each share of the players, and use the (local) addition procedure to compute $([\![a]\!], [\![b]\!], [\![ab]\!], [\![\hat{a}]\!], [\![\hat{a}]\!])$. Each player P_i adds his authenticated shares of $(a, b, ab, \hat{a}, \hat{a}b)$ to $\mathsf{AuthValues}_i$ and to $\mathsf{SacrificeValues}_i$.
2. Compute $[\![\alpha]\!] \leftarrow [\![x]\!] - [\![a]\!]$ and $[\![\beta]\!] \leftarrow [\![y]\!] - [\![b]\!]$, and call the open command of $\mathscr{F}_{\mathsf{i\text{-}prep}}$ to get (α, β) (which is a valid operation as both $x-a$ and $y-b$ are labeled as random by $\mathscr{F}_{\mathsf{i\text{-}prep}}$). Each player P_i adds $(\alpha, s_i(\alpha), \delta_i(\alpha))$ and $(\beta, s_i(\beta), \delta_i(\beta))$ to $\mathsf{OpenValues}_i$.
3. Set $[\![xy]\!] \leftarrow \alpha\,[\![b]\!] + \beta\,[\![a]\!] + \alpha\beta + [\![ab]\!]$.

Fig. 6: Covert Protocol (without the output protocol)

4.4 Implementing the Output Phase

In addition to the $\mathscr{F}_{\mathsf{i\text{-}prep}}$ functionality, we will assume that the players have access to an ideal *implicit checking functionality* $\mathscr{F}_{\mathsf{i\text{-}check}}$, whose purpose is to let the player commit to values, and implicitely check that a commitment contains a given value. We also assume an ideal *secure transfer functionality* $\mathscr{F}_{\mathsf{st}}$. The implicit checking functionality is represented Figure 7. $\mathscr{F}_{\mathsf{st}}$ has the following behavior: on input (send, P_j, x) from P_i, $\mathscr{F}_{\mathsf{st}}$ output (P_i, x) to P_j. The functionality leaks the bitsize of x. As $\mathscr{F}_{\mathsf{i\text{-}check}}$ relies on commitment schemes, we start by recalling their definition.

Commitment Scheme. Commitment schemes are an essential primitive in cryptography. A commitment scheme allows to lock a value in a box, so that the sender cannot change the locked value when

it opens the box (the *binding* property) but the receiver gets no information about the locked value from seeing the box (the *hiding* property).

Definition 7 (Commitment Scheme). *A commitment scheme over $\mathbb{F}$ (with randomness space $\mathbb{F}$) is defined by two algorithms* $(\mathsf{Setup}, \mathsf{Commit})$*:*

- $\mathsf{Setup}(1^\kappa)$*, generates the public parameters* pp*,*
- $\mathsf{Commit}(m; r)$*, given a message $m \in \mathbb{F}$ and some random coins $r \xleftarrow{\$} \mathbb{F}$, outputs a commitment c (*pp *is implicitely passed as a parameter to* Commit*).*

A commitment scheme is assumed to satisfy the following properties:

Hiding. No probabilistic polynomial-time adversary $\mathscr{A}$, that is first given $\mathsf{pp} \xleftarrow{\$} \mathsf{Setup}(1^\kappa)$, can distinguish commitments on two messages (m_0, m_1) of its choice.

Binding. For any commitment c, no probabilistic polynomial-time adversary $\mathscr{A}$ can find two pairs (m_0, r_0) and (m_1, r_1), with $m_0 \neq m_1$, such that $c = \mathsf{Commit}(m_0; r_0) = \mathsf{Commit}(m_1; r_1)$. The commitment scheme is said *perfectly binding* if this is impossible even for unbounded adversaries.

Uniform. To allow for a uniform UC implementation of the functionality $\mathscr{F}_{\mathsf{i\text{-}check}}$, we need the commitments to look like random bit-strings. We say that a commitment scheme is *uniform* if for any message m, the distribution $\{c : r \xleftarrow{\$} \mathbb{F}, c \leftarrow \mathsf{Commit}(\mathsf{pp}, m; r)\}$ is computationally indistinguishable from the uniform distribution over $\{0,1\}^t$ for some t. Note that the uniformity property of a commitment scheme implies its hiding property.

Functionality $\mathscr{F}_{\mathsf{i\text{-}check}}$

$\mathscr{F}_{\mathsf{i\text{-}check}}$ has the same description as $\mathscr{F}_{\mathsf{i\text{-}prep}}$. In addition, the functionality contains the description of a group $(\mathbb{G}, \cdot)$, and maintains a dictionary Commit. It also contains the description of a uniform commitment scheme $c : \mathbb{F}^2 \mapsto \mathbb{G}$. In addition to $\mathscr{F}_{\mathsf{i\text{-}prep}}$'s description, $\mathscr{F}_{\mathsf{i\text{-}check}}$ also has the following commands:

Commit: On input $(\mathsf{com}, m_i, r_i, \hat{\mathsf{id}}_i, \mathsf{id})$ from each player P_i, where $(m_i, r_i) \in \mathbb{F}^2$ and each m_i is an element of the pair $\mathsf{Val}[\hat{\mathsf{id}}_i]$, store $\mathsf{Commit}[\mathsf{id}] \leftarrow (\sum_i m_i, \sum_i r_i)$.

Linear Combination of Commitments: On command $(\mathsf{lcc}, \bar{\mathsf{id}}, \mathsf{id}_1, \cdots, \mathsf{id}_t, c_1, \cdots, c_t, c)$ from the honest parties, where $\mathsf{id}_1, \cdots, \mathsf{id}_t) \in \mathsf{Commit.keys}()$ and $(c_1, \cdots, c_t, c) \in \mathbb{F}^{t+1}$, set $\mathsf{Commit}[\bar{\mathsf{id}}] \leftarrow \sum_{i=1}^{t} c_i \cdot \mathsf{Commit}[\mathsf{id}_i] + c$.

Implicit Check: On input $(\mathsf{implicit\text{-}check}, \mathsf{id}, m, r_i)$ from each party P_i, check whether $\mathsf{Commit}[\mathsf{id}] = (m, \sum_i r_i)$. Wait for an input from the adversary. If this is $\mathsf{deliver}$ and if the check succeeds, pick uniformly random shares over $\mathbb{G}^n$ of the vector $\boldsymbol{v} = (1, \cdots, 1) \in \mathbb{G}^n$, and output one share to each player. Else, send n uniformly random elements of $\mathbb{G}$ to each players.

Reveal: On input $(\mathsf{reveal}, \mathsf{id})$ from all the players, where $\mathsf{id} \in \mathsf{Commit.keys}()$, retrieve $(m, r) \leftarrow \mathsf{Commit}[\mathsf{id}]$, output $c(m; r)$ to all the players, and terminate.

Fig. 7: Ideal Functionalities for Implicit Checking

The Output Command. We are now ready to describe the implementation of the output command of our covert protocol, in the $(\mathscr{F}_{\mathsf{i\text{-}prep}}, \mathscr{F}_{\mathsf{i\text{-}check}}, \mathscr{F}_{\mathsf{st}})$-hybrid model, assuming access to a uniform commitment scheme $c : \mathbb{F}^2 \mapsto \mathbb{G}$. Recall that an authenticated value $[\![x]\!]$ is a pair $(\langle x \rangle, \langle \Delta \cdot x \rangle)$, where $\Delta \in \mathbb{F}$ is uniformly shared between the players; the shares held by each player P_i are denoted $(s_i(x), \delta_i(x))$, where $\delta_i(x)$ is called the *authentication share* of P_i. The goal of the players will be to simultaneously verify the correctness of all the authenticated values (i.e., for each authenticated value a, that $\sum_i \delta_i(a) = \Delta \sum_i s_i(a)$), the correctness of all the partial openings, the absence of errors in all the triple generations (which will be verified using a sacrificing procedure), and the favorability of the result (z should be equal to 1). We assume that the players have access to a stateful pseudo-random generator $\mathsf{prg} : \mathbb{F} \mapsto \mathbb{F}^*$. The players sequentially perform the following operations:

Coin Flipping. The purpose of this phase is for the players to jointly generate a random seed m. Each player P_i picks $(m_i, r_i) \xleftarrow{\$} \mathbb{F}^2$, calls the command $(\mathsf{com}, m_i, r_i, \mathsf{id}_1)$ of $\mathscr{F}_{\mathsf{i\text{-}check}}$, and broadcasts m_i. Let $m \leftarrow \sum_i m_i$. Note that a corrupted player P_i could cheat by broadcasting $m_i' \neq m_i$ instead of m_i, but it will later be ensured that if they did so, this will just cause the protocol to fail.

Sacrifice. Let $(\llbracket u_j \rrbracket, \llbracket v_j \rrbracket, \llbracket u_j v_j \rrbracket, \llbracket \hat{u}_j \rrbracket, \llbracket \hat{u}_j v_j \rrbracket)_{j \leq s}$ be all the authenticated tuples stored in the sets $\mathsf{SacrificeValues}_i$ during the protocol. The players stretch $(r_j)_{j \leq s} \leftarrow \mathsf{prg}(m)$ and call s times the open command of $\mathscr{F}_{\mathsf{i\text{-}prep}}$ on each $\llbracket \rho_j \rrbracket \leftarrow r_j \cdot \llbracket u_j \rrbracket - \llbracket \hat{u}_j \rrbracket$ to obtain ρ_j. Let $\llbracket \sigma_j \rrbracket \leftarrow r_j \cdot \llbracket u_j v_j \rrbracket - \llbracket \hat{u}_j v_j \rrbracket - \rho_j \cdot \llbracket v_j \rrbracket$. Each player P_i adds $(\rho_j, \llbracket \rho_j \rrbracket)$ and $(0, \llbracket \sigma_j \rrbracket)$ to his set $\mathsf{OpenValues}_i$, for $j = 1$ to s.

