

Chapter 2

Definitions

In this chapter we present a number of definitions of security for secure computation. Specifically, in Sections 2.2 to 2.4 we present definitions of security for semi-honest, malicious and covert adversaries; all these definitions are based on the ideal/real-model paradigm for formulating security. We begin with the classic definitions of security in the presence of semi-honest and malicious adversaries, and then proceed to the more recent notion of security in the presence of covert adversaries. In Section 2.5, we show that it often suffices to consider restricted types of functionalities, which enables us to simplify the presentation of the general protocols in Chapters 3 to 5. In Section 2.6 we consider two relaxations of these definitions, for the case of malicious adversaries. Finally, in Section 2.7 we conclude with the issue of sequential composition of secure protocols. We stress that since the focus of this book is secure *two-party* computation, all of the definitions are presented for the case of two parties only.

2.1 Preliminaries

We begin by introducing notation and briefly reviewing some basic notions; see [30] for more details. 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 will represent 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 by $X \stackrel{c}{\equiv} Y$, 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 where its input is longer than its overall running time.) We sometimes use PPT as shorthand for *probabilistic polynomial time*.

For a set X , we denote by $x \leftarrow_R X$ the process of choosing an element x of X under the uniform distribution.

2.2 Security in the Presence of Semi-honest Adversaries

The model that we consider here is that of two-party computation in the presence of *static semi-honest* adversaries. Such an adversary controls one of the parties (statically, and so at the onset of the computation) and follows the protocol specification exactly. However, it may try to learn more information than allowed by looking at the transcript of messages that it received and its internal state. Since we only consider static semi-honest adversaries here, we will sometimes omit the qualification that security is with respect to such adversaries only. The definitions presented here are according to Goldreich in [32].

Two-party computation. A two-party protocol problem is cast by specifying a random process that maps pairs of inputs to pairs of outputs (one for each party). We refer to such a process as a **functionality** and denote it $f : \{0, 1\}^* \times \{0, 1\}^* \rightarrow \{0, 1\}^* \times \{0, 1\}^*$, where $f = (f_1, f_2)$. That is, for every pair of inputs $x, y \in \{0, 1\}^n$, the output-pair is a random variable $(f_1(x, y), f_2(x, y))$ ranging over pairs of strings. The first party (with input x) wishes to obtain $f_1(x, y)$ and the second party (with input y) wishes to obtain $f_2(x, y)$. We often denote such a functionality by $(x, y) \mapsto (f_1(x, y), f_2(x, y))$. Thus, for example, the oblivious transfer functionality [72] is specified by $((z_0, z_1), \sigma) \mapsto (\lambda, z_\sigma)$, where λ denotes the empty string. When the functionality f is probabilistic, we sometimes use the notation $f(x, y, r)$, where r is a uniformly chosen random tape used for computing f .

Privacy by simulation. Intuitively, a protocol is secure if whatever can be computed by a party participating in the protocol can be computed based on its input and output only. This is formalized according to the simulation paradigm. Loosely speaking, we require that a party's *view* in a protocol execution be simulatable given only its input and output. This then implies that the parties learn nothing from the protocol *execution* itself, as desired.

Definition of security. We begin with the following notation:

- Let $f = (f_1, f_2)$ be a probabilistic polynomial-time functionality and let π be a two-party protocol for computing f .

- The view of the i th party ($i \in \{1, 2\}$) during an execution of π on (x, y) and security parameter n is denoted by $\text{view}_i^\pi(x, y, n)$ and equals $(w, r^i, m_1^i, \dots, m_t^i)$, where $w \in \{x, y\}$ (its value depending on the value of i), r^i equals the contents of the i th party's *internal* random tape, and m_j^i represents the j th message that it received.
- The output of the i th party during an execution of π on (x, y) and security parameter n is denoted by $\text{output}_i^\pi(x, y, n)$ and can be computed from its own view of the execution. We denote the joint output of both parties by $\text{output}^\pi(x, y, n) = (\text{output}_1^\pi(x, y, n), \text{output}_2^\pi(x, y, n))$.

