Security Against Covert Adversaries: Efficient Protocols for Realistic Adversaries*

Yonatan Aumann[†] Yehuda Lindell[†]

March 8, 2009

Abstract

In the setting of secure multiparty computation, a set of mutually distrustful parties wish to securely compute some joint function of their private inputs. The computation should be carried out in a secure way, meaning that no coalition of corrupted parties should be able to learn more than specified or somehow cause the result to be "incorrect". Typically, corrupted parties are either assumed to be semi-honest (meaning that they follow the protocol specification) or malicious (meaning that they may deviate arbitrarily from the protocol). However, in many settings, the assumption regarding semi-honest behavior does not suffice and security in the presence of malicious adversaries is excessive and expensive to achieve.

In this paper, we introduce the notion of *covert adversaries*, which we believe faithfully models the adversarial behavior in many commercial, political, and social settings. Covert adversaries have the property that they may deviate arbitrarily from the protocol specification in an attempt to cheat, but do not wish to be "caught" doing so. We provide a definition of security for covert adversaries and show that it is possible to obtain highly efficient protocols that are secure against such adversaries. We stress that in our definition, we quantify over all (possibly malicious) adversaries and do not assume that the adversary behaves in any particular way. Rather, we guarantee that if an adversary deviates from the protocol in a way that would enable it to "cheat" (meaning that it can achieve something that is impossible in an ideal model where a trusted party is used to compute the function), then the honest parties are guaranteed to detect this cheating with good probability. We argue that this level of security is sufficient in many settings.

1 Introduction

1.1 Background

In the setting of secure multiparty computation, a set of parties with private inputs wish to jointly compute some functionality of their inputs. Loosely speaking, the security requirements of such a computation are that (i) nothing is learned from the protocol other than the output (privacy), (ii) the output is distributed according to the prescribed functionality (correctness), and (iii) parties cannot make their inputs depend on other parties' inputs. Secure multiparty computation forms the basis for a multitude of tasks, including those as simple as coin-tossing and agreement, and as complex as electronic voting, electronic auctions, electronic cash schemes, anonymous transactions, remote game playing (a.k.a. "mental poker"), and privacy-preserving data mining.

*An extended abstract of this work appeared in the 4^{th} *Theory of Cryptography Conference* (TCC), 2007.

[†]Department of Computer Science, Bar-Ilan University, Israel. email: {aumann,lindell}@cs.biu.ac.il. Work supported in part by an Infrastructures grant from the Ministry of Science, Israel. The second author was also supported by the israel science foundation (grant No. 781/07).

The security requirements in the setting of multiparty computation must hold even when some of the participating parties are adversarial. It has been shown that, with the aid of suitable cryptographic tools, *any* two-party or multiparty function can be securely computed [31, 14, 12, 4, 7], even in the presence of very strong adversarial behavior. However, the efficiency of the computation depends dramatically on the adversarial model considered. Classically, two main categories of adversaries have been considered:

1. *Malicious adversaries:* these adversaries may behave arbitrarily and are not bound in any way to following the instructions of the specified protocol. Protocols that are secure in the malicious model provide a very strong security guarantee as honest parties are "protected" irrespective of the adversarial behavior of the corrupted parties.

2. *Semi-honest adversaries:* these adversaries correctly follow the protocol specification, yet may attempt to learn additional information by analyzing the transcript of messages received during the execution. Security in the presence of semi-honest adversaries provides only a weak security guarantee, and is not sufficient in many settings. Semi-honest adversarial behavior primarily models inadvertent leakage of information, and is suitable only where participating parties essentially trust each other, but may have other concerns.

Secure computation in the semi-honest adversary model can be carried out very efficiently, but, as mentioned, provides weak security guarantees. Regarding malicious adversaries, it has been shown that, under suitable cryptographic assumptions, *any* multiparty probabilistic polynomial-time functionality can be securely computed for any number of *malicious* corrupted parties [14, 12]. However, this comes at a price. These feasibility results of secure computation typically do not yield protocols that are efficient enough to actually be implemented and used in practice (particularly if standard *simulation-based security* is required). Their importance is more in telling us that it is perhaps worthwhile searching for other efficient protocols, because we at least know that a solution exists in principle. However, the unfortunate state of affairs today – many years after these feasibility results were obtained – is that very few truly efficient protocols exist for the setting of malicious adversaries. Thus, we believe that some middle ground is called for: an adversary model that accurately models adversarial behavior in the real world, on the one hand, but for which efficient, secure protocols can be obtained, on the other.

1.2 Our Work – Covert Adversaries

In this work, we introduce a new adversary model that lies between the semi-honest and malicious models. The motivation behind the definition is that in many real-world settings, parties are willing to actively cheat (and as such are not semi-honest), but only if they are not caught (and as such they are not arbitrarily malicious). This, we believe, is the case in many business, financial, political and diplomatic settings, where honest behavior cannot be assumed, but where the companies, institutions and individuals involved cannot afford the embarrassment, loss of reputation, and negative press associated with being *caught* cheating. It is also the case, unfortunately, in many social settings, e.g. elections for a president of the country-club. Finally, in remote game playing, players may also be willing to actively cheat, but would try to avoid being caught, or else they may be thrown out of the game. In all, we believe that this type of *covert* adversarial behavior accurately models many real-world situations. Clearly, with such adversaries, it may be the case that the risk of being caught is weighed against the benefits of cheating, and it cannot be assumed that players would avoid being caught at any price and under all circumstances. Accordingly, our definition explicitly models the probability of catching adversarial behavior; a probability that can be tuned to the specific circumstances of the problem. In particular, we do not assume that adversaries are only willing to risk being caught with negligible probability, but rather allow for much higher probabilities.

The definition. Our definition of security is based on the classical *ideal/real simulation paradigm*,[1] and provides the guarantee that if the adversary cheats, then it will be caught by the honest parties (with some probability). In order to understand what we mean by this, we have to explain what we mean by "cheating". Loosely speaking, we say that an adversary successfully cheats if it manages to do something that is impossible in the ideal model. Stated differently, successful cheating is behavior that cannot be simulated in the ideal model. Thus, for example, an adversary who learns more about the honest parties' inputs than what is revealed by the output has cheated. In contrast, an adversary who uses pseudorandom coins instead of random coins (where random coins are what are specified in the protocol) has not cheated.

We are now ready to informally describe the guarantee provided by our definition. Let $0 < \epsilon \leq 1$ be a value (called the *deterrence factor*). Then, any attempt to cheat by a real adversary $\mathcal{A}$ is detected by the honest parties with probability at least ϵ. This is formalized by allowing the ideal-model simulator $\mathcal{S}$ to sometimes "fail" (meaning that the output distribution of the real protocol execution cannot be simulated in the standard ideal model for secure computation), with the requirement that in a real execution with $\mathcal{A}$ the honest parties would detect cheating with probability that is at least ϵ times the probability that the simulator fails. Note that when an adversary follows a strategy that can result in a successful cheat with some probability p, the honest parties are guaranteed to catch the adversary cheating with probability at least $\epsilon \cdot p$. Thus, provided that ϵ is sufficiently large, an adversary that wishes not to be caught cheating, will refrain from *attempting* to cheat, lest it be caught doing so. Clearly, the higher the value of ϵ, the greater the probability adversarial behavior is caught and thus the greater the *deterrent* to cheat. We therefore call our notion security in the presence of covert adversaries with ϵ-deterrent. Note that the security guarantee does not preclude successful cheating. Indeed, if the adversary decides to cheat then it may gain access to the other parties' private information or bias the result of the computation. The only guarantee is that if it attempts to cheat, then there is a fair chance that it will be caught doing so. This is in contrast to standard definitions, where absolute privacy and security are guaranteed, for the given type of adversary. We remark that by setting $\epsilon = 1$, our definition can be used to capture a requirement that cheating parties are always caught.

Further details on the definition. The above intuitive notion can be interpreted in a number of ways. We present three different formulations that form a strict hierarchy (i.e., the first definition is weaker than the second which is weaker than the third). We briefly describe the three definitions here (these descriptions are not complete and are only intended to give a flavor of the full definitions):

1. *Failed-simulation formulation:* In this definition, the ideal-model simulator is allowed to sometimes "fail" in the sense that the output distribution generated in the ideal model need not always be indistinguishable from the output distribution in a real protocol execution. Rather, it is guaranteed that if these output distributions can be distinguished with some

[1] According to this paradigm, security is formalized by comparing the execution of a real protocol to an ideal execution where a trusted party receives the parties' inputs, computes the function and returns the outputs. More formally, a protocol is secure if for every real-model adversary $\mathcal{A}$ attacking the protocol there exists an ideal-model adversary/simulator $\mathcal{S}$ (interacting in a world where a trusted party computes the function) such that the output distribution of the honest parties and $\mathcal{S}$ in an ideal execution is computationally indistinguishable from the output distribution of the honest parties and $\mathcal{A}$ in a real execution of the protocol. See Section 2 for more details.

probability Δ, then the honest parties will detect cheating by a corrupted party with probability at least $\epsilon \cdot \Delta$, where ϵ is the deterrence factor. On an intuitive level, this captures what we desire because executions that are successfully simulated are executions in which the adversary does not successfully cheat, whereas in failed simulations the adversary may have successfully cheated. The important point is that the probability that the honest parties will detect cheating is related (by ϵ) to the probability that the simulator may fail in its simulation.

2. *Explicit-cheat formulation:* In this definition, the ideal-model adversary/simulator is explicitly given the ability to cheat. Specifically, the ideal model is modified so that a special cheat instruction can be sent by the adversary to the trusted party. Upon receiving such an instruction, the trusted party hands all the honest parties' inputs to the adversary. Then, it tosses coins and with probability ϵ announces to the honest parties that cheating has taken place (by sending the message $\mathsf{corrupted}_i$ where party P_i is the corrupted party that sent the cheat instruction). However, with probability $1-\epsilon$, the trusted party does not announce that cheating has taken place, and so the adversary gets off scot-free. Observe that in the ideal model the adversary can always cheat. However, as required, if it chooses to do so it is guaranteed to be caught with probability ϵ. Here, the ideal-model simulator is required to generate an output distribution that is computationally indistinguishable from a real execution (but in the modified ideal model and not the standard one).

3. *Strong explicit-cheat formulation:* Here we make a small modification to the ideal model of the previous definition so that the adversary only receives the honest parties' inputs in the case that the honest parties do not detect its cheating. Specifically, if the trusted party announces that cheating has taken place, then the adversary learns absolutely nothing. This is stronger than the previous definition because when the adversary attempts to cheat, it must take the risk of being caught and gaining nothing. Thus the deterrence is higher. (Metaphorically speaking, there is less deterrence to not rob a bank if when you are caught you are allowed to keep the stolen money.)

In Section 3 we present all three definitions, and also discuss the relationships between them and the standard definitions of security in the presence of semi-honest and malicious adversaries. We also explain why we chose these specific formulations over other possibilities.

Composition. An important security property, and one that is guaranteed by the standard definition of security that is based on the ideal/real simulation paradigm, is that of *modular sequential composition*. Loosely speaking, this means that if a secure protocol ρ is used as a subprotocol inside some larger protocol π, then it suffices to analyze π in a model where instead of running ρ the parties send their inputs to ρ to a trusted party who carries out the computation for them (the fact that "sequential" composition is considered means that when ρ is being executed, no other subprotocols are executed). Such composition theorems significantly simplify proofs of security (making them "modular") and are also security goals within themselves (guaranteeing a higher level of security). We prove modular sequential composition theorems for the "explicit-cheat" and "strong explicit-cheat" definitions, and a weaker sequential composition theorem for the "failed-simulation" definition. (The weaker version states that if you run secure protocols sequentially, then their security properties are preserved. Thus, this provides appropriate security guarantees for running protocols sequentially, but does not allow modular construction of larger protocol π using a subprotocol ρ that has already been proven secure. We did not succeed in proving a modular composition theorem for this definition, but also do not have a counter-example.[2] We leave the existence of a modular sequential composition for the failed-simulation definition as an open question.)

Protocol constructions. As mentioned, the aim of this work is to provide a definition of security for which it is possible to construct highly efficient protocols. We demonstrate this by presenting a generic protocol for secure *two-party* computation in our model that is only mildly less efficient than the protocol of Yao [31], which is secure only for semi-honest adversaries. The first step of our construction is a protocol for oblivious transfer that is based on homomorphic encryption schemes.[3] Highly efficient protocols under this assumption are known [2, 23]. However, these protocols do not achieve *simulation-based* security. Rather, only privacy is guaranteed (with the plus that privacy is preserved even in the presence of fully malicious adversaries). We prove the following informally stated theorem:

Theorem 1.1 *Let $\epsilon = 1 - \frac{1}{k}$ where $k = \mathrm{poly}(n)$ and n is the security parameter. Assuming the existence of homomorphic encryption schemes, there exists an oblivious transfer protocol that is secure in the presence of covert adversaries with ϵ-deterrent, has four rounds of communication and requires $\mathcal{O}(k)$ homomorphic encryption operations.*

We remark that the constant hidden inside the $\mathcal{O}$ notation for $\mathcal{O}(k)$ is very small (to be exact, the protocol requires the generation of $2k$ pairs of encryption keys, and carrying out $2k$ encryptions, 2 homomorphic operations and one decryption). When setting $\epsilon = 1/2$ we have $k = 2$ and thus the protocol is highly efficient. (To compare, the analogous protocols that achieve only privacy without simulation require generating one encryption key, carrying out one encryption and one decryption, and computing two homomorphic operations. Thus our protocol is about four times slower.) We also show that when many oblivious transfers are run simultaneously, efficiency can be further improved because the generation of $2k$ pairs of encryption keys may be carried out only once.

Having constructed an oblivious transfer protocol that meets our definition, we use it in the protocol of Yao [31] in order to obtain efficient general two-party computation. We modify Yao's protocol so that a number ℓ of garbled circuits are sent, and then all but one are opened in order to check that they were constructed correctly (this follows the folklore cut-and-choose methodology for boosting the security of Yao's protocol for adversaries that may not be semi-honest). In addition, as it was pointed out in [24], when dealing with malicious adversaries it is necessary to modify the circuit so that each input bit is "split" into a number of random shares (see Section 6 for a full explanation as to why this is necessary). This modification has a significant effect on efficiency because an oblivious transfer is needed for every input bit. Thus, when each bit is split into m shares, we have that m oblivious transfers are needed for each input bit. We present a protocol for general secure two-party computation for which different values of ℓ and m can be plugged in (recall that ℓ denotes the number of garbled circuits that are constructed and sent, and m denotes the number of oblivious transfers per input bit). Our protocol achieves ϵ-deterrent for $\epsilon = (1 - \ell^{-1})(1 - 2^{-m+1})$. Thus, in order to achieve a deterrent of $\epsilon = 1/2$ it suffices to take $\ell = m = 3$. For a higher deterrent of $\epsilon \approx 9/10$ it is possible to take $\ell = m = 10$. We prove the following informally stated theorem:

[2] In previous versions of this work, we stated that we have a modular sequential composition theorem for all of our definitions. We retract that statement here.

[3] We remark that there is no need to show "feasibility" here because any protocol that is secure in the presence of malicious adversaries is secure in the presence of covert adversaries (with any ϵ). Thus, our focus is on constructing protocols that are highly efficient and not on using general assumptions.

Theorem 1.2 *Assume the existence of one-way functions and secure oblivious transfer. Then, for every ℓ and m and every probabilistic polynomial-time function f, there exists a protocol π that securely computes f in the presence of covert adversaries with ϵ-deterrent for $\epsilon = (1 - \ell^{-1})(1 - 2^{-m+1})$. Furthermore, the protocol π has a constant number of rounds, requires m oblivious transfers per input bit, and has communication complexity $\mathcal{O}(\ell \cdot n \cdot |C|)$ excluding the cost of the oblivious transfers, where $|C|$ is the size of the circuit computing f and n is the security parameter.*

It is sufficient for the oblivious transfer protocol referred to in Theorem 1.2 to be secure in the presence of covert adversaries (with the same ϵ achieved by protocol π). Thus, a protocol for general two-party computation with $\epsilon = 1/2$ can be constructed by combining Theorems 1.1 and 1.2, and the result is a protocol that is only a *constant factor* slower than the original protocol of Yao that is only secure for semi-honest adversaries. (Note that the protocol of Yao [31] has communication complexity $\mathcal{O}(n|C|)$ and requires one oblivious transfer per input bit.) Our protocol construction is for the case of non-reactive functionalities where the computation consists of a single stage in which parties first provide inputs and then receive their specified outputs. The more general case of reactive computation (where the computation consists of a number of stages in which inputs are provided and outputs are received) can be obtained in straightforward way by making modifications to the circuit as described in [12, Chapter 7].

We view our constructions as a "proof of concept" that highly efficient protocols are achievable in this model, and leave the construction of such protocols for specific tasks of interest for future work.

Comparison to efficient protocols in the malicious model. As we have mentioned, achieving secure computation in the presence of malicious adversaries seems to be significantly harder than for covert adversaries as introduced here. In order to demonstrate this, we compare the complexity of our protocol to the best-known general protocols for two-party computation in the presence of malicious adversaries. Recently, two efficient protocols for general two-party computation in the presence of malicious adversaries were presented [22, 25]. The protocol of [22] achieves universal composability under the decisional composite residuosity and strong RSA assumptions and assumes a common reference string. The protocol of [25] can be constructed under more general assumptions and is secure in the plain model, achieving stand-alone security. The protocol of [22] requires $\mathcal{O}(|C|)$ public-key operations and bandwidth of $\mathcal{O}(n \cdot |C|)$. Thus, for circuits that are not very small, the computational overhead is prohibitive (and significantly greater than for our protocol where for $\epsilon = 1/2$ a constant number of public-key operations is needed per input bit irrespective of the size of the circuit). The complexity of the protocol of [25] is comparable to ours with respect to public-key operations, but requires symmetric operations and bandwidth on the order of $\mathcal{O}(sn|C| + s^2k)$ where k is the length of the input, n is the (computational) security parameter and s is a statistical security parameter (currently s needs to be set at least 680 to achieve a reasonable error probability but it is conjectured that this can be reduced to 160). Thus, once again, our protocol is much more efficient for circuits that are not very small.

1.3 Related Work

The idea of allowing the adversary to cheat as long as it will be detected was first considered by [11] who defined a property called *t-detectability;* loosely speaking, a protocol fulfilling this property provides the guarantee that no coalition of t parties can cheat without being caught. The work of [11] differs to ours in that (a) they consider the setting of an honest majority, and (b) their definition is not simulation based. Another closely related work to ours is that of [6] which considered *honest-looking adversaries*. Such adversaries may deviate arbitrarily from the protocol specification, but only if this deviation cannot be detected. Our definition differs from that of [6] in a number of important ways. First, our definition provides security guarantees even for adversaries that are willing to be caught with high probability. Thus, we do not assume anything about the adversary's willingness or lack of willingness to be caught. Second, we place the onus of detecting any cheating by an adversary on the *protocol*. This is of importance because the fact that an adversary generates messages that are distributed differently to an honest party does not mean that the honest parties can or will detect this. (In order to see this, first note that the honest parties may not have the appropriate distinguisher. Second, the result of any single execution may not be enough to detect cheating. For example, if the protocol tells an honest party to send a random bit and the adversary always sends the bit 1, then the honest parties cannot deduce that the adversary is cheating in any given execution because an honest party also sends the bit 1 with probability 1/2.) Thus, in our formulation, the protocol specification itself has instructions that include outputting the fact that "party P_i has cheated". We remark also that our motivation and that of [6] is completely different: they considered a more stringent setting where all parties are either malicious or honest-looking. In contrast, we consider a *relaxation* of the adversary model (where parties are either fully honest or covert) with the aim of obtaining more efficient protocols.

The idea of allowing an adversary to cheat with non-negligible probability as long as it will be caught with good probability has been mentioned many times in the literature; see [20, 26] for just two examples. We stress, however, that none of these works formalized this idea. Furthermore, our experience in proving our protocol secure is that simple applications of cut-and-choose do not meet our definition (and there are actual attacks that can be carried out on the cut-and-choose technique used in [26], for example). Another approach to obtaining efficient protocols is to consider definitions of security that are weaker in the sense that they do not follow the simulation paradigm; see [18] for just one example. In contrast, our approach is to remain within the ideal/real simulation paradigm, thereby preserving the well-known advantages of this definitional paradigm.

We conclude by remarking that the works on covert secure two-party and multiparty computation of [1, 8] have no connection with this work; those works consider steganographic secure computation and so it is the computation that is covert, whereas in our work it is the adversarial behavior that is covert.

1.4 Organization

In Section 2 we review the standard definitions of secure two-party computation and in Section 3 we present formal definitions for the notion of security in the presence of covert adversaries. We present three formulations of this notion and prove relations between the different formulations. In addition, we show that our definitions fall in between malicious and semi-honest security (i.e., security in the presence of malicious adversaries implies security in the presence of covert adversaries for any ϵ, and security in the presence of covert adversaries with $\epsilon > 1/\text{poly}(n)$ implies security in the presence of semi-honest adversaries). In Section 4 we prove composition theorems for all three of our formulations of security in the presence of covert adversaries. We then proceed to construct efficient protocols under the strongest of our three definitions. In Section 5 we construct protocols for oblivious transfer (the basic protocol is given in Section 5.1 and extensions in Section 5.2). Then in Section 6 we present our protocol for general two-party computation and prove its security.

2 Preliminaries and Standard Definitions

2.1 Preliminaries

A function $\mu(\cdot)$ is negligible in n, or just negligible, if for every positive polynomial $p(\cdot)$ and all sufficiently large n's it holds that $\mu(n) < 1/p(n)$. A probability ensemble $X = \{X(a,n)\}_{a\in\{0,1\}^*;n\in\mathbb{N}}$ is an infinite sequence of random variables indexed by a and $n \in \mathbb{N}$. (The value a will represent the parties' inputs and n the security parameter.) Two distribution ensembles $X = \{X(a,n)\}_{a\in\{0,1\}^*;n\in\mathbb{N}}$ and $Y = \{Y(a,n)\}_{a\in\{0,1\}^*;n\in\mathbb{N}}$ are said to be computationally indistinguishable, denoted $X \stackrel{c}{\equiv} Y$, if for every non-uniform polynomial-time algorithm D there exists a negligible function $\mu(\cdot)$ such that for every $a \in \{0,1\}^*$ and every $n \in \mathbb{N}$,

$$|\Pr[D(X(a,n)) = 1] - \Pr[D(Y(a,n)) = 1]| \leq \mu(n)$$

All parties are assumed to run in time that is polynomial in the security parameter. (Formally, each party has a security parameter tape upon which that value 1^n is written. Then the party is polynomial in the input on this tape. We note that this means that a party may not even be able to read its entire input, as would occur in the case that its input is longer than its overall running time.)

2.2 Secure Multiparty Computation – Standard Definition

In this section we briefly present the standard definition for secure multiparty computation and refer to [12, Chapter 7] for more details and motivating discussion. The following description and definition is based on [12], which in turn follows [16, 27, 3, 5].

Multiparty computation. A multiparty protocol problem is cast by specifying a random process that maps sets of m-inputs to sets of m-outputs (one for each party). We will assume that the number of parties m is fixed, but as noted in [12], this can easily be generalized to the case that m is a parameter. We refer to such a process as a functionality and denote it $f : (\{0,1\}^*)^m \to (\{0,1\}^*)^m$, where $f = (f_1, \ldots, f_m)$. That is, for every vector of inputs $\overline{x} = (x_1, \ldots, x_m)$, the output-vector is a random variable $\overline{y} = (f_1(\overline{x}), \ldots, f_m(\overline{x}))$ ranging over vectors of strings. The i^{th} party P_i, with input x_i, wishes to obtain $f_i(\overline{x})$. We sometimes denote such a functionality by $(\overline{x}) \mapsto (f_1(\overline{x}), \ldots, f_m(\overline{x}))$. Thus, for example, the oblivious transfer functionality is denoted by $((x_0, x_1), \sigma) \mapsto (\lambda, x_\sigma)$, where (x_0, x_1) is the first party's input, σ is the second party's input, and λ denotes the empty string (meaning that the first party has no output). We assume the existence of special symbols abort and corrupted that are not in the range of f (these have special meaning, as will be seen later).

Adversarial behavior. Loosely speaking, the aim of a secure multiparty protocol is to protect honest parties against dishonest behavior by other parties. In this section, we present the definition for *malicious adversaries* who control some subset of the parties and may instruct them to arbitrarily deviate from the specified protocol. We also consider *static corruptions*, meaning that the set of corrupted parties is fixed at the onset.

Security of protocols (informal). The security of a protocol is analyzed by comparing what an adversary can do in a real protocol execution to what it can do in an ideal scenario that is secure by definition. This is formalized by considering an *ideal* computation involving an incorruptible *trusted third party* to whom the parties send their inputs. The trusted party computes the functionality on the inputs and returns to each party its respective output. Loosely speaking, a protocol is secure if any adversary interacting in the real protocol (where no trusted third party exists) can do no more harm than if it was involved in the above-described ideal computation. One technical detail that arises when considering the setting of no honest majority is that it is impossible to achieve fairness or guaranteed output delivery. That is, it is possible for the adversary to prevent the honest parties from receiving outputs. Furthermore, it may even be possible for the adversary to receive output while the honest parties do not. We consider malicious adversaries and static corruptions in all of our definitions in this paper.

Execution in the ideal model. As we have mentioned, some malicious behavior cannot be prevented (for example, early aborting). This behavior is therefore incorporated into the ideal model. Let the set of parties be $P_1, \ldots, P_m$ and let $I \subseteq [m]$ denote the indices of the corrupted parties, controlled by an adversary $\mathcal{A}$. An ideal execution proceeds as follows:

Inputs: Each party obtains an input; the i^{th} party's input is denoted x_i. The adversary $\mathcal{A}$ receives an auxiliary input denoted z.

Send inputs to trusted party: Any honest party P_j sends its received input x_j to the trusted party. The corrupted parties controlled by $\mathcal{A}$ may either abort (by replacing the input x_i with a special abort_i message), send their received input, or send some other input of the same length to the trusted party. This decision is made by $\mathcal{A}$ and may depend on the values x_i for $i \in I$ and its auxiliary input z. Denote the vector of inputs sent to the trusted party by $\overline{w}$ (note that $\overline{w}$ does not necessarily equal $\overline{x}$).

If the trusted party receives an input of the form abort_i for some $i \in I$, it sends abort_i to all parties and the ideal execution terminates. (If it receives abort_i for more than one i, then it takes any arbitrary one, say the smallest i, and ignores all others.) Otherwise, the execution proceeds to the next step.

Trusted party sends outputs to adversary: The trusted party computes $(f_1(\overline{w}), \ldots, f_m(\overline{w}))$ and sends $f_i(\overline{w})$ to party P_i, for all $i \in I$ (i.e., to all corrupted parties).

Adversary instructs trusted party to continue or halt: $\mathcal{A}$ sends either $\mathsf{continue}$ or abort_i to the trusted party (for some $i \in I$). If it sends $\mathsf{continue}$, the trusted party sends $f_j(\overline{w})$ to party P_j, for all $j \notin I$ (i.e., to all honest parties). Otherwise, if it sends abort_i, the trusted party sends abort_i to all parties P_j for $j \notin I$.

