Output Privacy in Secure Multiparty Computation

Emmanuel Bresson[1], Dario Catalano[2],
Nelly Fazio[3]⋆, Antonio Nicolosi[3]⋆, and Moti Yung[4]⋆⋆

[1] Cryptology Department, CELAR, France. emmanuel.bresson@polytechnique.org
[2] CNRS - École Normale Supérieure, Paris, France. dario.catalano@ens.fr
[3] New York University, USA. {fazio,nicolosi}@cs.nyu.edu
[4] Columbia University & RSA Labs, USA. moti@cs.columbia.edu

Abstract. In secure multiparty computation, a set of mutually mistrusting players engage in a protocol to compute an arbitrary, publicly known polynomial-sized function of the party's private inputs, in a way that does not reveal (to an adversary controlling some of the players) any knowledge about the remaining inputs, beyond what can be deduced from the obtained output(s). Since its introduction by Yao [39], and Goldreich, Micali and Wigderson [29], this powerful paradigm has received a lot of attention. All throughout, however, very little attention has been given to the privacy of the players' outputs. Yet, disclosure of (part of) the output(s) may have serious consequences for the overall security of the application *e.g.,* when the computed output is a secret key; or when the evaluation of the function is part of a larger computation, so that the function's output(s) will be used as input(s) in the next phase.

In this work, we define the notion of *private-output multiparty computation*. This newly revised notion encompasses (as a particular case) the classical definition and allows a set of players to jointly compute the output of a common function in such a way that the execution of the protocol reveals no information (to an adversary controlling some of the players) about (some part of) the *outputs* (other than what follows from the description of the function itself). Next, we formally verify that basically no function can be output-privately computed in the presence of an adversary who gets full access to the internal memory of the corrupted players. However, if one restricts the (computationally bounded) adversary to control only part of the *state* of corrupted players, any function can be output-privately computed, assuming that enhanced trapdoor permutations exist and that public communication channels are available. Moreover, we prove security is preserved under sequential composition.

We note that *partial* access to the internal state of some of the players (either part of the time *e.g.,* forward-security and intrusion-resiliency, or part of the space, *e.g.,* secure CPU/memory) is an assumption that has been used in various settings to formalize limits on the attacker's capabilities that can be enforced via reasonable physical and architectural restrictions. However, previous models were devised for specific cryptographic tasks (*e.g.,* encryption and signature schemes), whereas our formalization has a wider scope. We believe that the model we suggest may foster further studies of insider adversaries with partial control in the context of secure multiparty computation.

1 Introduction

Secure multi-party computation (MPC) [39,29] allows a set of mutually mistrusting parties to jointly compute a function, while keeping their inputs private. The MPC paradigm allows many settings and concerns to be modeled, and thus it is a strong tool in showing that solutions exist to very general cryptographic problems (*cf. e.g.,* [29,3,14,15,10,27]). The power of the framework stems from the fact that under corruption of some of the parties (according to various settings and constraints) it is possible to compile any polynomial sized function into a protocol that maintains input privacy. In

⋆ Research conducted while visiting École Normale Supérieure, Paris.
⋆⋆ Research conducted in part while visiting École Normale Supérieure, Paris.

particular, input privacy is assured facing an adversary that is assumed to monitor the entire state (memory) of corrupted parties (passive adversary) and one that in addition may control the corrupted parties' memory arbitrarily (malicious adversary).

Let us briefly recall just some of the settings considered in the literature. A basic distinction is between the *computational* setting [29] where all communication between the parties is visible to the adversary, and the *information-theoretic* one [3,14,38], where point-to-point communication links are completely protected, but the adversary is not restricted to probabilistic polynomial time. An orthogonal distinction is regarding static *vs.* adaptive corruption [10], whereas other directions investigate different classes of possible adversaries [32,19,7] or extended security notions [38,11,23,37,2,8].

MOTIVATION. In all the above flavors, the definition of secure MPC guarantees that no information on the inputs is leaked to the adversary. In other words, the privacy concern (*cf. e.g.* [15,1]) is about the parties' *inputs*, and no assurance is directly provided about the confidentiality of the *output(s)*. Disclosure of (part of) the output(s) may indeed have serious consequences on the overall security of the application *e.g.,* when the computed output is a secret key; or when the evaluation of the function is part of a larger computation, so that the function's output(s) will be used as input(s) in the next phase; or if there is a need to reveal the output only at a certain point locally by a participant, in "real time," and not before.

One could think that a simple way to provide output-privacy (with respect to some adversary) would be to compute the given function using standard multiparty computation techniques, but forcing the players to keep the output "distributed" among them all. Then, when the output is needed, the parties would privately exchange their "shares" (by means of encryption) and locally reconstruct the global output. This would seemingly protect against outsider adversaries, *i.e.,* adversaries who are only allowed to monitor the communication among the players.

Such *ad-hoc* solution, however, dodges the problem, rather than solving it. In particular, how can we prove that this is "secure" if current definitions of secure function evaluation do *not* model the security concerns that we want to address? How could protocols resulting from the above approach be composed, when the output is required to be at the same time secure and locally (rather than distributedly) available?

On the other hand, a more structured approach would clearly be preferable, possibly addressing issues of protocol composition, and encompassing more powerful adversaries who can exert some form of "active" control on the protocol participants. Though this is mostly a definitional effort, we believe that shedding light on the conditions under which privacy of the output(s) can be attained is important to improve our understanding of a central cryptographic tool such as MPC.

Postulating a limitation on the kind of control that the adversary can obtain on corrupted parties is akin in spirit to physical assumptions, such as the use of smart-cards and tamper-proof memory. Theoretical modeling and formal treatment of the issue of protected and tamper-proof storage (and not merely the more traditional protected communication lines) is a recent area of research, motivated by the advances in hardware technology and computer architectures. Reliance on some form of tamper-resistant hardware has been formally investigated in recent work, *e.g.*, the use of self-destructing capabilities for algorithmic tamper-proof security [26], of physical envelopes for collusion-free protocols [34,35,36], and of secure IPSec cards in "bump-in-the-wire" configuration [31].

THE PROBLEM, A NEW MODEL AND OUR RESULTS. This paper aims to answer the following question: *Is it possible to provide a framework for multiparty computation where concerns of confidentiality of the output can be properly expressed in the syntax of the functionality, and addressed in the definition of security?* A first difficulty arises from the fact that, for some functionalities, (part of) the output of one player may coincide with that of other players. For correctness, a secure protocol should then guarantee that this is actually the case. This implies that the corruption of one player might reveal, to the corrupting adversary, (part of) of the output of *all* players. Intuitively, this makes the notion of output-privacy hard to capture. For the sake of concreteness, in this work we will consider scenarios in which the concerns of confidentiality are focused on such common output, which we explicitly mark as *global* in the description of the functionality itself (*cf.* Definition 1). Our extended model will then provide privacy of the global output, whereas *local outputs* (*i.e.,* the part of each party's output which is not common to all players) will be protected only to the (limited) extent guaranteed by the classical setting.

The main technical hurdle toward a satisfactory definition of private-output multiparty computation lies in that confidentiality of the global output cannot seemingly be achieved with respect to traditional *active adversaries i.e.,* adversaries with total control over the behavior of (some of the) players. This is because, for correctness, each party should obtain the same global output at the end of a run of the protocol: this part of the output is thus available to the adversary, as soon as a player is corrupted. Thus, a good definition of private-output multiparty computation should be strict enough to encompass this issue, but also sufficiently general to leave room for "interesting" adversaries to consider.

The same intuition behind this impossibility issue suggests that achieving private-output security for the case of *outsider* (*i.e.* eavesdropping) adversaries may be feasible. In practice, however, security against this kind of adversaries cannot be deemed sufficient. Consequently, an important issue to investigate is to determine the highest attainable level of protection. To do so, the most natural thing is to consider adversaries having limited corrupting power, and then progressively increase the level of control that the adversary can exert over controlled parties as much as possible, without falling into a plain impossibility. We note that settings with partial corruption (either part of the time *e.g.,* forward-security [4] and key-insulation [18], or part of the space, *e.g.,* secure CPU/memory [26]) have been considered in many situations in the past for specific cryptographic tasks (*e.g.,* encryption and signature schemes), but not for general MPC.

To formalize a proper level of "limited corrupting power," we move from the following observation. Whenever the adversary corrupts a player, the available information, namely its input, its random coins, and the messages exchanged with the other parties, are already enough to derive the value of the global output. Hence, to there be any hope of protecting such value, the power of the adversary must be reduced by postulating that it cannot access (part of) these three quantities, *even for the corrupted players*.

Since communication among the parties occurs over an insecure network, removing the *received* messages from the adversary's view seems too strong an assumption. Similarly, the party's input is often decided by a higher-level protocol, and so it is quite likely to be known to the adversary.

Thus, the most natural way to restrict the view that the adversary can obtain by corrupting parties is to provide her with parties' inputs, along with all the messages exchanged with the other parties, while limiting her access to the controlled parties' randomness and outputs (which, in practice can be produced by a tamper-proof device at the user's computing environment).

Establishing how and when to limit the adversary's access to this randomness leads to new classes of adversaries of increasing capabilities (*i.e.,* the more randomness the adversary is allowed to access, the more powerful she is). Furthermore, this allows us to prove the strongest possible feasibility result. Informally, we prove that, under the assumption that enhanced trapdoor permutations exist, every function is output-privately computable even in the presence of an adversary with, basically, the sole restriction that she cannot see the global output computed by the players. In light of the above mentioned impossibility (which we formally verify), our result is optimal in the sense that it is the strongest attainable feasibility result.