Definition 2.2.1 (security w.r.t. semi-honest behavior): *Let $f = (f_1, f_2)$ be a functionality. We say that π securely computes f in the presence of static semi-honest adversaries if there exist probabilistic polynomial-time algorithms S_1 and S_2 such that*

$$\begin{aligned} \{(S_1(1^n, x, f_1(x, y)), f(x, y))\}_{x, y, n} &\stackrel{c}{=} \{(\text{view}_1^\pi(x, y, n), \text{output}^\pi(x, y, n))\}_{x, y, n}, \\ \{(S_2(1^n, y, f_2(x, y)), f(x, y))\}_{x, y, n} &\stackrel{c}{=} \{(\text{view}_2^\pi(x, y, n), \text{output}^\pi(x, y, n))\}_{x, y, n}, \\ x, y &\in \{0, 1\}^* \text{ such that } |x| = |y|, \text{ and } n \in \mathbb{N}. \end{aligned}$$

The above states that the view of a party can be simulated by a probabilistic polynomial-time algorithm given access to the party's *input and output only*. We emphasize that the adversary here is semi-honest and therefore its view in the execution of π is exactly as in the case where both parties follow the protocol specification. We note that it is not enough for the simulator S_i to generate a string indistinguishable from $\text{view}_i^\pi(x, y)$. Rather, the *joint distribution* of the simulator's output and the functionality output $f(x, y)$ must be indistinguishable from $(\text{view}_i^\pi(x, y), \text{output}^\pi(x, y))$. This is necessary for probabilistic functionalities; see [11, 32] for a full discussion.

A simpler formulation for deterministic functionalities. In the case where the functionality f is deterministic, a simpler definition can be used. Specifically, we do not need to consider the joint distribution of the simulator's output with the protocol output. Rather we separately require *correctness*, meaning that

$$\{\text{output}^\pi(x, y, n)\}_{x, y \in \{0, 1\}^*; n \in \mathbb{N}} \stackrel{c}{=} \{f(x, y)\}_{x, y \in \{0, 1\}^*}$$

and, in addition, that there exist PPT S_1 and S_2 such that

$$\{S_1(1^n, x, f_1(x, y))\}_{x, y \in \{0, 1\}^*; n \in \mathbb{N}} \stackrel{c}{=} \{\text{view}_1^\pi(x, y, n)\}_{x, y \in \{0, 1\}^*; n \in \mathbb{N}}, \quad (2.1)$$

$$\{S_2(1^n, y, f_2(x, y))\}_{x, y \in \{0, 1\}^*; n \in \mathbb{N}} \stackrel{c}{=} \{\text{view}_2^\pi(x, y, n)\}_{x, y \in \{0, 1\}^*; n \in \mathbb{N}} \quad (2.2)$$

The reason this suffices is that when f is deterministic, $\text{output}^\pi(x, y, n)$ must equal $f(x, y)$. Furthermore, the distinguisher for the ensembles can compute

$f(x, y)$ by itself (because it is given x and y , the indices of the ensemble). See [32, Section 7.2.2] for more discussion.

For simplicity of notation, we will often let n be the length of x and y . In this case, the simulators S_1 and S_2 do not need to receive 1^n for input, and we omit n from the VIEW and OUTPUT notations.

An equivalent definition. A different definition of security for two-party computation in the presence of semi-honest adversaries compares the output of a real protocol execution to the output of an *ideal* computation involving an incorruptible trusted third party (as described in the Introduction). The trusted party receives the parties' inputs, computes the functionality on these inputs and returns to each its respective output. Loosely speaking, a protocol is secure if any real-model adversary can be converted into an ideal-model adversary such that the output distributions are computationally indistinguishable. We remark that in the case of semi-honest adversaries, this definition is equivalent to the (simpler) simulation-based definition presented here; see [32]. This formulation of security will be used for defining security in the presence of malicious adversaries below.