Outputs: An honest party always outputs the message it obtained from the trusted party. The corrupted parties output nothing. The adversary $\mathcal{A}$ outputs any arbitrary (probabilistic polynomial-time computable) function of the initial inputs $\{x_i\}_{i \in I}$, the auxiliary input z, and the messages $\{f_i(\overline{w})\}_{i \in I}$ obtained from the trusted party.

This ideal model is different from that of [12] in that in the case of an "abort", the honest parties output abort_i and not a $\perp$ symbol. This means that the honest parties *know* the identity of the corrupted party who causes the abort. This is achieved by most multiparty protocols, including that of [14], but not all (e.g., the protocol of [17] does not meet this requirement).

Let $f : (\{0,1\}^*)^m \rightarrow (\{0,1\}^*)^m$ be an m-party functionality, where $f = (f_1, \ldots, f_m)$, let $\mathcal{A}$ be a non-uniform probabilistic polynomial-time machine, and let $I \subseteq [m]$ be the set of corrupted parties. Then, the ideal execution of f on inputs $\overline{x}$, auxiliary input z to $\mathcal{A}$ and security parameter n, denoted $\text{IDEAL}_{f,\mathcal{A}(z),I}(\overline{x}, n)$, is defined as the output vector of the honest parties and the adversary $\mathcal{A}$ from the above ideal execution.

Execution in the real model. We next consider the real model in which a real m-party protocol π is executed (and there exists no trusted third party). In this case, the adversary $\mathcal{A}$ sends all messages in place of the corrupted parties, and may follow an arbitrary polynomial-time strategy. In contrast, the honest parties follow the instructions of π. As is standard, we assume that the parties are connected via *authenticated channels*; this means that the adversary can see all messages sent between parties, but cannot modify them. As default, we also assume synchronous network and a broadcast channel (or public-key infrastructure for running authenticated Byzantine agreement [9]). This is the basic real model that is typically defined; we refer the reader to [12, Chapter 7] for more details. We stress that our definitions below for covert adversaries can be applied to any specification of a real model (synchronous or asynchronous communication, broadcast channel or not, and so on). We therefore take the basic model here, but this is no way limits our definitions.

Let f be as above and let π be an m-party protocol for computing f. Furthermore, let $\mathcal{A}$ be a non-uniform probabilistic polynomial-time machine and let I be the set of corrupted parties. Then, the real execution of π on inputs $\overline{x}$, auxiliary input z to $\mathcal{A}$ and security parameter n, denoted $\text{REAL}_{\pi,\mathcal{A}(z),I}(\overline{x}, n)$, is defined as the output vector of the honest parties and the adversary $\mathcal{A}$ from the real execution of π.

Security as emulation of a real execution in the ideal model. Having defined the ideal and real models, we can now define security of protocols. Loosely speaking, the definition asserts that a secure party protocol (in the real model) emulates the ideal model (in which a trusted party exists). This is formulated by saying that adversaries in the ideal model are able to simulate executions of the real-model protocol. We will consider executions where all inputs are of the same length (see discussion in [12]), and will therefore say that a vector $\overline{x} = (x_1, \ldots, x_m)$ is balanced if for every i and j it holds that $|x_i| = |x_j|$.

Definition 2.1 (secure multiparty computation): *Let f and π be as above. Protocol π is said to* securely compute f with abort in the presence of malicious adversaries *if for every non-uniform probabilistic polynomial-time adversary $\mathcal{A}$ for the real model, there exists a non-uniform probabilistic polynomial-time adversary $\mathcal{S}$ for the ideal model, such that for every $I \subseteq [m]$,*

$$\left\{\text{IDEAL}_{f,\mathcal{S}(z),I}(\overline{x}, n)\right\}_{\overline{x},z\in(\{0,1\}^*)^{m+1};n\in\mathbb{N}} \stackrel{\text{c}}{\equiv} \left\{\text{REAL}_{\pi,\mathcal{A}(z),I}(\overline{x}, n)\right\}_{\overline{x},z\in(\{0,1\}^*)^{m+1};n\in\mathbb{N}}$$

where $\overline{x}$ is a balanced vector.

We note that the above definition assumes that the parties (and adversary) know the input lengths (this can be seen from the requirement that $\overline{x}$ is balanced and so all the inputs in the vector of inputs are of the same length).[4] We remark that some restriction on the input lengths is unavoidable, see [12, Section 7.1] for discussion.

2.3 Functionalities that Provide Output to a Single Party

In the standard definition of secure computation, both parties receive output and these outputs may be *different*. However, the presentation of our two-party protocol is far simpler if we assume that only party P_2 receives output. We will show now that this suffices for the general case. That is, we claim that any protocol that can be used to securely compute *any* efficient functionality

[4]In the case that no parties are corrupted, we assume that the adversary receives the length of the inputs as part of its auxiliary input z.

$f(x, y)$ where only P_2 receives output, can be used to securely compute *any* efficient functionality $f = (f_1, f_2)$ where party P_1 receives $f_1(x_1, x_2)$ and party P_2 receives $f_2(x_1, x_2)$. For simplicity, we will assume that the length of the output of $f_1(x_1, x_2)$ is at most n, where n is the security parameter. This can be achieved by simply taking n to be larger in case it is necessary.

Let $f = (f_1, f_2)$ be a functionality. We wish to construct a secure protocol in which P_1 receives $f_1(x_1, x_2)$ and P_2 receives $f_2(x_1, x_2)$. As a building block we use a protocol for computing any efficient functionality with the limitation that only P_2 receives output. Let $r, a, b \in_R \{0, 1\}^n$ be randomly chosen strings. Then, in addition to x_1, party P_1's input includes the elements r, a and b. Furthermore, define a functionality g (that has only a single output) as follows:

$$g((r, a, b, x_1), x_2) = (\alpha, \beta, f_2(x_1, x_2))$$

where $\alpha = r + f_1(x_1, x_2)$, $\beta = a \cdot \alpha + b$, and the arithmetic operations are defined over $GF[2^n]$. Note that α is a one-time pad encryption of P_1's output $f_1(x, y)$, and β is an information-theoretic message authentication tag of α (specifically, $a\alpha + b$ is a pairwise-independent hash of α). Now, the parties compute the functionality g, using a secure protocol in which only P_2 receives output. Following this, P_2 sends the pair (α, β) to P_1. Party P_1 checks that $\beta = a \cdot \alpha + b$; if yes, it outputs $\alpha - r$, and otherwise it outputs abort_2.

It is easy to see that P_2 learns nothing about P_1's output $f_1(x_1, x_2)$, and that it cannot alter the output that P_1 will receive (beyond causing it to abort), except with probability 2^{-n}. We remark that it is also straightforward to construct a simulator for the above protocol. Applying the composition theorem of [5] (for standard security) or 4.2 (for covert adversaries – to be defined below), we have the following proposition:

Proposition 2.2 *Assume that there exists a protocol for securely computing any probabilistic polynomial-time functionality in which only a single party receives output. Then, there exists a protocol for securely computing any probabilistic polynomial-time functionality in which both parties receive output. This holds also for security in the presence of covert adversaries for Definitions 3.3 and 3.4.*

We remark that the circuit for computing g is only mildly larger than that for computing f. Thus, the construction above is also efficient and has only a mild effect on the complexity of the secure protocol (assuming that the complexity of the original protocol, where only P_2 receives output, is proportional to the size of the circuit computing f as is the case for our protocol below).

3 Definitions – Secure Computation with Covert Adversaries

3.1 Motivation

The standard definition of security (see Definition 2.1) is such that all possible (polynomial-time) adversarial behavior is simulatable. Here, in contrast, we wish to model the situation that parties may *successfully* cheat. However, if they do so, they are likely to be caught. There are a number of ways of defining this notion. In order to motivate ours, we begin with a somewhat naive implementation of the notion, and show its shortcoming.

First attempt: Define an adversary to be covert if the distribution over the messages that it sends during an execution is computationally indistinguishable from the distribution over the messages that an honest party would send. Then, quantify over all covert adversaries $\mathcal{A}$ for the real world (rather than all adversaries).[5] A number of problems arise with this definition.

- The fact that the distribution generated by the adversary can be distinguished from the distribution generated by honest parties does not mean that the honest parties can detect this in any specific execution. Consider for example a coin-tossing protocol where the honest distribution gives even probabilities to 0 and 1, while the adversary gives double the probability to the 1 outcome. Clearly, the distributions differ. However, in any given execution, even an outcome of 1 does not provide the honest players with sufficient evidence of any wrong-doing. Thus, it is not sufficient that the *distributions* differ. Rather, one needs to be able to detect cheating in any given execution.

- The fact that the distributions differ does not necessarily imply that the honest parties have an efficient distinguisher. Furthermore, in order to guarantee that the honest parties detect the cheating, they would have to analyze all traffic during an execution. However, this analysis *cannot* be part of the protocol because then the distinguishers used by the honest parties would be known (and potentially bypassed).

- Another problem is that, as mentioned in the introduction, adversaries may be willing to risk being caught with more than negligible probability, say 10^{-6}. With such an adversary, the definition would provide no security guarantee. In particular, the adversary may be able to *always* learn all parties' inputs, and risk being caught in one run in a million.

Second attempt. To solve the aforementioned problems, we first require that the protocol itself be responsible for detecting cheating. Specifically, in the case that a party P_i attempts to cheat, the protocol may instruct the honest parties to output a message saying that "party P_i has cheated" (we require that this only happens if P_i indeed cheated). This solves the first two problems. To solve the third problem, we explicitly quantify the probability that an adversary is caught cheating. Roughly, given a parameter ϵ, a protocol is said to be secure against covert adversaries with ϵ-deterrent if any cheating adversary will necessarily be caught with probability at least ϵ.

This definition captures the spirit of what we want, but is still problematic. To illustrate the problem, consider an adversary that plays honestly with probability 0.99, and cheats otherwise. Such an adversary can only ever be caught with probability 0.01 (because otherwise it is honest). However, when $\epsilon = 1/2$ for example, such an adversary must be caught with probability 0.5, which is impossible. We therefore conclude that an *absolute* parameter cannot be used, and the probability of catching the adversary must be related to the probability that it cheats.

Final definition. We thus arrive at the following approach. First, as mentioned, we require that the protocol itself be responsible for detecting cheating. That is, if a party P_i successfully cheats, then with good probability (ϵ), the honest parties in the protocol will all receive a message that "P_i cheated". Second, we do not quantify only over adversaries that are covert (i.e., those that are not detected cheating by the protocol). Rather, we allow all possible adversaries, even completely malicious ones. Then, we require either that this malicious behavior can be successfully simulated (as in Definition 2.1), or that the honest parties will receive a message that cheating has been detected, and this happens with probability at least ϵ times the probability that successful cheating takes place. We stress that when the adversary chooses to cheat, it may actually learn

[5] We remark that this is the conceptual approach taken by [6], and that there are important choices that arise when attempting to formalize the approach. In any case, as we have mentioned, the work of [6] differs greatly because their aim was to model all parties as somewhat adversarial.

secret information or cause some other damage. However, since it is guaranteed that such a strategy will likely be caught, there is strong motivation to refrain from doing so.

As it turns out, the above intuition can be formalized in three different ways, which form a hierarchy of security guarantees. In practice, the implementor should choose the formulation that best suites her needs, and for which sufficiently efficient protocols exists. All three definitions are based on the ideal/real simulation paradigm, as presented in Section 2. In each definition, the only change is to the ideal model; the real model is the same as for standard definitions of security for malicious adversaries (see Section 2.2). We now present the definitions in order of security, starting with the weakest (i.e., least secure) one.

3.2 Version 1: Failed Simulation Formulation

The first formulation we present is based on allowing the simulator to fail sometimes, where by "fail" we mean that its output distribution is not indistinguishable from the real one. This corresponds to an event of successful cheating. However, we guarantee that the probability that the adversary is caught cheating is at least ϵ times the probability that the simulator fails. The details follow.

Recall that we call a vector balanced if all of its items are of the same length. In addition, we denote the output vector of the honest parties and adversary $\mathcal{A}$ in an ideal execution of f by $\textsc{ideal}_{f,\mathcal{A}(z),I}(\overline{x}, n)$, where $\overline{x}$ is the vector of inputs, z is the auxiliary input to $\mathcal{A}$, I is the set of corrupted parties, and n is the security parameter, and denote the analogous outputs in a real execution of π by $\textsc{real}_{\pi,\mathcal{A}(z),I}(\overline{x}, n)$. We begin by defining what it means to "detect cheating":

Definition 3.1 *Let π be an m-party protocol, let $\mathcal{A}$ be an adversary, and let I be the index set of the corrupted parties. A party P_j is said to* detect cheating in π *if its output in π is* $\mathsf{corrupted}_i$*; this event is denoted* $\textsc{output}_j(\textsc{real}_{\pi,\mathcal{A}(z),I}(\overline{x})) = \mathsf{corrupted}_i$. *The protocol π is called* detection accurate *if for every $j, k \notin I$, the probability that P_j outputs* $\mathsf{corrupted}_k$ *is negligible.*

We require that all protocols be detection accurate (meaning that only corrupted parties can be "caught cheating"). This is crucial because otherwise a party that is detected cheating can just claim that it is due to a protocol anomaly and not because it really cheated. The definition follows:

Definition 3.2 (security – failed simulation formulation): *Let f and π be as in Definition 2.1, and let $\epsilon : \mathbb{N} \to [0, 1]$ be a function. Protocol π is said to* securely compute f in the presence of covert adversaries with ϵ-deterrent *if it is* detection accurate *and if for every non-uniform probabilistic polynomial-time adversary $\mathcal{A}$ for the real model, there exists a non-uniform probabilistic polynomial-time adversary $\mathcal{S}$ for the ideal model such that for every $I \subseteq [m]$, every balanced vector $\overline{x} \in (\{0,1\}^*)^m$, every auxiliary input $z \in \{0,1\}^*$, and every non-uniform polynomial-time distinguisher D, there exists a negligible function $\mu(\cdot)$ such that,*

$$\begin{aligned}\Pr\Big[\exists i \in I\ \forall j \notin I\ :\ \textsc{output}_j(\textsc{real}_{\pi,\mathcal{A}(z),I}(\overline{x}, n)) = \mathsf{corrupted}_i\Big] \\ \geq \epsilon(n) \cdot \Big|\Pr\left[D(\textsc{ideal}_{f,\mathcal{S}(z),I}(\overline{x}, n)) = 1\right] - \Pr\left[D(\textsc{real}_{\pi,\mathcal{A}(z),I}(\overline{x}, n)) = 1\right]\Big| - \mu(n)\end{aligned}$$

The parameter ϵ indicates the probability that successful adversarial behavior is detected (observe that when such a detection occurs, *all* honest parties must detect the same corrupted party). Clearly, the closer ϵ is to one, the higher the deterrence to cheat, and hence the level of security, assuming covert adversaries. Note that the adversary can decide to never be detected cheating, in which case the IDEAL and REAL distributions are guaranteed to be *computationally indistinguishable*, as in the standard definition of security. In contrast, it can choose to cheat with some noticeable probability, in which case the IDEAL and REAL output distribution may be distinguishable (while guaranteeing that the adversary is caught with good probability). This idea of allowing the ideal and real models to not be fully indistinguishable in order to model "allowed cheating" was used in [13].

We stress that the definition does not *require* the simulator to "fail" with some probability. Rather, it is *allowed* to fail with a probability that is at most $1/\epsilon$ times the probability that the adversary is caught cheating. As we shall see, this is what enables us to construct highly efficient protocols. We also remark that due to the required detection accuracy, the simulator cannot fail when the adversary behaves in a fully honest-looking manner (because in such a case, no honest party will output $\mathsf{corrupted}_i$). Thus, security is always preserved in the presence of adversaries that are willing to cheat arbitrarily, as long as their cheating is not detected.

We remark that the above definition (as with the ones that follow) requires that all honest parties *agree* on the identity of the cheating party P_i. This is important for ensuring that a party that cheats can be "punished" (if the honest different parties disagree about who cheated, then no action can be taken against the cheating party).

3.3 Version 2: Explicit Cheat Formulation

The main drawback of Definition 3.2 is that it does not rule out the ability of the adversary to make its cheat strategy (implicitly) depend on the honest parties' inputs or on the output. Specifically, it is possible that the adversary can act in a way that for some set of honest party inputs its behavior is like that of an honest party, while for another set of honest party inputs its behavior achieves successful cheating. For example, in oblivious transfer, a corrupted sender may carry out a strategy whereby if the receiver has input bit $\sigma = 0$ then the protocol terminates as with an honest sender, and if the receiver has input bit $\sigma = 1$ then the protocol terminates with the receiver detecting cheating. (Some natural protocols have this property.) In order to see that this phenomenon is not ruled out by Definition 3.2, observe that the probability that an honest party outputs $\mathsf{corrupted}_i$ may be different for every $\overline{x}$. Thus, in particular, the adversary's strategy may be such that for some inputs this probability is high and for others it is low. This phenomenon is undesirable since there may be honest parties' inputs for which it is more "worthwhile" for the adversary to risk being caught. Therefore, it may run a strategy that results in potentially successful cheating only when the honest parties have such worthwhile inputs. We therefore wish to force the adversary to explicitly decide whether or not to cheat, and have this decision be *independent* of the honest parties' inputs.

Another drawback of Definition 3.2 is that there is no explicit partition of the probability space of the ideal-model executions into "successful" and "unsuccessful" simulations (i.e., the definition does not mandate the existence of a subspace such that executions inside the subspace have the property that the honest parties detect cheating with probability ϵ, while outside of the subspace full security holds). Thus, there is no guarantee that the executions in which the honest parties detect cheating are fully correlated with the executions in which the adversary's behavior cannot be simulated. It is not clear that this is essential, but it is intuitively appealing (this notion is reminiscent of the fraction version of knowledge complexity in [15]). As more evidence that this drawback is main aesthetic, we note that honest parties cannot detect cheating when the corrupted parties behave honestly, because this would contradict the requirement of detection accuracy in Definition 3.1. Thus, there must be some correlation between unsuccessful simulations and the event that honest parties output $\mathsf{corrupted}$.

The above discussion brings us to an alternate definition, which is based on redefining the ideal functionality so as to explicitly include the option of cheating. Aside from overcoming the drawbacks described above, this alternate formulation has two additional advantages. First, it makes the security guarantees that are achieved more explicit. Second, it makes it easy to prove a sequential composition theorem (see below).

We modify the ideal model by adding new instructions that the adversary can send to the trusted party. Recall that in the standard ideal model, the adversary can send a special abort_i message to the trusted party, resulting in the honest parties receiving abort_i as output. In the ideal model for the explicit cheat formulation for covert adversaries, the adversary can send the following additional special instructions:

- *Special input* $\mathsf{corrupted}_i$*:* If the ideal-model adversary sends $\mathsf{corrupted}_i$ instead of an input, the trusted party sends $\mathsf{corrupted}_i$ to all honest parties and halts. This enables the simulation of behavior by a real adversary that always results in detected cheating. (It is not essential to have this special input, but it sometimes makes proving security easier.)

- *Special input* cheat_i*:* If the ideal-model adversary sends cheat_i instead of an input, the trusted party hands it all of the honest parties' inputs. Then, the trusted party tosses coins and with probability ϵ determines that this "cheat strategy" by P_i was detected, and with probability $1-\epsilon$ determines that it was not detected. If it was detected, the trusted party sends $\mathsf{corrupted}_i$ to all honest parties. If it was not detected, the trusted party gives the ideal-model adversary the ability to set the outputs of the honest parties to whatever values it wishes. Thus, a cheat_i input is used to model a protocol execution in which the real-model adversary decides to cheat. Such cheating is always successful in the ideal model in that the adversary learns the honest parties' inputs. However, as required, this cheating is also always detected with probability at least ϵ. Note also that if the cheat attempt is not detected then the adversary is given "full cheat capability" including the ability to determine the honest parties' outputs.

The idea behind our new ideal model is that given the above instructions, the adversary in the ideal model can choose to cheat, with the caveat that its cheating is guaranteed to be detected with probability at least ϵ. We stress that since the capability to cheat is given through an "input" that is provided to the trusted party, the adversary's decision to cheat must be made before the adversary learns anything (and thus independently of the honest parties' inputs and the output).

We are now ready to present the modified ideal model. Let $\epsilon : \mathbb{N} \to [0,1]$ be a function. Then, the ideal execution with ϵ proceeds as follows:

Inputs: Each party obtains an input; the i^{th} party's input is denoted by x_i; we assume that all inputs are of the same length, denoted n. The adversary receives an auxiliary-input z.

Send inputs to trusted party: Any honest party P_j sends its received input x_j to the trusted party. The corrupted parties, controlled by $\mathcal{A}$, may either send their received input, or send some other input of the same length to the trusted party. This decision is made by $\mathcal{A}$ and may depend on the values x_i for $i \in I$ and the auxiliary input z. Denote the vector of inputs sent to the trusted party by $\overline{w}$.

Abort options: If a corrupted party sends $w_i = \mathsf{abort}_i$ to the trusted party as its input, then the trusted party sends abort_i to all of the honest parties and halts. If a corrupted party sends $w_i = \mathsf{corrupted}_i$ to the trusted party as its input, then the trusted party sends $\mathsf{corrupted}_i$ to all of the honest parties and halts. If multiple parties send abort_i (resp., $\mathsf{corrupted}_i$), then the trusted party relates only to one of them (say, the one with the smallest i). If both $\mathsf{corrupted}_i$ and abort_j messages are sent, then the trusted party ignores the $\mathsf{corrupted}_i$ message.

Attempted cheat option: If a corrupted party sends $w_i = \mathsf{cheat}_i$ to the trusted party as its input, then the trusted party sends to the adversary all of the honest parties' inputs $\{x_j\}_{j \notin I}$ (as above, if multiple cheat_i messages are sent, the trusted party ignores all but one). In addition,

1. With probability ϵ, the trusted party sends $\mathsf{corrupted}_i$ to the adversary and all of the honest parties.
2. With probability $1 - \epsilon$, the trusted party sends $\mathsf{undetected}$ to the adversary. Following this, the adversary sends the trusted party output values $\{y_j\}_{j \notin I}$ of its choice for the honest parties. Then, for every $j \notin I$, the trusted party sends y_j to P_j.

The ideal execution then ends at this point.

If no w_i equals abort_i, $\mathsf{corrupted}_i$ or cheat_i, the ideal execution continues below.

Trusted party answers adversary: The trusted party computes $(f_1(\overline{w}), \ldots, f_m(\overline{w}))$ and sends $f_i(\overline{w})$ to $\mathcal{A}$, for all $i \in I$.

Trusted party answers honest parties: After receiving its outputs, the adversary sends either abort_i for some $i \in I$, or $\mathsf{continue}$ to the trusted party. If the trusted party receives $\mathsf{continue}$ then it sends $f_j(\overline{w})$ to all honest parties P_j ($j \notin I$). Otherwise, if it receives abort_i for some $i \in I$, it sends abort_i to all honest parties.

Outputs: An honest party always outputs the message it obtained from the trusted party. The corrupted parties output nothing. The adversary $\mathcal{A}$ outputs any arbitrary (probabilistic polynomial-time computable) function of the initial inputs $\{x_i\}_{i \in I}$, the auxiliary input z, and the messages obtained from the trusted party.

The output of the honest parties and the adversary in an execution of the above ideal model is denoted by $\text{IDEALC}^{\epsilon}_{f,\mathcal{S}(z),I}(\overline{x}, n)$.

Notice that there are two types of "cheating" here. The first is the classic abort and is used to model "early aborting" due to the impossibility of achieving fairness in general when there is no honest majority (as in Definition 2.1, the honest parties here are informed as to who caused the abort). The other type of cheating in this ideal model is more serious for two reasons: first, the ramifications of the cheat are greater (the adversary may learn all of the parties' inputs and may be able to determine their outputs), and second, the cheating is only guaranteed to be detected with probability ϵ. Nevertheless, if ϵ is high enough, this may serve as a deterrent. We stress that in the ideal model the adversary must decide whether to cheat obliviously of the honest-parties inputs and before it receives any output (and so it cannot use the output to help it decide whether or not it is "worthwhile" cheating). We have the following definition.

Definition 3.3 (security – explicit cheat formulation): *Let f, π and ϵ be as in Definition 3.2. Protocol π is said to* securely compute f in the presence of covert adversaries with ϵ-deterrent *if for every non-uniform probabilistic polynomial-time adversary $\mathcal{A}$ for the real model, there exists a non-uniform probabilistic polynomial-time adversary $\mathcal{S}$ for the ideal model such that for every $I \subseteq [m]$:*

$$\left\{\text{IDEALC}^{\epsilon}_{f,\mathcal{S}(z),I}(\overline{x}, n)\right\}_{\overline{x},z \in (\{0,1\}^*)^{m+1}; n \in \mathbb{N}} \stackrel{\text{c}}{\equiv} \left\{\text{REAL}_{\pi,\mathcal{A}(z),I}(\overline{x}, n)\right\}_{\overline{x},z \in (\{0,1\}^*)^{m+1}; n \in \mathbb{N}}$$

where $\overline{x}$ is a balanced vector.

Definition 3.3 and detection accuracy. We note that in Definition 3.3 it is not necessary to explicitly require that π be detection accurate because this is taken care of in the ideal model (in an ideal execution, only a corrupted party can send a cheat_i input).

3.4 Version 3: Strong Explicit Cheat Formulation

The third, and strongest version follows the same structure and formulation of the previous version (Version 2). However, we make a slight but important change to the ideal model. In the ideal model of the explicit cheat formulation, the adversary can always cheat and obtain the honest parties' inputs. Here, we modify the ideal model so that the adversary only learns the honest parties' inputs if its cheating goes undetected. Stated differently, if the adversary sends an input cheat_i, then the trusted party only sends it the honest parties' inputs in the event of $\mathsf{undetected}$ (which occurs with probability $1-\epsilon$). However, if the trusted party sends $\mathsf{corrupted}_i$ to the honest parties and the adversary (an event which happens with probability ϵ), then the adversary learns nothing and so its attempt to cheat fails completely. This is significantly stronger than the previous definition because the adversary must take the risk of being caught without knowing if it will gain anything at all. Formally, we modify the "attempted cheat option" in the ideal model as follows:

Attempted cheat option: If a corrupted party sends $w_i = \mathsf{cheat}_i$ to the trusted party as its input, then the trusted party works as follows:

1. With probability ϵ, the trusted party sends $\mathsf{corrupted}_i$ to the adversary and all of the honest parties.
2. With probability $1-\epsilon$, the trusted party sends $\mathsf{undetected}$ to the adversary along with the honest parties' inputs $\{x_j\}_{j\notin I}$. Following this, the adversary sends the trusted party output values $\{y_j\}_{j\notin I}$ of its choice for the honest parties. Then, for every $j \notin I$, the trusted party sends y_j to P_j.