2 Preliminaries

2.1 Notation

Let ℓ be a security parameter. In the following we denote with $\mathbb{N}$ the set of natural integers and with $\mathbb{R}^+$ the set of positive real numbers. We say that a function $\mathsf{negl} : \mathbb{N} \to \mathbb{R}^+$ is *negligible* if for every polynomial $p(\ell)$ there exists an $\ell_0 \in \mathbb{N}$ s.t. for all $\ell > \ell_0$, $\mathsf{negl}(\ell) \leq 1/p(\ell)$. PPT stands for Probabilistic Polynomial Time. If $\mathcal{A}$ is a PPT algorithm, we write $x \overset{r}{\leftarrow} \mathcal{A}(y)$ to denote that x is obtained by running $\mathcal{A}$ on input y. (We omit the 'r' when $\mathcal{A}$ is deterministic.) If S is a set, we denote with $x \overset{r}{\leftarrow} S$ the process of sampling x from S uniformly at random, and by $x \overset{\mathcal{D}}{\leftarrow} S$ the process of sampling x from S according to a given (efficiently samplable) distribution $\mathcal{D}$.

2.2 Global-Output Multi-party Functionalities

In the standard multi-party scenario, the goal is to realize functionalities that, given the security parameter, the inputs of the n parties and the random coins, return n (possibly different) outputs, one for each participant. In the case of global-output multi-party computation, instead, functionalities have a *global* output, which should be obtained by all the parties but at the same time ought to remain secret to the adversary, along with n *local* outputs, one for each participant:

Definition 1 (Global-Output Multi-party Functionality). A Global-Output Multi-party Functionality is a functionality of the form:

$$f : \mathbb{N} \times (\{0,1\}^*)^n \times \{0,1\}^* \to \{0,1\}^* \times (\{0,1\}^*)^n.$$

We are interested in functions that are computable in time that is polynomial in the security parameter. In particular, all inputs, as well as the global and local outputs, have length polynomial in the security parameter. We denote the local input of party P_i with x_i, and we let $\overrightarrow{x} \doteq (x_1, \ldots, x_n)$. Also, $f_0(\overrightarrow{x})$ denotes the global output, while $f_i(\overrightarrow{x})$ denotes the local outputs of party P_i.

2.3 Non-trivial Global-Output Multi-party Functionalities

Intuitively, a global-output multi-party functionality is *non-trivial*, with respect to a given distribution $\mathcal{D}$, if the global output of the function (on an input sampled from the given distribution $\mathcal{D}$) is not entirely determined by the specification of the function and of the distribution. The notion of non-trivial global-output multi-party function can be extended in a natural way, to the notion of *t-non-trivial function*, for which up to t inputs do not fully determine the output. The informal definition requires that no adversary, controlling up to t inputs and seeing the corresponding local outputs, can infer the value of the global output (see appendix A for a formal definition and a discussion about it).

In this paper, we consider protocols to compute global-output multi-party functions that are non-trivial with respect to the probability distribution used in the higher-level protocol to sample the players' inputs. In particular, and unless otherwise specified, by saying that a protocol π computes a (global-output multi-party) function f, we mean that π computes a global-output multi-party function f which is t-non-trivial (for some value of t) with respect to some probability distribution $\mathcal{D}$. (The exact value of t and the precise probability distribution $\mathcal{D}$ can additionally be specified if necessary.)

3 A Model for Private-Output Computation: The Two-Party Case

Our security definitions follow the real *vs.* ideal methodology, similarly to the case of standard multiparty computation (*cf. e.g.* [7]). Loosely speaking, such approach consists of three steps. First, one formalizes the notion of "real world" execution of a protocol. Second, an idealized computational process is defined in such a way that its security is immediately apparent. Finally, to prove a protocol secure, one shows that its execution in the real world is just as "safe" as running the idealized process.

In contrast to standard two-party computation, in our context we are interested in guaranteeing the privacy of the global output of the functionality. To this end, the idea is to define real/ideal models and real/ideal adversaries as close as possible to their counterparts in the standard two-party setting, while at the same time ensuring that an adversary with (partial) control over one of the parties cannot extract information about the function's global output from the protocol.

3.1 The Real Model: Overview

Before presenting our definitions of real-model protocols and their execution, we start with a motivating discussion about the types of attacks on the system that secure protocols should be able to withstand.

In standard multi-party computation, communication between the parties is assumed to be authenticated but not private.[1] Consequently, the weakest adversarial behavior considered in the literature consists of passively monitoring the network, without access neither to the two parties' inputs nor to their (local/global) outputs (*eavesdropping adversary*). A more powerful adversary is captured by the so-called *honest-but-curious* or *semi-honest* model, in which the adversary is still passive, but has complete access to the internal state of one of the parties. In the *augmented semi-honest* model, the adversary can additionally change the input and outputs of one of the parties, whereas in the *active* model the adversary has complete control over the corrupted party, so that in particular her interaction with the other, honest party does not necessarily follow the prescribed protocol.

Classifying the adversaries. Whereas the eavesdropping adversary models an *outsider* attack, all the other adversarial behaviors are instances of *insider* attacks. Clearly, output-privacy is impossible under any form of insider attack, since the adversary can just wait for an honest execution of the protocol to complete, and then read the resulting global output off the memory of the corrupted party. In other words, the traditional classification of adversaries is too restrictive for defining a notion of output-privacy in two-party computation.

At a closer look, the inadequacy of traditional corruption models stems from more fundamental considerations. In the case of global-output functionalities, which provide a common output to both parties, it is natural to assume that each party holds a certain level of trust on the other party (*e.g.,* that they are both interested in obtaining the correct outcome). However, such trust refers to the *human being* "on the other side of the line," and not to his/her computing environment. In particular, given the frequency and scale of worm and virus infections that afflict our computing systems nowadays, placing complete trust in the computing platform of anybody, no matter how trustworthy he/she may be as a person, is (at the very least) a very dangerous leap of faith.

To go beyond the "all-or-nothing" nature of traditional corruption models, we therefore suggest to take a more comprehensive look at the context in which the computation prescribed by the two-party protocol takes place, and try to derive a more detailed model capturing how a two-party protocol is carried out.

Components of a protocol. The outermost layer is the *application context*, which provides the input and is supposed to obtain the local and global outputs, as computed by the two-party protocol. The application context more or less corresponds to what is called the "environment" in the Universally Composable (UC) framework [8] (although in this work we only consider sequential composition of protocols).

The actual code implementing the two-party protocol (what we call a *strategy* in Section 3.2 below) is split into two well separated Turing machines: the *driver* and the *secure device*. Roughly speaking, the driver implements the high-level logic of the protocol, whereas the secure device is a piece of tamper-resistant hardware with *limited* capabilities, that carries out only the most sensitive computations related to the global output.

In our model, we see the secure device as part of the party's trusted computing base: in other words, we assume it to be bug-free, and fully complaint to its specification. Clearly, a necessary precondition for such assumption to be fulfilled is that the actual amount of code within the secure device ought to be as small as possible: otherwise, it would be unfeasible to subject the secure device to a thorough verification required to establish its correctness. The driver, on the other hand, can conceivably be a much larger piece of software, with plenty of frills and added features, which interacts with the secure device to implement the prescribed functionality.

The execution of the driver's code does not occur immediately within the application context; rather, its interaction with the application context is mediated by the *control wrapper*. The control wrapper gets the input from the application context, and provides a (possibly altered) value as input to the

[1] The recent work of [2] is an exception, but their approach cannot provide agreement, which instead is inherent in the context of global-output functionalities.

driver; at the end of the driver's execution, the control wrapper receives the local output from the driver, and forwards a (possibly altered) value as local output to the application context. The global output, however, is communicated by the secure device directly to the application context, and is thus never known to the control wrapper.

The control wrapper is also involved in the communication between the two parties in that the interaction between the two corresponding drivers is in fact carried out by the associated control wrappers. Since we are assuming the availability of authenticated channels, the control wrapper cannot alter the content of such messages; it can, however, drop them, effectively causing the execution of the protocol to abort.

Overall, the capabilities of the control wrapper amount to: 1) altering the input, possibly based on the value provided by the application context; 2) altering the local output, possibly based on the value computed by the driver; and 3) stopping the execution of the protocol at any point.

Modeling adversaries. The control wrapper aims at modeling the role that, in a real-life deployment, is played by the mechanism used within the application to invoke crypto library code (which is represented by the driver). Under normal circumstances, the control wrapper is just a dummy interface, that simply relays the values that it receives back and forth. However, by exploiting bugs in the system, a piece of malicious software (like a virus) could successfully subvert such mechanism (*e.g.*, by overwriting the entry point for the library function in the appropriate system table), and manage to intercept the communication between the application and the library code. It is thus reasonable to consider "partial" insider adversaries who are able to take over the control wrapper. We refer to this kind of adversary as *input-/output-controlling, communication-halting*; it closely resembles the *augmented semi-honest* model (*cf. e.g.*, [28, Chap. 7]).

A more sophisticated attack could conceivably replace the code for the driver altogether (rather than just intercepting all of its communication as described above). We refer to this kind of adversary as *state-controlling*. We find it reasonable to distinguish such kind of attack from the previous one, basically for the same reason that justifies the distinction between augmented semi-honest and malicious behavior for the (standard) two-party case: namely, the latter kind of attacks are more difficult to mount, for they require a deeper understanding of the details of the two-party protocol. Notice, however, that the state-controlling adversary is still restricted to use the interface provided by the tamper-resistant secure device, as specified by the protocol. Such restrictions are dropped for the case of *active* adversary, which obtains full control over the corrupted party. The active adversary is meant to represent a type of attack stronger than what any virus can mount, in that it can even break the tamper-resistance of the secure device.