Commit. Let $\llbracket y \rrbracket$ be the authenticated share of the target output of the protocol, and let $\llbracket z \rrbracket$ be the target output of the admissibility function. Let $((o_1, \llbracket o'_1 \rrbracket), \cdots, (o_t, \llbracket o'_t \rrbracket))$ be all the values stored in the sets $\mathsf{OpenValues}_i$ during the protocol, and let $(\llbracket a_i \rrbracket)_{i \leq u}$ be all the values stored in the sets $\mathsf{AuthValues}_i$ during the execution of the protocol. The players perform the following operations:

- All the players stretch $(\lambda_1, \cdots, \lambda_s, \lambda'_1, \cdots, \lambda'_{2t}, \mu) \leftarrow \mathsf{prg}(m)$. Let γ be the pseudorandom linear combination with these coefficients of all the partial openings $(o_1, \cdots, o_s, \rho_1, \cdots, \rho_t, 0, \cdots, 0, 1)$. For each $i \leq n$, P_i sets

$$\gamma_i \leftarrow \gamma \cdot s_i(\Delta) - \sum_{j=1}^{s} \lambda_j \delta_i(o'_j) + \sum_{j=1}^{t} \lambda'_j \delta_i(\rho'_j) + \sum_{j=1}^{t} \lambda'_{t+j} \delta_i(\sigma_j) + \mu \delta_i(z)$$

  Each player P_i picks $r'_i \xleftarrow{\$} \mathbb{F}$ and calls $(\mathsf{com}, \gamma_i, r'_i, \mathsf{id}_2)$. Note that if all the players participated, the result is favorable ($z = 1$), and no player cheated, $\mathsf{Commit}[\mathsf{id}_2]$ should now contain $\Delta \cdot \gamma - \Delta \cdot \gamma = 0$.
- All the players stretch $(\alpha_i)_{i \leq u} \leftarrow \mathsf{prg}(\lambda)$. Each player P_i sets

$$\zeta_i \leftarrow \sum_{j=1}^{u} \alpha_j s_i(a_j), \qquad \zeta'_i \leftarrow \sum_{j=1}^{u} \alpha_j \delta_i(a_j)$$

- Each party P_i picks $(\theta_i, \theta'_i) \xleftarrow{\$} \mathbb{F}^2$ and calls $(\mathsf{com}, s_i(y), \theta_i, \mathsf{id}_3)$ and $(\mathsf{com}, \delta_i(y), \theta'_i, \mathsf{id}_4)$. Note that if all the players participated honestly to the protocol, $\mathsf{Commit}[\mathsf{id}_3]$ should contain y and $\mathsf{Commit}[\mathsf{id}_4]$ should contain $\Delta \cdot y$.
- Each party P_i picks $(\tau_i, \tau'_i) \xleftarrow{\$} \mathbb{F}^2$ and calls $(\mathsf{com}, \zeta_i, \tau_i, \mathsf{id}_5)$ and $(\mathsf{com}, \zeta'_i, \tau'_i, \mathsf{id}_6)$. Note that if all the players participated honestly, $\Delta \cdot \mathsf{Commit}[\mathsf{id}_5] - \mathsf{Commit}[\mathsf{id}_6]$ should be equal to 0.

MAC Reconstruction. Each player P_i broadcasts his share of Δ. All the players reconstruct Δ.

Implicit Checking. Each player P_i calls $(\mathsf{lin\text{-}comb}, \mathsf{id}_7, \mathsf{id}_3, \mathsf{id}_4, \Delta, -1, 0)$ (this stores $\Delta\mathsf{Commit}[\mathsf{id}_3] - \mathsf{Commit}[\mathsf{id}_4]$ at $\mathsf{Commit}[\mathsf{id}_7]$; if all the players participated honestly, this value should be equal to 0) and $(\mathsf{lin\text{-}comb}, \mathsf{id}_8, \mathsf{id}_5, \mathsf{id}_6, \Delta, -1, 0)$ (this stores $\Delta\mathsf{Commit}[\mathsf{id}_5] - \mathsf{Commit}[\mathsf{id}_6]$ at $\mathsf{Commit}[\mathsf{id}_8]$; if all the players participated honestly, this value should be equal to 0). Then, each player P_i performs the following calls to $\mathscr{F}_{\mathsf{i\text{-}check}}$:

- $(\mathsf{implicit\text{-}check}, \mathsf{id}_1, m, r_i)$, to implicitely check that the coin flipping phase was performed honestly,
- $(\mathsf{implicit\text{-}check}, \mathsf{id}_2, 0, r'_i)$, to implicitely check that all the partial openings were correct and that the result is favorable (note that because of the sacrificing procedure, this does also implicitely ensures that all the multiplication tuples were correctly generated),
- $(\mathsf{implicit\text{-}check}, \mathsf{id}_7, 0, \theta'_i - \Delta\theta_i)$, to implicitely check that $\mathsf{Commit}[\mathsf{id}_3]$ does indeed contain the correct output value y,
- $(\mathsf{implicit\text{-}check}, \mathsf{id}_8, 0, \tau'_i - \Delta\tau_i)$, to implicitely check that all the inputs where correctly authenticated.

Let $(K_i^{(1)}, \cdots, K_i^{(n)})$ denote the component-wise product over $\mathbb{G}$ of the four outputs of $\mathscr{F}_{\mathsf{i\text{-}check}}$ for each player P_i.

Explicit Checking. The purpose of this phase is to allow the players to explicitely check whether all of the implicit checks did succeed.[3] Each player P_i picks a random key $K'_i \in \mathbb{G}$, sets $K_i^{(i)} \leftarrow K_i^{(i)} \cdot K'_i$, and broadcasts $(K_i^{(1)}, \cdots, K_i^{(n)})$. All the players reconstruct $(K_1, \cdots, K_n)$. Each player P_i checks that $K_i = K'_i$; if this is not the case, P_i sets his output to be $\perp$ and stops participating from this point (that is, P_i sends uniformly random bit-strings for the rest of the protocol).

[3] We emphasize that this does not allow the players to learn separate information on which implicit check did succeed, and which did fail.

Secure Transfer. Each player P_i calls $(\mathsf{send}, P_j, s_i(y)||\theta_i)$ for each $j \neq i$. All the players reconstruct (y, θ), where $\theta \leftarrow \sum_i \theta_i$.

Output. All the players call $(\mathsf{reveal}, \mathsf{id}_3)$, and get a commitment c_y. Each player P_i checks that $c_y = c(y; \theta)$. If the check succeeds, P_i terminates with output y; else, P_i terminates with output $\perp$.

Theorem 8. *The protocol Π securely implements $\mathcal{F}_{\mathsf{CABB}}$ in the covert model of [CGOS07], given access to the ideal functionalities $\mathcal{F}_{\mathsf{i\text{-}prep}}$, $\mathcal{F}_{\mathsf{i\text{-}check}}$, and $\mathcal{F}_{\mathsf{wr}}$.*

4.5 Security Proof

Let C be the set of parties corrupted by the adversary. Let $\mathcal{S}im$ be a simulator of the honest parties, running local copies of $\mathcal{F}_{\mathsf{i\text{-}prep}}, \mathcal{F}_{\mathsf{i\text{-}check}}$, and $\mathcal{F}_{\mathsf{wr}}$. We start by describing the simulation of all the commands, except the output command; we deal afterward with the simulation of the output command. The simulator $\mathcal{S}im$ maintains a flag flag to record cheating attempts from the corrupted parties; initially, he sets $\mathsf{flag} = \mathsf{OK}$.

Simulation of the Computation. The simulation of the initialization command is straightforward, as the commands implicit-triple and input-tuple do not involve any private information of the players, but only value identifiers. If the adversary introduces errors in any call to one of the commands (by sending random tokens), $\mathcal{S}im$ sets $\mathsf{flag} \leftarrow \mathsf{fail}$.

For the input command from P_i, two cases occur:

- If P_i is corrupted, $\mathcal{S}im$ computes and stores $x'_i \leftarrow r_i + \varepsilon'_i$ with identifier id_i, where r_i is a mask generated by $\mathcal{S}im$'s local copy of $\mathcal{F}_{\mathsf{i\text{-}prep}}$, and ε'_i is the (possibly erroneous) value broadcasted by P_i. $\mathcal{S}im$ inputs $(\mathsf{input}, P_i, \mathsf{id}_i, x'_i)$ to $\mathcal{F}_{\mathsf{CABB}}$ on behalf of the corrupted party P_i, and $(\mathsf{input}, P_i, \mathsf{id}_i)$ on behalf of the other parties.
- If P_i is honest, $\mathcal{S}im$ broadcasts a random value ε'_i on behalf of P_i in the real world, and inputs $(\mathsf{input}, P_i, \mathsf{id}_i)$ to $\mathcal{F}_{\mathsf{CABB}}$ on behalf of the corrupted parties in the ideal world. $\mathcal{S}im$ computes and stores $x'_i \leftarrow r_i + \varepsilon'_i$ with identifier id_i.

The add command is local, and therefore requires no simulation; $\mathcal{S}im$ simply sends $(\mathsf{add}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$ to $\mathcal{F}_{\mathsf{CABB}}$ (with appropriate identifiers). For the multiplication command, $\mathcal{S}im$ performs the authentication procedures honestly on behalf of the honest parties (this does not involve any of their private inputs). Then, $\mathcal{S}im$ broadcasts two random values on behalf of his local copy of $\mathcal{F}_{\mathsf{i\text{-}prep}}$ when the players call the open commands. $\mathcal{S}im$ sends $(\mathsf{mult}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$ to $\mathcal{F}_{\mathsf{CABB}}$ (with appropriate identifiers) and stores the result of the computation. It remains to simulate the output phase.

Simulation of the Output.

Coin Flipping. $\mathcal{S}im$ performs honestly the coin flipping procedure on behalf of all the honest parties.

Sacrifice. $\mathcal{S}im$ performs honestly the sacrifice procedure on behalf of all the honest parties, and emulates the open command honestly.