Everything else in the ideal model remains the same. We denote the resultant ideal model by $\text{IDEALSC}^{\epsilon}_{f,\mathcal{S}(z),I}(\overline{x}, n)$ and have the following definition:

Definition 3.4 (security – strong explicit cheat formulation): *Let f, π and ϵ be as in Definition 3.2. Protocol π is said to* securely compute f in the presence of covert adversaries with ϵ-deterrent *if for every non-uniform probabilistic polynomial-time adversary $\mathcal{A}$ for the real model, there exists a non-uniform probabilistic polynomial-time adversary $\mathcal{S}$ for the ideal model such that for every $I \subseteq [m]$:*

$$\left\{\text{IDEALSC}^{\epsilon}_{f,\mathcal{S}(z),I}(\overline{x}, n)\right\}_{\overline{x},z\in(\{0,1\}^*)^{m+1};n\in\mathbb{N}} \stackrel{\text{c}}{\equiv} \left\{\text{REAL}_{\pi,\mathcal{A}(z),I}(\overline{x}, n)\right\}_{\overline{x},z\in(\{0,1\}^*)^{m+1};n\in\mathbb{N}}$$

where $\overline{x}$ is a balanced vector.

The difference between the regular and strong explicit cheat formulations is perhaps best exemplified in the case that $\epsilon = 1$. In both versions, any potentially successful cheating attempt is detected. However, in the regular formulation, the adversary may learn the honest parties' private inputs (albeit, while being detected). In the strong formulation, in contrast, the adversary learns nothing when it is detected. Since it is always detected, this means that full security is achieved.

3.5 Cheating and Aborting

It is important to note that in all of our above definitions, a party that halts mid-way through the computation may be considered a "cheat" (we also use this in an inherent way when constructing our protocols later). Arguably, this may be undesirable due to the fact that an honest party's computer may crash (such unfortunate events may not even be that rare). Nevertheless, we argue that as a basic definition it suffices. This is due to the fact that it is possible for all parties to work by storing their input and random-tape on disk before they begin the execution. Then, before sending any message, the incoming messages that preceded it are also written to disk. The result of this is that if a party's machine crashes, it can easily reboot and return to its previous state. (In the worst case the party will need to request a retransmit of the last message if the crash occurred before it was written.) We therefore believe that honest parties cannot truly hide behind the excuse that their machine crashed (it would be highly suspicious that someone's machine crashed in an irreversible way that also destroyed their disk at the critical point of a secure protocol execution).

Despite the above, it is possible to modify the definition so that honest halting is never considered cheating. When considering the failed simulation formulation (Definition 3.2), this modification only needs to be made to the notion of "detection accuracy" and uses the notion of a fail-stop party who acts semi-honestly, except that it may halt early.

Definition 3.5 *A protocol π is* non-halting detection accurate *if it is detection accurate as in Definition 3.1 and if for every honest party P_j and fail-stop party P_k, the probability that P_j outputs $\mathsf{corrupted}_k$ is negligible.*

Definition 3.2 can then be modified so that protocol π is required to be *non-halting* detection accurate (and not just detection accurate). When considering Definitions 3.3 and 3.4, this strengthening must be explicitly added to the definition by requiring that π be non-halting detection accurate. (Recall that detection accuracy is not needed for these definitions. However, the requirement that $\mathsf{corrupted}_k$ is not output for a fail-stop party P_k does need to be added separately.)

We remark that although this strengthening is clearly desirable, it may also be prohibitive. We note that we are able to modify our main protocol so that it meets this stronger definition; see Section 6.3. In order to do so, we require an oblivious transfer that is secure in the presence of malicious adversaries (and not just covert). A highly efficient protocol for this task, with only a constant number of exponentiations per transfer, appears in [29].

3.6 Relations Between Security Models

Relations between covert security definitions. The three security definitions for covert adversaries constitute a strict hierarchy, with version 1 being strictly weaker than version 2, which in turn is strictly weaker than version 3. We begin by proving that version 1 is strictly weaker than version 2.

Proposition 3.6 *Let π be a protocol that securely computes some functionality f in the presence of covert adversaries with ϵ-deterrent by Definition 3.3. Then, π securely computes f in the presence of covert adversaries with ϵ-deterrent by Definition 3.2. Furthermore, assuming the existence of string oblivious transfer that is secure in the presence of malicious adversaries, there exist protocols that are secure by Definition 3.2 and not secure by Definition 3.3.*

Proof: Let f, π and ϵ be as in the proposition. Then, we first claim that π is detection accurate. This is due to the fact that in the ideal model of Definition 3.3, honest parties only output $\mathsf{corrupted}_i$

for $i \in I$. Therefore, this must hold also in the real model, except with negligible probability (as required by Definition 3.1). Now, let $\mathcal{A}$ be an adversary and let $\mathcal{S}$ be the simulator that is guaranteed to exist for IDEALC by Definition 3.3. We claim that the simulator $\mathcal{S}$ also works for Definition 3.2. In order to see this, let Δ be the probability that $\mathcal{S}$ sends $\mathsf{corrupted}_i$ or cheat_i for input for some $i \in I$ (this probability depends only on $\mathcal{A}$, the corrupted parties' inputs and the auxiliary input z). Now, when $\mathcal{S}$ sends input $\mathsf{corrupted}_i$, the honest parties all output $\mathsf{corrupted}_i$ with probability 1. In addition, when $\mathcal{S}$ sends input cheat_i, the honest parties all output $\mathsf{corrupted}_i$ with probability ϵ in the ideal model. It follows that the honest parties output $\mathsf{corrupted}_i$ with probability *at least* $\epsilon \cdot \Delta$. It remains, therefore, to show that the IDEAL and REAL distributions can be distinguished with probability at most Δ (because then the probability that the adversary is caught cheating is at least ϵ times the maximum distinguishing "gap" between the IDEAL and REAL distributions). However, this follows immediately from the fact that if $\mathcal{S}$ does *not* send any input of the form $\mathsf{corrupted}_i$ or cheat_i, then the ideal execution is the same as in the standard definitions (and so the same as in Definition 3.2). Thus, in the event that $\mathcal{S}$ does not send $\mathsf{corrupted}_i$ or cheat_i, the IDEAL and REAL of Definition 3.2 are computationally indistinguishable. Since $\mathcal{S}$ sends $\mathsf{corrupted}_i$ or cheat_i with probability Δ, we obtain that the IDEAL distribution can be distinguished from the REAL one with probability at most $\Delta + \mu(n)$ as desired.

For the furthermore part of the proposition, take any protocol for string oblivious transfer that is secure in the presence of *malicious adversaries*, as in Definition 2.1; denote it π. Then, define a protocol π' where the sender upon input (x_0, x_1) first computes the ciphertexts $c_0 = E_{k_0}(x_0)$ and $c_1 = E_{k_1}(x_1)$, where k_0 and k_1 are secret keys for a private-key encryption scheme chosen by the sender. We assume that the encryption scheme is such that not all keys are valid, and the decryption algorithm outputs $\perp$ in such a case (there are many examples of such encryption schemes). Then, the parties run the oblivious transfer protocol π where the sender inputs (k_0, k_1) and the receiver inputs σ. Finally, the sender sends the receiver the pair of ciphertexts (c_0, c_1). Upon receiving k_σ and (c_0, c_1), the receiver decrypts c_σ obtaining x_σ. If upon decryption the receiver obtains $\perp$ then it outputs $\mathsf{corrupted}_S$, where S denotes the sender. It is not difficult to show that π' is secure by Definition 3.2 with $\epsilon = 1$ (using the composition theorem for malicious adversaries of [5]), because by the security of π the only thing that a corrupted sender can do is to send invalid ciphertexts or keys, in which case the receiver always outputs $\mathsf{corrupted}_S$. However, π' is *not* secure by Definition 3.3 because a corrupted sender can send a valid key k_0 and an invalid key k_1. This means that R always outputs $\mathsf{corrupted}_S$ if $\sigma = 1$ and never outputs $\mathsf{corrupted}_S$ if $\sigma = 0$. This contradicts Definition 3.3 because the adversary must decide to cheat independently of the honest party's input. (Technically, the ideal-model simulator has no input to the trusted party that can result in this output distribution.) ■

Next we prove that Definition 3.3 is strictly weaker than Definition 3.4.

Proposition 3.7 *Let π be a protocol that securely computes some functionality f in the presence of covert adversaries with ϵ-deterrent by Definition 3.4. Then, π securely computes f in the presence of covert adversaries with ϵ-deterrent by Definition 3.3. Furthermore, assuming that there exist protocols that are secure by Definition 3.4, there exist protocols that are secure by Definition 3.3 and not secure by Definition 3.4.*

Proof: The fact that security under Definition 3.4 implies security under Definition 3.3 is immediate because the only difference is that in Definition 3.4 the ideal simulator may receive less information. (Formally this is shown by just constructing a simulator for Definition 3.3 that doesn't pass on the inputs to the simulator designed for Definition 3.4 in the case of $\mathsf{undetected}$.)

For the furthermore part of the proposition, take any protocol that is secure under Definition 3.4 and add an instruction that if a party receives the output $\mathsf{corrupted}_i$ then it sends party P_i its input and halts. Such a protocol is still secure under Definition 3.3 (because the ideal adversary receives the honest parties' inputs in the case that the honest parties receive $\mathsf{corrupted}_i$). However, if the functionality being computed does not reveal all the inputs, the modified protocol is no longer secure under Definition 3.4. ■

Relation to the malicious and semi-honest models. As a sanity check regarding our definitions, we present two propositions that show the relation between security in the presence of covert adversaries and security in the presence of malicious and semi-honest adversaries.

Proposition 3.8 *Let π be a protocol that securely computes some functionality f with abort in the presence of malicious adversaries, as in Definition 2.1. Then, π securely computes f in the presence of covert adversaries with ϵ-deterrent, for any of the three formulations (Definitions 3.2, 3.3, and 3.4) and for every $0 \leq \epsilon \leq 1$.*

This proposition follows from the simple observation that according to Definition 2.1, there exists a simulator that always succeeds in its simulation. Thus, Definition 3.2 holds even if the probability of detecting cheating is 0. Likewise, for Definitions 3.3 and 3.4 the same simulator works (there is simply no need to ever send a cheat input).

Next, we consider the relation between covert and semi-honest adversaries. We remark that security for malicious adversaries only implies security for semi-honest adversaries if the semi-honest adversary is allowed to modify its input before the execution begins [19].[6] Calling such an adversary *augmented semi-honest*, we have the following:

Proposition 3.9 *Let π be a protocol that securely computes some functionality f in the presence of covert adversaries with ϵ-deterrent, for any of the three formulations and for $\epsilon(n) \geq 1/\mathrm{poly}(n)$. Then, π securely computes f in the presence of augmented semi-honest adversaries.*

This proposition follows from the fact that due to the requirement of detection accuracy, no party outputs $\mathsf{corrupted}_i$ when the adversary is semi-honest. Since $\epsilon \geq 1/\mathrm{poly}(n)$ this implies that the REAL and IDEAL distributions can be distinguished with at most negligible probability, as required. We stress that if $\epsilon = 0$ (or is negligible) then the definition of covert adversaries requires nothing, and so the proposition does not hold for this case.

We conclude that, as one may expect, security in the presence of covert adversaries with ϵ-deterrent lies in between security in the presence of malicious adversaries and security in the presence of semi-honest adversaries. If $1/\mathrm{poly}(n) \leq \epsilon(n) \leq 1$ then it can be shown that Definitions 3.2 and 3.3 are strictly different to both the semi-honest and malicious models (this is not difficult to see and so details are omitted). Regarding Definition 3.4, the same is true for the case that $1/\mathrm{poly}(n) \leq \epsilon(n) \leq 1 - 1/\mathrm{poly}(n)$. However, as we show below, when $\epsilon(n) = 1 - \mu(n)$, Definition 3.4 is equivalent to security in the presence of malicious adversaries (Definition 2.1).

Strong explicit cheat formulation and the malicious model. The following proposition shows that the strong explicit cheat formulation "converges" to the malicious model as ϵ approaches 1. In order to make this claim technically, we need to deal with the fact that in the

[6] This situation is anti-intuitive because the ability to modify input only strengthens the adversary, and so it seems that this should in turn imply security for (ordinary) semi-honest adversaries. However, this intuition is false because when the real adversary is allowed to modify its input, so is the ideal adversary/simulator. Thus, the ideal adversary in this case is given more power than a standard semi-honest ideal adversary, enabling it to simulate some protocols that a standard semi-honest simulator cannot.

malicious model honest parties never output $\mathsf{corrupted}_i$, whereas this can occur in the strong explicit formulation even with $\epsilon = 1$. We therefore define a transformation of any protocol π to π' where the only difference is that if an honest party should output $\mathsf{corrupted}_i$ in π, then it outputs abort_i instead in π'. We have the following:

Proposition 3.10 *Let π be a protocol and μ a negligible function. Then π securely computes some functionality f in the presence of covert adversaries with $\epsilon(n) = 1 - \mu(n)$ under Definition 3.4 if and only if π' securely computes f with abort in the presence of malicious adversaries.*

This is true since, by definition, either the ideal adversary does not send cheat_i, in which case the ideal execution is the same as in the regular ideal model, or it does send cheat_i, in which case it is caught with probability that is negligibly close to 1 and so the protocol is aborted. Stated differently, when ϵ is negligibly close to 1, sending cheat_i is the same as sending abort_i (as long as the output of honest parties is changed from $\mathsf{corrupted}_i$ to abort_i as discussed above). We stress that Proposition 3.10 does not hold for Definitions 3.2 and 3.3 because in these definitions the adversary may learn the honest parties' private inputs when it is caught (something that is not allowed in the malicious model).

4 Modular Sequential Composition

Sequential composition theorems for secure computation are important for two reasons. First, they constitute a security goal within themselves and guarantee security even when parties run many executions, albeit sequentially. Second, they are useful tools that help in writing proofs of security. As such, we believe that when presenting a new definition, it is of great importance to also prove an appropriate composition theorem for that definition. We prove *modular* sequential composition theorems that are analogous to that of [5] for Definitions 3.3 and 3.4, and a weaker sequential composition theorem for Definition 3.2.

The weaker sequential composition theorem states that when a polynomial number of secure protocols are run sequentially, then security is maintained for the overall execution, with the deterrent being the minimum deterrent of any of the individual protocols.

For Definitions 3.3 and 3.4 we prove modular sequential composition. The basic idea behind such composition is that it is possible to design a protocol that uses an ideal functionality as a subroutine, and then analyze the security of the protocol when a trusted party computes this functionality. For example, assume that a protocol is constructed that uses oblivious transfer as a subroutine. Then, first we construct a protocol for oblivious transfer and prove its security. Next, we prove the security of the protocol that uses oblivious transfer as a subroutine, in a model where the parties have access to a trusted party computing the oblivious transfer functionality. The composition theorem then states that when the "ideal calls" to the trusted party for the oblivious transfer functionality are replaced by real executions of a secure protocol computing this functionality, the protocol remains secure. In the proofs below, for the sake of simplicity, we assume a synchronous model of communication. However, we remark that when output delivery is not guaranteed (as is the case in our definitions and in general when no honest majority is assumed) then this is the same as assuming asynchronous communication and having the parties include the round number in every message that they send. In order to ensure the effect of a synchronous network, an honest party only sends its $i+1^{\text{th}}$ message after receiving round-i messages from all parties (for this we also need to assume that all parties send and receive messages in all rounds). The adversary in such a case can easily prevent the protocol from terminating; however, as we have stated, this is allowed here as neither fairness nor output delivery are guaranteed.

4.1 Sequential Composition for Definition 3.2

In this section, we prove a basic sequential composition theorem for the failed-simulation formulation of security in the presence of covert adversaries. The guarantee provided is that sequential executions of secure protocols preserves the security guarantees. We first need to define what we mean by sequential execution. Let $f_1, \ldots, f_{p(n)}$ be multi-party functionalities. For each party P_j and functionality f_ℓ, let $M_\ell^{(j)}$ be a probabilistic polynomial-time *transition procedure* that generates P_j's input to f_ℓ based on P_j's private input and the outputs of the previous f_i computations ($i < \ell$). Denote by M_ℓ the set of $M_\ell^{(j)}$. Let f be the multiparty functionality resulting from applying M_1, then f_1, then M_2, then f_2, etc. up to $M_{p(n)}$ and then $f_{p(n)}$. We call f the *composition* of f_ℓ's and the M_ℓ's.

For each ℓ let π_ℓ be a protocol for computing f_ℓ. Let π be the protocol obtained by first applying M_1, then π_1, then M_2, then π_2, etc. up to $M_{p(n)}$ and then $\pi_{p(n)}$. We call π the *concatenation* of the π_ℓ's and the M_ℓ's.

Theorem 4.1 *Let $p(n)$ be a polynomial. Let $f_1, \ldots, f_{p(n)}$ be multiparty probabilistic polynomial-time functionalities, $M_1, \ldots, M_{p(n)}$ transition procedures (as defined above), and $\pi_1, \ldots, \pi_{p(n)}$ protocols that securely compute $f_1, \ldots, f_{p(n)}$ in the presence of covert adversaries with deterrents $\epsilon_1, \ldots, \epsilon_{p(n)}$, respectively, under Definition 3.2. Let f be the composition of f_ℓ's and the M_ℓ's and π the concatenation of the π_ℓ's and the M_ℓ's (as defined above). Then π securely computes f in the presence of covert adversaries with deterrent $\epsilon = \min_\ell\{\epsilon_\ell\}$, under Definition 3.2.*

Proof: First note that since the transition procedures do not include inter-party communication, we can eliminate reference to them, as follows. For each ℓ, let f'_ℓ be the functionality that is the composition of f_ℓ on M_ℓ, and in addition, appends each party's input to each party's output. Similarly, let π'_ℓ be the protocol obtained by concatenating π_ℓ to M_ℓ and also outputting its inputs (for each party separately). Then, since M_ℓ does not include any interaction between the players, if π_ℓ securely computes f_ℓ in the presence of covert adversaries with deterrence ϵ_ℓ, then so does π'_ℓ for f'_ℓ. Furthermore, f is the composition of the f'_ℓ and π is the concatenation of the π'_ℓ's.

Let $\mathcal{A}$ be an adversary attacking π. For each ℓ, let $\mathcal{A}_\ell$ be the restriction of $\mathcal{A}$ to π'_ℓ. For $\ell < p(n)$ the output of $\mathcal{A}_\ell$ is the full state of the adversary at the end of the execution of π'_ℓ. The input to $\mathcal{A}_\ell$ ($\ell > 1$) is the state of the adversary at the beginning of π'_ℓ. Let $\mathcal{S}_\ell$ be the simulator for $\mathcal{A}_\ell$ guaranteed by the security of π'_ℓ with respect to f'_ℓ. Finally, let $\mathcal{S}$ be the simulator that is obtained by running $\mathcal{S}_1, \ldots, \mathcal{S}_{p(n)}$ in turn, where the simulator $\mathcal{S}_\ell$ is run on the adversary $\mathcal{A}_\ell$ with its input being the state output by $\mathcal{S}_{\ell-1}$.

Denote by $\textsc{real}_{\pi,\mathcal{A}(z),I}(\overline{x}, n)$ the real execution of π with adversary $\mathcal{A}$ with axillary input z, and by $\textsc{ideal}_{f,\mathcal{S}(z),I}(\overline{x}, n)$ the ideal execution of f with simulator $\mathcal{S}$ running on adversary $\mathcal{A}$ with axillary input z. Further denote $\textsc{hybrid}_{\langle\pi'_1,\ldots,\pi'_\ell,f'_{\ell+1},\ldots,f'_{p(n)}\rangle,\mathcal{A}(z),\mathcal{S}(z),I}(\overline{x}, n)$ the sequential executions of $\pi'_1, \ldots, \pi'_\ell$ followed by sequential calls to a trusted party computing $f'_{\ell+1}, \ldots, f'_{p(n)}$. In the execution, $\pi'_1, \ldots, \pi'_\ell$ are executed by the adversaries $\mathcal{A}_1, \ldots, \mathcal{A}_\ell$, while the $f'_{\ell+1}, \ldots, f'_{p(n)}$ are with the simulators $\mathcal{S}_{\ell+1}, \ldots, \mathcal{S}_{p(n)}$ running on the residual adversary.

First observe that for $\ell = 1$ it holds that

$$\textsc{hybrid}_{\langle\pi'_1,\ldots,\pi'_{\ell-1},f'_\ell,\ldots,f'_{p(n)}\rangle,\mathcal{A}(z),\mathcal{S}(z),I}(\overline{x}, n) \equiv \textsc{ideal}_{f,\mathcal{S}(z),I}(\overline{x}, n)$$

and for $\ell = p(n)$ it holds that

$$\textsc{hybrid}_{\langle\pi'_1,\ldots,\pi'_\ell,f'_{\ell+1},\ldots,f'_{p(n)}\rangle,\mathcal{A}(z),\mathcal{S}(z),I}(\overline{x}, n) \equiv \textsc{real}_{\pi,\mathcal{A}(z),I}(\overline{x}, n).$$

We therefore have that:

$$\begin{aligned}&\left|\Pr[D(\textsc{ideal}_{f,\mathcal{S}(z),I}(\overline{x},n))=1]-\Pr[D(\textsc{real}_{\pi,\mathcal{A}(z),I}(\overline{x},n))=1]\right|\\ &\leq \sum_{\ell=1}^{p(n)}\Big|\Pr[D(\textsc{hybrid}_{\langle\pi'_1,\ldots,\pi'_{\ell-1},f'_\ell,\ldots,f'_{p(n)}\rangle,\mathcal{A}(z),\mathcal{S}(z),I}(\overline{x},n))=1]\\ &\qquad\qquad -\Pr[D(\textsc{hybrid}_{\langle\pi'_1,\ldots,\pi'_\ell,f'_{\ell+1},\ldots,f'_{p(n)}\rangle,\mathcal{A}(z),\mathcal{S}(z),I}(\overline{x},n))=1]\Big|.\end{aligned}$$

We begin by proving that for every $\ell = 1, \ldots, p(n)$ there exists a negligible function $\mu_\ell(n)$ such that

$$\begin{aligned}&\Big|\Pr[D(\textsc{hybrid}_{\langle\pi'_1,\ldots,\pi'_{\ell-1},f'_\ell,\ldots,f'_{p(n)}\rangle,\mathcal{A}(z),\mathcal{S}(z),I}(\overline{x},n))=1]\\ &\qquad -\Pr[D(\textsc{hybrid}_{\langle\pi'_1,\ldots,\pi'_\ell,f'_{\ell+1},\ldots,f'_{p(n)}\rangle,\mathcal{A}(z),\mathcal{S}(z),I}(\overline{x},n))=1]\Big|\\ &\leq \frac{1}{\epsilon_\ell}\cdot\Pr\left[\exists i\in I\ \forall j\notin I\ :\ \textsc{output}_j^{\pi'_\ell}(\textsc{real}_{\pi,\mathcal{A}(z),I}(\overline{x},n))=\mathsf{corrupted}_i\right]-\mu_\ell(n).\end{aligned}$$

where $\textsc{output}_j^{\pi'_\ell}(\textsc{real}_{\pi,\mathcal{A}(z),I}(\overline{x},n)) = \mathsf{corrupted}_i$ is the event that j outputs $\mathsf{corrupted}_i$ in the real execution of π *during the execution of* π'_ℓ.

Assume by contradiction that there exists an ℓ ($1 \leq \ell \leq p(n)$) and a non-negligible function δ such that

$$\begin{aligned}&\Big|\Pr[D(\textsc{hybrid}_{\langle\pi'_1,\ldots,\pi'_{\ell-1},f'_\ell,\ldots,f'_{p(n)}\rangle,\mathcal{A}(z),\mathcal{S}(z),I}(\overline{x},n))=1]\\ &\qquad -\Pr[D(\textsc{hybrid}_{\langle\pi'_1,\ldots,\pi'_\ell,f'_{\ell+1},\ldots,f'_{p(n)}\rangle,\mathcal{A}(z),\mathcal{S}(z),I}(\overline{x},n))=1]\Big|\\ &> \frac{1}{\epsilon_\ell}\cdot\Pr\left[\exists i\in I\ \forall j\notin I\ :\ \textsc{output}_j^{\pi'_\ell}(\textsc{real}_{\pi,\mathcal{A}(z),I}(\overline{x},n))=\mathsf{corrupted}_i\right]+\delta(n)\end{aligned}\tag{1}$$

This implies that there must be a vector of inputs $\overline{x}_\ell$ for the honest parties in the ℓ-th execution (π'_ℓ or f'_ℓ) and a state s for $\mathcal{A}$ after the executions $\pi'_1, \ldots, \pi'_{\ell-1}$ such that Eq. (1) holds when the honest parties' inputs to the ℓth execution are $\overline{x}_\ell$ and the state of $\mathcal{A}$ is s. This follows from a straightforward averaging argument. Specifically, if for all possible states and vectors, Eq. (1) does not hold, then when summing over all possibilities the inequality would not be achieved. Now, consider an adversary $\mathcal{A}_{\pi'_\ell}$ that is given a state s as input and interacts with honest parties upon the input vector $\overline{x}_\ell$. The strategy of $\mathcal{A}_{\pi'_\ell}$ is to run $\mathcal{A}$ from state s, and to output the state of $\mathcal{A}$ at the end of the execution. In addition, we construct a distinguisher $D_{\pi'_\ell}$ who receives the inputs/outputs of the honest parties, the original vector $\overline{x}$ of inputs, and the output of $\mathcal{A}_{\pi'_\ell}$. $D_{\pi'_\ell}$ then emulates the rest of the ideal executions, applies D to the result and outputs whatever D outputs. For s and $\overline{x}_\ell$ as above, it follows that

$$\Pr[D_{\pi'_\ell}(\textsc{real}_{\pi'_\ell,\mathcal{A}_{\pi'_\ell}(s),I}(\overline{x}_\ell,n))=1]=\Pr[D(\textsc{hybrid}_{\langle\pi'_1,\ldots,\pi'_\ell,f'_{\ell+1},\ldots,f'_{p(n)}\rangle,\mathcal{A}(z),\mathcal{S}(z),I}(\overline{x},n))=1\mid(s,\overline{x}_\ell)]$$

where the conditioning on $(s, \overline{x}_\ell)$ means an s and $\overline{x}_\ell$ for which Eq. (1) holds. Observe now that when the ideal simulator $\mathcal{S}_\ell$ that is guaranteed to exist for π'_ℓ is applied to this adversary, the result is exactly an execution of $\textsc{hybrid}_{\langle\pi'_1,\ldots,\pi'_{\ell-1},f'_\ell,\ldots,f'_{p(n)}\rangle,\mathcal{A}(z),\mathcal{S}(z),I}(\overline{x},n)$. Thus,

$$\Pr[D_{\pi'_\ell}(\textsc{ideal}_{f'_\ell,\mathcal{S}_\ell(s),I}(\overline{x}_\ell,n))=1]=\Pr[D(\textsc{hybrid}_{\langle\pi'_1,\ldots,\pi'_{\ell-1},f'_\ell,\ldots,f'_{p(n)}\rangle,\mathcal{A}(z),\mathcal{S}(z),I}(\overline{x},n))=1\mid(s,\overline{x}_\ell)]$$

Finally, note that the probability that some P_j outputs $\mathsf{corrupted}_i$ in such a stand-alone execution of π'_ℓ equals the probability that it outputs it in π'_ℓ in a real execution of $\pi'_1, \ldots, \pi'_{p(n)}$. This is because the first ℓ executions are identical in both cases, and we are considering the event of $\mathsf{corrupted}_i$ being output before these ℓ executions end. We thus conclude that

$$\begin{aligned}
&\Big|\Pr[D_{\pi'_\ell}(\text{IDEAL}_{f'_\ell,\mathcal{S}_{\pi'_\ell}(s),I}(\overline{x}_\ell,n))=1] \\
&\qquad -\Pr[D_{\pi'_\ell}(\text{REAL}_{\pi'_\ell,\mathcal{A}_{\pi'_\ell}(s),I}(\overline{x}_\ell,n))=1]\Big| \\
&> \frac{1}{\epsilon_\ell}\cdot\Pr\left[\exists i\in I\ \forall j\notin I\ :\ \text{OUTPUT}_j(\text{REAL}_{\pi'_\ell,\mathcal{A}_{\pi'_\ell}(s),I}(\overline{x}_\ell,n))=\mathsf{corrupted}_i\right]+\delta(n)
\end{aligned}$$

in contradiction to the assumption that π'_ℓ securely computes f'_ℓ in the presence of covert adversaries with ϵ_ℓ-deterrent. Let $\epsilon=\mathsf{min}\{e_\ell\}_{\ell=1}^{p(n)}$. We conclude that

$$\begin{aligned}
&\left|\Pr[D(\text{IDEAL}_{f,\mathcal{S}(z),I}(\overline{x},n))=1]-\Pr[D(\text{REAL}_{\pi,\mathcal{A}(z),I}(\overline{x},n))=1]\right| \\
&\leq \sum_{\ell=1}^{p(n)}\frac{1}{\epsilon}\cdot\Pr\left[\exists i\in I\ \forall j\notin I\ :\ \text{OUTPUT}_j^{\pi'_\ell}(\text{REAL}_{\pi,\mathcal{A}(z),I}(\overline{x},n))=\mathsf{corrupted}_i\right]-\sum_{\ell=1}^{p(n)}\mu_\ell(n) \\
&= \frac{1}{\epsilon}\cdot\Pr\left[\exists i\in I\ \forall j\notin I\ :\ \text{OUTPUT}_j(\text{REAL}_{\pi,\mathcal{A}(z),I}(\overline{x},n))=\mathsf{corrupted}_i\right]-\mu(n)
\end{aligned}$$

for some negligible function $\mu(n)$, as required. ■

4.2 Composition for Definitions 3.3 and 3.4

In this section, we prove a modular sequential composition theorem for the stronger Definitions 3.3 and 3.4. We begin by presenting some background and notation.