We remark here that such distinction between state-controlling and active adversaries is only meaningful assuming that each secure devices embeds some kind of randomness (*e.g.*, cryptographic keys) which is certified by a common Public-Key Infrastructure (PKI). Indeed, if the operation of the secure device were only based on uncertified randomness, a state-controlling adversary in control of the driver could just never invoke the secure device, and instead pick some fresh randomness and (perfectly) simulate the execution of the secure device, according to its publicly specified capabilities. Without a PKI to certify the randomness used within each secure device, the other party would have no way to notice the deceit. Therefore, by just interacting with the other party according to the protocol, the state-controlling adversary would eventually obtain the global output, exactly as the real secure device would.[2]

The new classification. To summarize, we suggest the following types of adversaries (in order of increasing powers):

[2] In the absence of a PKI, the only difference between the state-controlling and the active adversaries would be that the former cannot cause the real secure device to output a wrong value, but this is immaterial for the security guarantees that we want.

1. *eavesdropping*: can only eavesdrop the communication between the two honest parties;
2. *input-/output-controlling, communication-halting*: can alter the input to the corrupted party, abort the execution at any moment, and modify the *local* output that is returned to the application context;
3. *state-controlling*: can compute the messages to be exchanged with the other party arbitrarily, but can only use the secure device in accordance to its interface (as prescribed by the protocol);
4. *active*: can break the tamper-resistance of the secure device, and thus can exert total control over the corrupted party during the execution of the protocol.

We believe the above classification to be reasonable, since it provides a spectrum of "partial" insider attacks, progressively bridging the gap between outsider and insider adversaries. Besides, in Section 5, we prove that output-privacy is attainable against state-controlling adversaries, which are much more powerful than eavesdropping adversaries, thus tightening the gap between possible and impossible in the context of private-output multi-party computation.

As a final note, we remark that in the following we will focus on *static* adversaries only. Thus, we assume that the adversary is initialized with the identity of the controlled party, some auxiliary information (such as the security parameter ℓ) and her own randomness $r_{\mathcal{A}}$.

3.2 The Real Model: Definitions

Definition 2. A two-party *protocol* Π is a pair of strategies (Π_1, Π_2), where each *strategy* Π_i consists of two Turing machines: the *driver* D_i and the *secure device* SD_i. Intuitively, the secure device contains a tamper-resistant memory storing data that must be kept secret in order to protect the global output. The driver D_i, instead, is an interactive Turing machine which communicates with the other party's driver, while at the same time querying the associated secure device SD_i to perform computations involving the protected data stored within SD_i. The interaction between the driver D_i and the secure device SD_i takes place via a fixed set of queries (called the *interface* of SD_i) which is determined by the specification of the secure device itself.

It is up to the protocol designer to make sure that the driver can accomplish its task with as little help from the secure device as possible: in particular, the secure device required for our completeness result (*cf.* Section 5 and Appendix E) only needs to be able to perform a handful of basic cryptographic operations, which could be implemented on low-power smart-cards. We stress that such basic set of operations should be the same for many functionalities, *i.e.*, a given implementation of the secure device should not be specific to a given functionality. In other words, once the design criteria of the secure device are specified, these criteria should allow any protocol, respecting those criteria, to compute any desired functionality, using the *same* secure device. In Appendix C we describe an illustrative example of such an implementation, that it is indeed *sufficient* for our completeness theorem. There, the randomness that has to be kept stored in the secure device is just a secret key (for a corresponding public-key cryptosystem). Moreover, our proposed secure device is required to be able to perform only a couple of very simple operations. See Appendix C for further details.

Definition 3. An (honest) *execution* of a two-party protocol begins with the two parties P_1 and P_2 receiving their inputs from the application context, and proceeds as an alternation of P_1- and P_2-rounds. In a P_i-round, driver D_i computes the next message to be sent to the other party based on its input, its randomness, the messages received so far from party P_{3-i}, and the interaction with the secure device SD_i. In the last round, each driver D_i submits a Finalize-query to the associated secure device SD_i. In response, SD_i returns a "dummy" value to D_i to acknowledge its query, and writes the *global* output on its own (tamper-resistant) global output tape. At this point, driver D_i produces the *local* output. Notice, once again, that the driver never gets to see the value of the global output.

We now introduce a few random variables related to the execution of a private-output two-party protocol. To this end, the key aspect to consider is the randomness used by each party. The whole *random tape* of P_i, denoted r_i, is broken up into two components: the *driver randomness* r_i^D (used within the driver D_i), and the *protected randomness* r_i^P (used within the secure device SD_i), which intuitively represents the piece of randomness that must be protected in order to prevent the adversary from recovering the global output:

$$r_i \doteq (r_i^D, r_i^P), \qquad \text{for } i = 1, 2.$$

The *partial view* of P_i is a random variable[3] consisting of the private input x_i, the driver randomness r_i^P, the incoming messages m_j^I and the answers a_ℓ that the secure device SD_i provides to the queries of the driver D_i:

$$\text{VIEW}_i^{\Pi,D}(x_1, x_2) \doteq (x_i, r_i^D, m_1^I, \ldots, m_t^I, a_1, \ldots, a_s), \qquad \text{for } i = 1, 2.$$

(The rationale for including the answers to the driver's queries in the partial view is that they may influence the outgoing messages produced by D_i.)

The *local output* of P_i, produced by the driver D_i at the end of the computation, is denoted with $\text{OUTPUT}_i^{\Pi,L}(x_1, x_2)$ and it is implicit in P_i's partial view.

The *complete view* of P_i is a random variable consisting of the private input x_i, the *full* randomness r_i, and all the incoming messages m_j^I generated by P_{3-i}:

$$\text{VIEW}_i^{\Pi,C}(x_1, x_2) \doteq (x_i, r_i, m_1^I, \ldots, m_t^I), \qquad \text{for } i = 1, 2.$$

Notice that it is not necessary to include the responses provided by the secure device SD_i in the complete view, since these are already determined by the protected randomness r_i^P (which is part of r_i). Thus, the complete view encompasses more information than the corresponding partial view; it is also clear that the latter can be easily derived from the former.

The *global output* of P_i, denoted $\text{OUTPUT}_i^{\Pi,G}(x_1, x_2)$, is the value output by the secure device SD_i at the end of the protocol (upon the driver's call to the Finalize-query); it is implicit in P_i's complete view.

Intuitively, at the end of a run of Π, the global outputs of the two parties should agree. We model this by introducing a random variable $\text{OUTPUT}^{\Pi,G}(x_1, x_2)$, defined as follows:

$$\text{OUTPUT}^{\Pi,G}(x_1, x_2) \doteq \begin{cases} \text{OUTPUT}_1^{\Pi,G}(x_1, x_2) & \text{if } \text{OUTPUT}_1^{\Pi,G}(x_1, x_2) = \text{OUTPUT}_2^{\Pi,G}(x_1, x_2) \\ \perp & \text{otherwise} \end{cases}$$

where $\perp$ is a special "failure" symbol.

We also denote with $\text{OUTPUT}^{\Pi}(x_1, x_2)$ the tuple:

$$(\text{OUTPUT}^{\Pi,G}(x_1, x_2), \text{OUTPUT}_1^{\Pi,L}(x_1, x_2), \text{OUTPUT}_2^{\Pi,L}(x_1, x_2)).$$

Definition 4. We say that a two-party protocol Π *implements* a private-output two-party functionality f if, for any inputs x_1, x_2 for the two parties, (honest) execution of Π results in the two secure devices outputting the same global output, and the distribution of the tuple consisting of such global output, followed by the local outputs produced by D_1 and D_2 is (computationally) indistinguishable from the distribution of the functionality f on input x_1, x_2; or, in formula:

$$\{f(x_1, x_2)\}_{x_1,x_2} \overset{c}{\equiv} \{\text{OUTPUT}^{\Pi}(x_1, x_2)\}_{x_1,x_2}$$

where $\overset{c}{\equiv}$ denotes computational indistinguishability by PPT distinguishers.

[3] All the random variables we define are over the probability space induced by the random coins of the two parties.

Real-Model Adversaries. We now discuss how we model the influence that an adversary can exert on the execution of a two-party protocol. For short, we will use $\textsc{x} \in \{\textsc{eave}, \textsc{i/o}, \textsc{state}, \textsc{act}\}$ (standing for "eavesdropping," "input-/output-controlling, communication-halting," "state-controlling," and "active," respectively) to denote the type of adversary.

A real-model adversary $\mathcal{A}$ (of any kind $\textsc{x}$) is initialized with an auxiliary input z (which includes the security parameter ℓ) and some random coins $r_{\mathcal{A}}$. Additionally, different types of adversaries obtain different information about the execution of the protocol (possibly in an interactive and adaptive way). In the following, we model this by introducing random variables $\textsc{x\_view}^{\Pi}_{\mathcal{A}(z),1}(x_1, x_2)$ and $\textsc{x\_view}^{\Pi}_{\mathcal{A}(z),2}(x_1, x_2)$ that an adversary of type $\textsc{x}$ contributes to define. Then, $\mathcal{A}$ gets to see $\textsc{x\_view}^{\Pi}_{\mathcal{A}(z),\bar{\imath}}(x_1, x_2)$, for the index $\bar{\imath}$ corresponding to the party controlled[4] by $\mathcal{A}$. Following a common practice for standard two-party computation [28], in the transcript of the protocol we replace the local output of the controlled party with the adversary's output. Also, w.l.o.g. we assume $\mathcal{A}$'s output to consist of all the information that $\mathcal{A}$ sees during a protocol's execution, namely z, $r_{\mathcal{A}}$ and $\textsc{x\_view}^{\Pi}_{\mathcal{A}(z),\bar{\imath}}(x_1, x_2)$.

Eavesdropping adversaries do not affect the execution of the protocol in any way. The view of a eavesdropping adversary consists just of the messages exchanged between the two parties:

$$\textsc{eave\_view}^{\Pi}_{\mathcal{A}(z),\bar{\imath}}(x_1, x_2) \doteq (m^I_1, \ldots, m^I_t, m^O_1, \ldots, m^O_t).$$

where m^I_j and m^O_j denotes respectively incoming and outgoing messages received by and sent from party $P_{\bar{\imath}}$.