Commit. $\mathcal{S}im$ calls the Output command of the ideal functionality $\mathcal{F}_{\mathsf{CABB}}$ on behalf of the corrupted parties in the ideal world, and gets an output X. If $X = \perp$, $\mathcal{S}im$ sets $\mathsf{flag} = \mathsf{fail}$. Then, he plays as follows:

- $\mathcal{S}im$ locally computes all the values γ_i (with the values stored in his local copy of $\mathcal{F}_{\mathsf{i\text{-}prep}}$) of the corrupted parties; let γ' denote their sum. If $\mathsf{flag} = \mathsf{OK}$, $\mathcal{S}im$ commits to uniformly random shares of $-\gamma'$ on behalf of the honest players; otherwise, $\mathcal{S}im$ commits to random values on behalf of the honest players.
- $\mathcal{S}im$ locally computes the sum of the values $s_i(y)$ and the sum of the values $\delta_i(y)$ for all the corrupted parties P_i; let (s', δ') denote the sums. If $\mathsf{flag} = \mathsf{OK}$, $\mathcal{S}im$ commits to random shares of $X - s'$ and $\Delta X - \delta'$ on behalf of the honest parties; otherwise, $\mathcal{S}im$ commits to random values on their behalf. Let $(s'_i(y))_i$ denote the values $\mathcal{S}im$ committed to with identifier id_3 on behalf of the honest parties, and let $(\theta_i)_i$ be the associated random coins.

– $\mathscr{S}im$ locally computes the sum of the values ζ_i and the sum of the values ζ'_i for all the corrupted parties P_i; let (ζ', ζ'') denote the sums. If $\mathsf{flag} = \mathsf{OK}$, $\mathscr{S}im$ picks $x \xleftarrow{\$} \mathbb{F}$ and commits to random shares of $x - \zeta'$ and $\Delta x - \zeta''$ on behalf of the honest parties; otherwise, $\mathscr{S}im$ commits to random values on their behalf.

MAC Reconstruction. $\mathscr{S}im$ honestly broadcasts the MAC share of each honest party. If the reconstructed MAC is not equal to Δ, $\mathscr{S}im$ sets $\mathsf{flag} = \mathsf{fail}$.

Implicit Checking. If $\mathsf{flag} = \mathsf{OK}$, $\mathscr{S}im$ plays honestly on behalf of each honest party, and emulates honestly the implicit-check command of his local copy of $\mathscr{F}_{\mathsf{i\text{-}check}}$. Otherwise, $\mathscr{S}im$ emulates implicit-check as if all the checks had failed.

Explicit Checking. $\mathscr{S}im$ honestly plays the explicit checking phase on behalf of the honest parties, and stops participating on behalf of the honest parties for which the check failed.

Secure Transfer. $\mathscr{S}im$ honestly emulates the secure transfer protocol. $\mathscr{S}im$ securely transfers $s'_i(y)||\theta_i$ to all the parties on behalf of each honest party P_i.

Output. $\mathscr{S}im$ computes the appropriate random coin θ and returns $c(X, \theta)$ when the parties call the reveal command of $\mathscr{F}_{\mathsf{i\text{-}check}}$.

Indistinguishability. We now argue that the view produced by $\mathscr{S}im$ is computationally indistinguishable from an honest run of the protocol. This is straightfowrward for the implicit-triple, input-tuple, and add commands. The simulation of the input phase is perfect, as the only leaked value is uniformly random over $\mathbb{F}$ in both the honest protocol and the simulated protocol. The simulation is also perfect for the multiplication command, as only uniformly random-looking values are returned by the open command.

Let us look at the output phase. The emulation of the coin flipping phase, the sacrifice phase, and the MAC reconstruction phase are perfect, as $\mathscr{S}im$ plays honestly in all of them (the private inputs of the honest parties are not required in these phases). By the uniformity property of the commitment scheme (which implies in particular its hiding property), each commitment (whose random coins are drawn uniformly at random) is indistinguishable from a uniformly random bit-string, no matter the value it commits to, hence the emulation of the commit phase is indistinguishable from a honest execution of the commit phase, under the uniformity property of the commitment scheme. In the implicit checking phase, the view of any strict subset of the parties is indistinguishable from the random distribution over $\mathbb{G}$, as the values are uniformly random all-or-nothing shares. We now look at the explicit checking phase. The core observation, which is a classical result in MPC, is that the probability that a random linear combination of committed values is equal to zero is negligible, unless all the committed values are equal to 0. Therefore, the keys $(K_i^{(1)}, \cdots, K_i^{(n)})$ held by each player P_i form uniformly random shares of $(1, \cdots, 1)$ over $\mathbb{G}^n$ if and only if the following conditions are all satisfied:

1. The commitment with identifier id_1 commits to the seed m (this ensure that the values m_i sent by the corrupted parties were not adaptively chosen using the values sent by the honest parties)
2. All the partially opened values are consistent with the corresponding authenticated shares
3. The output of the protocol is favorable ($z = 1$)
4. The commitment with identifier id_3 contains the same value y that the authenticated share $[\![y]\!]$ of the output of the protocol
5. All the values a authenticated during the protocol satisfy $\sum_i \delta_i(a) = \Delta \cdot \sum_i s_i(a)$
6. All the players participated to the protocol (as if any player was not participating, the above conditions would break with overwhelming probability).

Therefore, the only step in which an adversary could potentially introduce deviations while passing the checks is the generation of multiplication triples. But this is handled by the sacrificing phase: our sacrificing phase is perfectly identical to the sacrifice procedure in the implementation of the triple generation functionality in [KOS16], except that the sacrifice procedure of [KOS16] involves an explicit check that $[\![\rho_j]\!]$ does indeed open to ρ_j, and that $[\![\sigma_j]\!]$ does indeed open to 0. But as $(\rho_j, [\![\rho_j]\!])$ and $(0, [\![\sigma_i]\!])$ are added by each player to his set $\mathsf{OpenValues}_i$, this is captured by the condition 2 above. The exact same argument as in [KOS16] therefore shows that the probability that both checks pass is negligible, unless the corresponding triple $([\![u_j]\!], [\![v_j]\!], [\![u_j v_j]\!])$ is indeed a valid triple.

Therefore, if all of these conditions hold, then it implies that the corrupted players remained perfectly honest up to the explicit checking phase of the protocol. In the simulated protocol, any attempt from a corrupted player to cheat in a partial opening, in an authentication procedure, during a triple generation, or during the MAC reconstruction phase, is detected by $\mathscr{S}im$, which sets $\mathsf{flag} = \mathsf{fail}$ when it happens. If any corrupted party attempts to cheat during the coin flipping phase or the commit phase, this is automatically detected and handled by $\mathscr{S}im$'s local copy of $\mathscr{F}_{\mathsf{i\text{-}check}}$. Finally, $\mathscr{S}im$ learns whether all the players participated and the output is favorable from $\mathscr{F}_{\mathsf{CABB}}$. Therefore, the term-by-term products of the keys returned during the simulation of implicit-check will form random-looking shares of $(1, \cdots, 1)$ exactly when it happens in a real execution of the protocol.

As each player P_i multiplies his i'th share by a random element K'_i of $\mathbb{G}$, all the values with index $i \neq j$ are indistinguishable from random elements of $\mathbb{G}$ for each player P_j. Moreover, as $K_i^{(i)}$ remains perfectly hidden from the view of any player P_j for $j \neq i$, the probability for all the other players to pass P_i's check is negligible, unless the keys $(K_j^{(i)})_{j \leq n}$ do indeed reconstruct to 1. Therefore, it holds with overwhelming probility that the check performed by each player P_i passes if and only if all the conditions listed above are verified, and the players broadcasted their correct shares of the keys – that is, if all the players participated honestly to the protocol up to that point, and the output was favorable. Note that a corrupted player could cause the check to fail for a subset of the parties only; this correspond to the adversary being able to chose which honest parties will get the output of the protocol.

The simulation of the secure transfer and the output is straightforward, as the reveal phase only involve values stored in $\mathscr{S}im$'s local copy of $\mathscr{F}_{\mathsf{i\text{-}prep}}$. By the uniformity property of the commitment scheme, it reveals nothing on the output y. The final check performed by the players will succeed if an only if the players send correct shares of their opening, by the binding property of the commitment scheme. Therefore, if all the players were honest up to that point, including the secure transfer phase, they indeed obtain the correct output y of the protocol; else, either the explicit check fails, or the final check fails (in which case each honest players know that all the players participated to the protocol and the output was favorable, but is prevented by the adversary to get his final output).

Eventually, it is straightforward to see that $\mathscr{S}im$ sends only uniformly random values during the simulation. As the presence of non-participating players does not affect whether any of the functionalities $\mathscr{F}_{\mathsf{i\text{-}prep}}$ and $\mathscr{F}_{\mathsf{i\text{-}check}}$ returns an output (see subsection 4.2), and if any player does not participate, the explicit check phase will already break down for all the players in the real protocol (hence before any call to $\mathscr{F}_{\mathsf{st}}$) and in the simulation (as $\mathscr{S}im$ receives $\perp$ from $\mathscr{F}_{\mathsf{CABB}}$), both the real protocol and the simulation do not leak any information regarding non-participating players. Therefore, our covert protocol Π implements $\mathscr{F}_{\mathsf{CABB}}$ in the $(\mathscr{F}_{\mathsf{i\text{-}prep}}, \mathscr{F}_{\mathsf{i\text{-}check}}, \mathscr{F}_{\mathsf{st}})$-hybrid uniform UC model. □

4.6 Implementing the Ideal Preprocessing Functionality

We now turn our attention to the ideal functionality $\mathscr{F}_{\mathsf{i\text{-}prep}}$. We start by introducing a notion that will be useful to describe our implementations.