The hybrid model. We consider a *hybrid model* where parties both interact with each other (as in the real model) and use trusted help (as in the ideal model). Specifically, the parties run a protocol π that contains "ideal calls" to a trusted party computing some functionalities $f_1,\ldots,f_{p(n)}$. These ideal calls are just instructions to send an input to the trusted party. Upon receiving the output back from the trusted party, the protocol π continues. The protocol π is such that f_i is called before f_{i+1} for every i (this just determines the "naming" of the calls as $f_1,\ldots,f_{p(n)}$ in that order). We stress that honest parties all send their input to the trusted party in the same round and do not send other messages until they receive back their output (this is because we consider *sequential composition* here). Of course, the trusted party may be used a number of times throughout the π-execution. However, each time is independent (i.e., the trusted party does not maintain any state between these calls). We call the regular messages of π that are sent amongst the parties standard messages and the messages that are sent between parties and the trusted party ideal messages. *We stress that in the hybrid model, the trusted party behaves as in the ideal model of the definition being considered.* Thus, when proving security in the hybrid model for Definitions 3.3 and 3.4, the trusted party computing $f_1,\ldots,f_{p(n)}$ follows the instructions of the trusted party in Definition 3.3 and 3.4, respectively. Formally, we define an (f,ϵ)-hybrid model that is the same as the regular hybrid model except that the trusted party is as in IDEALC^ϵ (when considering Definition 3.3) or as in IDEALSC^ϵ (when considering Definition 3.4).

Let $f_1,\ldots,f_{p(n)}$ be probabilistic polynomial-time functionalities and let π be an m-party protocol that uses ideal calls to a trusted party computing $f_1,\ldots,f_{p(n)}$. Furthermore, let $\mathcal{A}$ be a non-uniform probabilistic polynomial-time machine and let I be the set of corrupted parties. Then, the $f_1,\ldots,f_{p(n)}$-hybrid execution of π on inputs $\overline{x}$, auxiliary input z to $\mathcal{A}$ and security parameter n, denoted $\text{HYBRID}^{f_1,\ldots,f_{p(n)}}_{\pi,\mathcal{A}(z),I}(\overline{x})$, is defined as the output vector of the honest parties and the adversary $\mathcal{A}$ from the hybrid execution of π with a trusted party computing $f_1,\ldots,f_{p(n)}$.

Sequential modular composition. Let $f_1,\ldots,f_{p(n)}$ and π be as above, and let $\rho_1,\ldots,\rho_{p(n)}$ be protocols. We assume that each ρ_i has a fixed number rounds that is the same for all parties. Consider the real protocol $\pi^{\rho_1,\ldots,\rho_{p(n)}}$ that is defined as follows. All standard messages of π are unchanged. When a party P_i is instructed to send an ideal message x to the trusted party to compute f_j, it begins a real execution of ρ_j with input x instead. When this execution of ρ_j concludes with output y, party P_i continues with π as if y was the output received by the trusted party for f_j (i.e. as if it were running in the hybrid model). If a party receives $\mathsf{corrupted}_k$ as output from ρ_j, then it behaves as instructed in π. Note that $\mathsf{corrupted}_k$ may be received as output when ρ_j is run and when f_j is run. This is due to the fact that the ideal model used is that of IDEALC or IDEALSC, and in these ideal models parties may receive $\mathsf{corrupted}_k$ for output.

The composition theorem of [5] for malicious adversaries states that if $\rho_1,\ldots,\rho_{p(n)}$ securely compute $f_1,\ldots,f_{p(n)}$ respectively, and π securely computes some functionality g in the f-hybrid model, then $\pi^{\rho_1,\ldots,\rho_{p(n)}}$ securely computes g (in the real model). We remark that our proof below is an almost direct corollary of the theorem of [5] (after casting the models of Definitions 3.3 and 3.4 in a different, yet equivalent, model).

Theorem 4.2 *Let $p(n)$ be a polynomial, let $f_1,\ldots,f_{p(n)}$ be multiparty probabilistic polynomial-time functionalities and let $\rho_1,\ldots,\rho_{p(n)}$ be protocols that securely compute $f_1,\ldots,f_{p(n)}$ in the presence of covert adversaries with deterrent $\epsilon_1,\ldots,\epsilon_{p(n)}$, respectively. Let g be a multiparty functionality and let π be a secure protocol for computing g in the $(f_1,\epsilon_1),\ldots,(f_{p(n)},\epsilon_{p(n)})$-hybrid model (using a single call to each f_i) in the presence of covert adversaries with ϵ-deterrent. Then, $\pi^{\rho_1,\ldots,\rho_{p(n)}}$ securely computes g in the presence of covert adversaries with ϵ-deterrent. The above holds for Definitions 3.3 and 3.4 by taking the appropriate ideal model in each case.*

Proof Sketch: Theorem 4.2 can be derived as an almost immediate corollary from the composition theorem of [5] in the following way. First, define a special functionality interface that follows the instructions of the trusted party in Definition 3.3 (respectively, in Definition 3.4). That is, define a *reactive functionality* that receives inputs and writes outputs (this functionality is modeled by an interactive Turing machine). The appropriate reactive functionality here acts exactly like the trusted party (e.g., if it receives a cheat_i message when computing f_ℓ, then it tosses coins and with probability ϵ_ℓ outputs $\mathsf{corrupted}_i$ to all parties and with probability $1-\epsilon_\ell$ gives the adversary all of the honest parties' inputs and lets it chooses their outputs). Next, consider the standard ideal model of Definition 2.1 with functionalities of the above form. It is easy to see that a protocol securely computes some functionality f under Definition 3.3 (resp., under Definition 3.4) *if and only if* it is securely computes the appropriately defined reactive functionality under Definition 2.1. This suffices because the composition theorem of [5] can be applied to Definition 2.1, yielding the result.[7] ■

Observe that in Theorem 4.2 the protocols $\rho_1,\ldots,\rho_{p(n)}$ and π may all have different deterrent values. Thus the proof of π in the hybrid model must take into account the actual deterrent values $\epsilon_1,\ldots,\epsilon_{p(n)}$ of the protocols $\rho_1,\ldots,\rho_{p(n)}$, respectively.

[7] Two remarks are in place here. First, the composition theorem of [5] is formally proven for standard (non-reactive) functionalities and the case of an honest majority. Nevertheless, the proof can be extended to these cases in a straightforward way with almost no changes. Second, the composition theorem of [5] assumes a strict polynomial-time simulator. This is fine because we also required this in our definitions.

5 Oblivious Transfer

In the oblivious transfer functionality [30, 10], a sender has two inputs (x_0, x_1) and a receiver has an input bit σ. The sender receives no output (and, in particular, learns nothing about the receiver's bit), while the receiver learns x_σ (but learns nothing about $x_{1-\sigma}$). This variant of oblivious transfer is often called 1-out-of-2 oblivious transfer.

In this section we will construct an efficient oblivious transfer protocol that is secure in the presence of covert adversaries with ϵ-deterrent. We will first present the basic scheme that considers a single oblivious transfer and $\epsilon = 1/2$. We will then extend this to enable the simultaneous execution of many oblivious transfers and also higher values of ϵ. Our constructions all rely on the existence of secure homomorphic encryption schemes.

Homomorphic encryption. Intuitively, a public-key encryption scheme is homomorphic if given two ciphertexts $c_1 = E_{pk}(m_1)$ and $c_2 = E_{pk}(m_2)$ it is possible to efficiently compute $E_{pk}(m_1 + m_2)$ without knowledge of the secret decryption key. Of course this assumes that the plaintext message space is a group; we actually assume that both the plaintext and ciphertext spaces are groups (with respective group operations $+$ and $\cdot$). A natural way to define this is to require that for all pairs of keys (pk, sk), all $m_1, m_2 \in \mathcal{P}$ and $c_1, c_2 \in \mathcal{C}$ with $m_1 = D_{sk}(c_1)$ and $m_2 = D_{sk}(c_2)$, it holds that $D_{sk}(c_1 \cdot c_2) = m_1 + m_2$. However, we actually need a *stronger property*. Specifically, we require that the result of computing $c_1 \cdot c_2$ when c_2 is a random encryption of m_2 is a random encryption of $m_1 + m_2$ (by a random encryption we mean a ciphertext generated by encrypting the plaintext with uniformly distributed coins). This property ensures that if one party generated c_1 and the other party applied a series of homomorphic operations to c_1 in order to generate c, then the only thing that the first party can learn from c is the underlying plaintext. In particular, it learns nothing about the steps taken to arrive at c (e.g., it cannot know if the second party added m_3 and then m_4 where $m_2 = m_3 + m_4$ or if it just added m_2). We stress that this holds even if the first party knows the secret key of the encryption scheme. We formalize the above by requiring that the distribution of $\{pk, c_1, c_1 \cdot c_2\}$ is *identical* to the distribution of $\{pk, E_{pk}(m_1), E_{pk}(m_1+m_2)\}$, where in the latter case the encryptions of m_1 and $m_1 + m_2$ are generated independently of each other, using uniformly distributed random coins. We denote by $E_{pk}(m)$ the random variable generated by encrypting m with public-key pk using uniformly distributed random coins. We have the following formal definition.

Definition 5.1 *A public-key encryption scheme (G, E, D) is* homomorphic *if for all n and all (pk, sk) output by $G(1^n)$, it is possible to define groups $\mathcal{M}, \mathcal{C}$ such that:*

- *The plaintext space is $\mathcal{M}$, and all ciphertexts output by E_{pk} are elements of $\mathcal{C}$,*[8] *and*
- *For every $m_1, m_2 \in \mathcal{M}$ it holds that*

$$\{pk, c_1 = E_{pk}(m_1), c_1 \cdot E_{pk}(m_2)\} \equiv \{pk, E_{pk}(m_1), E_{pk}(m_1 + m_2)\} \tag{2}$$

  where the group operations are carried out in $\mathcal{C}$ and $\mathcal{M}$, respectively.

Note that in the left distribution in Eq. (2) the ciphertext c_1 is used to generate an encryption of $m_1 + m_2$ using the homomorphic operation, whereas in the right distribution the encryptions of m_1 and $m_1 + m_2$ are independent. An important observation is that any such scheme supports the

[8] The plaintext and ciphertext spaces may depend on pk; we leave this implicit.

multiplication of a ciphertext by a scalar, that can be achieved by computing multiple additions. We also assume that (G, E, D) has *no decryption errors*; this means that for every key-pair (pk, sk) in the range of $G(1^n)$ and for every m in the message space $\Pr[D_{sk}(E_{sk}(m)) = m] = 1$. Such encryption schemes can be constructed under the quadratic-residuosity, N-residuosity, decisional Diffie-Hellman (DDH) and other assumptions; see [28, 2, 23] for some references. By convention, no ciphertext is invalid. That is, any ciphertext that is not in the ciphertext group $\mathcal{C}$ is interpreted as an encryption of the identity element of the plaintext group $\mathcal{M}$.

5.1 The Basic Protocol

Protocol 5.2 (oblivious transfer from errorless homomorphic encryption):

- **Inputs:** *The sender S has a pair of strings (x_0, x_1) for input; the receiver R has a bit σ. Both parties have the security parameter 1^n as auxiliary input. (In order to satisfy the constraints that all inputs are of the same length, it is possible to define $|x_0| = |x_1| = k$ and give the receiver $(\sigma, 1^{2k-1})$.)*

- **Assumption:** *We assume that the group determined by the homomorphic encryption scheme with security parameter n is large enough to contain all strings of length k. Thus, if the homomorphic encryption scheme only works for single bits, we will only consider $k = 1$ (i.e., bit oblivious transfer).*

- **The protocol:**

  1. *The receiver R chooses two sets of two pairs of keys:*
     (a) $(pk_1^0, sk_1^0) \leftarrow G(1^n)$; $(pk_2^0, sk_2^0) \leftarrow G(1^n)$ *using random coins* r_G^0, *and*
     (b) $(pk_1^1, sk_1^1) \leftarrow G(1^n)$; $(pk_2^1, sk_2^1) \leftarrow G(1^n)$ *using random coins* r_G^1

     R sends (pk_1^0, pk_2^0) and (pk_1^1, pk_2^1) to the sender S.
  2. Key-generation challenge:
     (a) *S chooses a random coin $b \in_R \{0, 1\}$ and sends b to R.*
     (b) *R sends S the random-coins r_G^b that it used to generate (pk_1^b, pk_2^b).*
     (c) *S checks that the public keys output by the key-generation algorithm G when given input 1^n and the appropriate portions of the random-tape r_G^b equal pk_1^b and pk_2^b. If this does not hold, or if R did not send any message here, S outputs* $\mathsf{corrupted}_R$ *and halts. Otherwise, it proceeds.*
     Denote $pk_1 = pk_1^{1-b}$ and $pk_2 = pk_2^{1-b}$.
  3. *R chooses two random bits $\alpha, \beta \in_R \{0, 1\}$. Then:*
     (a) *R computes*
     $$c_0^1 = E_{pk_1}(\alpha) \qquad c_0^2 = E_{pk_2}(1 - \alpha)$$
     $$c_1^1 = E_{pk_1}(\beta) \qquad c_1^2 = E_{pk_2}(1 - \beta)$$
     using random coins r_0^1, r_0^2, r_1^1 and r_1^2, respectively.
     (b) *R sends (c_0^1, c_0^2) and (c_1^1, c_1^2) to S.*
  4. Encryption-generation challenge:
     (a) *S chooses a random bit $b' \in_R \{0, 1\}$ and sends b' to R.*
     (b) *R sends $r_{b'}^1$ and $r_{b'}^2$ to S (i.e., R sends an opening to the ciphertexts $c_{b'}^1$ and $c_{b'}^2$).*

(c) *S checks that one of the ciphertexts $\{c^1_{b'}, c^2_{b'}\}$ is an encryption of 0 and the other is an encryption of 1. If not (including the case that no message is sent by R), S outputs $\mathsf{corrupted}_R$ and halts. Otherwise, it continues to the next step.*

5. *R sends a "re-ordering" of the ciphertexts $\{c^1_{1-b'}, c^2_{1-b'}\}$. Specifically, if $\sigma = 0$ then it sets c_0 to be the ciphertext that is an encryption of 1, and sets c_1 to be the ciphertext that is an encryption of 0. Otherwise, if $\sigma = 1$ then it sets c_0 to be the encryption of 0, and c_1 to be the encryption of 1. (Only the ordering needs to be sent and not the actual ciphertexts. Furthermore, this can be sent together with the openings in Step 4b.)*

6. *S uses the homomorphic property and c_0, c_1 as follows.*

   (a) *S computes $\tilde{c}_0 = x_0 \cdot_E c_0$ (this operation is relative to the key pk_1 or pk_2 depending if c_0 is an encryption under pk_1 or pk_2)*

   (b) *S computes $\tilde{c}_1 = x_1 \cdot_E c_1$ (this operation is relative to the key pk_1 or pk_2 depending if c_1 is an encryption under pk_1 or pk_2)*

   S sends $\tilde{c}_0$ and $\tilde{c}_1$ to R. (Notice that one of the ciphertexts is encrypted with key pk_1 and the other is encrypted with key pk_2.)

7. *If $\sigma = 0$, the receiver R decrypts $\tilde{c}_0$ and outputs the result (if $\tilde{c}_0$ is encrypted under pk_1 then R outputs $x_0 = D_{sk_1}(\tilde{c}_0)$; otherwise it outputs $x_0 = D_{sk_2}(\tilde{c}_0)$). Otherwise, if $\sigma = 1$, R decrypts $\tilde{c}_1$ and outputs the result.*

8. *If at any stage during the protocol, S does not receive the next message that it expects to receive from R or the message it receives is invalid and cannot be processed, it outputs abort_R (unless it was already instructed to output $\mathsf{corrupted}_R$). Likewise, if R does not receive the next message that it expects to receive from S or it receives an invalid message, it outputs abort_S.*

We remark that the reordering message of Step 5 can actually be sent by R together with the message in Step 4b. Furthermore, the messages of the key-generation challenge can be piggybacked with later messages, as long as they conclude before the final step. We therefore have that the number of rounds of communication can be exactly *four* (each party sends two messages).

Before proceeding to the proof of security, we present the intuitive argument showing why Protocol 5.2 is secure. We begin with the case that the receiver is corrupt. First note that if the receiver follows the instructions of the protocol, it learns only a single value x_0 or x_1. This is because one of c_0 and c_1 is an encryption of 0. If it is c_0, then $\tilde{c}_0 = x_0 \cdot_E c_0 = E_{pk}(0 \cdot x_0) = E_{pk}(0)$ (where $pk \in \{pk_1, pk_2\}$, and so nothing is learned about x_0; similarly if it is c_1 then $\tilde{c}_1 = E_{pk}(0)$ and so nothing is learned about x_1. However, in general, the receiver may not generate the encryptions $c^1_0, c^1_1, c^2_0, c^2_1$ properly (and so it may that at least one of the pairs (c^1_0, c^2_0) and (c^1_1, c^2_1) are *both* encryptions of 1, in which case the receiver could learn both x_0 and x_1). This is prevented by the encryption-generation challenge. That is, if the receiver tries to cheat in this way then it is guaranteed to be caught with probability at least $1/2$. The above explains why a malicious receiver can learn only one of the outputs, unless it is willing to be caught cheating with probability $1/2$. This therefore demonstrates that "privacy" holds. However, we actually need to prove security via simulation, which involves showing how to *extract* the receiver's implicit input and how to *simulate* its view. Extraction works by first providing the corrupted receiver with the encryption-challenge bit $b' = 0$ and then rewinding it and providing it with the challenge $b' = 1$. If the corrupted receiver replies to both challenges, then the simulator can construct σ from the opened ciphertexts and the reordering provided. Given this input, the simulation can be completed in a straightforward manner; see the proof below. A crucial point here is that if the receiver does not reply to both challenges then an honest sender would output $\mathsf{corrupted}_R$ with probability 1/2, and so this corresponds to a cheat_R input in the ideal world.

We now proceed to discuss why the protocol is secure in the presence of a corrupt sender. In this case, it is easy to see that such a sender cannot learn anything about the receiver's input because the encryption scheme is semantically secure (and so a corrupt sender cannot determine σ from the unopened ciphertexts). However, as above, we need to show how extraction and simulation works. Extraction here works by providing encryptions so that in one of the pairs (c_0^1, c_0^2) or (c_1^1, c_1^2) both of the encrypted values are 1. If this pair is the one used (and not the one opened), then we have that $\tilde{c}_0$ is an encryption of x_0 and $\tilde{c}_1$ is an encryption of $\tilde{c}_1$. An important point here is that unlike a real receiver, the simulator can do this without being "caught". Specifically, the simulator generates the ciphertexts so that for a random $b' \in_R \{0,1\}$ it holds that $c_{1-b'}^1$ and $c_{1-b'}^2$ are both encryptions of 1, whereas $c_{b'}^1$ and $c_{b'}^2$ are general correctly, one being an encryption of 0 and the other an encryption of 1. Then, the simulator "hopes" that the corrupted sender asks it to open the ciphertexts $c_{b'}^1$ and $c_{b'}^2$ which look as they should. In such a case, the simulator proceeds and succeeds in extracting both x_0 and x_1. However, if the corrupted sender asks the simulator to open the other ciphertexts (that are clearly invalid), the simulator just rewinds the corrupted sender and tries again. Thus, extraction can be achieved. Regarding the simulation of the sender's view, this follows from the fact that the only differences between the above and a real execution are the values encrypted in the ciphertexts $c_0^1, c_0^2, c_1^1, c_1^2$. These distributions are therefore indistinguishable by the semantic security of the encryption scheme.

We now formally prove that Protocol 5.2 meets Definition 3.4 with $\epsilon = \frac{1}{2}$ (of course, this immediately implies security under Definitions 3.2 and 3.3 as well).

Theorem 5.3 *Assuming that (G, E, D) constitutes a semantically secure homomorphic encryption scheme (with errorless decryption), Protocol 5.2 securely computes the oblivious transfer functionality $((x_0, x_1), \sigma) \mapsto (\lambda, x_\sigma)$ in the presence of covert adversaries with ϵ-deterrent for $\epsilon = \frac{1}{2}$, under Definition 3.4.*

Proof: We will separately consider the case that no parties are corrupted, the case that the receiver is corrupted and the case that the sender is corrupted (the case that both parties are corrupted is trivial). We note that although we construct three different simulators (one for each corruption case), a single simulator as required by the definition can be constructed by simply combining the three simulators into one machine, and working appropriately given the corruption set I.

No corruptions. We first consider the case that no parties are corrupted (i.e., $I = \phi$). In this case, the real adversary $\mathcal{A}$'s view can be generated by a simulator Sim that simply runs S and R honestly, with inputs $x_0 = x_1 = 0^k$ and $\sigma = 0$ (recall that in this case we assume that the adversary's auxiliary input contains the input length k). The fact that this simulation is indistinguishable from a real execution (with the honest parties' real inputs) follows from the indistinguishability property of encryption scheme. The proof is straightforward and is therefore omitted. We remark that in order to show that the REAL and IDEAL outputs are indistinguishable, we also have to show that the honest parties' outputs in a real execution are correct (because this is the case in the ideal world). The sender's output is defined as λ and so this clearly holds. Regarding the receiver, recall that $\tilde{c}_0 = x_0 \cdot_E c_0$ and $\tilde{c}_1 = x_1 \cdot_E c_1$. Thus, if $\sigma = 0$ it holds c_0 is an encryption of 1 and so $\tilde{c}_0 = E_{pk_1}(x_0 \cdot 1) = E_{pk_1}(x_0)$; likewise, if $\sigma = 1$ then c_1 is an encryption of 1 and so $\tilde{c}_1 = E_{pk_1}(x_1)$. This implies that the receiver correctly obtains x_σ as required.

Corrupted receiver: Let $\mathcal{A}$ be a real adversary that controls the receiver R. We construct a simulator Sim that works as follows:

1. Sim receives $(\sigma, 1^{2k-1})$ and z as input and invokes $\mathcal{A}$ on this input.

2. Sim plays the honest sender with $\mathcal{A}$ as receiver.

3. When Sim reaches the key-generation challenge step, it first sends $b = 0$ and receives back $\mathcal{A}$'s response. Then, Sim rewinds $\mathcal{A}$, sends $b = 1$ and receives back $\mathcal{A}$'s response.

   (a) If both of the responses from $\mathcal{A}$ would cause a $\mathsf{corrupted}$-output (meaning a response that would cause S to output $\mathsf{corrupted}_R$ in a real execution), Sim sends $\mathsf{corrupted}_R$ to the trusted party, simulates the honest S aborting due to detected cheating, and outputs whatever $\mathcal{A}$ outputs.

   (b) If $\mathcal{A}$ sends back exactly one response that would cause a $\mathsf{corrupted}$-output, then Sim sends cheat_R to the trusted party.

      i. If the trusted party replies with $\mathsf{corrupted}_R$, then Sim rewinds $\mathcal{A}$ and hands it the query for which $\mathcal{A}$'s response would cause a $\mathsf{corrupted}$-output. Sim then simulates the honest S aborting due to detected cheating, and outputs whatever $\mathcal{A}$ outputs.

      ii. If the trusted party replies with $\mathsf{undetected}$ and the honest S's input pair (x_0, x_1), then Sim plays the honest sender with input (x_0, x_1) in the remainder of the execution with $\mathcal{A}$ as the receiver. At the conclusion, Sim outputs whatever $\mathcal{A}$ outputs.

   (c) If neither of $\mathcal{A}$'s responses would cause a $\mathsf{corrupted}$-output, then Sim rewinds $\mathcal{A}$, gives it a random b' and proceeds as below.

4. Sim receives ciphertexts $c_0^1, c_0^2, c_1^1, c_1^2$ from $\mathcal{A}$.

5. Next, in the encryption-generation challenge step, Sim first sends $b' = 0$ and receives back $\mathcal{A}$'s response, which includes the reordering of the ciphertexts (recall that the reordering message are actually sent together with the ciphertext openings). Then, Sim rewinds $\mathcal{A}$, sends $b' = 1$ and receives back $\mathcal{A}$'s response.

   (a) If both of the responses from $\mathcal{A}$ would cause a $\mathsf{corrupted}$-output, Sim sends $\mathsf{corrupted}_R$ to the trusted party, simulates the honest S aborting due to detected cheating, and outputs whatever $\mathcal{A}$ outputs.