In the presence of the other kind of adversaries, the view of both parties' maintains the same format as in the honest case:

$$\textsc{x\_view}^{\Pi}_{\mathcal{A}(z),i}(x_1, x_2) \doteq (x_i, r^D_i, m^I_1, \ldots, m^I_t, a_1, \ldots, a_s), \qquad \text{for } i = 1, 2$$

where $\textsc{x} \in \{\textsc{i/o}, \textsc{state}, \textsc{act}\}$. However, the way in which such views are computed differs substantially from the honest execution, in ways that we now describe.

In the case of an $\textsc{i/o}$-adversary, the execution begins with $\mathcal{A}$ seeing the input $x_{\bar{\imath}}$ and the randomness $r^D_{\bar{\imath}}$ for the controlled party's driver $D_{\bar{\imath}}$. Then, $\mathcal{A}$ gets to decide (based on z, $r_{\mathcal{A}}$, $x_{\bar{\imath}}$ and $r^D_{\bar{\imath}}$) the value $\hat{x}_{\bar{\imath}}$ that $D_{\bar{\imath}}$ should use in the protocol: we denote this with the notation $\hat{x}_{\bar{\imath}} \doteq \mathcal{A}(\bar{\imath}, z, r_{\mathcal{A}}, x_{\bar{\imath}}, r^D_{\bar{\imath}})$. Similarly, during the execution, $\mathcal{A}$ gets to see all the information sent to/from the driver of the controlled party $P_{\bar{\imath}}$ and can additionally stop the driver $D_{\bar{\imath}}$ at any moment, effectively causing the protocol to abort. To denote this, for any (j, ℓ)-prefix $(m^I_1, \ldots, m^I_j, a_1, \ldots, a_\ell)$ of messages (from $D_{3-\bar{\imath}}$) and answers (from $SD_{\bar{\imath}}$), we let $\mathcal{A}(\bar{\imath}, z, r_{\mathcal{A}}, x_{\bar{\imath}}, r^D_{\bar{\imath}}, m^I_1, \ldots, m^I_j, a_1, \ldots, a_\ell)$ denote a $\textsc{yes}/\textsc{no}$-value indicating whether $\mathcal{A}$ decides to abort execution at the (j, ℓ)-prefix or not. At the end of the execution, $\mathcal{A}$ obtains the local output from $D_{\bar{\imath}}$, and outputs her entire view. Notice that $\mathcal{A}$ does not obtain the global output from the secure device $SD_{\bar{\imath}}$; however, $\mathcal{A}$ can prevent $SD_{\bar{\imath}}$ from producing a global output altogether by stopping the $\mathsf{Finalize}$-query that the driver $D_{\bar{\imath}}$ issues to the $SD_{\bar{\imath}}$ at the end of all its computation.

In the case of a state-controlling adversary $\mathcal{A}$ (controlling party $P_{\bar{\imath}}$), $\mathcal{A}$ gets to play the role of the driver $D_{\bar{\imath}}$. Thus, $\mathcal{A}$ obtains the input $x_{\bar{\imath}}$, and all $P_{\bar{\imath}}$-round are carried out according to $\mathcal{A}$'s (rather than $D_{\bar{\imath}}$'s) code. We remark that $\mathcal{A}$ can also interact with the secure device, querying it on arbitrary values, but such interaction still has to occur using the interface provided by $SD_{\bar{\imath}}$. As for the notation, for a $\textsc{state}$-adversary $\mathcal{A}$, we use $\mathcal{A}(\bar{\imath}, z, r_{\mathcal{A}}, x_{\bar{\imath}}, r^D_{\bar{\imath}}, m^I_1, \ldots, m^I_j, a_1, \ldots, a_\ell)$ to denote the next message that $\mathcal{A}$ wishes to send to $P_{3-\bar{\imath}}$ (or the next query to be asked to $SD_{\bar{\imath}}$, whichever is appropriate).

In the case of an active adversary, all computations occur as for the state-controlling adversary, except that now $\mathcal{A}$ can additionally play the role of the secure device $SD_{\bar{\imath}}$ (as well as the driver's of party $P_{\bar{\imath}}$), so that now the global output is exposed to $\mathcal{A}$. Notation remains as in the $\textsc{state}$ case, except that there is no secure device to query.

[4] Although a $\textsc{eave}$-adversary does not actually control any party, for notational convenience we assume that one of the parties (*e.g.*, party P_1) is controlled in a "null" way.

3.3 The Ideal Model

We now provide definitions of private-output two-party *ideal process* and of *ideal-model adversaries*. By analogy to the real model (*cf.* Section 3.1), each party P_i in the ideal model is a pair of "dummy" Turing machines: a driver D_i and a secure device SD_i. The driver has access to the security parameter ℓ and is attached to two tapes: the *input* tape and the *local output* tape. The secure device can write to the *global output* tape, and it exports a $\mathsf{Finalize}$-query (that the driver may or may not call during the ideal execution).

As in [7], the ideal process is parameterized by the functionality f that the parties wish to evaluate. Recall (*cf.* Section 2) that a two-party functionality, in the private-output setting, is defined as $f : \mathbb{N} \times (\{0,1\}^*)^2 \times \{0,1\}^* \to \{0,1\}^* \times (\{0,1\}^*)^2$. In the ideal model we assume the existence of an incorruptible probabilistic trusted third party $\mathcal{T}$, which knows f and ℓ.

An ideal-process adversary $\mathcal{B}$ is an interactive PPT Turing machine which may influence the behavior of the controlled party in several ways. Since we are considering the static case, $\mathcal{B}$ is initialized with the identity of the controlled party (if any), some random input $r_{\mathcal{B}}$, along with an auxiliary input (which includes the security parameter ℓ).

In what follows, we describe the execution of the ideal process assuming the presence of an *input-/output-controlling, communication-halting* adversary controlling one of the two party. Note, however, that the definition remains the same in the case of *state-controlling* adversaries—being in the ideal model, there is no communication between the drivers that could be altered by such adversaries. Furthermore, this definition also applies to the *active* scenario: in particular, an ideal active adversary is not conceded access to the secure device since, in the ideal model, the tamper-resistance of the secure device ought to be ideal *i.e.*, unbreakable. In the case of *eavesdropping* adversary, instead, none of the parties is ever controlled by the adversary. Thus, the following definition applies to adversaries of any type $\mathrm{X} \in \{\mathrm{EAVE}, \mathrm{I/O}, \mathrm{STATE}, \mathrm{ACT}\}$.

Input Stage

Inputs to the parties: Each party obtains an input x_i. The controlled party communicates its input to the adversary. Recall that the parties do not have random input in the ideal model.

Inputs to the trusted party: An honest party always sends x_i to the trusted party $\mathcal{T}$. A controlled party may, depending on the adversary's strategy (which is based on $x_{\bar{\imath}}$, as well as on the auxiliary input and her randomness), either abort or send some $\widehat{x_{\bar{\imath}}} \in \{0,1\}^{|x_{\bar{\imath}}|}$ to $\mathcal{T}$.

Actual Computation

$\mathcal{T}$ provides the local output to the first party: $\mathcal{T}$ evaluates the functionality on the received inputs (and fresh randomness, in case f is a probabilistic functionality), and sends the first local output to the first party's driver. If $\mathcal{T}$ received only one input, then it sends $\perp$ to both parties' drivers and secure devices.

$\mathcal{T}$ provides the local output to the second party: If the first party is controlled, it hands off its local output (just obtained from $\mathcal{T}$) to the adversary. Then $\mathcal{B}$ may, depending on all the information received so far, decide to "stop" the trusted party. In this case, $\mathcal{T}$ sends $\perp$ to the second party's driver and to both parties' secure devices. Otherwise, $\mathcal{T}$ sends the second local output to the second party's driver.

$\mathcal{T}$ provides the global output to the first party: If the second party is controlled, it hands off its local output (just obtained from $\mathcal{T}$) to the adversary. Then $\mathcal{B}$ may, depending on all the information received so far, decide to "stop" the trusted party. In this case, $\mathcal{T}$ sends $\perp$ to both parties' secure devices. Otherwise, $\mathcal{T}$ sends the global output to the first party's secure device.

$\mathcal{T}$ provides the global output to the second party: If the first party is controlled, the adversary may, depending on all the information received so far,[5] decide to "stop" the trusted party. In this case, $\mathcal{T}$ sends $\perp$ to the second party's secure device. Otherwise, $\mathcal{T}$ sends the global output to the second party's secure device.

[5] Note that the information available to the adversary at this point is the same as that available in the second step of the Actual Computation stage, since the global output is stored in the secure device and is therefore

Output Stage

Local output: An honest party's driver always writes the local output received from the trusted party to its local output tape. A controlled party's driver may write an arbitrary value to its local output tape, depending on the strategy of the adversary.

Global output: An honest party's driver always invokes the Finalize-query on the secure device, which causes it to write (on its global output tape) whatever value received from $\mathcal{T}$. A controlled party's driver may or may not invoke the Finalize-query on the corresponding secure device. If such query is not invoked, we assume that the global output tape of this party contains $\perp$ (as an initial default value). If the Finalize-query is invoked, the secure device of the controlled party writes the global output received from $\mathcal{T}$ to its global output tape.