Definition 9 (Encodable Set). *A set S is* k-encodable *if there is an efficiently computable and efficiently invertible map* $\mathsf{map} : S \times \mathfrak{R} \mapsto \{0,1\}^k$*, where $\mathfrak{R}$ is the space of random coins, such that* $\{\mathsf{map}(x; r) \mid x \xleftarrow{\$} S, r \xleftarrow{\$} \mathfrak{R}\}$ *is the uniform distribution over* $\{0,1\}^k$.

Note that most groups used in crypto (e.g. elliptic curves, prime-order fields, and subgroups of modular squares) do have such efficient mappings in the uniform distribution.

Importing the Implementation of MASCOT. In [KOS16, Section 5,6], $\mathscr{F}_{\mathsf{prep}}$ is implemented with a protocol $\Pi_{[\![\cdot]\!]}$ using and ideal functionality for correlated oblivious product evalution with errors (COPEe), that we described Subsection 3.2. Our implementation of $\mathscr{F}_{\mathsf{i\text{-}prep}}$ is exactly the protocol $\Pi'_{[\![\cdot]\!]}$ obtained by removing all the checking procedures from $\Pi_{[\![\cdot]\!]}$ (as well as the authentication and sacrificing steps in the triple generation). The simulation is therefore completely straitghforward, as the functionality $\mathscr{F}_{\mathsf{i\text{-}prep}}$ is identical to $\mathscr{F}_{\mathsf{prep}}$, except that the adversary is allowed to introduce errors at every step of the computation, which models the fact that the adversary might now manipulate

the outcome of some commands (as no checking procedures are performed). $\mathscr{F}_{\mathsf{i\text{-}prep}}$ also distinguishes between random and non-random inputs, but each call to the open command in $\Pi_{[\![\cdot]\!]}$ is already on a random value. The multiplication triple generation procedure is also different, as it produces unauthenticated shares in $\mathscr{F}_{\mathsf{i\text{-}prep}}$, but this corresponds exactly to the procedure implemented by the protocol Π_{triple} of [KOS16, Section 5] without the authentication, sacrifice, and checking steps. Let $\mathscr{S}im'$ be the simulator constructed exactly as the simulator for $\Pi_{[\![\cdot]\!]}$, except that it does not simulate any checking procedure. We observe that all the values sent to any strict subset of the players by the simulator of $\Pi_{[\![\cdot]\!]}$ are uniformly random values over $\mathbb{F}$, except for the output of the checking procedure, which we entirely remove. Therefore, $\mathscr{S}im'$ does only send uniformly random values over $\mathbb{F}$, which are statistically close to uniformly random bit-strings.

The protocol $\Pi_{[\![\cdot]\!]}$ implements $\mathscr{F}_{\mathsf{prep}}$ in the $(\mathscr{F}_{\mathsf{COPEe}}, , \mathscr{F}_{\mathsf{com}})$-hybrid model, where $\mathscr{F}_{\mathsf{com}}$ is an ideal commitment functionality. However, $\mathscr{F}_{\mathsf{com}}$ is only invoked in the MAC checking subprotocol (that we completely remove in our protocol). As $\mathscr{S}im'$ does only send uniformly random values over $\mathbb{F}$, we conclude that the protocol $\Pi'_{[\![\cdot]\!]}$ implements $\mathscr{F}_{\mathsf{i\text{-}prep}}$ in the uniform UC $\mathscr{F}_{\mathsf{COPEe}}$-hybrid model. By the uniform UC theorem (theorem 15), it suffices to replace $\mathscr{F}_{\mathsf{COPEe}}$ by an implementation of $\mathscr{F}_{\mathsf{COPEe}}$ in the uniform UC model to get a uniform UC implementation of $\mathscr{F}_{\mathsf{i\text{-}prep}}$ in the real world.

Correlated Oblivious Product Evaluation with Errors. It therefore remains to discuss the implementation of $\mathscr{F}_{\mathsf{COPEe}}$. We implement it exactly as in [KOS16, Section 4]: indeed, the distribution of all the values sent to a strict subset of the players (which corresponds here to a single player) by the simulator of this implementation is already trivially uniform.[4] The protocol COPEe requires ideal access to an oblivious transfer functionality. We note that a small modification to the DDH-based UC oblivious transfer of [PVW08] leads to a uniform UC oblivious transfer protocol secure against malicious adversaries: all the values exchanged in this protocol are indistinguishable (under the DDH assumption) from uniformly random values over some group $\mathbb{G}_{\mathsf{OT}}$ where the DDH assumption is believed to hold. Therefore, we modify the protocol as follow: we pick $\mathbb{G}_{\mathsf{OT}}$ as an encodable group, and let all the players use the associated map map to encode all their inputs before sending them. As the map is efficiently invertible, the players can decode all the flows they receive. As the distribution of messages sent in the UC oblivious transfer of [PVW08] is indistinguishable from random over $\mathbb{G}_{\mathsf{OT}}$ under the DDH assumption, the messages in our modified protocol are all indistinguishable from random bit-strings under the same assumption, hence it implements the oblivious transfer functionality in the uniform UC model.

4.7 Implementing the Implicit Checking Functionality

We now turn our attention to the implicit checking functionality. Our implementation relies on two primitives: a *homomorphic smooth projective hash function* and a *plaintext-checkable homomorphic commitment scheme*. We start by introducing these primitives.

Smooth Projective Hash Functions. Smooth projective hash functions (SPHF), introduced by Cramer and Shoup in [CS02] under the name of hash proof system, were initially used to construct $\mathsf{IND\text{-}CCA}$-secure encryption schemes out of of $\mathsf{IND\text{-}CPA}$-secure encryptions schemes; however, it has since proven useful in many settings, such as commitment schemes [ACP09], or password-authenticated key-exchanges [BBC+13]. We recall the definition from [ACP09]. Intuitively, an SPHF allows to hash a word with some secret key, so that if the word belongs to some language $\mathscr{L}$, then the hash can also be computed using a witness for $x \in \mathscr{L}$ an some public key. If $x \notin \mathscr{L}$, then the hash value is statistically instinguishable from random, even given the public key. More formally, an SPHF with domain Π on a word x for a langage $\mathscr{L}$ is a tuple of probabilistic polynomial-time algorithms $(\mathsf{HashKG}, \mathsf{ProjKG}, \mathsf{Hash}, \mathsf{ProjHash})$ such that:

- $\mathsf{HashKG}(1^\kappa)$ outputs a hashing key hk.
- $\mathsf{ProjKG}(\mathsf{hk}, \mathscr{L}, x)$ outputs a projection key hp.

[4] The values sent by the simulator are uniformly random over $\mathbb{F}$, which is statistically close to the uniform distribution over bit-strings as the order of $\mathbb{F}$ is of the form $2^a + b$ with a, b polynomials in κ.

– $\mathsf{Hash}(\mathsf{hk}, \mathscr{L}, x)$ outputs a hash value H.
– $\mathsf{ProjHash}(\mathsf{hp}, \mathscr{L}, x, w)$, where w is a witness for the statement $x \in \mathscr{L}$, outputs a projective hash value H'.

Definition 10. *(smooth projective hash function) A smooth projective hash function is a tuple of algorithm as defined above such that the following conditions hold:*

1. ***Correctness.*** *For every $x \in \mathscr{L}$ with witness w, $\mathsf{hk} \leftarrow \mathsf{HashKG}(1^\kappa)$, $\mathsf{hp} \leftarrow \mathsf{ProjKG}(\mathsf{hk}, \mathscr{L}, x)$, $\mathsf{Hash}(\mathsf{hk}, \mathscr{L}, x) = \mathsf{ProjHash}(\mathsf{hp}, \mathscr{L}, x, w)$.*
2. ***Smoothness.*** *For every $x \notin \mathscr{L}$, $\mathsf{hk} \leftarrow \mathsf{HashKG}(1^\kappa)$, $\mathsf{hp} \leftarrow \mathsf{ProjKG}(\mathsf{hk}, \mathscr{L}, x)$, the distributions $\{H \in \Pi \mid H = \mathsf{Hash}(\mathsf{hk}, \mathscr{L}, x)\}$ and $\{H \xleftarrow{\$} \Pi\}$ are statistically indistinguishable, even given hp.*

Homomorphic SPHF. An SPHF is homomorphic if the hashing space $\mathbb{H}$ and the witness space $\mathbb{W}$ are groups (with a law denoted $\cdot$ for $\mathbb{H}$ and $+$ for $\mathbb{W}$) and for any $x \in \mathscr{L}$ with witness w, and any values $w_1 \cdots w_t$ such that $\sum_i w_i = w$,

$$\mathsf{ProjHash}(\mathsf{hp}, \mathscr{L}, x, w) = \prod_{i=1}^{t} \mathsf{ProjHash}(\mathsf{hp}, \mathscr{L}, x, w_i)$$

Uniform SPHF. As for the oblivious transfer protocol, we can construct uniform SPHF out of standard SPHF, by using SPHF where the group of projection keys and the group of hash values are all encodable; we note that this does hold for most instantiations of the SPHF framework of [BBC+13], which encompasses all known constructions of SPHFs.

Commitment Scheme. We let $\mathsf{com} : \mathbb{F}^2 \mapsto \mathbb{G}$ denote a uniform commitment scheme, which satisfies in addition the following properties:

Homomorphic. A commitment scheme is *homomorphic* if there is an operator $\boxplus$ such that for any $(m_0, r_0, m_1, r_1) \in \mathbb{F}^4$, letting $c_0 \leftarrow \mathsf{Commit}(m_0; r_0)$, $c_1 \leftarrow \mathsf{Commit}(m_1; r_1)$, and $c \leftarrow c_0 \boxplus c_1$, it holds that $c = \mathsf{Commit}(m_0 + m_1; r_0 + r_1)$. We let $\bullet$ denote the external multiplication by an integer, and $\boxminus$ denote the substraction operator.