   (b) If $\mathcal{A}$ sends back exactly one response that would cause a $\mathsf{corrupted}$-output, then Sim sends cheat_R to the trusted party.

      i. If the trusted party replies with $\mathsf{corrupted}_R$, then Sim rewinds $\mathcal{A}$ and hands it the query for which $\mathcal{A}$'s response would cause a $\mathsf{corrupted}$-output. Sim then simulates the honest S aborting due to detected cheating, and outputs whatever $\mathcal{A}$ outputs.

      ii. If the trusted party replies with $\mathsf{undetected}$ and the honest S's input pair (x_0, x_1), then Sim plays the honest sender with input (x_0, x_1) and completes the execution with $\mathcal{A}$ as the receiver. (Note that the sender has not yet used its input at this stage of the protocol. Thus, Sim has no problem completing the execution like an honest sender.) At the conclusion, Sim outputs whatever $\mathcal{A}$ outputs.

   (c) If neither of $\mathcal{A}$'s responses would cause a $\mathsf{corrupted}$-output, then Sim uses the reorderings to determine the value of σ. Specifically, Sim chooses a random b' and takes the reordering that relates to $c_{1-b'}^1$ and $c_{1-b'}^2$ (if $c_{1-b'}^1$ is an encryption of 1, then Sim determines $\sigma = 0$ and otherwise it determines $\sigma = 1$). The value b' chosen is the one that Sim sends to $\mathcal{A}$ and appears in the final transcript.

Sim sends σ to the trusted party and receives back $x = x_\sigma$. Simulator Sim then completes the execution playing the honest sender and using $x_0 = x_1 = x$.

6. If at any point $\mathcal{A}$ sends a message that would cause the honest sender to halt and output abort_R, simulator Sim immediately sends abort_R to the trusted party, halts the simulation and proceeds to the final "output" step.

7. *Output:* At the conclusion, Sim outputs whatever $\mathcal{A}$ outputs.

This completes the description of Sim. Denoting Protocol 5.2 as π and noting that I here equals $\{R\}$ (i.e., the receiver is corrupted), we need to prove that for $\epsilon = \frac{1}{2}$,

$$\left\{\text{IDEALSC}^{\epsilon}_{ot,\mathcal{S}(z),I}(((x_0, x_1), \sigma), n)\right\} \stackrel{\text{c}}{\equiv} \left\{\text{REAL}_{\pi,\mathcal{A}(z),I}(((x_0, x_1), \sigma), n)\right\}$$

It is clear that the simulation is perfect if Sim sends $\mathsf{corrupted}_R$ or cheat_R at any stage. This is due to the fact that the probability that an honest S outputs $\mathsf{corrupted}_R$ in the simulation is identical to the probability in a real execution (probability 1 in the case that $\mathcal{A}$ responds incorrectly to both challenges and probability 1/2 otherwise). Furthermore, in the case that Sim sends cheat_R and receives back undetected it concludes the execution using the true input of the sender. The simulation until the last step is perfect (it involves merely sending random challenges); therefore the completion using the true sender's input yields a perfect simulation. The above is clearly true of abort_R as well (because this can only occur before the last step where the sender's input is used).

It remains to analyze the case that Sim does not send $\mathsf{corrupted}_R$, cheat_R or abort_R to the trusted party. Notice that in this case, $\mathcal{A}$ responded correctly to both the key-generation challenges and the encryption-generation challenges. In particular, this implies that the keys pk_1 and pk_2 are correctly generated, and that Sim computes σ based on the encrypted values sent by $\mathcal{A}$ and the reordering.

Now, if $\sigma = 0$, then Sim hands $\mathcal{A}$ the ciphertexts $\tilde{c}_0 = E_{pk}(x_0)$ and $\tilde{c}_1 = E_{pk'}(0)$, where $pk, pk' \in \{pk_1, pk_2\}$ and $pk \neq pk'$, and if $\sigma = 1$, it hands $\mathcal{A}$ the ciphertexts $\tilde{c}_0 = E_{pk}(0)$ and $\tilde{c}_1 = E_{pk'}(x_1)$. This follows from the instructions of Sim and the honest party (Sim plays the honest party with $x_0 = x_1 = x_\sigma$ and so $\tilde{c}_\sigma$ is an encryption of x_σ and $\tilde{c}_{1-\sigma}$ is an encryption of 0). The important point to notice is that these messages are distributed identically to the honest sender's messages in a real protocol; the fact that Sim does not know $x_{1-\sigma}$ makes no difference because for every x' it holds that $x' \cdot E_{pk}(0) = E_{pk}(0)$. We note that this assumes that the homomorphic property of the encryption scheme holds, but this is given by the fact that pk_1 and pk_2 are correctly formed. Regarding the rest of the messages sent by Sim, these are generated independently of the sender-input and so exactly like an honest sender.

We conclude that the view of $\mathcal{A}$ as generated by the simulator Sim is *identical* to the distribution generated in a real execution. Thus, its output is identically distributed in both cases. (Since the sender receives no output, we do not need to consider the output distribution of the honest sender in the real and ideal executions.) We conclude that

$$\left\{\text{IDEALSC}^{\epsilon}_{ot,\mathcal{S}(z),I}(((x_0, x_1), \sigma), n)\right\} \equiv \left\{\text{REAL}_{\pi,\mathcal{A}(z),I}(((x_0, x_1), \sigma), n)\right\}$$

completing this corruption case.

Corrupted sender: Let $\mathcal{A}$ be a real adversary that controls the sender S. We construct a simulator Sim that works as follows:

1. Sim receives (x_0, x_1) and z and invokes $\mathcal{A}$ on this input.

2. Sim interacts with $\mathcal{A}$ and plays the honest receiver until Step 3 of the protocol.

3. In Step 3 of the protocol, Sim works as follows:

   (a) Sim chooses random bits $b, \alpha \in_R \{0,1\}$

   (b) Sim computes:

$$c_b^1 = E_{pk_1}(\alpha) \qquad c_b^2 = E_{pk_2}(1-\alpha)$$
$$c_{1-b}^1 = E_{pk_1}(1) \qquad c_{1-b}^2 = E_{pk_2}(1)$$

   (c) Sim sends $c_0^1, c_0^2, c_1^1, c_1^2$ to $\mathcal{A}$.

4. In the next step (Step 4 of the protocol), $\mathcal{A}$ sends a bit b'. If $b' = b$, then Sim opens the ciphertexts c_b^1 and c_b^2 as the honest receiver would (note that the ciphertexts are "correctly" constructed). Otherwise, Sim returns to Step 3 of the simulation above (i.e., it returns to the beginning of Step 3 of the protocol) and tries again with fresh randomness.[9]

5. Sim sends a random reordering of the ciphertexts c_{1-b}^1 and c_{1-b}^2 (the actual order doesn't matter because they are both encryptions of 1).

6. The simulator Sim receives from $\mathcal{A}$ the ciphertexts $\tilde{c}_0$ and $\tilde{c}_1$. Sim computes $x_0 = D_{sk_1}(\tilde{c}_0)$ and $x_1 = D_{sk_2}(\tilde{c}_1)$ (or $x_0 = D_{sk_2}(\tilde{c}_0)$ and $x_1 = D_{sk_1}(\tilde{c}_1)$, depending on which of c_0, c_1 is encrypted with pk_1 and which with pk_2), and sends the pair (x_0, x_1) to the trusted party as S's input.

7. If at any stage in the simulation $\mathcal{A}$ does not respond, or responds with an invalid message that cannot be processed, then Sim sends abort_S to the trusted party for the sender's inputs. (Such behavior from $\mathcal{A}$ can only occur before the last step and so before any input (x_0, x_1) has already been sent to the trusted party.)

8. Sim outputs whatever $\mathcal{A}$ outputs.

Notice that Sim never sends cheat_S to the trusted party. Thus we actually prove standard security in this corruption case. That is, we prove that:

$$\left\{\text{IDEAL}_{ot,\mathsf{Sim}(z),I}((x_0,x_1,\sigma),n)\right\} \stackrel{\text{c}}{\equiv} \left\{\text{REAL}_{\pi,\mathcal{A}(z),I}((x_0,x_1,\sigma),n)\right\} \tag{3}$$

By Proposition 3.8, this implies security for covert adversaries as well. In order to prove Eq. (3), observe that the only difference between the view of the adversary $\mathcal{A}$ in a real execution and in the simulation by Sim is due to the fact that Sim does not generate c_b^1, c_b^2 correctly. Thus, intuitively, Eq. (3) follows from the security of the encryption scheme. That is, we begin by showing that if the view of $\mathcal{A}$ in the real and ideal executions can be distinguished, then it is possible to break the security of the encryption scheme. We begin by showing that the view of $\mathcal{A}$ when interacting with an honest sender with input $\sigma = 0$ is indistinguishable from the view of $\mathcal{A}$ when interacting in a simulation with Sim.

Let $\mathcal{A}'$ be an adversary that attempts to distinguish encryptions under a key pk.[10] Adversary $\mathcal{A}'$ receives a key pk, chooses a random bit $\gamma \in_R \{0,1\}$ and a random index $\ell \in_R \{1,2\}$ and sets

[9] This yields an expected polynomial-time simulation because these steps are repeated until $b' = b$. A strict polynomial-time simulation can be achieved by just halting after n attempts. The probability that $b' \neq b$ in all of these attempts can be shown to be negligible, based on the hiding property of the encryption scheme.

[10] The game that $\mathcal{A}'$ plays is that it receives a key pk, outputs a pair of plaintexts m_0, m_1, receives back a *challenge ciphertext* $E_{pk}(m_b)$ for some $b \in \{0,1\}$, and outputs a "guess" bit b'. An encryption scheme is indistinguishable if the probabilities that $\mathcal{A}'$ outputs $b' = 1$ when $b = 1$ and when $b = 0$ are negligibly close.

$pk_\ell^{1-\gamma} = pk$. It then chooses the keys $pk_{3-\ell}^{1-\gamma}$, pk_1^γ and pk_2^γ by itself and sends $\mathcal{A}$ the keys (pk_1^0, pk_2^0) and (pk_1^1, pk_2^1). When $\mathcal{A}$ replies with a bit b, adversary $\mathcal{A}'$ acts as follows. If $b = \gamma$, then $\mathcal{A}'$ opens the randomness used in generating (pk_1^b, pk_2^b) as the honest receiver would ($\mathcal{A}'$ can do this because it chose $(pk_1^\gamma, pk_2^\gamma)$ by itself and $\gamma = b$). If $b \neq \gamma$, then $\mathcal{A}'$ cannot open the randomness as an honest receiver would. Therefore, $\mathcal{A}'$ just halts. If $\mathcal{A}$ continues, then it sets $pk_1 = pk_1^{1-\gamma}$ and $pk_2 = pk_2^{1-\gamma}$ (and so pk_ℓ is the public-key pk that $\mathcal{A}'$ is "attacking"). Now, $\mathcal{A}'$ computes the ciphertexts $c_0^1, c_0^2, c_1^1, c_1^2$ in the following way. $\mathcal{A}'$ chooses α and β at random, as the honest receiver would. Then, for a random ζ adversary $\mathcal{A}'$ computes $c_\zeta^1 = E_{pk_1}(\alpha)$, $c_\zeta^2 = E_{pk_2}(1-\alpha)$, and $c_{1-\zeta}^{3-\ell} = E_{pk_{3-\ell}}(1)$. However, $\mathcal{A}'$ does not compute $c_{1-\zeta}^{\ell} = E_{pk_\ell}(1)$. Rather, it outputs a pair of plaintexts $m_0 = 0, m_1 = 1$ and receives back $c = E_{pk}(m_b) = E_{pk_\ell}(m_b)$ (for $b \in_R \{0,1\}$). Adversary $\mathcal{A}'$ sets $c_{1-\zeta}^{\ell} = c$ (i.e., to equal the challenge ciphertext) and continues playing the honest receiver until the end. In this simulation, $\mathcal{A}'$ sets the reordering so that c_0 equals $c_{1-\zeta}^{3-\ell}$ (that is, it is an encryption of 1). The key point here is that if $\mathcal{A}'$ does not halt and $b = 0$, then the simulation by $\mathcal{A}'$ is identical to a real execution between $\mathcal{A}$ and an honest receiver R who has input $\sigma = 0$ (because $c_0 = c_{1-\zeta}^{3-\ell}$ is an encryption of 1 and $c_1 = c_{1-\zeta}^{\ell}$ is an encryption of 0, as required). In contrast, if $\mathcal{A}'$ does not halt and $b = 1$, then the simulation by $\mathcal{A}'$ is identical to the simulation carried out by Sim (because in this case they are both encryptions of 1). Finally, note that $\mathcal{A}'$ halts with probability exactly 1/2 in both cases (this is due to the fact that the distribution of the keys is identical for both choices of γ). Combining the above together, we have that if it is possible to distinguish the view of $\mathcal{A}$ in the simulation by Sim from a real execution with a receiver who has input 0, then it is possible to distinguish encryptions. Specifically, $\mathcal{A}'$ can just run the distinguisher that exists for these views and output whatever the distinguisher outputs.

The above shows that the view of $\mathcal{A}$ in the simulation is indistinguishable from its view in a real execution with an honest receiver with input $\sigma = 0$. However, we actually have to show that when the honest receiver has input $\sigma = 0$, the *joint distribution* of $\mathcal{A}$ and the honest receiver's outputs in a real execution is indistinguishable from the joint distribution of Sim and the honest receiver's outputs in the ideal model. The point to notice here is that the output of the honest receiver in both the real and ideal models is the value obtained by decrypting $\tilde{c}_0$ using key $pk_{3-\ell}$. (In the real model this is what the protocol instructs the honest party to output and in the ideal model this is the value that Sim sends to the trusted party as the sender's input x_0.) However, in this reduction $\mathcal{A}'$ knows the associated secret-key to $pk_{3-\ell}$, because it chose $pk_{3-\ell}$ itself. Thus, $\mathcal{A}'$ can append the decryption of $\tilde{c}_0$ to the view of $\mathcal{A}$, thereby generating a joint distribution. It follows that if $\mathcal{A}'$ received an encryption of $m_0 = 0$ then it generates the joint distribution of the outputs in the real execution, and if it received an encryption of $m_1 = 1$ then it generates the joint distribution of the outputs in the ideal execution. By the indistinguishability of the encryption scheme we have the real and ideal distributions are indistinguishable, completing the proof of Eq. (3) for the case that $\sigma = 0$. The case for $\sigma = 1$ follows from an almost identical argument as above. Combining these two cases, we have the output distribution generated by the simulator in the ideal model is computationally indistinguishable from the output distribution of a real execution. It remains to show that Sim runs in expected polynomial-time. Note that Sim rewinds if in the simulation it holds that $b' \neq b$. Now, in the case that the ciphertexts $c_0^1, c_0^2, c_1^1, c_1^2$ are generated as by the honest party (each pair containing an encryption of 0 and an encryption of 1), the probability that $b' \neq b$ is exactly 1/2 because the value of b' is information-theoretically hidden. In contrast, in the simulation this is not the case because c_b^1, c_b^2 are "correctly" constructed, whereas c_{1-b}^1, c_{1-b}^2 are both encryptions of 1. Nevertheless, if the probability that $b' \neq b$ is non-negligibly far from 1/2, then this can be used to distinguish an encryption of 0 from an encryption of 1 (the actual reduction can be derived from the reduction already carried out above and is thus omitted). It follows that the expected number of rewindings is at most slightly greater than 2, implying that the overall simulation runs in expected polynomial-time. As we have mentioned in Footnote 9, the simulation can be made to run in strict polynomial-time by aborting if for n consecutive trials it holds that $b' \neq b$. By the argument given above, such an abort can only occur with negligible probability. This concludes the proof of this corruption case, and thus of the theorem. ■

Discussion. The proof of Protocol 5.2 in the case that the receiver is corrupted relies heavily on the fact that the simulator can send cheat and therefore does not need to complete a "standard" simulation. Take for example the case that $\mathcal{A}$ (controlling the receiver) only replies with one valid response to the encryption-generation challenge. In this case, the receiver can learn both x_0 and x_1 with probability 1/2. However, the simulator in the ideal model can never learn both x_0 and x_1. Therefore, the simulator cannot generate the correct distribution. However, by allowing the simulator to declare a cheat, it can complete the simulation as required. This demonstrates why it is possible to achieve higher efficiency for this definition of security. We remark that the above protocol is *not* non-halting detection accurate (see Definition 3.5). For example, a cheating receiver can send $c_0^1 = E_{pk_1}(\alpha)$ and $c_0^2 = E_{pk_1}(\alpha)$. Then, if the sender chooses $b' = 1$ (thus testing c_1^1 and c_1^2), the adversary succeeds in cheating and learning both of the sender's inputs. However, if the sender chooses $b' = 0$, the receiver can just abort at this point. This means that such an early abort must be considered an attempt to cheat, and so a sender running with a fail-stop receiver must also output $\mathsf{corrupted}_R$.

The proof of security for a corrupted sender. We stress that we have actually proven something stronger. Specifically, we have shown that Protocol 5.2 is secure in the presence of a covert receiver with 1/2-deterrent as stated. However, we have also shown that Protocol 5.2 is (fully) secure with abort in the presence of a *malicious sender*.

Efficiently recognizable public keys. We remark that in the case that it is possible to efficiently recognize that a public-key is in the range of the key-generator of the public-key encryption scheme, it is possible to skip the key-generation challenge step in the protocol (the sender can verify for itself if the key is valid).

5.2 Extensions

String oblivious transfer. In Protocol 5.2, x_0 and x_1 are elements in the group over which the homomorphic encryption scheme is defined. If this group is large, then we can carry out string oblivious transfer. This is important because later we will use Protocol 5.2 to exchange symmetric encryption keys. However, if the group contains only 0 and 1, then this does not suffice. In order to extend Protocol 5.2 to deal with string oblivious transfer, even when the group has only two elements, we only need to change the last two steps of the protocol. Specifically, instead of S computing a single encryption for x_0 and a single encryption for x_1, it computes an encryption for each bit. That is, denote the bits of x_0 by $x_0^1, \ldots, x_0^n$, and likewise for x_1. Then, S computes:

$$\tilde{c}_0 = x_0^1 \cdot_E c_0, \ldots, x_0^n \cdot_E c_0 \quad \text{and} \quad \tilde{c}_1 = x_1^1 \cdot_E c_1, \ldots, x_1^n \cdot_E c_1 \ .$$

Note that the receiver can still only obtain one of the strings because if $\sigma = 0$ then $\tilde{c}_1$ just contains encryptions to zeroes, and vice versa if $\sigma = 1$.

Simultaneous oblivious transfer. We will use Protocol 5.2 in Yao's protocol for secure two-party computation. This means that we will run one oblivious transfer for every bit of the input. In principle, these oblivious transfers can be run in parallel, as long as the protocol being used remains secure under parallel composition. The classical notion of parallel composition considers the setting where the honest parties run each execution obliviously of the others (this is often called "stateless composition"). We do not know how to prove that our protocol composes in parallel in this sense. Nevertheless, we can modify Protocol 5.2 so that it is possible to simultaneously run many oblivious transfers with a cost that is *less* than running Protocol 5.2 the same number of times in parallel. We call this *simultaneous oblivious transfer* in order to distinguish it from "parallel oblivious transfer" (which considers stateless parallel composition, as described above). The simultaneous oblivious transfer functionality is defined as follows:

$$((x_1^0, x_1^1), \ldots, (x_n^0, x_1^n), (\sigma_1, \ldots, \sigma_n)) \mapsto (\lambda, (x_1^{\sigma_1}, \ldots, x_n^{\sigma_n}))$$

Thus, we essentially have n oblivious transfers where in the i^{th} such transfer, the sender has input (x_i^0, x_i^1) and the receiver has input σ_i.

The extension to Protocol 5.2 works as follows. First, the same public-key pair (pk_1, pk_2) can be used in all executions. Therefore, Steps 1 and 2 remain unchanged. Then, Step 3 is carried out *independently* for all n bits $\sigma_1, \ldots, \sigma_n$. That is, for every i, two pairs of ciphertexts encrypting 0 and 1 (in random order) are sent. The important change comes in Step 4. Here, the *same* challenge bit b' is used for every i. The sender then replies as it should, opening the $c_{b'}^1$ and $c_{b'}^2$ ciphertexts for every i. The protocol then concludes by the sender computing the $\tilde{c}_0$ and $\tilde{c}_1$ ciphertexts for every i, and the receiver decrypting.

The proof of the above extension is almost identical to the proof of Theorem 5.3. The main point is that since only a single challenge is used for both the key-generation challenge and encryption-generation challenge, the probability of achieving $b' = b$ (as needed for the simulation) and $b = \gamma$ (as needed for the reduction to the security of the encryption scheme) remains one half. Furthermore, the probability that a corrupted R will succeed in cheating remains the same because if there is any i for which the encryptions are not correctly formed, then the receiver will be caught with probability one half.

Higher values of ϵ. Finally, we show how it is possible to obtain higher values of ϵ with only minor changes to Protocol 5.2. The basic idea is to increase the probability of catching a corrupted receiver in the case that it attempts to generate an invalid key-pair or send ciphertexts in Step 3 that do not encrypt the same value. Let $k = \text{poly}(n)$ be an integer. Then, first the receiver generates k pairs of public-keys $(pk_1^1, pk_2^1), \ldots, (pk_1^k, pk_2^k)$ instead of just two pairs. The sender then asks the receiver to reveal the randomness used in generating all the pairs except for one (the unrevealed key-pair is the one used in the continuation of the protocol). Note that if a corrupted receiver generated even one key-pair incorrectly, then it is caught with probability $1 - 1/k$. Likewise, in Step 3, the receiver sends k pairs of ciphertexts where in each pair one ciphertext is an encryption of 0 and the other an encryption of 1. Then, the sender asks the receiver to open all pairs of encryptions of σ_i except for one pair. Clearly, the sender still learns nothing about σ because the reordering is only sent on the ciphertext pair that is not opened. Furthermore, if the receiver generates even one pair of ciphertexts so that the ciphertexts are not correctly formed, then it will be caught with probability $1 - 1/k$. The rest of the protocol remains the same. We conclude that the resulting protocol is secure in the presence of covert adversaries with ϵ-deterrent where $\epsilon = 1 - 1/k$. Notice that this works as long as k is polynomial in the security parameter and thus ϵ can be made to be very close to 1, if desired. (Of course, this methodology *cannot* be used to make ϵ negligibly close to 1, because then k has to be super-polynomial.)

Summary. We conclude with the following theorem, derived by combining the extensions above:

Theorem 5.4 *Assume that there exist semantically secure homomorphic encryption schemes with errorless decryption. Then, for any $k = \text{poly}(n)$ there exists a protocol that securely computes the* simultaneous string oblivious transfer *functionality*

$$((x_1^0, x_1^1), \ldots, (x_n^0, x_1^n), (\sigma_1, \ldots, \sigma_n)) \mapsto (\lambda, (x_1^{\sigma_1}, \ldots, x_n^{\sigma_n}))$$

in the presence of covert adversaries with ϵ-deterrent for $\epsilon = 1 - \frac{1}{k}$. Furthermore, the protocol has four rounds of communication, and involves generating $2k$ encryption keys, carrying out $2kn$ encryption operations, $2n$ homomorphic multiplications and n decryptions.

Note that the amortized complexity of each oblivious transfer is: $2k$ encryptions, 2 scalar multiplications with the homomorphic encryption scheme and 1 decryption. (The key generation which is probably the most expensive is run $2k$ times independently of n. Therefore, when many oblivious transfers are run, this becomes insignificant.)

6 Secure Two-Party Computation

In this section, we show how to securely compute any two-party functionality in the presence of covert adversaries. We present a protocol for the strong explicit cheat formulation, with parameters that can be set to obtain a wide range of values for the ϵ-deterrent. Our protocol is based on Yao's protocol for semi-honest adversaries [31]. We will base our description on the write-up of [24] of this protocol, and will assume familiarity with it. Nevertheless, in Appendix A, we briefly describe Yao's garbled circuit construction and present an important lemma regarding it.

6.1 Overview of the Protocol

The original protocol of Yao is not secure when the parties may be malicious. Intuitively, there are two main reasons for this. First, the circuit constructor P_1 may send P_2 a garbled circuit that computes a completely different function. Second, the oblivious transfer protocol that is used when the parties can be malicious must be secure for this case. The latter problem is solved here by using the protocol guaranteed by Theorem 5.4. The first problem is solved by having P_1 send P_2 a number of garbled circuits; denote this number by ℓ. Then, P_2 asks P_1 to open all but one of the circuits (chosen at random) in order to check that they are correctly constructed. This opening takes place before P_1 sends the keys corresponding to its input, so nothing is revealed by opening the circuits. The protocol then proceeds similarly to the semi-honest case. The main point here is that if the unopened circuit is correct, then this will constitute a secure execution that can be simulated. However, if it is not correct, then with probability $1-1/\ell$ party P_1 will have been caught cheating and so P_2 will output $\mathsf{corrupted}_1$ (recall, ℓ denotes the number of circuits sent). While the above intuition forms the basis for our protocol, the actual construction of the appropriate simulator is somewhat delicate, and requires a careful construction of the protocol. We note some of these subtleties hereunder.

First, it is crucial that the oblivious transfers are run before the garbled circuits are sent by P_1 to P_2. This is due to the fact that the simulator sends a corrupted P_2 a fake garbled circuit that evaluates to the exact output received from the trusted party (and only this output), as described in Lemma A.1. However, in order for the simulator to receive the output from the trusted party, it must first send it the input used by the corrupted P_2. This is achieved by first running the oblivious transfers, from which the simulator is able to extract the corrupted P_2's input.

The second subtlety relates to an issue we believe may be a problem for many other implementations of Yao that use cut-and-choose. The problem is that the adversary can construct (at least in theory) a garbled circuit with two sets of keys, where one set of keys decrypt the circuit to the specified one and another set of keys decrypt the circuit to an incorrect one. This is a problem because the adversary can supply "correct keys" to the circuits that are opened and "incorrect keys" to the circuit that is computed. Such a strategy cannot be carried out without risk of detection for the keys that are associated with P_2's input because these keys are obtained by P_2 in the oblivious transfers *before* the garbled circuits are even sent (thus if incorrect keys are sent for one of the circuits, P_2 will detect this if that circuit is opened). However, it is possible for a corrupt P_1 to carry out this strategy for the input wires associated with its own input. We prevent this by having P_1 commit to these keys and send the commitments together with the garbled circuits. Then, instead of P_1 just sending the keys associated with its input, it sends the appropriate decommitments.

A third subtlety that arises is connected to the difference between Definitions 3.2 and 3.3 (where the latter is the stronger definition where the decision by the adversary to cheat is not allowed to depend on the honest parties' inputs or on the output). Consider a corrupted P_1 that behaves exactly like an honest P_1 except that in the oblivious transfers, it inputs an invalid key in the place of the key associated with 0 as the first bit of P_2. The result is that if the first bit of P_2's input is 1, then the protocol succeeds and no problem arises. However, if the first bit of P_2's input is 0, then the protocol will always fail and P_2 will always detect cheating. Thus, P_1's decision to cheat may depend on P_2's private input, something that is impossible in the ideal models of Definitions 3.3 and 3.4. In summary, this means that such a protocol achieves Definition 3.2 (with $\epsilon = 1/\ell$) but not Definition 3.3. In order to solve this problem, we use a circuit that computes the function $g(x_1, x_2^1, \ldots, x_2^m) = f(x_1, \oplus_{i=1}^m x_2^i)$, instead of a circuit that directly computes f. Then, upon input x_2, party P_2 chooses random $x_2^1, \ldots, x_2^{m-1}$ and sets $x_2^m = (\oplus_{i=1}^{m-1} x_2^i) \oplus x_2$. This makes no difference to the result because $\oplus_{i=1}^m x_2^i = x_2$ and so $g(x_1, x_2^1, \ldots, x_2^m) = f(x_1, x_2)$. However, this modification makes every bit of P_2's input uniform when considering any proper subset of $x_2^1, \ldots, x_2^m$. This helps because as long as P_1 does not provide invalid keys for all m shares of x_2, the probability of failure is independent of P_2's actual input (because any set of $m-1$ shares is independent of x_2). Since $m-1$ invalid shares are detected with probability $1 - 2^{-m+1}$ we have that P_2 detects the cheating by P_1 with this probability, independently of its input value. This method was previously used in [25] (however, there they must set m to equal the security parameter).