Ideal-Model Adversaries. The ideal-model adversary affecting the execution of the ideal process as described above can be captured by the following definition:

Definition 5 (Ideal-Model Adversary). An ideal-model adversary is a PPT Turing machine $\mathcal{B}$ such that, if $\bar{\imath}$ is the index of the controlled party, $x_{\bar{\imath}}$ is the corresponding input, and z and $r_{\mathcal{B}}$ are $\mathcal{B}$'s auxiliary input and random coins, then:

- $\mathcal{B}(\bar{\imath}, z, r_{\mathcal{B}}, x_{\bar{\imath}})$ outputs an altered input $\hat{x}_{\bar{\imath}} \in \{0,1\}^{|x_{\bar{\imath}}|}$ for $P_{\bar{\imath}}$;
- $\mathcal{B}(\bar{\imath}, z, r_{\mathcal{B}}, x_{\bar{\imath}}, y_{\bar{\imath}})$ outputs $(\hat{y}_{\bar{\imath}}, b_{\text{halt}}, b_{\text{fin}}) \in \{0,1\}^* \times \{\text{EARLY}, \text{LATE}, \text{NEVER}\} \times \{\text{YES}, \text{NO}\}$, where:
  1. $\hat{y}_{\bar{\imath}}$ is the altered output for party $P_{\bar{\imath}}$;
  2. b_{halt} denotes $\mathcal{B}$'s decision on aborting the execution (EARLY = abort right after getting $y_{\bar{\imath}}$; LATE = abort after $SD_{\bar{\imath}}$ gets y_0; NEVER = do not abort);
  3. b_{fin} specifies whether or not $\mathcal{B}$ wishes to allow $SD_{\bar{\imath}}$ to output y_0.

3.4 The Security Definition

Let $f : \mathbb{N} \times (\{0,1\}^*)^2 \times \{0,1\}^* \to \{0,1\}^* \times (\{0,1\}^*)^2$ be a two-party functionality and Π be a private-output two-party protocol implementing f. Let $\text{X} \in \{\text{EAVE}, \text{I/O}, \text{STATE}, \text{ACT}\}$, and let $\mathcal{A}$ be a real-model adversary of type X, and $\mathcal{B}$ be an ideal-model adversary.

Definition 6. The *joint real-model execution of Π under $\mathcal{A}$ controlling party $P_{\bar{\imath}}$ on input pair (x_1, x_2) and auxiliary input z*, is defined as:

$$\text{X\_REAL}^{\Pi}_{\mathcal{A}(\bar{\imath},z)}(x_1,x_2) \doteq ((\text{OUTPUT}^{\Pi,G}_1(x_1,x_2), \Gamma_1), (\text{OUTPUT}^{\Pi,G}_2(x_1,x_2), \Gamma_2))$$

where we have $\Gamma_{\bar{\imath}} \doteq (z, r_{\mathcal{A}}, \text{X\_VIEW}^{\Pi,L}_{\bar{\imath}}(x_1,x_2))$, $\Gamma_{3-\bar{\imath}} \doteq \text{OUTPUT}^{\Pi,L}_{3-\bar{\imath}}(x_1,x_2)$, and the random variables $\text{X\_VIEW}^{\Pi,L}_1(x_1,x_2)$, $\text{X\_VIEW}^{\Pi,L}_2(x_1,x_2)$, $\text{OUTPUT}^{\Pi,G}_1(x_1,x_2)$, $\text{OUTPUT}^{\Pi,G}_2(x_1,x_2)$, and $\text{OUTPUT}^{\Pi,L}_{3-\bar{\imath}}(x_1, x_2)$ refer to the same execution of Π in the presence of $\mathcal{A}(z)$ controlling party $P_{\bar{\imath}}$, and are computed as described in Section 3.

Definition 7. The *joint real-model execution of Π under $\mathcal{A}$* is defined as the ensemble:

$$\text{X\_REAL}^{\Pi}_{\mathcal{A}} \doteq \{\text{X\_REAL}^{\Pi}_{\mathcal{A}(\bar{\imath},z)}(x_1,x_2)\}_{x_1,x_2,\bar{\imath},z}$$

Definition 8. The *joint ideal-model execution of f under $\mathcal{B}$ controlling party $P_{\bar{\imath}}$ on input pair (x_1, x_2) and auxiliary input z*, is defined as:

$$\text{IDEAL}^{f}_{\mathcal{B}(\bar{\imath},z)}(x_1,x_2) \doteq ((y_0^1, \hat{y}_1), (y_0^2, \hat{y}_2))$$

where:

$$\hat{x}_{\bar{\imath}} \xleftarrow{r} \mathcal{B}(\bar{\imath}, z, r_{\mathcal{B}}, x_{\bar{\imath}}),\ \ \hat{x}_{3-\bar{\imath}} \leftarrow x_{3-\bar{\imath}},\ \ (y_0, y_1, y_2) \xleftarrow{r} f(\hat{x}_1, \hat{x}_2),\ \ (\hat{y}_{\bar{\imath}}, b_{\text{halt}}, b_{\text{fin}}) \xleftarrow{r} \mathcal{B}(\bar{\imath}, z, r_{\mathcal{B}}, x_{\bar{\imath}}, y_{\bar{\imath}}),$$

$$\hat{y}_{3-\bar{\imath}} \leftarrow \begin{cases} \perp & \text{if } \bar{\imath} = 1 \wedge b_{\text{fin}} = \text{EARLY} \\ y_{3-\bar{\imath}} & \text{otherwise} \end{cases} \quad y_0^{\bar{\imath}} \leftarrow \begin{cases} \perp & \text{if } b_{\text{fin}} = \text{NO} \\ y_0 & \text{otherwise} \end{cases} \quad y_0^{3-\bar{\imath}} \leftarrow \begin{cases} \perp & \text{if } b_{\text{halt}} \neq \text{NEVER} \\ y_0 & \text{otherwise} \end{cases}$$

not revealed to the adversary. We prefer to split the strategy of the adversary in two steps only for clarity of exposition.

Definition 9. The *joint ideal-model execution of f under $\mathcal{B}$* is defined as the ensemble:

$$\text{IDEAL}^f_{\mathcal{B}} \doteq \{\text{IDEAL}^f_{\mathcal{B}(\bar{\imath},z)}(x_1, x_2)\}_{x_1,x_2,\bar{\imath},z}$$

Definition 10. Protocol Π is said to *output-privately* compute f with respect to X-adversaries (where X is one of $\{\text{EAVE}, \text{I/O}, \text{STATE}, \text{ACT}\}$) if Π implements f (in the sense of Definition 4), and for every PPT real-model adversary $\mathcal{A}$ of type X, there exists a PPT ideal-model adversary $\mathcal{B}$, such that:

$$\text{IDEAL}^f_{\mathcal{B}} \overset{c}{\equiv} \text{X\_REAL}^{\Pi}_{\mathcal{A}}.$$

3.5 Further Comments

It is quite easy to see that our definition can be viewed as a generalization of that by Canetti [7]. Such comparison is discussed in appendix B. Also the extension to n-party functionalities is quite straightforward.

4 Impossibility Result

In this section, we formally state the impossibility result briefly discussed in Section 3. This is interesting because it shows that our definition formally captures the intuition that private-output multiparty computation should be impossible when facing active adversaries.

The following theorem (whose proof is included in Appendix D) is given for the two-party case, but it can be easily generalized to the multi-party case.

Theorem 11. *Let f be any 1-non-trivial two-party function (with respect to some distribution $\mathcal{D}$) and π be any two-party protocol that computes f. Then there exists an active adversary $\mathcal{A}$ (controlling one player) such that, for any ideal process active adversary $\mathcal{B}$, the two distribution ensembles $\text{IDEAL}^f_{\mathcal{B}}$ and $\text{ACT\_REAL}^{\pi}_{\mathcal{A}}$ are not computationally indistinguishable, that is, there cannot exist any π that can output-privately compute f in the presence of $\mathcal{A}$.*

5 A Completeness Theorem for State-Controlling Adversaries

Given the impossibility result for active adversaries (*cf.* Section 4), in this section we focus on the problem of realizing secure private-output two-party computation in the presence of state-controlling adversaries. In particular, we show that if enhanced trapdoor permutations [29,28] exist, then any functionality f can be privately computed in the presence of computationally bounded state-controlling adversaries.

Theorem 12. *Let ℓ be a security parameter. If enhanced trapdoor permutations exist, then for any private-output two-party functionality f, there exists a protocol π output-privately implementing f with respect to* STATE*-adversaries.*

An overview of the resulting construction and proof are included in Appendix E. For the sake of modularity, we prove the theorem for the case of I/O-adversaries (and then we use standard techniques to extend the proof to the case of STATE-adversaries). In a nutshell our proof goes as follows. As a first step we describe an adequate hybrid model. Next we show that this model can be used to *reduce* the problem to that of realizing deterministic functionalities. Finally, we exhibit a construction for any deterministic function seen as an arithmetic circuit.

Here we note that as an interesting by-product we get a (sequential) composition theorem (Theorem 15) for the framework of private-output two-party computation. See Appendix E for full details.