Plaintext-Checkable. A perfectly binding commitment is *plaintext-checkable* if the setup additionally outputs a trapdoor τ, and there is an efficient algorithm $\mathsf{Check}(c, m, \tau)$ which, given the trapdoor, the public parameters, a commitment and a candidate plaintext, outputs 1 if c is a valid commitment of m, and 0 else.

Implicit Checking Protocol. We now describe the protocol $\Pi_{\text{i-check}}$, in the $\mathscr{F}_{\text{i-prep}}$-hybrid model. Let $(\mathsf{com}, \mathsf{Setup}, \mathsf{Check})$ be a uniform plaintext-checkable homomorphic commitment scheme. For each $m \in \mathbb{F}$, let

$$\mathscr{L}_m = \{(c, m) \mid \exists r, c = \mathsf{Commit}(m; r)\}$$

We fix a value m. Let $(\mathsf{HashKG}, \mathsf{ProjKG}, \mathsf{Hash}, \mathsf{ProjHash})$ be a uniform homomorphic SPHF for the language $\mathscr{L}_m$.

Commit. To jointly commit to a value $m = \sum_i m_i$, with the m_i being values stored in $\mathscr{F}_{\text{i-prep}}$, each player P_i picks $r_i \xleftarrow{\$} \mathbb{F}$ and broadcasts $c_i \leftarrow \mathsf{com}(m_i; r_i)$. All the players compute $c \leftarrow \boxplus_i c_i$.

Linear Combination. Linear combinations are locally performed using the homomorphic properties of com.

Implicit Check. Let c be a commitment, let m be a plaintext, and let $(r_i)_{i \le n}$ be the shares of the random coins held by the parties. For $k = 1$ to n, the players repeat the following operations:

1. Each player P_i computes $\mathsf{hk}_i \xleftarrow{\$} \mathsf{HashKG}(1^\kappa)$, $\mathsf{hp}_i \leftarrow \mathsf{ProjKG}(\mathsf{hk}_i, \mathscr{L}_m, c)$, and broadcasts hp_i.
2. Each player P_i compute $H_i \leftarrow \mathsf{Hash}(\mathsf{hk}_i, \mathscr{L}_m, c)$ and broadcasts the hash values $(H'_{ij})_{j \neq i} \leftarrow (\mathsf{ProjHash}(\mathsf{hp}_j, \mathscr{L}_m, c, r_i))_{j \neq i}$. Each player P_i sets $K_i^{(k)} \leftarrow H_i^{-1} \cdot \prod_{j \neq i} H'_{ij}$.

Reveal. The reveal command does not require any implementation, as the players already have the corresponding output of any valid input to the command.

Instantiating the primitives. We identify $\mathbb{F}$ to $\mathbb{Z}_p$. Let $q = kp + 1$ be a prime, for some small k. We implement com as the additive version of the famous ElGamal encryption scheme, over the multiplicative subgroup $\mathbb{G}$ of $(\mathbb{Z}_q^*, \cdot)$ of order p. The setup outputs a generator $g \xleftarrow{\$} \mathbb{G}$ of $\mathbb{G}$, and $h \leftarrow g^\tau$ for a random $\tau \xleftarrow{\$} \mathbb{Z}_p$. It sets τ to be the trapdoor.

We now describe an efficient mapping map between $\mathbb{G}$ and $\{0,1\}^{\log p}$: let $\mathbb{G}_k$ be the multiplicative subgroup of $\mathbb{Z}_q^*$ of order k; let $\mathsf{map}_{\mathsf{cano}}$ be the canonical bijective mapping which maps elements of $\mathbb{G}\times\mathbb{G}_k$ to elements of $\mathbb{Z}_q^*$. To encode an element x of $\mathbb{G}$, one picks $y \xleftarrow{\$} \mathbb{G}_k$ and sets $\mathsf{map}(x; y) \leftarrow \mathsf{map}_{\mathsf{cano}}(x, y)$. Decoding is simply done by computing $\mathsf{map}_{\mathsf{cano}}^{-1}(z)$ and dropping the second part of the pair. Encodings of uniformly random elements of $\mathbb{G}$ are uniformly random over $\mathbb{Z}_q^*$, hence they are statistically close to uniformly random over $\{0,1\}^{\log q}$.

To commit to m with random coin $r \in \mathbb{F}$, the sender picks $(r_0, r_1) \xleftarrow{\$} \mathbb{G}_k^2$ and computes $(\mathsf{map}(g^r; r_0), \mathsf{map}(h^r g^m; r_1))$. The scheme is perfectly binding; it's hiding property relies on the DDH assumption over $\mathbb{G}$. The scheme is also homomorphic. The plaintext-checking algorithm works as follows: given a commitment (c_0, c_1) and a candidate message m, the algorithm checks whether $c_1/c_0^s = g^m$. Moreover, by the DDH assumption, a random encrytion c of an arbitrary message m is indistinguishable from a random element of $\mathbb{Z}_q^2$, which are statistically close to uniformly random over $\{0,1\}^{2\log p}$.

The framework of [BBC+13] allows to instantiate SPHFs from various assumptions; a possibility is to instantiate it over $\mathbb{G}$, based again on the DDH assumption. With this instantiation, the language of ElGamal commitments to some plaintext is naturally and efficiently handled; we note that the DDH-based SPHFs of [BBC+13] are already homomorphic. The same mapping from $\mathbb{G}$ to $\{0,1\}^{\log p}$ makes both the projection keys and the hash values indistinguishable from random bit-strings under the DDH assumption.

Proof. We prove that this protocol securely implements $\mathscr{F}_{\mathsf{i\text{-}check}}$ in the uniform UC model. It is straightforward to see that the security of the implementation of the commit command directly reduces to the hiding and biding properties of com; furthermore, the implementation is perfect for the linear combination and the reveal commands, at they involve no interactions. We now turn our attention to the implicit checking command.

From the correctness of the SPHF and the homomorphic properties, we get that if $c \in \mathscr{L}_m$, it should hold that $\prod_{i\neq j} H'_{ij} = H_j$, hence $\prod_i H_i^{-1} \cdot \prod_{j\neq i} H'_{ij} = 1$. From the smoothness property, we get that if $c \notin \mathscr{L}$, then each hash H_i is indistinguishable from a uniformly random element of $\mathbb{H}$ from the view of all the players $(P_j)_{j\neq i}$. The simulation proceeds as follows: $\mathscr{S}im$ retrieves the values m_i that the corrupted player should commit too, from his local copy of $\mathscr{F}_{\mathsf{i\text{-}prep}}$; let m' be the sum of these values. Then, $\mathscr{S}im$ commits to uniformly random shares of $m - m'$ on behalf of the honest parties. Using the fact that the commitment scheme is plaintext-checkable, $\mathscr{S}im$ checks whether the resulting commitment c commits to m using his trapdoor τ. If this is the case, $\mathscr{S}im$ computes honestly the projective hash values on behalf of the honest players (which he can do as he picked the plaintext and the random coins for them). $\mathscr{S}im$ also computes H_i honestly on behalf of each honest player. Else, if c does not commit to m, $\mathscr{S}im$ broadcasts uniformly random elements of $\mathbb{H}$ on behalf of the honest parties. Moreover, as the SPHF and the commitment scheme are both uniform, the view of any strict subgroup of the players (which consists of commitments with uniformly random coins and all the projection keys and hash values broadcasted during the protocol together with the values H_i computed by these players) is indistinguishable from a distribution of uniformly random bits. Therefore, the protocol $\Pi_{\mathsf{i\text{-}check}}$ implements $\mathscr{F}_{\mathsf{i\text{-}check}}$ in the uniform UC model.

4.8 Implementing the Secure Transfer Functionality

The functionality $\mathscr{F}_{\mathsf{st}}$ allows the players to communicate through perfectly secure authenticated point-to-point channels. We must implement it in the uniform UC model – i.e., implement it with a protocol in which only random-looking bit-strings are exchanged. This task is of interest outside of the scope of covert multiparty computation; indeed, this problem was already considered in [HLv02, vAH04] under the name of *public-key steganography*, by the same authors that introduced the concept of covert two-party computation. In particular, [vAH04] gives a construction whose security relies on

the DDH assumption. The construction uses a simple variant of the ElGamal encryption scheme; it is straightforward to show that it implements our secure transfer functionality in the uniform UC model.

4.9 Complexity

At the exception of the output phase, our protocol has the same complexity as the MASCOT protocol, minus the cost of the checking procedures that we do not perform. The cost of the output phase is dominated by the implicit checking procedure: our implementation of the implicit checking procedure communicates $O(n^3)$ group elements, which amounts to $O(\kappa n^3)$ bits exchanged.

References

ACP09. M. Abdalla, C. Chevalier, and D. Pointcheval. Smooth projective hashing for conditionally extractable commitments. In *CRYPTO 2009*, *LNCS* 5677, pages 671–689. Springer, Heidelberg, August 2009.

BBC+13. F. Benhamouda, O. Blazy, C. Chevalier, D. Pointcheval, and D. Vergnaud. New techniques for SPHFs and efficient one-round PAKE protocols. In *CRYPTO 2013, Part I*, *LNCS* 8042, pages 449–475. Springer, Heidelberg, August 2013.

Can01. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

CDN15. R. Cramer, I. B. Damgård, and J. B. Nielsen. *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press, New York, NY, USA, 1st edition, 2015.

CDSJ16. C. Cho, D. Dachman-Soled, and S. Jarecki. Efficient concurrent covert computation of string equality and set intersection. In *CT-RSA 2016*, *LNCS* 9610, pages 164–179. Springer, Heidelberg, February / March 2016.

CGOS07. N. Chandran, V. Goyal, R. Ostrovsky, and A. Sahai. Covert multi-party computation. In *48th FOCS*, pages 238–248. IEEE Computer Society Press, October 2007.

CKL03. R. Canetti, E. Kushilevitz, and Y. Lindell. On the limitations of universally composable two-party computation without set-up assumptions. In *EUROCRYPT 2003*, *LNCS* 2656, pages 68–86. Springer, Heidelberg, May 2003.