Intuitively, an adversary can cheat by providing an incorrect circuit or by providing invalid keys for shares. However, it is then detected with the probabilities described above. Below, we show that when using ℓ circuits and splitting P_2's input into m shares, we obtain $\epsilon = (1 - 1/\ell)(1 - 2^{-m+1})$. This enables us to play around with the values of m and ℓ in order to optimize efficiency versus ϵ-deterrent. For example, if we wish to obtain $\epsilon = 1/2$ we can use the following parameters:

1. *Set $\ell = 2$ and $m = n$:* This yields $\epsilon = (1 - 1/2)(1 - 2^{-n+1})$ which is negligibly close to 1/2. However, since in Yao's protocol we need to run an oblivious transfer for every one of P_2's input bits, this incurs a blowup of the number of oblivious transfers (and thus exponentiations) by n. Thus, this setting of parameters results in a considerable computational blowup.

2. *Set $\ell = 3$ and $m = 3$:* This yields $\epsilon = (1 - 1/3)(1 - 1/4) = 1/2$. The computational cost incurred here is much less than before because we only need 3 oblivious transfers for each of P_2's input bits. Furthermore, the cost of sending 3 circuits is not much greater than 2, and so the overall complexity is much better.

Before proceeding to the protocol, we provide one more example of parameters. In order to achieve $\epsilon = 9/10$ it is possible to set $\ell = 25$ and $m = 5$ (setting $\ell = m = 10$ gives 0.898 which is very close). This gives a significantly higher value of ϵ. We remark that such a setting of ϵ also assumes a value of $\epsilon = 9/10$ for the oblivious transfer protocol. As we have seen, this involves a blowup of 5 times more computation than for oblivious transfer with $\epsilon = 1/2$.

6.2 The Protocol for Two-Party Computation

We are now ready to describe the actual protocol.

Protocol 6.1 (two-party computation of a function f):

- **Inputs:** *Party P_1 has input x_1 and party P_2 has input x_2, where $|x_1| = |x_2|$. In addition, both parties have parameters ℓ and m, and a security parameter n. For simplicity, we will assume that the lengths of the inputs are n.*

- **Auxiliary input:** *Both parties have the description of a circuit C for inputs of length n that computes the function f. The input wires associated with x_1 are $w_1, \ldots, w_n$ and the input wires associated with x_2 are $w_{n+1}, \ldots, w_{2n}$.*

- **The protocol:**

  1. *Parties P_1 and P_2 define a new circuit C' that receives $m+1$ inputs $x_1, x_2^1, , \ldots, x_2^n$ each of length n, and computes the function $f(x_1, \oplus_{i=1}^m x_2^i)$. Note that C' has $n + mn$ input wires. Denote the input wires associated with x_1 by $w_1, \ldots, w_n$, and the input wires associated with x_2^i by $w_{n+(i-1)m+1}, \ldots, w_{n+im}$, for $i = 1, \ldots, n$.*
  2. *Party P_2 chooses $m-1$ random strings $x_2^1, \ldots, x_2^{m-1} \in_R \{0,1\}^n$ and defines $x_2^m = (\oplus_{i=1}^{m-1} x_2^i) \oplus x_2$, where x_2 is P_2's original input (note that $\oplus_{i=1}^m x_2^i = x_2$). The value $z_2 \stackrel{\text{def}}{=} x_2^1, \ldots, x_2^m$ serves as P_2's new input of length mn to C'. (The input wires associated with P_2's new input are $w_{n+1}, \ldots, w_{n+mn}$.)*
  3. *For each $i = 1, \ldots, mn$ and $\beta = 0, 1$, party P_1 chooses ℓ encryption keys by running $G(1^n)$, the key generator for the encryption scheme, ℓ times. The j^{th} key associated with a given i and β is denoted $k_{w_{n+i},\beta}^j$; note that this is the key associated with the bit β for the input wire w_{n+i} in the j^{th} circuit. The result is an ℓ-tuple, denoted:*

     $$[k_{w_{n+i},\beta}^1, \ldots, k_{w_{n+i},\beta}^\ell]$$

     (This tuples constitutes the keys that are associated with the bit β for the input wire w_{n+i} in all ℓ circuits.)
  4. *P_1 and P_2 run mn executions of an oblivious transfer protocol, as follows. In the i^{th} execution, party P_1 inputs the pair*

     $$\left([k_{w_{n+i},0}^1, \ldots, k_{w_{n+i},0}^\ell], [k_{w_{n+i},1}^1, \ldots, k_{w_{n+i},1}^\ell]\right)$$

     and party P_2 inputs the bit z_2^i (P_2 receives the keys $[k_{w_{n+i},z_2^i}^1, \ldots, k_{w_{n+i},z_2^i}^\ell]$ as output). The executions are run using a simultaneous oblivious transfer functionality, as in Theorem 5.4. If a party receives a $\mathsf{corrupted}_i$ or abort_i message as output from the oblivious transfer, it outputs it and halts.

5. *Party P_1 constructs ℓ garbled circuits $GC_1, \ldots, GC_\ell$ using independent randomness (the circuits are garbled versions of C' described above). The keys for the input wires $w_{n+1}, \ldots,$ w_{n+mn} in the garbled circuits are taken from above (i.e., in GC_j the keys associated with w_{n+i} are $k^j_{w_{n+i},0}$ and $k^j_{w_{n+i},1}$). The keys for the inputs wires $w_1, \ldots, w_n$ are chosen randomly, and are denoted in the same way.*

   P_1 sends the ℓ garbled circuits to P_2.

6. *P_1 commits to the keys associated with its inputs. That is, for every $i = 1, \ldots, n$, $\beta = 0, 1$ and $j = 1, \ldots, \ell$, party P_1 computes*

$$c^j_{w_i,\beta} = \mathsf{Com}(k^j_{w_i,\beta}; r^j_{i,\beta})$$

   where Com is a perfectly-binding commitment scheme, $\mathsf{Com}(x; r)$ denotes a commitment to x using randomness r, and $r^j_{i,\beta}$ is a random string of sufficient length to commit to a key of length n.

   P_1 sends all of the above commitments. The commitments are sent as ℓ vectors of pairs (one vector for each circuit); in the j^{th} vector the i^{th} pair is $\{c^j_{w_i,0}, c^j_{w_i,1}\}$ in a random order (the order is randomly chosen independently for each pair).

7. *Party P_2 chooses a random index $\gamma \in_R \{1, \ldots, \ell\}$ and sends γ to P_1.*

8. *P_1 sends P_2 all of the keys for the input wires in all garbled circuits except for GC_γ (this enables a complete decryption of the garbled circuit), together with the associated mappings and the decommitment values. (I.e. for every $i = 1, \ldots, n + mn$ and $j \neq \gamma$, party P_1 sends the keys and mappings $(k^j_{w_i,0}, 0), (k^j_{w_i,1}, 1)$. In addition, for every $i = 1, \ldots, n$ and $j \neq \gamma$ it sends the decommitments $r^j_{i,0}, r^j_{i,1}$.)*

9. *P_2 checks that everything that it received is in order. That is, it checks:*
   - *That the keys it received for all input wires in circuits GC_j ($j \neq \gamma$) indeed decrypt the circuits (when using the received mappings), and the decrypted circuits are all C'.*
   - *That the decommitment values correctly open all the commitments $c^j_{w_i,\beta}$ that were received, and these decommitments reveal the keys $k^j_{w_i,\beta}$ that were sent for P_1's wires.*
   - *That the keys received in the oblivious transfers earlier match the appropriate keys that it received in the opening (i.e., if it received $[k^1_i, \ldots, k^\ell_i]$ in the i^{th} oblivious transfer, then it checks that k^j_i from the transfer equals $k^j_{w_{n+i},z^i_2}$ from the opening).*

   If all the checks pass, it proceeds to the next step. If not, it outputs $\mathsf{corrupted}_1$ and halts. In addition, if P_2 does not receive this message at all, it outputs $\mathsf{corrupted}_1$.

10. *P_1 sends decommitments to the input keys associated with its input for the unopened circuit GC_γ. That is, for $i = 1, \ldots, n$, party P_1 sends P_2 the key $k^\gamma_{w_i,x_i}$ and decommitment r^γ_{i,x_i}, where x_i is the i^{th} bit of P_1's input.*

11. *P_2 checks that the values received are valid decommitments to the commitments received above. If not, it outputs abort_1. If yes, it uses the keys to compute $C'(x_1, z_2) = C'(x_1, x^1_2, \ldots, x^m_2) = C(x_1, x_2)$, and outputs the result. If the keys are not correct (and so it is not possible to compute the circuit), or if P_2 doesn't receive this message at all, it outputs abort_1.*

Note that steps 7–10 are actually a single step of P_1 sending a message to P_2, followed by P_2 carrying out a computation. If during the execution, any party fails to receive a message or it receives one that is ill-formed, it outputs abort_i (where P_i is the party who failed to send the message). This holds unless the party is explicitly instructed above to output $\mathsf{corrupted}_i$ instead (as in Step 9).

For reference throughout the proof, we provide a high-level diagram of the protocol in Figure 1.

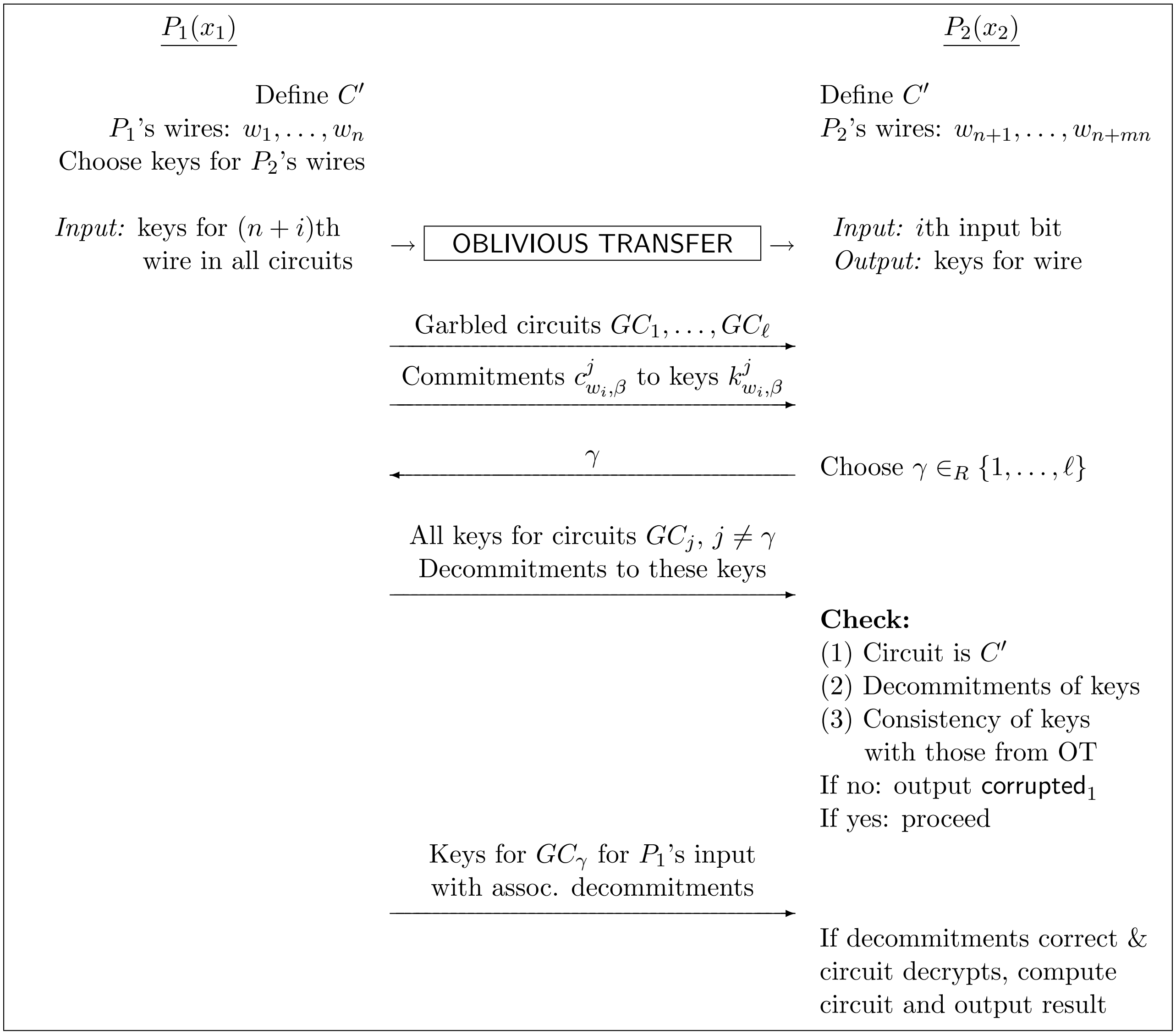

Figure 1: A high-level diagram of the protocol

We have motivated the protocol construction above and thus proceed directly to prove its security. Note that we assume that the oblivious transfer protocol is secure with the same ϵ as above (of course, one can also use an oblivious transfer protocol that is secure in the presence of malicious adversaries, because this is secure in the presence of covert adversaries for any ϵ).

Theorem 6.2 *Let ℓ and m be parameters in the protocol that are both upper-bound by* $\text{poly}(n)$*, and set $\epsilon = (1-1/\ell)(1-2^{-m+1})$. Let f be any probabilistic polynomial-time function. Assume that the encryption scheme used to generate the garbled circuits has indistinguishable encryptions under chosen-plaintext attacks (and has an elusive and efficiently verifiable range), and that the oblivious transfer protocol used is secure in the presence of covert adversaries with ϵ-deterrent according to Definition 3.4. Then, Protocol 6.1 securely computes f in the presence of covert adversaries with ϵ-deterrent according to Definition 3.4.*

Proof: Our analysis of the security of the protocol is in the (OT, ϵ)-hybrid model, where the parties are assumed to have access to a trusted party computing the oblivious transfer functionality following the ideal model of 3.4; see Section 4. Thus the simulator that we describe will play the trusted party in the oblivious transfer, when simulating for the adversary. We separately consider the different corruption cases (when no parties are corrupted, and when either one of the parties is corrupted). In the case that no parties are corrupted, the security reduces to the semi-honest case which has already been proven in [24] (the additional steps in Protocol 6.1 don't make a difference here).

Party P_2 is corrupted. Intuitively, the security in this case relies on the fact that P_2 can only learn a single set of keys in the oblivious transfers and thus can decrypt the garbled circuit to only a single value as required. Formally, let $\mathcal{A}$ be a probabilistic polynomial-time adversary controlling P_2. The simulator $\mathcal{S}$ fixes $\mathcal{A}$'s random-tape to a uniformly distributed tape and works as follows:

1. $\mathcal{S}$ chooses ℓ sets of mn random keys as P_1 would.

2. $\mathcal{S}$ plays the trusted party for the oblivious transfers with $\mathcal{A}$ as the receiver. $\mathcal{S}$ receives the input that $\mathcal{A}$ sends to the trusted party (as its input as receiver to the oblivious transfers):

   (a) If the input is abort_2 or $\mathsf{corrupted}_2$, then $\mathcal{S}$ sends abort_2 or $\mathsf{corrupted}_2$ (respectively) to the trusted party computing f, simulates P_1 aborting and halts (outputting whatever $\mathcal{A}$ outputs).

   (b) If the input is cheat_2, then $\mathcal{S}$ sends cheat_2 to the trusted party. If it receives back $\mathsf{corrupted}_2$, then it hands $\mathcal{A}$ the message $\mathsf{corrupted}_2$ as if it received it from the trusted party, simulates P_1 aborting and halts (outputting whatever $\mathcal{A}$ outputs). If it receives back $\mathsf{undetected}$ (and thus P_1's input x_1 as well), then $\mathcal{S}$ works as follows. First, it hands $\mathcal{A}$ the string $\mathsf{undetected}$ together with the nm random keys that it chose (note that $\mathcal{A}$ expects to receive the inputs of P_1 to the oblivious transfers in the case of $\mathsf{undetected}$). Next, $\mathcal{S}$ uses the input x_1 of P_1 that it received in order to perfectly emulate P_1 in the rest of the execution. That is, it runs P_1's honest strategy with input x_1 while interacting with $\mathcal{A}$ playing P_2 for the rest of the execution. Let y_1 be the output for P_1 that it receives. $\mathcal{S}$ sends y_1 to the trusted party (for P_1's output) and outputs whatever $\mathcal{A}$ outputs. The simulation ends here in this case.

   (c) If the input is a series of bits $z_2^1, \ldots, z_2^{mn}$, then $\mathcal{S}$ hands $\mathcal{A}$ the keys from above that are "chosen" by the z_2^i bits, and proceeds with the simulation below.

3. $\mathcal{S}$ defines $x_2 = \oplus_{i=0}^{m-1}(z_2^{i \cdot n+1}, \ldots, z_2^{i \cdot n+n})$ and sends x_2 to the trusted party computing f. $\mathcal{S}$ receives back some output y.

4. $\mathcal{S}$ chooses a random value ζ and computes the garbled circuits GC_j for $j \neq \zeta$ correctly (using the appropriate input keys from above as P_1 would). However, for the garbled circuit GC_ζ, the simulator $\mathcal{S}$ does not use the true circuit for computing f but rather a circuit $\widetilde{GC}$ that always evaluates to y (the value it received from the trusted party), using Lemma A.1. $\mathcal{S}$ uses the appropriate input keys from above also in generating GC_ζ. $\mathcal{S}$ also computes commitments to the keys associated with P_1's input in an honest way.

5. $\mathcal{S}$ sends $GC_1, \ldots, GC_\ell$ and the commitments to $\mathcal{A}$ and receives back an index γ.

6. If $\gamma \neq \zeta$ then $\mathcal{S}$ rewinds $\mathcal{A}$ and returns to Step 4 above (using fresh randomness).

Otherwise, if $\gamma = \zeta$, then $\mathcal{S}$ opens all the commitments and garbled circuits GC_j for $j \neq \gamma$, as the honest P_1 would, and proceeds to the next step.

7. $\mathcal{S}$ hands $\mathcal{A}$ arbitrary keys associated with the input wires of P_1. That is, for $i = 1, \ldots, n$, $\mathcal{S}$ hands $\mathcal{A}$ an arbitrary one of the two keys associated with the input wire w_i in GC_γ (one key per wire), together with its correct decommitment.

8. If at any stage, $\mathcal{S}$ does not receive a response from $\mathcal{A}$, it sends abort_2 to the trusted party (resulting in P_1 outputting abort_2). If the protocol proceeds successfully to the end, $\mathcal{S}$ sends $\mathsf{continue}$ to the trusted party and outputs whatever $\mathcal{A}$ outputs.

Denoting Protocol 6.1 as π and $I = \{2\}$ (i.e., party P_2 is corrupted), we prove that:

$$\left\{\text{IDEALSC}^{\epsilon}_{f,\mathcal{S}(z),I}((x_1, x_2), n)\right\} \stackrel{\text{c}}{\equiv} \left\{\text{HYBRID}^{ot,\epsilon}_{\pi,\mathcal{A}(z),I}((x_1, x_2), n)\right\} \tag{4}$$

In order to prove Eq. (4) we separately consider the cases of abort (including a "$\mathsf{corrupted}$" input), cheat or neither. If $\mathcal{A}$ sends abort_2 or $\mathsf{corrupted}_2$ as the oblivious transfer input, then $\mathcal{S}$ sends abort_2 or $\mathsf{corrupted}_2$ (respectively) to the trusted party computing f. In both cases the honest P_1 outputs the same (abort_2 or $\mathsf{corrupted}_2$) and the view of $\mathcal{A}$ is identical. Thus, the IDEAL and HYBRID output distributions are identical. The exact same argument is true if $\mathcal{A}$ sends cheat_2 and the reply to $\mathcal{S}$ from the trusted party is $\mathsf{corrupted}_2$. In contrast, if $\mathcal{A}$ sends cheat_2 and $\mathcal{S}$ receives back the reply $\mathsf{undetected}$, then the execution does not halt immediately. Rather, $\mathcal{S}$ plays the honest P_1 with its input x_1. Since $\mathcal{S}$ follows the exact same strategy as P_1, and the output received by P_1 from the execution is the same y_1 that $\mathcal{S}$ receives from the protocol execution, it is clear that once again the output distributions are identical (recall that in the ideal model, P_1 outputs the same y_1 obtained by $\mathcal{S}$). We remark that the probability of the trusted party answering $\mathsf{corrupted}_2$ or $\mathsf{undetected}$ is the same in the hybrid and ideal executions (i.e., ϵ), and therefore the output distributions in the cases of abort, $\mathsf{corrupted}$ or cheat are identical. We denote the event that $\mathcal{A}$ sends an abort, $\mathsf{corrupted}$ or cheat message in the oblivious transfers by bad_{OT}. Thus, we have shown that

$$\left\{\text{IDEALSC}^{\epsilon}_{f,\mathcal{S}(z),I}((x_1, x_2), n) \mid \mathsf{bad}_{\text{OT}}\right\} \equiv \left\{\text{HYBRID}^{ot,\epsilon}_{\pi,\mathcal{A}(z),I}((x_1, x_2), n) \mid \mathsf{bad}_{\text{OT}}\right\}$$

We now show that the IDEAL and HYBRID distributions are computationally indistinguishable in the case that $\mathcal{A}$ sends valid input in the oblivious transfer phase (i.e., in the event $\neg\mathsf{bad}_{\text{OT}}$). In order to show this, we consider a modified simulator $\mathcal{S}'$ who is also given the honest party P_1's real input x_1. Simulator $\mathcal{S}'$ works exactly as $\mathcal{S}$ does, except that it constructs GC_ζ honestly, and not as $\widetilde{GC}$ from Lemma A.1. Furthermore, in Step 7 it sends the keys associated with P_1's input x_1 and not arbitrary keys. It is straightforward to verify that the distribution generated by $\mathcal{S}'$ is identical to the distribution generated by $\mathcal{A}$ in an execution of the real protocol. This is due to the fact that all ℓ circuits received by $\mathcal{A}$ are honestly constructed and the keys that it receives from $\mathcal{S}'$ are associated with P_1's real input. The only difference is the rewinding. However, since ζ is chosen uniformly, this has no effect on the output distribution. Thus:

$$\left\{\text{IDEALSC}^{\epsilon}_{f,\mathcal{S}'(z,x_1),I}((x_1, x_2), n) \mid \neg\mathsf{bad}_{\text{OT}}\right\} \equiv \left\{\text{HYBRID}^{ot}_{\pi,\mathcal{A}(z),I}((x_1, x_2), n) \mid \neg\mathsf{bad}_{\text{OT}}\right\}$$

Next we prove that conditioned on the event that bad_{OT} does not occur, the distributions generated by $\mathcal{S}$ and $\mathcal{S}'$ are computationally indistinguishable. That is,

$$\left\{\text{IDEALSC}^{\epsilon}_{f,\mathcal{S}(z),I}((x_1, x_2), n) \mid \neg\mathsf{bad}_{\text{OT}}\right\} \stackrel{\text{c}}{\equiv} \left\{\text{IDEALSC}^{\epsilon}_{f,\mathcal{S}'(z,x_1),I}((x_1, x_2), n) \mid \neg\mathsf{bad}_{\text{OT}}\right\}$$

In order to see this, notice that the only difference between $\mathcal{S}$ and $\mathcal{S}'$ is in the construction of the garbled circuit GC_ζ. By Lemma A.1 it follows immediately that these distributions are computationally indistinguishable. (Note that we do not need to consider the joint distribution of $\mathcal{A}$'s view and P_1's output because P_1 has no output from Protocol 6.1.) This yields the above equation. In order to complete the proof of Eq. (4), note that the probability that the event $\mathsf{bad}_{\rm OT}$ happens is *identical* in the IDEAL and HYBRID executions. This holds because the oblivious transfer is the first step of the protocol and $\mathcal{A}$'s view in this step with $\mathcal{S}$ is identical to its view in a protocol execution with a trusted party computing the oblivious transfer functionality. Combining this fact with the above equations we derive Eq. (4).

We remark that the simulator $\mathcal{S}$ described above runs in expected polynomial-time. In order to see this, note that by Lemma A.1, a fake garbled circuit is indistinguishable from a real one. Therefore, the probability that $\gamma = \zeta$ is at most negligibly far from $1/\ell$ (otherwise, this fact alone can be used to distinguish a fake garbled circuit from a real one). It follows that the expected number of attempts by $\mathcal{S}$ is close to ℓ, and so its expected running-time is polynomial (by the assumption on ℓ). By our definition, $\mathcal{S}$ needs to run in strict polynomial-time. However, this is easily achieved by having $\mathcal{S}$ halt if it fails after $n\ell$ rewinding attempts. Following the same argument as above, such a failure can occur with at most negligible probability.

We conclude that $\mathcal{S}$ meets the requirements of Definition 3.4. (Note that $\mathcal{S}$ only sends cheat_2 due to the oblivious transfer. Thus, if a "fully secure" oblivious transfer protocol were to be used, the protocol would meet the standard definition of security for malicious adversaries for the case that P_2 is corrupted.)

Party P_1 is corrupted. The proof of security in this corruption case is considerably more complex. Intuitively, security relies on the fact that if P_1 does not construct the circuits correctly or does not provide the same keys in the oblivious transfers and circuit openings, then it will be caught with probability at least ϵ. In contrast, if it does construct the circuits correctly and provide the same keys, then its behavior is effectively the same as an honest party and so security is preserved. Formally, let $\mathcal{A}$ be an adversary controlling P_1. The simulator $\mathcal{S}$ works as follows:

1. $\mathcal{S}$ invokes $\mathcal{A}$ and plays the trusted party for the oblivious transfers with $\mathcal{A}$ as the sender. $\mathcal{S}$ receives the input that $\mathcal{A}$ sends to the trusted party (as its input to the oblivious transfers):
   (a) If the input is abort_1 or $\mathsf{corrupted}_1$, then $\mathcal{S}$ sends abort_1 or $\mathsf{corrupted}_1$ (respectively) to the trusted party computing f, simulates P_2 aborting and halts (outputting whatever $\mathcal{A}$ outputs).
   (b) If the input is cheat_1, then $\mathcal{S}$ sends cheat_1 to the trusted party. If it receives back $\mathsf{corrupted}_1$, then it hands $\mathcal{A}$ the message $\mathsf{corrupted}_1$ as if it received it from the trusted party, simulates P_2 aborting and halts (outputting whatever $\mathcal{A}$ outputs). If it receives back $\mathsf{undetected}$ (and thus P_2's input x_2 as well), then $\mathcal{S}$ works as follows. First, it hands $\mathcal{A}$ the string $\mathsf{undetected}$ together with the input string z_2 that an honest P_2 upon input x_2 would have used in the oblivious transfers (note that $\mathcal{A}$ expects to receive P_2's input to the oblivious transfers in the case of $\mathsf{undetected}$). We remark that $\mathcal{S}$ can compute z_2 by simply following the instructions of an honest P_2 with input x_2 from the start (nothing yet has depended on P_2's input so there is no problem of consistency). Next, $\mathcal{S}$ uses the derived input z_2 that it computed above in order to perfectly emulate P_2 in the rest of the execution. That is, it continues P_2's honest strategy with input z_2 while interacting with $\mathcal{A}$ playing P_1 for the rest of the execution. Let y_2 be the output for P_2 that it receives. $\mathcal{S}$ sends y_2 to the trusted party (for P_2's output) and outputs whatever $\mathcal{A}$ outputs. The simulation ends here in this case.