Theorem 13 (informal). *Assume a protocol Π^g output-privately computes a functionality g, and a protocol $\Pi^{f|g}$ output-privately computes f using ideal calls to g. Then the composed protocol $\Pi^f \doteq \Pi^{f|g} \circ \Pi^g$ output-privately computes f in the real model.*

References

1. R. Bar-Yehuda, B. Chor, E. Kushilevitz, and A. Orlitsky. Privacy, Additional Information, and Communication. *IEEE IT* 39(6):1930-1943, 1993.
2. B. Barak, R. Canetti, Y. Lindell, R. Pass and T. Rabin Secure Computation without Authentication. *Crypto'05*, pp. 361–377. Springer.
3. M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. *STOC '88*, pp. 1–10. ACM.
4. M. Bellare and S. Miner. A Forward-Secure Digital Signature Scheme. *Crypto '99*, pp. 431–448. Springer.
5. R. Canetti, R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Adaptive security for threshold cryptosystems. *Crypto '99*, pp. 98–115. Springer.
6. R. Canetti. Studies in Multi-party computation. PhD thesis, 1995.
7. R. Canetti. Security and Composition of Multi-party Cryptographic Protocols. *J. of Cryptology*, 13(1):143–202, 2000.
8. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. *FOCS '01*, pp. 136–145. IEEE. Full version available from `http://eprint.iacr.org/2001/067/`.
9. R. Canetti, I. Damgård, S. Dziembowski, Y. Ishai and T. Rabin. On Adaptive vs. Non-Adaptive Security of Multi-party Protocols. *Eurocrypt'2001*, pp. 262–279. Springer.
10. R. Canetti, U. Feige, O. Goldreich and M. Naor Adaptively Secure Computation *STOC '96*, pp. 639–648. ACM.
11. R. Canetti and R. Gennaro. Incoercible Multiparty Computation. *FOCS '96*, pp. 504–513. IEEE.
12. R. Canetti, S. Halevi and A. Herzberg. Maintaining Authenticated Communication in the Presence of Break-Ins. *J. of Cryptology*, 13(1):61–106, 2000.
13. R. Canetti, E. Kushilevitz, and Y. Lindell. On the Limitations of Universally Composable Two-Party Computation Without Set-Up Assumptions. *Eurocrypt' 03*, pp. 68–86. Springer.
14. D. Chaum, C. Crepeau, and I. Damgard. Multiparty Unconditionally Secure Protocols. *STOC '88*, pp.11–19. ACM.
15. B. Chor and E. Kushilevitz. A zero-one law for boolean privacy. *SIAM J. on Discrete Mathematics*, 4(1):36–47, 1991. Earlier version in *STOC '89*, pp. 62–72. ACM.
16. R. Cramer, I. Damgård, S. Dziembowski, M. Hirt, and T. Rabin. Efficient multi-party computations secure against an adaptive adversary. *Eurocrypt '99*, pp. 311–326. Springer.
17. W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE IT*, 22(6):644–654, 1976.
18. Y. Dodis, J. Katz, S. Xu and M. Yung. Key-Insulated Public Key Cryptosystems. *Eurocrypt '02*, pp. 65–82. Springer.
19. D. Dolev, C. Dwork, O. Waarts and M. Yung. Perfectly Secure Message Transmission. *J. of the ACM*, 40(1):17–47, 1993.
20. U. Feige, J. Kilian, and M. Naor. A minimal model for secure computation. *STOC '94*, pp. 554–563. ACM.
21. M. Franklin and R. Wright. Secure communication in minimal connectivity models. *J. of Cryptology*, 13(1):9–30, 2000.
22. M. Franklin and M. Yung. Secure hypergraphs: Privacy from partial broadcast. *STOC '95*, pp. 36–44. ACM.
23. M. Franklin and M. Yung. Communication complexity of secure computation. *STOC '92*, pp. 699–710. ACM.
24. R. Gennaro, Y. Ishai, E. Kushilevitz and T. Rabin. On 2-Round Secure Multi-Party Computation. *Crypto '02*, pp. 178-193. Springer.
25. R. Gennaro, Y. Ishai, E. Kushilevitz and T. Rabin. The Round Complexity of verifiable Secret Sharing and Secure Multicast. *STOC '01*, pp. 580-589, ACM.
26. R. Gennaro, A. Lysysanskaya, T. Malkin, S. Micali and T. Rabin. Algorithmic Tamper-Proof Security: Theoretical Foundations for Security against Hardware Tampering. *TCC '04*, pp. 258–277. Springer.
27. R. Gennaro, M. Rabin and T. Rabin Simplified VSS and Fast-track Multiparty Computations with Applications to Threshold Cryptography. *PODC '98*, pp. 101–111. ACM.
28. O. Goldreich. Foundations of cryptography, Volume 2. Basic Applications. 2004.
29. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game — a completeness theorem for protocols with honest majority. *STOC '87*, pp. 218–229. ACM.
30. S. Goldwasser and S. Micali. Probabilistic Encryption. *JCSS*, 28(2):270–299, 1984.
31. S. Halevi, D. Naor and P. Karger. Enforcing Confinement in Distributed Storage and a Cryptographic Model for Access Control `http://eprint.iacr.org/2005/169/`, 2005.

32. M. Hirt and U. Maurer. Complete Characterization of Adversaries Tolerable in General Multiparty Computation. *PODC '97*, pp. 25-34. ACM.
33. M. Hirt and U. Maurer. Player Simulation and General Adversary Structures in Perfect Multi-party Computation. *J. of Cryptology*, 13(1):31–60, 2000.
34. M. Lepinski, S. Micali, C. Peikert, and A. Shelat. Completely Fair SFE and coalition-safe cheap talk. *PODC '04*, pp. 1-10. ACM.
35. M. Lepinski, S. Micali, and A. Shelat. Fair Zero Knowledge. *TCC 05*, pp. 245-263. Springer.
36. M. Lepinski, S. Micali, and A. Shelat. Collusion-free Protocols. *STOC '05*, pp. 543-552. ACM.
37. S. Micali and P. Rogaway. Secure Computation (Unpublished Manuscript). Preliminary version in *Crypto '92*, pp. 392–404. Springer.
38. T. Rabin and M. Ben-Or Verifiable Secret Sharing and Secure Computation with Honest Majority. *STOC '89*, pp. 73–85. ACM.
39. A. Yao. Protocols for Secure Computations. *FOCS '82*, pp. 160-164. IEEE.

A Formal Definition of Non-Triviality

In Section 2, we informally introduced the notion of t-non-triviality for global-output multi-party functionalities as requiring that no adversary controlling up to t of the inputs, and seeing the corresponding local outputs, can completely determine the global output, when the honest parties' inputs are sampled according to a given distribution.

More precisely, let $\mathcal{D}$ be the distribution according to which players' inputs are assumed to be sampled (notice that this allows the inputs to be arbitrarily correlated). Let $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ be a probabilistic, polynomially bounded, two-stage adversary controlling up to t players. In the following $\mathcal{A}$ is allowed to access the private inputs and the local outputs of t players. For simplicity, and without loss of generality, we will assume that $\mathcal{A}$ controls players $P_1, \ldots, P_t$.

Definition 14 (Non-triviality). Let $\mathcal{D}$ be a samplable distribution. We say that a global-output multi-party function f is t-non-trivial with respect to $\mathcal{D}$, if no PPT adversary $\mathcal{A}$ can predict the global output of the function with probability negligibly close to 1, when the inputs of the players are chosen according to $\mathcal{D}$. More formally, we require that for any $\mathcal{A}$ and sufficiently large ℓ,

$$\Pr\left[y_0 = y_0' \;\middle|\; \begin{array}{l} (x_1, \ldots, x_n) \overset{\mathcal{D}}{\leftarrow} \{0,1\}^{\ell}; \rho \overset{r}{\leftarrow} \{0,1\}^{\ell_1}; \\ (\hat{x}_1, \ldots, \hat{x}_t, \mathrm{STATE}) \overset{r}{\leftarrow} \mathcal{A}_1(\ell, x_1, \ldots, x_t); \\ (y_0, y_1, \ldots, y_n) \leftarrow f(\ell, \hat{x}_1, \ldots, \hat{x}_t, x_{t+1}, \ldots, x_n, \rho); \\ y_0' \overset{r}{\leftarrow} \mathcal{A}_2(\mathrm{STATE}, y_1, \ldots, y_t) \end{array} \right] \leq 1 - \mu(\ell)$$

where ℓ_1 is a parameter polynomially related to ℓ, $\mu(\cdot)$ is a non-negligible function, and the probability is over the random coin tosses of $\mathcal{A}$, the random choice of ρ in $\{0,1\}^{l_1}$ and the random choice of the inputs according to $\mathcal{D}$.

We emphasize that the notion of non-triviality is deeply different from that of t-private functions as defined by Kushilevitz *et al*. [15,1], in which the adversary, when seeing up to t inputs, *as well as the output*, tries to get additional information about the other inputs. Our notion is instead related to the notion of *unpredictable function*, as defined by Lindell *et al.* in [13]. Since our setting deals with global-output multi-party functionality, the notion of non-triviality needs to take into account the distinction of global and local outputs, and is consequently more stringent. Indeed, a t-non-trivial function is required to be unpredictable with respect to a *distribution* (rather than just to some specific inputs). Moreover, in our context the adversary should not be able to predict the global output y_0, even if she is given access to the local outputs $y_1, \ldots, y_t$ of the controlled parties.
Remark. Note that we refer to non-triviality as a property of a function with respect to a distribution. Intuitively, this is because efficiently computable multi-party functions may not be non-trivial with respect to all (efficiently samplable) distributions (*e.g.*, consider a deterministic functionality and an input distribution that, conditioned on any fixed values for the first t inputs $x_1, \ldots, x_t$, assigns the entire marginal probability mass to a single tuple for $x_{t+1}, \ldots, x_n$).

B Further Comments on the Model

B.1 Extension to n-Party Functionalities, $n > 2$

The definition given in Section 3 for the two-party setting can be easily generalized to the multi-party case. For the case of eavesdropping adversaries the definition of secure private-output multi-party computation is pretty much the same as in the two-party case. Following [28], we consider two different scenarios to model general adversarial behaviors. The first model is very similar to the two-party case: the adversary is permitted to control even a majority of the participants and early abortion is allowed. In the second model the adversary is only allowed to corrupt a strict minority of the players and early abortion can actually be prevented.

B.2 Comparison with Canetti's Definition

We now briefly discuss the relation between our approach and the one by Canetti [7], and argue that our definition actually generalizes the one presented in [7] for the setting of standard multi-party computation secure against static adversaries.

Let us briefly recall the ideal process as described in [7]. The input-substitution phase is basically identical to ours: the ideal process adversary sees (and possibly alters) the inputs of the corrupted parties. In the computation stage the parties hand the inputs to $\mathcal{T}$. The latter performs the required computation (*i.e.*, $\mathcal{T}$ evaluates the f on the given inputs) and sends back to each player the corresponding output value y_i. The output phase goes exactly as our *local* output phase, whereas no global output is defined in [7].