CS02. R. Cramer and V. Shoup. Universal hash proofs and a paradigm for adaptive chosen ciphertext secure public-key encryption. In *EUROCRYPT 2002*, *LNCS* 2332, pages 45–64. Springer, Heidelberg, April / May 2002.

DDM+06. A. Datta, A. Derek, J. C. Mitchell, A. Ramanathan, and A. Scedrov. Games and the impossibility of realizable ideal functionality. In *TCC 2006*, *LNCS* 3876, pages 360–379. Springer, Heidelberg, March 2006.

DPSZ12. I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO 2012*, *LNCS* 7417, pages 643–662. Springer, Heidelberg, August 2012.

DZ13. I. Damgård and S. Zakarias. Constant-overhead secure computation of Boolean circuits using preprocessing. In *TCC 2013*, *LNCS* 7785, pages 621–641. Springer, Heidelberg, March 2013.

GJ10. V. Goyal and A. Jain. On the round complexity of covert computation. In *42nd ACM STOC*, pages 191–200. ACM Press, June 2010.

HLv02. N. J. Hopper, J. Langford, and L. von Ahn. Provably secure steganography. In *CRYPTO 2002*, *LNCS* 2442, pages 77–92. Springer, Heidelberg, August 2002.

IKNP03. Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In *CRYPTO 2003*, *LNCS* 2729, pages 145–161. Springer, Heidelberg, August 2003.

Jar14. S. Jarecki. Practical covert authentication. In *PKC 2014*, *LNCS* 8383, pages 611–629. Springer, Heidelberg, March 2014.

KL11. D. Kidron and Y. Lindell. Impossibility results for universal composability in public-key models and with fixed inputs. *Journal of Cryptology*, 24(3):517–544, July 2011.

KOS16. M. Keller, E. Orsini, and P. Scholl. Mascot: Faster malicious arithmetic secure computation with oblivious transfer. Cryptology ePrint Archive, Report 2016/505, 2016. `http://eprint.iacr.org/2016/505`.

PVW08. C. Peikert, V. Vaikuntanathan, and B. Waters. A framework for efficient and composable oblivious transfer. In *CRYPTO 2008*, *LNCS* 5157, pages 554–571. Springer, Heidelberg, August 2008.

vAH04. L. von Ahn and N. J. Hopper. Public-key steganography. In *EUROCRYPT 2004*, *LNCS* 3027, pages 323–341. Springer, Heidelberg, May 2004.

vAHL05. L. von Ahn, N. J. Hopper, and J. Langford. Covert two-party computation. In *37th ACM STOC*, pages 513–522. ACM Press, May 2005.

Supplementary Material

A Formal Proof of the Covert UC Theorem

A.1 Preliminaries

We will not cover the details of the original framework of [Can01]. Instead, we use an alternative definition of the UC framework which was developped in great details in Cramer, Damgård, and Nielsen's book on multiparty computation [CDN15]; we find this presentation of the UC model to be simpler and more natural to work with. Some of the definitions stated in this section are taken almost verbatim from [CDN15]. We omit some details to keep the presentation short; the interested reader should refer to [CDN15] for fully detailed definitions.

Interactive Agents and Systems. In the UC framework, players are modeled by *interactive agents*. An interactive agent is a computational device which receives and sends messages on input and output ports, and maintain some internal state. More formally, an interactive agent is a tuple $(\mathsf{In}, \mathsf{Out}, \mathsf{State}, \mathsf{Msg}, T, \sigma_0)$ where In and Out are finit sets of inports and outports, State is a set of possible states, Msg a set of possible messages, T a transition algorithm which takes as input a message, a state, and a security parameter κ, and σ_0 is the initial state of the agent. A running agent is said to be activated; when it outputs some result on an outport named *op*, the activation token and the result are sent to the agents who have an input port named *op*. A *responsive* agent is an agent that will eventually return a result when activated in any context; if it returns after a number of steps polynomial in κ, it is called *polytime responsive*.

By connecting outports to inports with the same name, we define an interaction pattern between the agents, which forms an *interactive system*. When an outport *op* of an agent has the same name as the inport of another agent in the interactive system, we say that they are *connected*: any message output on the outport *op* is input on the corresponding inport. If no two agents have inports with the same name, or outports with the same name, we say that they are *port compatible*. A port compatible interactive system is an interactive systems whose agents are all port compatible. If two interactive systems is_1 and is_2 are port compatible, then we can define a new interactive system as $is_1 \cup is_2$: this is the *composition* operation, and it will be denoted $\diamond$. By definition, $\diamond$ is a commutative and associative operation on interactive systems. When composing two interactive systems (π_1, π_2), if they have no connected input and output ports, we write $\pi_1 \diamond \pi_2 = \perp$.

Environments and Indistinguishability of Interactive Systems. All the inports and outports which are not connected to any port in the system are said to be *open*; an interactive system with no open ports is said to be *closed*. An interactive system is *executable* if it is closed and contains an agent with a special *activation port* and a special *return port*. Then, given an interactive system I, a *closure* of I is an interactive system Z connected to I through its activation and return ports, such that $I \diamond Z$ is executable. An environment of an interactive system I is an interactive system taking the security paramter κ as input, which is a closure of I so that $(Z \diamond I)(\kappa)$ will eventually output a *guess* in $\{0, 1\}$ on the special return port of I. Two interactive systems (I_1, I_2) are computationally indistinguishable with respect to some set of environments Env for both I_1 and I_2, if for every environment $Z \in \mathsf{Env}$, it holds that the distributions of outputs of $(I_1 \diamond Z)(\kappa)$ and $(I_2 \diamond Z)(\kappa)$ are computationally indistinguishable.[5] This is denoted $I_1 \overset{\mathsf{Env}}{\equiv} I_2$. This relation can easily be shown to be transitive.

Security and Ideal Functionalities. The purpose of the UC framework is to formalize the security of interactive protocols: its privacy, stating that all the values leaked from the protocol belong to some set of allowed values, and its robustness, stating that the only influence a malicious adversary can have on the behavior of the protocol belongs to some set of allowed influence. Those two properties are interleaved, and are defined by the mean of an *ideal functionality* I_F. An ideal functionality is

[5] Note that we can define similarly perfect and statistical indistinguishability of interactive systems.

an extremely simple interactive system which has some name F, takes some inputs, performs some computation and returns the result; this functionality models the expected behavior of the protocol. It containts n inports and outports $(F.in_i, F.out_i)_{i\leq n}$ and two *special ports*, a leakage port $F.leak$ and an influence port $F.inf$, which are used to model the allowed leakage and the allowed influence on the protocol.

Protocols. A *simple protocol* is an interactive system with n agents $P_1, \cdots, P_n$ called the *players*. A protocol π has a protocol name F, and a resource name R; these name will allow to link together a protocol and an interactive system called the *resource*. The resource is a system connected to all the players, which will allow us to compose protocols in a natural way. Each agent P_i has six ports: two *resource ports* $(R.in_i, R.out_i)$ that connect to the resource (the resource has the corresponding input port $R.out_i$ and output port $R.in_i$), an open inport $F.in_i$ on which it receives its input, and an open outport $F.out_i$ on which it writes its output (the *protocol ports*), and two *special ports* $(R.leak_i, R.inf_i)$ which will allow to model any influence or leakage from a protocol π: they will receive tags indicating which corruption model is chosen by the environment (passive or active corruption of some subset of the parties, or any other model of corruption), influence the behavior of the functionality, and specify what the corrupted players will output on their leakage port. We denote by Pro the set of protocols. When a protocol π_1 has resource name R, and protocol π_2 has protocol name F, we say that the protocol $\pi_1 \diamond \pi_2$ is a *composed protocol* which has the protocol name of π_1 and the resource name of π_2. This composed protocol contains *composed parties*, which are natural composition of the parties of π_1 and the parties of π_2 (each party of π_1 has its resource ports connected to the input and output ports of a party in π_2; note that in the composed protocol, each composed party might have more than two special properties).

Simulators. Now, to argue that a protocol π_F with name F securely implements some ideal functionality I_F with the same name F when using resource I_R, we would like to say that $\pi_F \diamond I_R$ is indistinguishable from I_F; however, those two interactive systems have no reason to be indistinguishable as, in particular, they will not have the same open ports (although they have the same set of input and output ports as they have the same name F, the ideal functionality does only have a single open pair of leakeage / influence ports, while the protocol contains open leakage and influence port for each player). This is solved by introducing a *simulator*: a simulator S for π_F related to I_F is a polytime interactive system so that the systems $I_F \diamond S$ and $\pi_F \diamond I_R$ have the same set of open ports (*id est* S has the leakage and influence ports of all the (possibly composed) players in π_F, and S connects to the leakage and influence port of I_F). An additional technical requirement for the simulator is that it must be corruption-preserving, which means that it does not corrupt players unless told so by the environment. We denote by $\mathscr{S}im$ the set of simulators (i.e. interactive systems that are a simulator of some protocol π_F for some functionality I_F). Having defined simulators, we can now define environment classes: an environment class Env is a set of environments so that for any $Z \in \mathsf{Env}$, and any $\pi_F \in \mathsf{Pro}$ and $S \in \mathscr{S}im$ with a port structure matching that of Z, $\pi_F \diamond Z \in \mathsf{Env}$ and $S \diamond Z \in \mathsf{Env}$.

Modeling Corruption. We have already stated that corruption is modeled using the leakage and influence ports of the protocol. We will now make this statement more precise. In this paper, we will consider active corruption of the players (*aka* malicious security). Let π_F be an n-party protocol with name F and resource name R. The *standard corruption behavior* for active corruption of π_F is defined in [CDN15] as follows: if a party P_i receives the special symbol "active corrupt" on $R.inf_i$, it outputs its current state on $R.leak_i$. Then, it follows the following rules, and only these rules:

- On input "read, p" on $R.inf_i$ where p is the name of one of its inports, P_i reads the next message m on p and returns m on $R.leak_i$.
- On input "send, p, m" on $R.inf_i$ where p is the name of one of its outports, P_i sends m on p.