(c) If the input is a series of mn pairs of ℓ-tuples of keys

$$\Big([k^1_{w_{n+i},0}, \ldots, k^\ell_{w_{n+i},0}], [k^1_{w_{n+i},1}, \ldots, k^\ell_{w_{n+i},1}]\Big)$$

for $i = 1, \ldots, mn$, then $\mathcal{S}$ proceeds below.

2. $\mathcal{S}$ receives from $\mathcal{A}$ a message consisting of ℓ garbled circuits $GC_1, \ldots, GC_\ell$ and a series of commitments.

3. For $j = 1, \ldots, \ell$, simulator $\mathcal{S}$ sends $\mathcal{A}$ the message $\gamma = j$, receives its reply and rewinds $\mathcal{A}$ back to the point before $\mathcal{A}$ receives γ.

4. $\mathcal{S}$ continues the simulation differently, depending on the validity of the circuit openings. In order to describe the cases, we introduce some terminology.

Legitimate circuit: We say that a garbled circuit GC_j is *legitimate* if in at least one of its openings, in response to a challenge $\gamma \neq j$, it is decrypted to the auxiliary input circuit C'. Note that if a circuit is legitimate then in all valid decryptions of the circuit (for all $\gamma \neq j$) it decrypts to C'. Furthermore, if a circuit is *illegitimate* then in *all* openings it is not correctly decrypted.

Inconsistent key: This notion relates to the question of whether the keys provided by P_1 in the oblivious transfers are the same as those committed to and thus revealed in a circuit opening. We say that a (committed) key $k^j_{w_i,\beta}$ received in an oblivious transfer is inconsistent if it is different from the analogous key committed to by P_1. We stress that the keys obtained in the oblivious transfers (and of course the committed keys) are *fixed* before this point of the simulation and thus this event is well defined.

Inconsistent wire: A wire w_i is inconsistent if there *exists* a circuit GC_j such that either $k^j_{w_i,0}$ or $k^j_{w_i,1}$ is an inconsistent key.

Totally inconsistent input: An original input bit x^i_2 is *totally inconsistent* if all of the wires associated with the shares of x^i_2 are inconsistent (recall that x^i_2 is split over m input wires). Note that the different inconsistent wires need not be inconsistent in the same circuit, nor need they be inconsistent with respect to the same value (0 or 1). Note that the determination that a wire is inconsistent is independent of the value γ sent by $\mathcal{S}$ because the oblivious transfers and commitments to keys take place before $\mathcal{S}$ sends γ in step 3 above.

Before proceeding to describe how $\mathcal{S}$ works, we remark that our strategy below is to have $\mathcal{S}$ use the different possibilities regarding the legitimacy of circuit and consistency of keys to cause the honest party in an ideal execution to output $\mathsf{corrupted}_1$ with the same probability as the honest P_2 catches $\mathcal{A}$ cheating in a real execution. Furthermore $\mathcal{S}$ does this while ensuring that γ is uniformly distributed and the bits chosen as shares of each x^i_2 are also uniformly distributed. In this light, we describe the expected probabilities of catching $\mathcal{A}$ in three cases:

- *There exists an illegitimate circuit GC_{j_0}:* in this case P_2 certainly catches $\mathcal{A}$ cheating unless $\gamma = j_0$. Thus, P_2 catches $\mathcal{A}$ with probability at least $1 - 1/\ell$. We stress that P_2 may catch $\mathcal{A}$ with higher probability depending on whether or not there are other illegitimate circuits of inconsistent inputs.

- *There exists a totally inconsistent wire:* if the inconsistent values of the wire belong to different circuits then P_2 will always catch $\mathcal{A}$. However, if they belong to one circuit GC_{j_0} then $\mathcal{A}$ will be caught if $\gamma \neq j_0$, or if $\gamma = j_0$ and the keys chosen in the oblivious transfer are all consistent (this latter event happens with probability at most 2^{-m+1} because $m-1$ bits of the sharing are chosen randomly. Thus, P_2 catches $\mathcal{A}$ with probability at least $(1-\ell^{-1})(1-2^{m+1})$.
- *None of the above occurs but there are inconsistent keys:* in this case, P_2 catches $\mathcal{A}$ if the inconsistent keys are those chosen and otherwise does not.

We are now ready to proceed. $\mathcal{S}$ works according to the follows cases:

(a) *Case 1 – at least one circuit is illegitimate:* Let GC_{j_0} be the first illegitimate circuit. Then, $\mathcal{S}$ sends $w_1 = \mathsf{cheat}_1$ to the trusted party. By the definition of the ideal model, with probability $\epsilon = (1-1/\ell)(1-2^{-m+1})$ it receives the message $\mathsf{corrupted}_1$, and with probability $1-\epsilon$ it receives the message $\mathsf{undetected}$ together with P_2's input x_2:

  i. If $\mathcal{S}$ receives the message $\mathsf{corrupted}_1$ from the trusted party, then it chooses $\gamma \neq j_0$ at random and sends γ to $\mathcal{A}$. Then, $\mathcal{S}$ receives back $\mathcal{A}$'s opening for the circuits, including the illegitimate circuit GC_{j_0}, and simulates P_2 aborting due to detected cheating. $\mathcal{S}$ then outputs whatever $\mathcal{A}$ outputs and halts.

  ii. If $\mathcal{S}$ receives the message $\mathsf{undetected}$ from the trusted party (together with P_2's input x_2), then with probability $p = \frac{\ell^{-1}}{1-\epsilon}$ it sets $\gamma = j_0$, and with probability $1-p$ it chooses $\gamma \neq j_0$ at random. It then sends γ to $\mathcal{A}$, and continues to the end of the execution emulating the honest P_2 with the input x_2 it received from the trusted party. (When computing the circuit, $\mathcal{S}$ takes the keys from the oblivious transfer that P_2 would have received when using input x_2 and when acting as the honest P_2 to define the string z_2.) Let y_2 be the output that $\mathcal{S}$ received when playing P_2 in this execution. $\mathcal{S}$ sends y_2 to the trusted party (to be the output of P_2) and outputs whatever $\mathcal{A}$ outputs). Note that if the output of P_2 in this emulated execution would have been $\mathsf{corrupted}_1$ then $\mathcal{S}$ sends $y_2 = \mathsf{corrupted}_1$ to the trusted party.[11]
  (We remark that below we will show below that the above probabilities result in γ being uniformly distributed in $\{1,\dots,\ell\}$.)

(b) *Case 2 – All circuits are legitimate but there is a totally inconsistent input:* Let x_2^i be the first totally inconsistent input and, for brevity, assume that the inconsistent keys are all for the 0-value on the wires (i.e. there are inconsistent keys $k^{j_1}_{w_{n+(i-1)m+1},0},\dots,k^{j_m}_{w_{n+im},0}$ for some $j_1,\dots,j_m \in \{1,\dots,\ell\}$). In this case, $\mathcal{S}$ sends $w_1 = \mathsf{cheat}_1$ to the trusted party. With probability ϵ it receives the message $\mathsf{corrupted}_1$, and with probability $1-\epsilon$ it receives the message $\mathsf{undetected}$ together with P_2's input x_2:

  i. If $\mathcal{S}$ receives the message $\mathsf{corrupted}_1$ from the trusted party, then it chooses random values for the bits on the wires $w_{n+(i-1)m+1},\dots,w_{n+im-1}$, subject to the constraints that not all are 1; i.e. at least one of these wires gets a value with an inconsistent key.[12] Let $w_{n+(i-1)m+t}$ be the first of these that is 0, and let G_{j_0} be the first circuit

[11]We remark that P_2 may output $\mathsf{corrupted}_1$ with probability that is higher than ϵ (e.g., if more than one circuit is illegitimate or if inconsistent keys are presented as well). This possibility is dealt with by having $\mathcal{S}$ play P_2 and force a $\mathsf{corrupted}_1$ output if this would have occurred in the execution.

[12]Recall that the input wires associated with P_2's input bit x_2^i are $w_{n+(i-1)m+1},\dots,w_{n+im}$. Thus, the simulator here fixes the values on all the wires except the last (recall also that the first $m-1$ values plus P_2's true input bit fully determine the value for the last wire w_{n+im}).

for which the key of this wire is inconsistent. $\mathcal{S}$ chooses $\gamma \neq j_0$ at random and sends it to $\mathcal{A}$. Among other things, $\mathcal{S}$ receives back $\mathcal{A}$'s opening of GC_{j_0}, and simulates P_2's aborting due to detected cheating. (Note that the probability that a real P_2 will make these two choices – choose the values for the first $m-1$ wires so that not all are 1, and choose $\gamma \neq j_0$ – is exactly ϵ.) $\mathcal{S}$ then outputs whatever $\mathcal{A}$ outputs and halts.

ii. If $\mathcal{S}$ receives the message undetected (and thus the real input x_2 of P_2) from the trusted party, it first determines the values for the shares of x_2^i and for the value γ, as follows:

- With probability $p = \frac{2^{-m+1}}{1-\epsilon}$, for all $t = 1, \ldots, m-1$ it sets the value on the wire $w_{n+(i-1)m+t}$ to equal 1 (corresponding to *not* choosing the inconsistent keys), and the value on the wire w_{n+im} to equal the XOR of x_2^i with the values set on the wires $w_{n+(i-1)m+1}, \ldots, w_{n+(i-1)m+m-1}$. The value γ is chosen at random (out of $1, \ldots, \ell$).
- With probability $1-p$, for all $t = 1, \ldots, m-1$ it sets the value on the wire $w_{n+(i-1)m+t}$ to a random value, subject to the constraint that not all are 1 (i.e. at least one of the shares has an inconsistent key), and it sets the value on the wire w_{n+im} to equal the XOR of x_2^i with the values set on the wires $w_{n+(i-1)m+1}, \ldots, w_{n+(i-1)m+m-1}$. Let $w_{n+(i-1)m+t}$ be the first wire that is 0, and let j_0 be the first circuit for which the key of this share is inconsistent. Then $\mathcal{S}$ sets $\gamma = j_0$.

The values for shares of all other input bits are chosen at random (subject to the constraint that their XOR is the input value obtained from the trusted party, as an honest P_2 would choose). S now sends γ to $\mathcal{A}$, and completes the execution emulating an honest P_2 using these shares and γ. It outputs whatever $\mathcal{A}$ would output, and sets P_2's output to whatever P_2 would have received in the executions, including $\mathsf{corrupted}_1$, if this would be the output (this is as described at the end of step 4(a)ii above).

(c) *Case 3 – All circuits are legitimate and there is no totally inconsistent input:* For each inconsistent wire (i.e. a wire for which there *exists* an inconsistent key), if there are any, $\mathcal{S}$ chooses a random value, and checks whether the value it chose corresponds to an inconsistent key. There are two cases:

i. *Case 3a – $\mathcal{S}$ chose bits with inconsistent keys:* In this case, $\mathcal{S}$ sends $w_1 = \mathsf{cheat}_1$ to the trusted party. With probability ϵ it receives the message $\mathsf{corrupted}_1$, and with probability $1-\epsilon$ it receives the message undetected together with P_2's input x_2. Let w_{i_0} be the first of the wires for which the bit chosen has an inconsistent key, and let GC_{j_0} be the first circuit in which the key is inconsistent:

A. If $\mathcal{S}$ receives the message $\mathsf{corrupted}_1$ from the trusted party, then it chooses $\gamma \neq j_0$ at random and sends it to $\mathcal{A}$. $\mathcal{S}$ then simulates P_2 aborting due to detected cheating. $\mathcal{S}$ then outputs whatever $\mathcal{A}$ outputs and halts.

B. If $\mathcal{S}$ receives the message undetected, together with $x_2 = (x_2^1, \ldots, x_2^n)$, from the trusted party, then first it chooses bits for the remaining (consistent) shares at random, subject to the constraint that for any input bit x_2^i, the XOR of all its shares equals the value of this bit, as provided by the trusted party. In addition:

- With probability $p = \frac{\ell-1}{1-\epsilon}$, simulator $\mathcal{S}$ sets $\gamma = j_0$.
- With probability $1-p$, simulator $\mathcal{S}$ chooses $\gamma \neq j_0$ at random.

In both cases, $\mathcal{S}$ sends γ to $\mathcal{A}$ and completes the execution emulating an honest P_2 using the above choice of shares, and outputting the values as explained in step 4(a)ii above (in particular, if the output of the emulated P_2 is $\mathsf{corrupted}_1$, then $\mathcal{S}$ causes this to be the output of P_2 in the ideal model).

ii. *Case 3b – $\mathcal{S}$ chose only bits with consistent keys:* $\mathcal{S}$ reaches this point of the simulation if all garbled circuits are legitimate and if either all keys are consistent or it is simulating the case that no inconsistent keys were chosen. Thus, intuitively, the circuit and keys received by $\mathcal{S}$ from $\mathcal{A}$ are the same as from an honest P_1. The simulator $\mathcal{S}$ begins by choosing a random γ and sending it to $\mathcal{A}$. Then, $\mathcal{S}$ receives the opening of the other circuits, as before. In addition, $\mathcal{S}$ receives from $\mathcal{A}$ the set of keys and decommitments (for the wires $w_1, \ldots, w_n$) for the unopened circuit GC_γ. If anything in this process is invalid (i.e. any of the circuits is not correctly decrypted, or the decommitments are invalid, or the keys cannot be used in the circuit), then $\mathcal{S}$ sends abort_1 or $\mathsf{corrupted}_1$ to the trusted party causing P_2 to output abort_1 or $\mathsf{corrupted}_1$, respectively (the choice of whether to send abort_1 or $\mathsf{corrupted}_1$ is according to the protocol description and what causes P_2 to output abort_1 and what causes it to output $\mathsf{corrupted}_1$). Otherwise, $\mathcal{S}$ uses the opening of the circuit GC_γ obtained above, together with the keys obtained in order to derive the input x_1' used by $\mathcal{A}$. Specifically, in step 3, the simulator $\mathcal{S}$ receives the opening of all circuits and this reveals the association between the keys on the input wires and the input values. Thus, when $\mathcal{A}$ sends the set of keys associated with its input in circuit GC_γ, simulator $\mathcal{S}$ can determine the exact input x_1' that is defined by these keys. $\mathcal{S}$ sends the trusted party x_1' (and $\mathsf{continue}$) and outputs whatever $\mathcal{A}$ outputs.

This concludes the description of $\mathcal{S}$. For reference throughout the analysis below, we present a high-level outline and summary of the simulator in Figures 2 and 3. We present it in the form of a "protocol" between the simulator $\mathcal{S}$ and the real adversary $\mathcal{A}$.

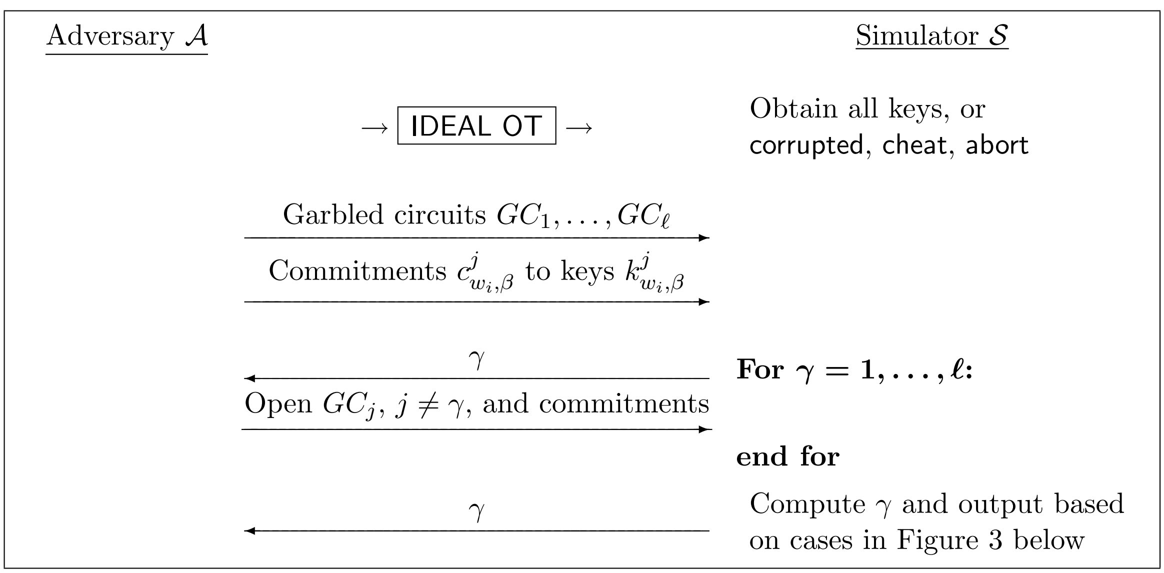

Figure 2: A high-level diagram of the simulator (P_1 corrupted)

Case 1 – at least one illegitimate circuit: Send cheat_1 to trusted party. Then:

1. If receive $\mathsf{corrupted}_1$: set $\gamma \neq j_0$ at random.
2. If receive $\mathsf{undetected}$: with probability $p = \frac{\ell^{-1}}{1-\epsilon}$ set $\gamma = j_0$; with probability $1-p$ set $\gamma \neq j_0$ at random; complete execution using real x_2.

Case 2 – there exists a totally inconsistent input: Send cheat_1 to trusted party.

1. If receive $\mathsf{corrupted}_1$: choose values for inconsistent input so at least one inconsistent key chosen. Set $\gamma \neq j_0$ at random.
2. If receive $\mathsf{undetected}$: with probability $p = \frac{2^{-m+1}}{1-\epsilon}$ choose values so no inconsistent key chosen and choose $\gamma \in_R \{1, \ldots, \ell\}$; with probability $1-p$ choose values so at least one inconsistent key chosen and set $\gamma = j_0$.

   Complete execution using real x_2.

Case 3 – all other cases: choose random values for inconsistent wires (if exist).

1. If chose a bit with an inconsistent key: send cheat_1. If receive $\mathsf{corrupted}_1$ set $\gamma \neq j_0$ at random. If receive $\mathsf{undetected}$, choose rest of values under constraint that consistent with real input of P_2. With probability $p = \frac{\ell^{-1}}{1-\epsilon}$ set $\gamma = j_0$; with probability $1-p$ choose $\gamma \neq j_0$ at random.
2. If no inconsistent keys chosen: derive input from keys and openings sent by $\mathcal{A}$. Send it to trusted party and conclude simulation (checking for abort or $\mathsf{corrupted}$ as in protocol specification).

Figure 3: Cases for the simulator $\mathcal{S}$ (P_1 corrupted)

Denote by $\mathsf{bad}_{\rm OT}$ the event that $\mathcal{A}$ sends abort_1, $\mathsf{corrupted}_1$ or cheat_1 in the oblivious transfers. The analysis of the event $\mathsf{bad}_{\rm OT}$ is identical to the case that P_2 is corrupted and so denoting π as Protocol 6.1 and $I = \{1\}$ (i.e., party P_1 is corrupted), we have that:

$$\left\{\text{IDEALSC}^{\epsilon}_{f,\mathcal{S}(z),I}((x_1,x_2),n) \mid \mathsf{bad}_{\rm OT}\right\} \equiv \left\{\text{HYBRID}^{ot}_{\pi,\mathcal{A}(z),I}((x_1,x_2),n) \mid \mathsf{bad}_{\rm OT}\right\}$$

It remains to analyze the case that $\neg\mathsf{bad}_{\rm OT}$ (i.e., the oblivious transfer is not aborted). We will prove the case following the same case analysis as in the description of the simulator. Before doing so, notice that the only messages that $\mathcal{A}$ receives in a protocol execution are in the oblivious transfers and the challenge value γ. Thus, when analyzing Protocol 6.1 in a hybrid model with a trusted party computing the oblivious transfer functionality, its view consists only of the value γ. Thus, in order to show that $\mathcal{A}$'s view in the simulation is indistinguishable from its view in a real execution, it suffices to show that the value γ that $\mathcal{S}$ hands $\mathcal{A}$ is (almost) uniformly distributed in $\{1, \ldots, \ell\}$. We stress that this is not the case when considering the *joint distribution* including P_2's output (because cheating by $\mathcal{A}$ can cause P_2 to output an incorrect value). The focus of the proof below is thus to show that the distribution over the challenge value γ sent by $\mathcal{S}$ during the simulation is uniform, and that the joint distribution over $\mathcal{A}$'s view and the output of P_2 in the simulation is statistically close to a real execution.

1. *Case 1 – at least one circuit is illegitimate:* We first show that the value γ sent by $\mathcal{S}$ in the simulation is uniformly distributed over $\{1, \ldots, \ell\}$, just like the value sent by P_2 in a real execution. In order to see this, we distinguish between the case that $\mathcal{S}$ receives $\mathsf{corrupted}_1$ and the case that it receives $\mathsf{undetected}$. We first prove that $\gamma = j_0$ with probability $1/\ell$:

$$\begin{aligned}\Pr[\gamma = j_0] &= \Pr[\gamma = j_0 \mid \mathsf{corrupted}_1]\Pr[\mathsf{corrupted}_1] + \Pr[\gamma = j_0 \mid \mathsf{undetected}]\Pr[\mathsf{undetected}] \\ &= 0 \cdot \Pr[\mathsf{corrupted}_1] + \frac{\ell^{-1}}{1-\epsilon} \cdot \Pr[\mathsf{undetected}] \\ &= \frac{1}{\ell} \cdot \frac{1}{1-\epsilon} \cdot (1-\epsilon) = \frac{1}{\ell}\end{aligned}$$

where the second equality is by the simulator's code, and the third follows from the fact that $\Pr[\mathsf{undetected}] = 1 - \epsilon$, by definition. We now proceed to prove that for every $j \neq j_0$ it also holds that $\Pr[\gamma = j] = 1/\ell$. For every $j = 1, \ldots, \ell$ with $j \neq j_0$:

$$\begin{aligned}
\Pr[\gamma = j] &= \Pr[\gamma = j \mid \mathsf{corrupted}_1]\Pr[\mathsf{corrupted}_1] + \Pr[\gamma = j \mid \mathsf{undetected}]\Pr[\mathsf{undetected}] \\
&= \Pr[\gamma = j \mid \mathsf{corrupted}_1] \cdot \epsilon + \Pr[\gamma = j \mid \mathsf{undetected}] \cdot (1 - \epsilon) \\
&= \left(\frac{1}{\ell - 1}\right) \cdot \epsilon + \left(\left(1 - \frac{1}{\ell(1-\epsilon)}\right) \cdot \frac{1}{\ell - 1}\right) \cdot (1 - \epsilon) \\
&= \frac{1}{\ell - 1} \cdot \left(\epsilon + \left(1 - \frac{1}{\ell(1-\epsilon)}\right) \cdot (1 - \epsilon)\right) \\
&= \frac{1}{\ell - 1} \cdot \left(\epsilon + (1 - \epsilon) - \frac{1-\epsilon}{\ell(1-\epsilon)}\right) \\
&= \frac{1}{\ell - 1} \cdot \left(1 - \frac{1}{\ell}\right) = \frac{1}{\ell}
\end{aligned}$$

where, once again, the third equality is by the code of the simulator. (Recall that when $\mathsf{undetected}$ is received, then with probability $1 - p$ for $p = \frac{\ell^{-1}}{(1-\epsilon)}$ the value γ is uniformly distributed under the constraint that it does not equal j_0. Thus, when $\mathsf{undetected}$ occurs, the probability that γ equals a given $j \neq j_0$ is $\frac{1}{\ell-1}$ times $1 - p$.)

We now proceed to show that the joint distribution of $\mathcal{A}$'s view and P_2's output in a real execution (or more exactly, a hybrid execution where the oblivious transfers are computed by a trusted party) is identical to the joint distribution of $\mathcal{S}$ and P_2's output in an ideal execution. We show this separately for the case that $\gamma \neq j_0$ and the case that $\gamma = j_0$. Now, when a real P_2 chooses $\gamma \neq j_0$, then it always outputs $\mathsf{corrupted}_1$. Likewise, in an ideal execution where the trusted party sends $\mathsf{corrupted}_1$ to P_2, the simulator $\mathcal{S}$ sets $\gamma \neq j_0$. Thus, when $\gamma \neq j_0$, the honest party outputs $\mathsf{corrupted}_1$ in both the real and ideal executions. Next consider the case that $\gamma = j_0$. In the simulation by $\mathcal{S}$, this only occurs when $\mathcal{S}$ receives back $\mathsf{undetected}$, in which case $\mathcal{S}$ perfectly emulates a real execution because it is given the honest party's real input x_2. Thus P_2's output is distributed identically in both the real and ideal executions when $\gamma = j_0$. (Note that P_2 may output $\mathsf{corrupted}_1$ in this case as well. However, what is important is that this will happen with exactly the same probability in the real and ideal executions.) Finally recall from above that γ as chosen by $\mathcal{S}$ is uniformly distributed, and thus the two cases (of $\gamma \neq j_0$ and $\gamma = j_0$) occur with the same probability in the real and ideal executions. We therefore conclude that the overall distributions are identical. This completes this case.

2. *Case 2 – All circuits are legitimate but there is a totally inconsistent input:* We analyze this case in an analogous way to above. Let '$\mathsf{all{=}1}$' denote the case that in a *real* execution all of the $m - 1$ first wires associated with the totally inconsistent input are given value 1 (and so the inconsistent keys determined for those wires are not revealed). Since the values on these wires are chosen by P_2 uniformly, we have that $\Pr[\text{'}\mathsf{all{=}1}\text{'}] = 2^{-m+1}$. Noting also that γ is chosen by P_2 independently of the values on the wires, we have that in a real execution:

$$\Pr[\gamma \neq j_0 \ \& \ \neg\text{'}\mathsf{all{=}1}\text{'}] = \left(1 - \frac{1}{\ell}\right)\left(1 - \frac{1}{2^{m-1}}\right) = \epsilon$$

where the second equality is by the definition of ϵ (recall that j_0 is the index of the first circuit for which an inconsistent key is chosen by $\mathcal{S}$). Now, the trusted party sends $\mathsf{corrupted}_1$ with probability exactly ϵ. Furthermore, in this case, $\mathcal{S}$ generates a transcript for which the event $\gamma \neq j_0$ & $\neg$'all=1' holds (see item (i) of case (2) of the simulator), and such an event in a real execution results in P_2 certainly outputting $\mathsf{corrupted}_1$. We thus have that the $\mathsf{corrupted}_1$ event in the ideal model is mapped with probability exactly ϵ to a sub-distribution over the real transcripts in which P_2 outputs $\mathsf{corrupted}_1$.