In our model, on the other hand, after terminating the actual computation stage, the trusted party $\mathcal{T}$ first hands to each player P_i his *local* output y_i, and then (in the *global* output stage) hands to each P_i the (common) value y_0. Now, if we set $y_0 = \perp$, our definition becomes identical to the one proposed by Canetti. In particular, the separation of each party's strategy in two components (the driver and the secure device) becomes immaterial—when there is no global output y_0, there is no information requiring the additional security provided by the secure device, which is thus unnecessary. For the same reason, in such case the definitions of real-model state-controlling and active adversary coincide, and the impossibility result of Section 4 no longer applies (since a functionality whose global output is always $\perp$ cannot be non-trivial in the sense of Definition 14).

Our definition, however, is more general, because it allows to consider more general scenarios such as those where no local data is sent to the players (*i.e.*, $y_i = \perp$) or those where an "hybrid" solution is required (*i.e.* where both y_0 and y_i are different from $\perp$).

C Informal Description of the Secure Devices' Capabilities

In this section, we informally describe a simple interface for the secure devices that is actually sufficient to prove our completeness theorem.

We assume that each secure device comes with a public key pk (for a corresponding encryption scheme) which is certified by an adequate authority and that is made available, together with its certificate, by the driver that uses it. The corresponding secret key sk is the only secret that the secure device is required to store.

The driver communicates with the secure device through the following set of queries. Let $\mathcal{M}$ be the message space for the public-key cryptosystem used by the secure device. For simplicity, we assume that $\mathcal{M}$ is the same for all the parties participating to the protocol (*i.e.*, drivers and secure devices).

Decrypt&Combine: This query has the following syntax. It takes an input of the form $(n, \odot, (C_1, \ldots, C_n), a)$, where:

- $\odot$ is an associative operator (typically $\oplus$, $+$ or $\cdot$);
- n is an integer such that if $n \neq 0$, $(C_1, .., C_n)$ is a vector of ciphertexts, produced using the secure device's public key, of messages $m_1, .., m_n \in \mathcal{M}$;
- k is an integer; and

– a is an element of $\mathcal{M}$.

The secure device, upon receiving such a query proceeds as follows.

– If $n > 0$, it decrypts the received ciphertexts and combines them all with a using the $\odot$ operator.

– If $n = 0$, the result is simply a.

Let y_0 be the obtained result: the secure device sends a dummy value to the driver to acknowledge its query and stops.

Finalize**:** Whenever the secure device receives a Finalize-query from driver D, it returns to D a pointer p and writes the global output y_0 (if this is defined) on its own (tamper resistant) output tape. The pointer p is interpreted as follows. It is used by the driver to have implicit access to the global output. More precisely, the driver cannot use the pointer to actually access (*i.e.* see) the global output. Rather it uses p to be able to perform future operations using y_0 (for example if y_0 has to be used as input for future protocols).

If a Finalize-query is asked after several Decrypt&Combine-queries, the secure device simply writes on its output tape all the global outputs computed so far, and sends several pointers to the driver.

D Proof of Theorem 11

To prove the theorem we show an active, polynomially bounded, real-model adversary $\mathcal{A}$ such that for any active, polynomially bounded, ideal-process adversary $\mathcal{B}$, the produced distributions are efficiently distinguishable. Let π be any two-party protocol for computing f. At the beginning of the protocol, $\mathcal{A}$ obtains the input of the corrupted player (without loss of generality we can assume that P_1 is the corrupted player). Next $\mathcal{A}$ merely monitors the internal memory of P_1 during the entire execution of the protocol. When the output y_0 of f is available, $\mathcal{A}$ reads it and, at the end of the process, she outputs y_0 together with all the informations gathered during the execution of π. Notice that since we are assuming that f is computed by π, the output of the honest player will be (or at least it will contain) y_0.

Let $\mathcal{I}_{\mathcal{A}}$ be the set of active, ideal adversaries whose running time is polynomially related to the running time of $\mathcal{A}$. Notice that the crucial difference between $\mathcal{A}$ and *any* adversary $\mathcal{B} \in \mathcal{I}_{\mathcal{A}}$, is that $\mathcal{A}$ can access the input, the local output and the global output of the corrupted player, while $\mathcal{B}$ can see only the input x_1 and the local output y_1 (which are assumed not to disclose y_0 due to the 1-non-triviality of f). This implies that $\mathcal{B}$ cannot "guess" y_0 with probability higher than $1 - \mu(\ell)$ for some non-negligible quantity $\mu(\ell)$ (ℓ is the usual security parameter).

Therefore we can construct a polynomial time distinguisher Δ for the two distribution ensembles $\text{ACT\_REAL}^{\pi}_{\mathcal{A}}$ and $\text{IDEAL}^{f}_{\mathcal{B}}$ as follows. Δ receives on input a challenge sampled according to either the $\text{IDEAL}^{f}_{\mathcal{B}}$ distribution ensemble or the $\text{ACT\_REAL}^{\pi}_{\mathcal{A}}$ one (depending on some secret bit b). Then Δ simply checks whether the output of the adversary equals that of the honest player or not. If this is the case, it outputs 1 (as its guess for the real process), otherwise it outputs 0.

Let us analyze the probability of success of Δ. We define the advantage "Real/Ideal" of Δ as follows

$$\mathsf{Adv}^{\mathsf{RI}}(\Delta) = |\Pr[\Delta \to 1|b = 1] - \Pr[\Delta \to 1|b = 0]| \,.$$

Our goal is to prove that this quantity is non negligible. Clearly, one has that $\Pr[\Delta \to 1|b = 1] = 1$. On the other hand if the challenge comes from an ideal execution of the protocol, then any ideal adversary $\mathcal{B}$ can produce a distribution which is indistinguishable from the real one only if it "guesses" y_0 correctly. Thus

$$\mathsf{Adv}^{\mathsf{RI}}(\Delta) = 1 - \Pr[\Delta \to 1|b = 0] = 1 - \Pr[\mathcal{B}(x_1, y_1) \to y_0] > \mu(\ell),$$

where the last inequality comes from the 1-non-triviality of the functionality f. This concludes the proof. $\square$

E Proof of Theorem 12

The construction of a secure protocol for an output-private two-party functionality f is based on the same technical tools as for the case of standard two-party computation. In particular, below we present the outline of the proof of a variant of Theorem 12 for the case of I/O-adversaries; its extension to the case of STATE-adversaries relies on the use of authentication, commitments and zero-knowledge proofs in a way that is essentially identical to the compilation of private two-party protocols into secure ones. We defer the details to a full version of this paper.

E.1 The Hybrid Model

In this section, we describe the so-called "hybrid model" we are going to use in our proof. The motivation for the model, however, is more general; specifically, the goal is to enable composition of cryptographic protocols with output privacy, following the modular approach as described by [7] for designing such protocols. The hybrid model we are about to define is very similar to that used in standard multi-party computation, so we just recall the main lines here. For more technical details, we refer the reader to [7].

Let g be a private-output two-party functionality. A protocol for the g-hybrid model is defined as in Definition 2, augmented with a mechanism for "g-oracle calls". Such oracle is available to both parties, and must be called with two inputs, one from each party; the result of the oracle call is a pair of local outputs (one for each party's driver), along with a global output to be delivered to both parties' secure devices. In particular, each party's driver has an additional read/write oracle-tape, while secure devices have an extra read-only oracle-tape. Oracle calls are performed as follows:

1. The invoking party's driver (for instance, D_1) writes its input x_1^g for the g-functionality on the oracle tape, and then sends a special message oracle request to the other driver (to notify that an oracle invocation has been initiated).
2. In response, the other driver writes on its own oracle tape its input x_2^g to the g-functionality, and answers with another special oracle call message.
3. Both drivers inform the associated secure device that an oracle call is occurring. This again, is modeled with a special query to the secure devices.
4. At this point, the oracle is invoked (intuitively, the functionality g is evaluated on input (x_1^g, x_2^g)), and the local outputs y_1^g and y_2^g are written on the oracle tapes of the call initiator and responder, respectively. Additionally, the global output of the functionality y_0^g is written on the oracle tapes of both secure devices.
5. Finally, each secure device SD_i returns to the driver D_i a special *handle*, or *pointer* to the global output y_0^g. We stress that such pointer does not allow D_i to read the value of y_0^g, but is needed to enable D_i to make future references to such value in subsequent interactions with SD_i.

The notions of *protocol execution* and *output-private views* w.r.t. X-adversaries (X $\in$ {EAVE, I/O, STATE, ACT}), as defined in Section 3, can be directly extended to the g-hybrid model to defined the *output-private security* of the protocol in the hybrid model; we omit the details for conciseness. Additionally, one can define a notion of *output-private reduction*: a protocol $\Pi^{f|g}$ output-privately reduces f to g w.r.t. X-adversaries, if $\Pi^{f|g}$ output-privately computes f (w.r.t. X-adversaries) in the g-hybrid model.

E.2 Composition of Protocols under I/O-adversaries.

In this paragraph we show that the standard notion of protocol composition still applies in our setting.

Let Π^g be a protocol implementing g, and $\Pi^{f|g}$ be a protocol implementing f in the g-hybrid world. The composed protocol $\Pi^f \doteq \Pi^{f|g} \circ \Pi^g$ is obtained from $\Pi^{f|g}$, by replacing invocations of the g functionality in $\Pi^{f|g}$ with real-world executions of Π^g. Notice that both $\Pi^{f|g}$ and Π^g are pairs, consisting of a driver and a secure device; thus, Π^f will also be a pair, whose driver D^f may ask queries (namely, Decrypt&Combine and Finalize) to the secure device SD^f.