We now describe how to model the active corruption of an ideal functionality I_F. On input "active corrupt, i" on $F.inf$, the ideal functionality I_F records that the party i is corrupted and outputs the ideal internal state of party i on $F.leak$. Then it starts ignoring all inputs on $F.in_i$ and stops giving outputs on $F.out_i$. Instead, whenever it gets an input "input, i, x" on $F.inf$, it behaves exactly as if x

had arrived on $F.in_i$, and whenever I_F is about to output some value y on $F.out_i$, it instead outputs "output, i, y" on $F.leak$.

A.2 The UC Framework.

We are now ready to define what it means for a protocol to be UC-secure.

Definition 11 (UC security [CDN15]). *Let I_F be an ideal functionality with name F, let $\pi_F \in \mathsf{Pro}$ be a protocol with protocol name F and resource name R, and let I_R be an ideal functionality with name R. Let* Env *be an environment class. We say that $\pi_F \diamond I_R$* securely implements *I_F in environments* Env *if there exists a simulator $S \in \mathscr{S}im$ for π_F such that $\pi_F \diamond I_R \overset{\mathsf{Env}}{\equiv} I_F \diamond S$. We can also write this as $\pi_F \diamond I_R \overset{\mathsf{Env}}{\geq} I_F$.*

The choice of the environment class to consider defines the model for the computation. We have already mentionned passive and active corruption, but many other restrictions are standard: considering only the class of environments that corrupt at most t parties defines the *threshold security* of the protocol. Environments that indicate the parties they will corrupt at the start of the protocol define *static security*, while environments using arbitrary corruption patters define *adapative security*. We can also define *computational security* by considering the class of polytime environments. Now, we can finally state the UC theorem:

Theorem 12 (UC theorem [CDN15]). *Let* Env *be an environment class. Let $\pi_F \in \mathsf{Pro}$ be a protocol with protocol name F and resource name G. Let π_G be a protocol with protocol name G and resource name H and for which $\pi_F \diamond \pi_G \neq \perp$. Let I_F, I_G and I_H be ideal functionalities with names F, G respectively H. If $\pi_F \diamond I_G \overset{\mathsf{Env}}{\geq} I_F$ and $\pi_G \diamond I_H \overset{\mathsf{Env}}{\geq} I_G$, then $(\pi_F \diamond \pi_G) \diamond I_H \overset{\mathsf{Env}}{\geq} I_F$.*

In the following, we will denote $\mathsf{Env}_{\mathsf{static}}$ the set of environments who indicate the subset of parties that they will corrupt before the start of the protocol; it was shown in [CDN15] that this is indeed an environment class.

A.3 Uniform UC Computation

In this section, we construct a variant of the UC framework that we will use in our framework for covert multiparty computation. We start by introducing a new security notion, which we call *uniformity*.

Leakage List. Recall that the port $G.leak$ of a protocol π_F with resource I_G for an ideal functionality I_F is used to emulate the actual leakage of the real protocol. Let us formalize this a bit: the leakage port delivers information regarding the messages exchanged between the players. Therefore, we will consider, without loss of generality, that each message output by a functionality G on its leakage port $G.leak$ must be of the form "i, Y, $l(m)$" where $i \in [n]$ indicates a player, $Y \subseteq [n] \setminus \{i\}$ is a subset of the other players, and $l(m)$ is a leakage. This tuple reads as follows: player i delivered some message m to the players in Y, on which $l(m)$ has leaked, where l is a public leakage function. For example, a simulator for a secure transfer functionality would use $l : m \mapsto |m|$ as leakage function.

We call the *leakage list* of $\pi_F \diamond I_G$ the list of tuples "i, Y, $l(m)$" sent on $G.leak$ during an execution of $\pi_F \diamond I_G$.

Uniform Simulators. We are now ready to define *uniform simulators*. Intuitively, a uniform simulator is a simulator that sends only uniformly random values: it leaks only values $y = l(m)$ where m is a uniformly random bit-string. More formally,

Definition 13 (Uniform Simulator). *Let $\mathsf{Env} \subseteq \mathsf{Env}_{\mathsf{static}}$ be an environment class. Let $S \in \mathscr{S}im$ be a simulator of a protocol π_F with resource name G for an ideal functionality I_F. Then S is a* uniform simulator *of π_F for I_F if for any environment $Z \in \mathsf{Env}$ for $I_F \diamond S$ corrupting a strict subset of size n' of the players indexed by $X \subsetneq [n]$, for any execution of $Z \diamond I_F \diamond S$, given all the input messages to the ideal functionality on the ports $(F.in_i)_{i \in X}$ of $I_F \diamond S$, the leakages $(y_1, \cdots, y_{n-n'})$ that appear in a*

tuple "i, Y, y" on the leakage list of $\pi_F \diamond I_G$, so that $i \in [n] \setminus X$ and $Y \subseteq [n] \setminus \{i\}$, are of the form $(l(m_1), \cdots, l(m_{n-n'}))$ for uniformly random bit-strings $(m_j)_{j \leq n-n'}$. We denote by $\mathscr{S}im_{\mathsf{uni}}$ the set of uniform simulators.

This notion is crucial to capture the intuitive notion of uniformity, as players who take part to protocols implementing some uniform functionality must not be distinguishable from non-participating players, who send only uniformly random values, from the view of any adversary corrupting a strict subset of the parties. To simplify the exposition, we will call *l-uniform* a distribution of n leakages the form $\{l(m_1), \cdots, l(m_n) \mid (m_1 \cdots m_n) \xleftarrow{\$} \{0,1\}^{t_1} \times \cdots \times \{0,1\}^{t_n}\}$ for integers $t_1 \cdots t_n$. Note that this notion can be naturally extended to situations where different leakage functions $l_1, \cdots, l_n$ are used.

Uniform Universal Composability. We are now ready to formalize what it means to uniformly implement an ideal functionality:

Definition 14 (Uniform UC Security). *Let I_F be an ideal functionality with name F and n input and output ports, let $\pi_F \in \mathsf{Pro}$ be a protocol with protocol name F and resource name R, and let I_R be an ideal functionality with name R. Let $\mathsf{Env} \subseteq \mathsf{Env}_{\mathsf{static}}$ be an environment class. We say that $\pi_F \diamond I_R$* uniformly implements *I_F in environments Env if there exists a simulator $S \in \mathscr{S}im_{\mathsf{uni}}$ for π_F such that $\pi_F \diamond I_R \overset{\mathsf{Env}}{\equiv} I_F \diamond S$. We can also write this as $\pi_F \diamond I_R \overset{\mathsf{Env}}{\geq}_{\mathsf{uni}} I_F$.*

One should note that this model of security does not involve the participation data of the players at all. It is rather a restriction of the standard notion of UC security: any uniform UC implementation π_F of a functionality I_F is in particular a UC implementation of I_F. The main result that will allow us to introduce our framework is the following theorem:

Theorem 15 (Uniform UC theorem). *Let $\mathsf{Env} \subseteq \mathsf{Env}_{\mathsf{static}}$ be an environment class. Let $\pi_F \in \mathsf{Pro}$ be a protocol with protocol name F and resource name G. Let π_G be a protocol with protocol name G and resource name H and for which $\pi_F \diamond \pi_G \neq \bot$. Let I_F, I_G, and I_H be ideal functionalities with names F, G, and H. If $\pi_F \diamond I_G \overset{\mathsf{Env}}{\geq}_{\mathsf{uni}} I_F$ and $\pi_G \diamond I_H \overset{\mathsf{Env}}{\geq}_{\mathsf{uni}} I_G$, then $(\pi_F \diamond \pi_G) \diamond I_H \overset{\mathsf{Env}}{\geq}_{\mathsf{uni}} I_F$.*

Proof. Let $Z \in \mathsf{Env}$ be an environment for which composition makes sense, and let $S \in \mathscr{S}im_{\mathsf{uni}}$ be a simulator such that $\pi_F \diamond I_G \diamond Z \overset{\mathsf{Stat}}{\equiv} I_F \diamond S \diamond Z$. As Env is an environment class, it holds that $Z' \leftarrow Z \diamond \pi_F \in \mathsf{Env}$. Therefore, let $S' \in \mathscr{S}im$ be a simulator such that $\pi_G \diamond I_H \diamond Z' \overset{\mathsf{Stat}}{\equiv} I_G \diamond S' \diamond Z'$. Moreover, $\pi_G \diamond I_H \diamond Z' = \pi_F \diamond \pi_G \diamond I_H \diamond Z$ and $I_G \diamond S' \diamond Z' = \pi_F \diamond I_G \diamond S' \diamond Z$, by definition of Z' and commutativity of $\diamond$. Therefore, $\pi_F \diamond \pi_G \diamond I_H \diamond Z \overset{\mathsf{Stat}}{\equiv} \pi_F \diamond I_G \diamond S' \diamond Z$. Let $T \leftarrow S \diamond S' \in \mathscr{S}im$; by plugging in the equation $\pi_F \diamond I_G \diamond Z \overset{\mathsf{Stat}}{\equiv} I_F \diamond S \diamond Z$, we get:

$$\pi_F \diamond \pi_G \diamond I_H \diamond Z \overset{\mathsf{Stat}}{\equiv} I_F \diamond T \diamond Z$$

It remains to show that $T \in \mathscr{S}im_{\mathsf{uni}}$. It is important to note that this does not follow immediatly from the fact that both S and S' output a l-uniform leakage from the viewpoint of Z: indeed, while each leakage is independently l-uniform from the viewpoint of Z, it might be that the joint distributions of the two leakages is not l-uniform anymore if, for example, one of the leakages depends from the other.