Next we analyze the case that not all values on the wires are 1, but $\gamma = j_0$. In a real execution, we have that this event occurs with the following probability:

$$\Pr[\gamma = j_0 \ \& \ \neg\text{`}\mathsf{all}\text{=1'}] = \frac{1}{\ell} \cdot \left(1 - 2^{-m+1}\right)$$

By the description of $\mathcal{S}$, this occurs in the simulation with probability $(1-\epsilon)(1-p)$ where $p = 2^{-m+1}/(1-\epsilon)$; see the second bullet of Case (2) subitem (ii), and observe that γ is always set to j_0 in this case. Now,

$$\begin{aligned}
(1-\epsilon)(1-p) &= (1-\epsilon) \cdot \left(1 - \frac{2^{-m+1}}{1-\epsilon}\right) \\
&= 1 - \epsilon - 2^{-m+1} \\
&= 1 - \left(1 - 2^{-m+1}\right)\left(1 - \ell^{-1}\right) - 2^{-m+1} \\
&= 1 - \left(1 - \frac{1}{\ell} - 2^{-m+1} + \frac{2^{-m+1}}{\ell}\right) - 2^{-m+1} \\
&= \frac{1}{\ell} - \frac{2^{-m+1}}{\ell} \\
&= \frac{1}{\ell} \cdot \left(1 - 2^{-m+1}\right).
\end{aligned}$$

Thus, the probability of this event in the simulation by $\mathcal{S}$ is exactly the same as in a real execution. Furthermore, the transcript generated by $\mathcal{S}$ in this case (and the output of P_2) is identical to in a real execution, because $\mathcal{S}$ runs an emulation using P_2's real input.

Thus far, we have analyzed the output distributions in the events ($\gamma \neq j_0$ & $\neg$'all=1') and ($\gamma = j_0$ & $\neg$'all=1'), and so have covered the case $\neg$'all=1'. It remains for us to analyze the event 'all=1'. That is, it remains to consider the case that all $m-1$ wires do equal 1; this case is covered by the simulation in the first bullet of Case (2), subitem (ii). In a real execution, this case occurs with probability 2^{-m+1}. Likewise, in the simulation, $\mathcal{S}$ reaches subitem (ii) with probability $1-\epsilon$ and then proceeds to the first bullet with probability $p = 2^{-m+1}/(1-\epsilon)$. Therefore, this case appears with overall probability 2^{-m+1} exactly as in a real execution. Furthermore, as above, the simulation by $\mathcal{S}$ is perfect because it emulates using P_2's real input.

We have shown that for the events ($\gamma \neq j_0$ & $\neg$'all=1'), ($\gamma = j_0$ & $\neg$'all=1'), and 'all=1', the joint output distribution generated by $\mathcal{S}$ is identical to that in a real execution. Furthermore, we have shown that these events occur with the same probability in the real and ideal executions. Since these events cover all possibilities, we conclude that the simulation by $\mathcal{S}$ in this case is perfect. (By perfect, we mean that when all circuits are legitimate but there is a totally inconsistent input, the joint output distribution of $\mathcal{S}$ and P_2 in an ideal execution is identical to the joint output distribution of $\mathcal{A}$ and P_2 in a hybrid execution of the protocol where a trusted party is used for the oblivious transfers.)

3. *Case 3 – all circuits are legitimate and there is no totally inconsistent input:* We have the following subcases:

   (a) *Case 3a – $\mathcal{S}$ chose values with inconsistent keys:* First observe that $\mathcal{S}$ chooses values with inconsistent keys with exactly the same probability as P_2 in a real execution. This holds because there are no totally inconsistent values and thus the choice of values on the wires with inconsistent keys is uniform. (Note that P_2's strategy for choosing values is equivalent to choosing any subset of $m-1$ values uniformly and then choosing the last value so that the XOR equals the associated input bit. Since there is at least one wire where both keys are consistent, we can look at this wire as being the one that determines the actual unknown input bit of P_2 and all others are chosen uniformly by $\mathcal{S}$ and P_2. Thus, the probability that $\mathcal{S}$ chooses an inconsistent key is the same as P_2.) We therefore fix the choice of values for the wires and proceed to analyze the transcripts generated by the simulator, *conditioned on this choice of keys.*

   In a real execution in which P_2 chose inconsistent keys, it outputs $\mathsf{corrupted}_1$ if the circuit in which the inconsistent keys were chosen is opened (it may also output $\mathsf{corrupted}_1$ if the circuit is opened but this is not relevant here). Now, if the trusted party sends $\mathsf{corrupted}_1$, then the simulator ensures that the circuit in which the inconsistent keys were chosen is opened (it does this by choosing γ uniformly under the constraint that $\gamma \neq j_0$; see subitem (A) of subitem (i) in Case 3a). In contrast, if the trusted party sends $\mathsf{undetected}$, then $\mathcal{S}$ runs a perfect emulation using P_2's real input; the two subcases (with probability p and $1-p$) are to ensure that γ is chosen uniformly. Thus, it remains to show that in this case, for every $j = 1, \ldots, \ell$ we have $\Pr[\gamma = j] = 1/\ell$. As above, we separately analyze the probability for $j = j_0$ and $j \neq j_0$. The computation is almost the same as in Case 1 above and we are therefore brief:

$$\begin{aligned}
\Pr[\gamma = j_0] &= \Pr[\gamma = j_0 \mid \mathsf{corrupted}_1] \cdot \epsilon + \Pr[\gamma = j_0 \mid \mathsf{undetected}] \cdot (1-\epsilon) \\
&= 0 \cdot \epsilon + \frac{\ell^{-1}}{1-\epsilon} \cdot (1-\epsilon) = \frac{1}{\ell} \, .
\end{aligned}$$

   In addition, for all $j \neq j_0$:

$$\begin{aligned}
\Pr[\gamma = j] &= \Pr[\gamma = j \mid \mathsf{corrupted}_1] \cdot \epsilon + \Pr[\gamma = j \mid \mathsf{undetected}] \cdot (1-\epsilon) \\
&= \left(\frac{1}{\ell-1}\right) \cdot \epsilon + \left(\left(1 - \frac{1}{\ell(1-\epsilon)}\right) \cdot \frac{1}{\ell-1}\right) \cdot (1-\epsilon) = \frac{1}{\ell}
\end{aligned}$$

   Thus, in this case, $\mathcal{S}$ chooses γ uniformly in $\{1, \ldots, \ell\}$. Furthermore, the transcript in each subcase is exactly as in a real execution, as required.

   (b) *Case 3b – $\mathcal{S}$ chose only values with consistent keys:* As above, the probability that $\mathcal{S}$ chose only values with consistent keys is identical to the probability that a real P_2 chooses only values with consistent keys. Now, in such a case, all circuits are legitimate, and in addition, all keys that are retrieved by P_2 are consistent (this includes the keys for the opened circuits and for the circuit that is computed). This means that the computation of the circuit using the keys retrieved by P_2 is identical to the computation of an honestly generated circuit. (Note that P_2 may abort or output $\mathsf{corrupted}_1$ in this case. However, here we are interested in the result of the computation of the circuit G_γ, *if* it is computed by P_2.) We also note that the keys provided by P_1 that are associated with its own input are provided via decommitments. Thus, P_1 can either not provide valid decommitments, or must provide decommitments that yield keys that result in the circuit being decrypted correctly. This also means that the associations made by $\mathcal{S}$ between the input keys of P_1 and the string x_1' that it sends to the trusted party are correct. We conclude that in this case, the joint output of $\mathcal{A}$ and the real P_2 in a real execution is identical to the joint output of $\mathcal{S}$ and P_2 in an ideal execution, as required.

This completes the proof of security in (OT, ϵ)-hybrid model. Applying Theorem 4.2 (sequential composition), we have that Protocol 6.1 is secure in the real model, when using a real oblivious transfer protocol that is secure in the presence of covert adversaries with ϵ-deterrent. ■

6.3 Non-Halting Detection Accuracy

It is possible to modify Protocol 6.1 so that it achieves *non-halting detection accuracy*; see Definition 3.5. Before describing how we do this, notice that the reason that we need to recognize a halting-abort as cheating in Protocol 6.1 is that if P_1 generates one faulty circuit, then it can always just refuse to continue (i.e., abort) in the case that P_2 asks it to open the faulty circuit. This means that if aborting is not considered cheating, then a corrupted P_1 can form a strategy whereby it is never detected cheating, but succeeds in actually cheating with probability $1/\ell$. In order to solve this problem, we construct a method whereby P_1 does not know if it will be caught or not. We do so by having P_2 receive the circuit openings via a fully secure 1-out-of-ℓ oblivious transfer protocol, rather than having P_1 send it explicitly. This forces P_1 to either abort before learning anything, or to risk being caught with probability $1 - 1/\ell$. In order to describe this in more detail, we restate the circuit opening stage of Protocol 6.1 as follows:

1. Party P_1 sends ℓ garbled circuits $GC_1, \ldots, GC_\ell$ to party P_2.
2. P_2 sends a random challenge $\gamma \in_R \{1, \ldots, \ell\}$.
3. P_1 opens GC_j for all $j \neq \gamma$ by sending decommitments, keys and so on. In addition, it sends the keys associated with its own input in GC_γ.
4. P_2 checks the circuits GC_j for $j \neq \gamma$ and computes GC_γ (using the keys from P_1 in the previous step and the keys it obtained earlier in the oblivious transfers). P_2's output is defined to be the output of GC_γ.

Notice that P_2 only outputs $\mathsf{corrupted}_1$ if the checks from the circuit that is opened do not pass. As we have mentioned, there is no logical reason why an adversarial P_1 would ever actually reply with an invalid opening; rather it would just abort. Consider now the following modification:

1. Party P_1 sends ℓ garbled circuits $GC_1, \ldots, GC_\ell$ to party P_2.
2. P_1 and P_2 participate in a (fully secure) 1-out-of-ℓ oblivious transfer with the following inputs:
   (a) P_1 defines its inputs $(x_1, \ldots, x_\ell)$ as follows. Input x_i consists of the opening of circuits GC_j for $j \neq i$ together with the keys associated with its own input in GC_i.
   (b) P_2's input is a random value $\gamma \in_R \{1, \ldots, \ell\}$.
3. P_2 receives an opening of $\ell - 1$ circuits together with the keys needed to compute the other and proceeds as above.

Notice that this modified protocol is essentially equivalent to Protocol 6.1 and thus its proof of security is very similar. However, in this case, an adversarial P_1 who constructs one faulty circuit must decide *before* the oblivious transfer if it wishes to abort (in which case there is no successful cheating) or if it wishes to proceed (in which case P_2 will receive an explicitly invalid opening). Note that due to the security of the oblivious transfer, P_1 cannot know what value γ party P_2 inputs, and so cannot avoid being detected.

The price of this modification is that of one fully secure 1-out-of-ℓ oblivious transfer and the replacement of all of the original oblivious transfer protocols with fully secure ones. (Of course, we could use oblivious transfer protocols that are secure in the presence of covert adversaries with non-halting detection accuracy, but we do not know how to construct such a protocol more efficiently than a fully secure one.) A highly efficient oblivious transfer protocol with a constant number of exponentiations per execution was recently shown in [29] (we remark that the protocol of [29] is designed in the common reference string model, however, coin-tossing can be used to generate the reference string). Using this protocol, we achieve non-halting detection accuracy at a similar cost. As we have mentioned, this is a significant advantage. (We remark that one should not be concerned with the lengths of $x_1, \ldots, x_\ell$ in P_1's input to the oblivious transfer. This is because P_1 can send them encrypted ahead of time with independent symmetric keys $k_1, \ldots, k_\ell$. Then the oblivious transfer takes place only on the keys.)

Acknowledgements

We would like to thank Oded Goldreich and Carmit Hazay for some helpful comments on the write-up, and Tal Zarsky for discussions on the social and legal implications of this adversary model. Finally, we thank the anonymous referees for their many helpful comments.

References

[1] L. von Ahn, N. Hopper and J. Langford. Covert Two-Party Computation. In 37*th STOC*, pages 513–522, 2005.

[2] W. Aiello, Y. Ishai and O. Reingold. Priced Oblivious Transfer: How to Sell Digital Goods. In *EUROCRYPT 2001*, Springer-Verlag (LNCS 2045), pages 119–135, 2001.

[3] D. Beaver. Foundations of Secure Interactive Computing. In *CRYPTO'91*, Springer-Verlag (LNCS 576), pages 377–391, 1991.

[4] M. Ben-Or, S. Goldwasser and A. Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. In 20*th STOC,* pages 1–10, 1988.

[5] R. Canetti. Security and Composition of Multiparty Cryptographic Protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

[6] R. Canetti and R. Ostrovsky. Secure Computation with Honest-Looking Parties: What If Nobody Is Truly Honest? In 31*st STOC*, pages 255–264, 1999.

[7] D. Chaum, C. Crépeau and I. Damgard. Multi-party Unconditionally Secure Protocols. In 20*th STOC*, pages 11–19, 1988.

[8] N. Chandran, V. Goyal, R. Ostrovsky and A. Sahai. Covert Multiparty Computation. In 48*th FOCS*, 2007.

[9] D. Dolev and H.R. Strong. Authenticated Algorithms for Byzantine Agreement. *SIAM Journal on Computing*, 12(4):656–665, 1983.

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

[11] M.K. Franklin and M. Yung. Communication Complexity of Secure Computation. In 24*th STOC*, 699–710, 1992.

[12] O. Goldreich. *Foundations of Cryptography: Volume 2 – Basic Applications*. Cambridge University Press, 2004.

[13] O. Goldreich and Y. Lindell. Session-Key Generation using Human Passwords Only. *Journal of Cryptology,* 19(3):241–340, 2006.

[14] O. Goldreich, S. Micali and A. Wigderson. How to Play any Mental Game – A Completeness Theorem for Protocols with Honest Majority. In 19*th STOC,* pages 218–229, 1987.

[15] O. Goldreich and E. Petrank. Quantifying Knowledge Complexity. In *Computational Complexity*, 8(1):50–98, 1999.

[16] S. Goldwasser and L. Levin. Fair Computation of General Functions in Presence of Immoral Majority. In *CRYPTO'90,* Springer-Verlag (LNCS 537), pages 77–93, 1990.

[17] S. Goldwasser and Y. Lindell. Secure Computation Without Agreement. *Journal of Cryptology,* 18(3):247–287, 2005.

[18] S. Halevi and Y.T. Kalai. Smooth Projective Hashing and Two-Message Oblivious Transfer. *Cryptology ePrint Archive*, report 2007/118, 2007.

[19] Y. Ishai. Personal Communication, 2007.

[20] Y. Ishai, J. Kilian, K. Nissim and E. Petrank. Extending Oblivious Transfers Efficiently. In *CRYPTO 2003,* Springer-Verlag (LNCS 2729), pages 145–161, 2003

[21] Y. Ishai, E. Kushilevitz, Y. Lindell and E. Petrank. Black-Box Constructions for Secure Computation. In 38*th STOC,* pages 99–108, 2006.

[22] S. Jarecki and V. Shmatikov. Efficient Two-Party Secure Computation on Committed Inputs. In *Eurocrypt '07*, Springer-Verlag (LNCS 4515), pages 97–114, 2007.

[23] Y.T. Kalai. Smooth Projective Hashing and Two-Message Oblivious Transfer. In *EUROCRYPT 2005*, Springer-Verlag (LNCS 3494) pages 78–95, 2005.

[24] Y. Lindell and B. Pinkas. A Proof of Yao's Protocol for Secure Two-Party Computation. *Cryptology ePrint Archive,* Report 2004/175, 2004. To appear in the *Journal of Cryptology.*

[25] Y. Lindell and B. Pinkas. An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries. In *EUROCRYPT 2007*, Springer-Verlag (LNCS 4515), pages 52-78, 2007.

[26] D. Malkhi, N. Nisan, B. Pinkas and Y. Sella. Fairplay – A Secure Two-Party Computation System. In the 13*th USENIX Security Symposium*, pages 287–302, 2004.

[27] S. Micali and P. Rogaway. Secure Computation. Unpublished manuscript, 1992. Preliminary version in *CRYPTO'91*, Springer-Verlag (LNCS 576), pages 392–404, 1991.

[28] P. Paillier Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT '99*, Springer-Verlag (LNCS 1592), pages 223–238, 1999.

[29] C. Peikert, V. Vaikuntanathan and B. Waters. A Framework for Efficient and Composable Oblivious Transfer. In *CRYPTO 2008*, Springer-Verlag (LNCS 5157), pages 554–571, 2008.

[30] M. Rabin. How to Exchange Secrets by Oblivious Transfer. Tech. Memo TR-81, Aiken Computation Laboratory, Harvard U., 1981.

[31] A. Yao. How to Generate and Exchange Secrets. In 27*th FOCS*, pages 162–167, 1986.

A Yao's Protocol for Semi-Honest Adversaries

We now describe Yao's protocol for secure two-party computation (in the presence of semi-honest adversaries) which is proven secure in [24]. Yao's protocol is based on the following "garbled-circuit" construction.

The garbled circuit construction. Let C be a Boolean circuit that receives two inputs $x_1, x_2 \in \{0,1\}^n$ and outputs $C(x_1, x_2) \in \{0,1\}^n$ (for simplicity in this description, we assume that the input length, output length and the security parameter are all of the same length n). We also assume that C has the property that if a circuit-output wire comes from a gate g, then gate g has no wires that are input to other gates.[13] (Likewise, if a circuit-input wire is itself also a circuit-output, then it is not input into any gate.) The reduction uses a private key encryption scheme (G, E, D) that has indistinguishable encryptions for multiple messages, and also a special property called an elusive efficiently verifiable range; see [24].[14]

We begin by describing the construction of a single garbled gate g in C. The circuit C is Boolean, and therefore any gate is represented by a function $g : \{0,1\} \times \{0,1\} \to \{0,1\}$. Now, let the two input wires to g be labelled w_1 and w_2, and let the output wire from g be labelled w_3. Furthermore, let $k_1^0, k_1^1, k_2^0, k_2^1, k_3^0, k_3^1$ be six keys obtained by independently invoking the key-generation algorithm $G(1^n)$; for simplicity, assume that these keys are also of length n. Intuitively, we wish to be able to compute $k_3^{g(\alpha,\beta)}$ from k_1^α and k_2^β, without revealing any of the other three values $k_3^{g(1-\alpha,\beta)}, k_3^{g(\alpha,1-\beta)}, k_3^{g(1-\alpha,1-\beta)}$. The gate g is defined by the following four values

$$c_{0,0} = E_{k_1^0}(E_{k_2^0}(k_3^{g(0,0)})) \qquad c_{0,1} = E_{k_1^0}(E_{k_2^1}(k_3^{g(0,1)}))$$
$$c_{1,0} = E_{k_1^1}(E_{k_2^0}(k_3^{g(1,0)})) \qquad c_{1,1} = E_{k_1^1}(E_{k_2^1}(k_3^{g(1,1)}))$$

The actual gate is defined by a *random permutation* of the above values, denoted as c_0, c_1, c_2, c_3; from here on we call them the garbled table of gate g. Notice that given k_1^α and k_2^β, and the values

[13] This requirement is due to our labelling of gates described below, that does not provide a unique label to each wire (see [24] for more discussion). We note that this assumption on C increases the number of gates by at most n.

[14] Loosely speaking, an encryption scheme has an elusive range if without knowing the key, it is hard to generate a ciphertext that falls in the range. An encryption scheme has a verifiable range if given the key and a ciphertext, it is easy to verify that the ciphertext is in the range. Such encryption schemes can be constructed using pseudorandom functions by encrypting the message together with n zeroes. It is easy to see that this provides both an elusive range and an efficiently verifiable one. We denote by $\perp$ the result of decrypting a value not in the range.

c_0, c_1, c_2, c_3, it is possible to compute the output of the gate $k_3^{g(\alpha,\beta)}$ as follows. For every i, compute $D_{k_2^\beta}(D_{k_1^\alpha}(c_i))$. If more than one decryption returns a non-$\perp$ value, then output abort. Otherwise, define k_3^γ to be the only non-$\perp$ value that is obtained. (Notice that if only a single non-$\perp$ value is obtained, then this will be $k_3^{g(\alpha,\beta)}$ because it is encrypted under the given keys k_1^α and k_2^β. By the properties of the encryption scheme, it can be shown that except with negligible probability, only one non-$\perp$ value is indeed obtained.)

We are now ready to show how to construct the entire garbled circuit. Let m be the number of *wires* in the circuit C, and let $w_1, \ldots, w_m$ be labels of these wires. These labels are all chosen uniquely with the following exception: if w_i and w_j are both output wires from the same gate g, then $w_i = w_j$ (this occurs if the fan-out of g is greater than one). Likewise, if an input bit enters more than one gate, then all circuit-input wires associated with this bit will have the same label. Next, for every label w_i, choose two independent keys $k_i^0, k_i^1 \leftarrow G(1^n)$; we stress that all of these keys are chosen independently of the others. Now, given these keys, the four garbled values of each gate are computed as described above and the results are permuted randomly. Finally, the output or decryption tables of the garbled circuit are computed. These tables simply consist of the values $(0, k_i^0)$ and $(1, k_i^1)$ where w_i is a *circuit-output wire*. (Alternatively, output gates can just compute 0 or 1 directly. That is, in an output gate, one can define $c_{\alpha,\beta} = E_{k_1^\alpha}(E_{k_2^\beta}(g(\alpha,\beta)))$ for every $\alpha, \beta \in \{0,1\}$.) The entire garbled circuit of C, denoted $G(C)$, consists of the garbled table for each gate and the output tables. We note that the structure of C is given, and the garbled version of C is simply defined by specifying the output tables and the garbled table that belongs to each gate. This completes the description of the garbled circuit.

Let $x_1 = x_1^1 \cdots x_1^n$ and $x_2 = x_2^1 \cdots x_2^n$ be two n-bit inputs for C. Furthermore, let $w_1, \ldots, w_n$ be the input labels corresponding to x_1, and let $w_{n+1}, \ldots, w_{2n}$ be the input labels corresponding to x_2. It is shown in [24] that given the garbled circuit $G(C)$ and the strings $k_1^{x_1^1}, \ldots, k_n^{x_1^n}, k_{n+1}^{x_2^1}, \ldots, k_{2n}^{x_2^n}$, it is possible to compute $C(x_1, x_2)$, except with negligible probability.

Yao's protocol. Yao's protocol works by designating one party, say P_1, to be the circuit constructor. P_1 builds a garbled circuit to compute f and hands it to P_2. In addition, P_1 sends P_2 the keys $k_1^{x_1^1}, \ldots, k_n^{x_1^n}$ that are associated with its input x_1. Finally, P_2 obtains the keys $k_{n+1}^{x_2^1}, \ldots, k_{2n}^{x_2^n}$ associated with its input via (semi-honest) oblivious transfer. That is, for every $i = 1, \ldots, n$, parties P_1 and P_2 run an oblivious transfer protocol. In the i^{th} execution, P_1 plays the sender with inputs (k_{n+i}^0, k_{n+i}^1) and P_2 plays the receiver with input x_2^i. Following this, P_2 has the keys $k_1^{x_1^1}, \ldots, k_n^{x_1^n}, k_{n+1}^{x_2^1}, \ldots, k_{2n}^{x_2^n}$ and so, as stated above, it can compute the circuit to obtain $C(x_1, x_2)$. Furthermore, since it has only these keys, it cannot compute the circuit for any other input.

A Lemma. In our proof of security, we will use the following lemma:

Lemma A.1 *Given a circuit C with inputs wires $w_1, \ldots, w_{2n}$ and an output value y (of the same length as the output of C) it is possible to efficiently construct a garbled circuit $\widetilde{GC}$ such that:*

1. *The output of $\widetilde{GC}$ is always y, regardless of the garbled values that are provided for P_1 and P_2's input wires, and*

2. *If $y = f(x_1, x_2)$, then no non-uniform probabilistic polynomial-time adversary $\mathcal{A}$ can distinguish between the distribution ensemble consisting of $\widetilde{GC}$ and a single arbitrary key for every input wire, and the distribution ensemble consisting of a real garbled version of C, together with the keys $k_1^{x_1^1}, \ldots, k_n^{x_1^n}, k_{n+1}^{x_2^1}, \ldots, k_{2n}^{x_2^n}$.*

Proof Sketch: The proof of this lemma is taken from [24] (it is not stated in this way there, but is proven). We sketch the construction of $\widetilde{GC}$ here for the sake of completeness, and refer the reader to [24] for a full description and proof. The first step in the construction of the fake circuit $\widetilde{GC}$ is to choose two random keys k_i and k'_i for every wire w_i in the circuit C. Next, the gate tables of C are computed: let g be a gate with input wires w_i, w_j and output wire w_ℓ. The table of gate g contains encryptions of the single key k_ℓ that is associated with wire w_ℓ, under *all four combinations* of the keys k_i, k'_i, k_j, k'_j that are associated with the input wires w_i and w_j to g. (This is in contrast to a real construction of the garbled circuit that involves encrypting both k_ℓ and k'_ℓ, depending on the function that the gate in question computes.) That is, the following values are computed:

$$c_{0,0} = E_{k_i}(E_{k_j}(k_\ell))$$
$$c_{0,1} = E_{k_i}(E_{k'_j}(k_\ell))$$
$$c_{1,0} = E_{k'_i}(E_{k_j}(k_\ell))$$
$$c_{1,1} = E_{k'_i}(E_{k'_j}(k_\ell))$$

The gate table for g is then just a random ordering of the above four values. This process is carried out for all of the gates of the circuit. It remains to describe how the output decryption tables are constructed. Denote the n-bit output y by $y_1 \cdots y_n$, and denote the circuit-output wires by $w_{m-n+1}, \ldots, w_m$. In addition, for every $i = 1, \ldots, n$, let k_{m-n+i} be the (single) key encrypted in the gate whose output wire is w_{m-n+i}, and let k'_{m-n+i} be the other key (as described above). Then, the output decryption table for wire w_{m-n+i} is given by: $[(0, k_{m-n+i}), (1, k'_{m-n+i})]$ if $y_i = 0$, and $[(0, k'_{m-n+i}), (1, k_{m-n+i})]$ if $y_i = 1$. This completes the description of the construction of the fake garbled circuit $\widetilde{GC}$.

Notice that by the above construction of the circuit, the output keys (or garbled values) obtained by P_2 for *any* set of input keys (or garbled values), equals $k_{m-n+1}, \ldots, k_m$. Furthermore, by the above construction of the output tables, these keys $k_{m-n+1}, \ldots, k_m$ decrypt to $y = y_1 \cdots y_n$ exactly. Thus, property (1) of the lemma trivially holds. The proof of property (2) follows from a hybrid argument in which the gate construction is changed one at a time from the real construction to the above fake one (indistinguishability follows from the indistinguishability of encryptions). The construction and proof of this hybrid are described in full in [24]. ■