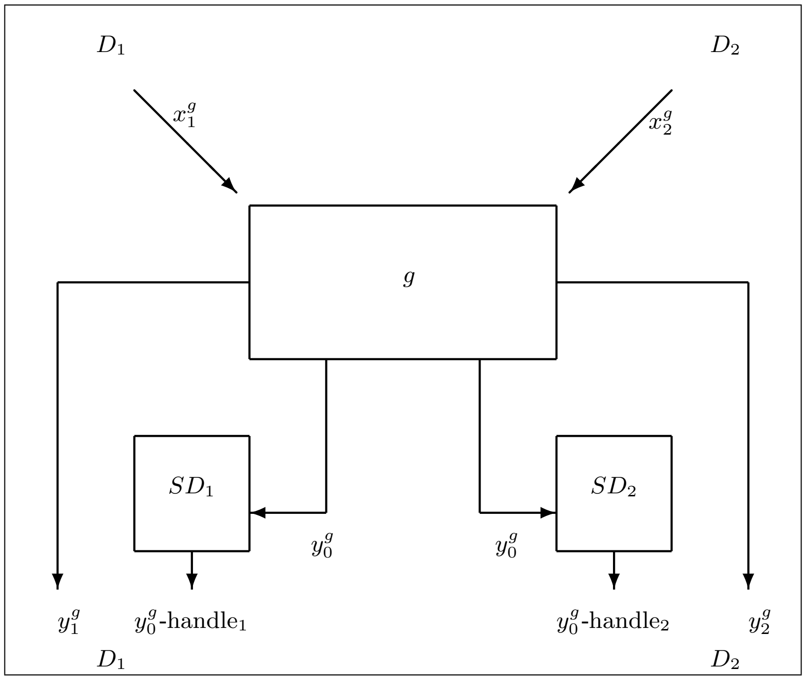

Fig. 1. The hybrid model

By a standard argument, it is possible to prove that when the two components are secure w.r.t. I/O-adversaries, the composed protocol Π^f is also secure w.r.t. the same class of adversaries, thus leading to the following result:

Theorem 15 (Composition Theorem — I/O-adversaries). *Given a protocol $\Pi^{f|g}$ output-privately reducing f to g (w.r.t.* I/O*-adversaries), and a protocol Π^g output-privately computing g (w.r.t.* I/O*-adversaries), the composed protocol $\Pi^f \doteq \Pi^{f|g} \circ \Pi^g$ output-privately computes f (w.r.t.* I/O*-adversaries).*

Proof (Sketch). Given the security of $\Pi^{f|g}$ and Π^g, there exist simulators $S^{f|g}$ and S^g. Proving the theorem amounts to exhibit a simulator S^f for Π^f. Such simulator is built by composing the existing simulators.

1. First $S^{f|g}$ is run to obtain a simulated execution of $\Pi^{f|g}$ in the g-hybrid world (it is easy to see that S^f can feed its own input to $S^{f|g}$ in this phase); during the execution $S^{f|g}$ has to make a polynomial number of oracle calls to the sub-protocol g;
2. Then S^g is used to fill in the "gaps" left by the simulation created in the previous step. In other words, for each invocation of the g-functionality occurring in the simulated transcript, the simulator S^g is invoked on the corresponding input/output values to obtain a suitable transcript for the sub-protocol Π^g.

Now we show that such simulator S^f produces a transcript which is computationally indistinguishable from one resulting from a real execution of Π^f with the same inputs and outputs. This is done using a standard hybrid argument: let N be the number of (sequential) invocations of the g that occur in $\Pi^{f|g}$. Then, one goes through a sequence of $N + 1$ hybrids experiments:

1. the first experiment produces the transcript according to the simulator S^f;

2. the second experiment consists in first running $\Pi^{f|g}$ and then simulating g-calls using S^g (indistiguishability from the previous experiment follows by the security of $\Pi^{f|g}$);
3. subsequent hybrids replace, one after the other, simulated executions produced by S^g with real execution of Π^g (indistinguishability follows from the security of Π^g).

The composition theorem follows from the observation that the last experiment in the sequence of hybrids is the real protocol. □

E.3 Deterministic *vs.* probabilistic functionalities

We now show how the computation with private output for general (probabilistic) functionalities can be reduced to that of deterministic functionalities [28, p.638].

Since a probabilistic functionality with inputs x and y, using randomness r can be viewed as a deterministic functionality of (x, y, r), we can construct the deterministic counter part of any functionality f by adequately sharing the randomness between the two parties.

This mechanism allow us to focus on the problem of realizing deterministic functionalities. More formally, a randomized functionality $f(x_1, x_2; r)$ (where r denotes the randomness) can be reduced to its deterministic counterpart $\bar{f}$:

$$\bar{f}((x_1, r_1), (x_2, r_2)) \doteq f(x_1, x_2; r_1 \oplus r_2)$$

where r_1, r_2 are chosen uniformly at random by Party 1 and Party 2, respectively.

It is easy to see that we can now consider an hybrid model in order to compute any functionality. Indeed, the reduction from the general case to the deterministic case basically consists of the realizing the following protocol $\Pi^{f|\bar{f}}$ for the $\bar{f}$-hybrid world: given his input x_i, each party P_i randomly selects a value r_i, and simply invokes the $\bar{f}$-functionality on input (x_i, r_i). The probabilistic functionality f is then inherently computed. For this reason, we now have to concentrate on protocol realizing $\Pi^{\bar{f}}$.

E.4 Construction for Deterministic Functionalities.

We now describe a construction of output-private secure (w.r.t. I/O-adversaries) protocols Π^f for any deterministic functionality f. Following the approach of [28], we think of f as an arithmetic circuit, thus consisting of a sequence of additions and multiplications in a given finite field.

The general way of going is to evaluate the circuit gate by gate in a distributed fashion. This is nothing but the historical result of [29]. For the addition gates, the operation is trivial and requires no interaction between the players. While for the multiplication gates, one needs interaction and the security relies on the Oblivious Transfer (OT) primitive; it is a well-known result that OT exists if (enhanced) trapdoor permutations exist.

Definition 16 (Enhanced trapdoor permutations). A indexed family of functions is said to be *enhanced trapdoor* if it is a trapdoor permutation family, and additionally, the one-wayness property still holds for a random element in the domain, even if the coins used to sample this element are made public.

Intuitively speaking, it is clear that the rationale in [29] can be viewed as a special case of private output computation, where the secure devices are *not* used and the global output is always $\perp$.

In the rest of this section, we then address the following points:

1. How to share the inputs between the parties;
2. How to evaluate securely the arithmetic circuit for f, in a distributed manner;
3. How to reconstruct the outputs.

Description of the Protocol. In the first phase, each party's driver D_i splits its own input x_i into two shares x_i^1, x_i^2, with $x_i = x_i^1 + x_i^2$. Then, D_1 sends x_1^2 to D_2 and D_2 sends x_2^1 to D_1. The two drivers proceed by evaluating the circuit gate by gate.

Then we turn to evaluating the arithmetic circuit. Addition gates are easy to take care of, since shares of the sum can be non-interactively obtained from the addends. As for multiplication gates, it is possible to use essentially the same oblivious-transfer-based protocol, originally proposed by [29]. The only difference is that, in our setting, at the end of the protocol, the two secure devices will contain $\perp$ as global output. In other word, we use the same paradigm using a (slightly) modified OT-protocol, in which both drivers and secure devices get an output, however the output for secure devices is always set to $\perp$ (*i.e.*, the SDs are not used in practice). Clearly, this minor modification does not affect the security properties of the original construction, which thus securely reduces evaluation of multiplication gates to (modified) oblivious transfer. Similarly, since the (classical) oblivious transfer functionality does not have a global output, standard two-party protocols for oblivious transfer can be straightforwardly translated to the private-output setting in order to implement this modified version of OT.

Finally, it remains to discuss how the two parties reconstruct the global output y_0 and the local outputs y_1, y_2, once all gates of the circuit have been evaluated. This phase involves the secure devices as follows. First note that at the end of the evaluation phase, each party holds three shares: y_0^i, y_1^i, y_2^i for the global output and the two local outputs, respectively. Also we recall that each secure device SD_i has a public key PK_i (certified via a PKI); the public key is made available by the party.

The reconstruction phase proceeds by having driver D_1 sending y_2^1 and the certified PK_1 to D_2, and *vice versa*. At this point, each D_i computes its local output by recombining the shares y_i^1 and y_i^2 into $y_i = y_i^1 + y_i^2$. Next, D_1 sends the value y_0^1, encrypted under PK_2, while D_2 sends the value y_0^2, encrypted under PK_1. Being encrypted, such values are not intelligible by the drivers. Thus, each driver D_i invokes his secure device to use the *received* encryption, say C_i, along with the share y_0^i, through a query $\mathsf{Decrypt\&Combine}(C_i, y_0^i, +)$. A $\mathsf{Finalize}$-query then give (handle) access to the global output y_0 for future needs.

Security Analysis. Given the security of the basic building blocks (*i.e.*, Oblivious Transfer, which relies on the existence of enhanced trapdoor permutations), to get a proof of security it remains to prove that the overall construction is secure in the OT-hybrid model, namely that there exists a simulator that can produce a transcript indistinguishable from that of the real execution, given oracle access to the OT-functionality.

To this end, we need to simulate the view of the adversary, which includes (1) the messages exchanged between the drivers during the gate-by-gate evaluation of the circuit, (2) the messages exchanged to recombine the shares of the local and global outputs, and (3) the communication between the driver of the controlled party and the corresponding secure device.

Messages of type (1) can be simulated as in [29], essentially proceeding backwards from the local outputs, deriving known shares while choosing at random the shares of unknown quantities.

Messages of type (2) and (3) consist just of public information and ciphertexts, that can be simulated by encrypting a fixed message, assuming semantic security of the underlying encryption scheme. $\square$