However, as Env is an environment class, it holds that $Z \diamond S'$ is an environment. Therefore, by the uniform UC security of $\pi_F \diamond I_G$, the leakage distribution output by S' is l-uniform from the viewpoint of Z, and by the uniform UC security of $\pi_G \diamond I_H$, the leakage distribution output by S is l-uniform from the viewpoint of $Z \diamond S'$, which already includes the leakage from S'. Moreover, simulators are by definition *corruption preserving*: it holds that the subset of players corrupted by $Z \diamond S'$ is exactly the subset X of players corrupted by Z. Taken altogether, this implies that the leakage distribution output by $T = S \diamond S'$ is l-uniform from the viewpoint of any environment Z corrupting a strict subset X of the players. □

A.4 Covert UC Computation

We are now ready to introduce our UC-like framework for covert MPC protocols. We will demonstrate the following result: if a protocol π_F is secure in the covert model of [CGOS07] when given oracle access to an ideal functionality I_F, then it remains secure in the covert model when I_F is replaced by a protocol $\pi_F \diamond I_R$ which *uniformly implements* I_F. This is in contrast with standard UC protocols in which the same security notion is considered for the protocol and for the subprotocol it relies on. In order to state our result formally, we first have to reformulate the covert model of [CGOS07] in the language of interactive systems introduced in [CDN15].

Covert Player. A covert player is defined as standard players (which are interactive agents with a particular six-port structure), except that it also maintains in its state a bit p called its *participation data*. It has an additional standard behavior rule: when $p = 0$, each time the player should send a message m of bitlength ℓ on some outport, it draws a uniformly random $m' \xleftarrow{\$} \{0,1\}^\ell$ and sends m' on that outport instead. Covert protocols are defined as standard protocols, with covert players instead of standard players. We can define composed covert protocols, and composed covert players, in the natural way. Note that covert protocols can be connected to standard simulators or ideal functionalities, which will behave as if connected to a standard protocol.

Covert Ideal Functionality. A covert ideal functionality is an ideal functionality parametrized by two functions (f, g). Let $\hat{f}$ be the function that maps $\boldsymbol{y}$ to $f(\boldsymbol{y})$ if $g(\boldsymbol{y}) = 1$, and to $\perp$ else. When receiving a participation vector $\boldsymbol{p}$ and an input vector $\boldsymbol{x}$ on its input ports $(F.in_i)_{i\leq n}$ and a list L of honest players on its influence port $F.inf$, it checks whether all the entrie of $\boldsymbol{p}$ are 1. If this check succeeds, it sends $\hat{f}(\boldsymbol{x})$ on $(F.out_i)_{i\in L}$ and $\perp$ on $(F.out_i)_{i\in[n]\setminus L}$. Else, it sends $\perp$ on all its outports $(F.out_i)_{i\leq n}$.

Covert Security. We denote by Env the class of environments that perform static corruption of a strict subset of the players and follow the standard corruption behavior defined before, with the following additional rule: when connected to a protocol with resource name R, they send on $R.inf$ a subset L of the honest players that should receive the output of the computation (this can easily be shown to be an environment class). Covert simulators are defined exactly as before, but with respect to covert protocols instead of standard protocols. We can then define covert security:

Definition 16 (Covert Security). *Let I_F be a covert ideal functionality with name F and n input and output ports, let π_F be a covert protocol with protocol name F and resource name R, and let I_R be an ideal functionality with name R. We say that $\pi_F \diamond I_R$* covertly implements *I_F in environments* Env *if there exists a covert simulator S for π_F such that $\pi_F \diamond I_R \overset{\mathsf{Env}}{\equiv} I_F \diamond S$. We can also write this as $\pi_F \diamond I_R \overset{\mathsf{Env}}{\geq}_{\mathsf{cov}} I_F$.*

Having reformulated the covert model of Chandran *et al.* in the framework of [CDN15], we are almost ready to state the main result of this section, which provides sufficient conditions to use protocols secure in the *uniform UC model* inside covert protocols. But protocols secure in the uniform UC model are standard protocols, and we have not defined yet how to compose standard protocols with covert protocols. Therefore, we need to first extend the composition operator as follows:

Definition 17 (Extended composition). *Let π be a covert protocol with n covert players $(P_i)_{i\leq n}$, and let π' be a protocol with $n' < n$ players $(P'_i)_{i\leq n'}$. Let $\boldsymbol{p}$ be the vector of participation data of $(P_i)_{i\leq n}$ in π. Let $\pi'_{\boldsymbol{p}}$ be the covert protocol defined exactly as π', except that the players of π' are replaced by covert players whose global participation data is chosen according to $\boldsymbol{p}$. We write $\pi \diamond \pi'$ to denote the* covert protocol *$\pi \diamond \pi'_{\boldsymbol{p}}$.*

We are now ready to state our covert UC theorem.

Theorem 18 (Covert UC theorem). *Let π_F be a covert protocol with name F and resource name G. Let π_G be a protocol with protocol name G and resource name H and for which $\pi_F \diamond \pi_G \neq \perp$. Let I_F be a covert ideal functionality with name F, and let I_G, I_H be ideal functionalities with names G, H. If $\pi_F \diamond I_G \overset{\mathsf{Env}}{\geq}_{\mathsf{cov}} I_F$, and $\pi_G \diamond I_H \overset{\mathsf{Env}}{\geq}_{\mathsf{uni}} I_G$, then $(\pi_F \diamond \pi_G) \diamond I_H \overset{\mathsf{Env}}{\geq}_{\mathsf{cov}} I_F$.*

Proof. Let S be a covert simulator such that for all $Z \in \mathsf{Env}$, $\pi_F \diamond I_G \diamond Z \stackrel{c}{\equiv} I_F \diamond S \diamond Z$. Let $S' \in \mathscr{S}im_{\mathsf{uni}}$ be a simulator such that for all $Z \in \mathsf{Env}$, $\pi_G \diamond I_H \diamond Z \stackrel{c}{\equiv} I_G \diamond S' \diamond Z$. Let $Z \in \mathsf{Env}$ be an environment that corrupt a subset $X \subsetneq [n]$ of the players. Let $\boldsymbol{p}$ be the participation data of the honest players (in $[n] \setminus X$). Let $\pi_{G,\boldsymbol{p}}$ be the covert protocol obtained by replacing the players of π_G by covert players, with participation data $\boldsymbol{p}$ for the honest players (corrupted participating parties are not differentiated from corrupted non-participating parties as they are both controlled by the environment, so they can be replaced by covert players with any participation data). Note that $\pi_{G,\boldsymbol{p}}$ has the same port structure as π_G, hence S' is also a simulator for $\pi_{G,\boldsymbol{p}}$. If $\boldsymbol{p} = \mathbf{1} = (1, \cdots, 1)$, the behavior of $\pi_{G,\boldsymbol{p}}$ is exactly the same as the behavior of π_G, hence, as $\pi_G \diamond I_H \diamond Z \stackrel{c}{\equiv} I_G \diamond S' \diamond Z$, it also holds that $\pi_{G,\mathbf{1}} \diamond I_H \diamond Z \stackrel{c}{\equiv} I_G \diamond S' \diamond Z$, where S' is now seen as a covert simulator for $\pi_{G,\mathbf{1}}$. Therefore, let us focus on the case $\boldsymbol{p} \neq \mathbf{1}$.

If $\boldsymbol{p} \neq \mathbf{1}$, the protocol $\pi_{G,\boldsymbol{p}}$ contains at least one honest non-participating player. We show that S' still simulates the protocol $\pi_{G,\boldsymbol{p}} \diamond I_H$ when connected to I_G, by considering successive modifications of the protocol π_G.

- We start with the protocol π_G unmodified. Recall that $\pi_G \diamond I_H \diamond Z \stackrel{c}{\equiv} I_G \diamond S' \diamond Z$.
- Now, consider the protocol π'_G, in which some honest player is replaced by a non-participating player, which means that all the messages he sends are replaced by uniformly random values. Let $(y_1, \cdots, y_t)$ be the leakage of all the messages sent by the original player, and let $(y'_1, \cdots, y'_t)$ be the leakage of the non-participating player that replaced him. By definition, $(y'_1, \cdots, y'_t)$ follows a l-uniform distribution.

We argue that $\pi_G \diamond I_H \diamond Z \stackrel{c}{\equiv} \pi'_G \diamond I_H \diamond Z$. This follows directly from the fact that $\pi_G \diamond I_H$ is a uniform UC implementation of I_G: as it is simulated by a uniform simulator, the distribution of its leakage is computationally indistinguishable from the l-uniform distribution. Therefore, if $\pi_G \diamond I_H \diamond Z$ and $\pi'_G \diamond I_H \diamond Z$ are not computationally indistinguishable, then Z is a polytime algorithm that distinguishes $(y_1, \cdots, y_t)$ from the l-uniform distribution, which contradicts the existence of S'.

The above process of replacing a honest player by a non-participating player can be repeated, and the same argument allows to argue that $S' \diamond I_G$ does still simulate the modified protocol. In particular, for any $Z \in \mathsf{Env}$ it holds that $\pi_{G,\boldsymbol{p}} \diamond I_H \diamond Z \stackrel{c}{\equiv} I_G \diamond S' \diamond Z$, where S' is seen as a covert simulator with respect to the covert protocol $\pi_{G,\boldsymbol{p}}$. As Env is an environment class, as in the proof of Theorem 15, we can use the fact that $Z' \leftarrow Z \diamond \pi_F \in \mathsf{Env}$ and plug the result in the equation $\pi_F \diamond I_G \diamond Z \stackrel{c}{\equiv} I_F \diamond S \diamond Z$ to show:

$$(\pi_F \diamond \pi_G) \diamond I_H \diamond Z \stackrel{c}{\equiv} I_F \diamond T \diamond Z$$

where $T \leftarrow S \diamond S'$ is a covert simulator, and $\pi_F \diamond \pi_G = \pi_F \diamond \pi_{G,\boldsymbol{p}}$ is a composed covert protocol. This concludes the proof. □