Augmented semi-honest adversaries. Observe that by the definition above, a semi-honest party *always* inputs its prescribed input value, even if it is corrupted. We argue that it often makes sense to allow a corrupted semi-honest party to modify its input, as long as it does so before the execution begins. This is due to the following reasons. First, on a subjective intuitive level it seems to us that this is in the spirit of semi-honest behavior because choosing a different input is not “improper behavior”. Second, when protocols achieving security in the presence of semi-honest adversaries are used as a stepping stone for obtaining security in the presence of malicious adversaries, it is necessary to allow the semi-honest adversary to modify its input. Indeed, Goldreich introduces the notion of an *augmented semi-honest adversary* that may modify its input before the execution begins when showing how to obtain security against malicious adversaries from protocols that are secure only in the presence of semi-honest adversaries [32, Sec. 7.4.4.1]. Finally, as we discuss in Section 2.3.3, it is natural that any protocol that is secure in the presence of malicious adversaries also be secure in the presence of semi-honest adversaries. Although very counterintuitive, it turns out that this only holds when the semi-honest adversary is allowed to change its input; see Section 2.3.3 for a full discussion. We present the definition of semi-honest adversaries above, where a corrupted party cannot change its input, for historical reasons only. However, we strongly prefer the notion of augmented semi-honest adversaries. We remark that a formal definition of this notion is easily obtained via the ideal/real-model paradigm; see Section 2.3 below.

2.3 Security in the Presence of Malicious Adversaries

In this section, we present the definition of security for the case of malicious adversaries who may use any efficient attack strategy and thus may arbitrarily deviate from the protocol specification. In this case, it does not suffice to construct simulators that can generate the view of the corrupted party. First and foremost, the generation of such a view depends on the actual input used by the adversary; indeed this input affects the actual output received. However, in contrast to the case of semi-honest adversaries, the adversary may not use the input that it is provided. Thus, a simulator for the case where P_1 is corrupted cannot just take x and $f(x, y)$ and generate a view (in order to prove that nothing more than the output is learned), because the adversary may not use x at all. Furthermore, beyond the possibility that a corrupted party may learn more than it should, we require correctness (meaning that a corrupted party cannot cause the output to be incorrectly distributed) and independence of inputs (meaning that a corrupted party cannot make its input depend on the other party's input). As discussed in the overview in Section 1.1, in order to capture these threats, and others, the security of a protocol is analyzed by comparing what an adversary can do in the protocol 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 were involved in the above-described ideal computation. See [11, 32] for more discussion on the advantages of this specific formulation.

We remark that since we consider the two-party case, there is no honest majority. It is therefore impossible to achieve fairness in general. Therefore, in the ideal setting we allow the adversary to obtain the corrupted party's output, without the honest party necessarily obtaining its output. We also remark that in defining security for two parties it is possible to consider only the setting where one of the parties is corrupted, or to also consider the setting where none of the parties are corrupted, in which case the adversary seeing the transcript between the parties should learn nothing. Since this latter case can easily be achieved by using encryption between the parties we present the simpler formulation security that assumes that exactly one party is always corrupted.

2.3.1 The Definition

Execution in the ideal model. As we have mentioned, in the case of no honest majority, it is in general impossible to achieve guaranteed output delivery and fairness. This “weakness” is therefore incorporated into the ideal model by allowing the adversary in an ideal execution to abort the execution or obtain output without the honest party obtaining its output. Denote the participating parties by P_1 and P_2 and let $i \in \{1, 2\}$ denote the index of the corrupted party, controlled by an adversary \mathcal{A} . An ideal execution for a function $f : \{0, 1\}^* \times \{0, 1\}^* \rightarrow \{0, 1\}^* \times \{0, 1\}^*$ proceeds as follows:

Inputs: Let x denote the input of party P_1 , and let y denote the input of party P_2 . The adversary \mathcal{A} also has an auxiliary input denoted by z .

Send inputs to trusted party: The honest party P_j sends its received input to the trusted party. The corrupted party P_i controlled by \mathcal{A} may either abort (by replacing the input with a special abort_i message), send its 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 input value of P_i and the auxiliary input z . Denote the pair of inputs sent to the trusted party by (x', y') (note that if $i = 2$ then $x' = x$ but y' does not necessarily equal y , and vice versa if $i = 1$).

Early abort option: If the trusted party receives an input of the form abort_i for some $i \in \{1, 2\}$, it sends abort_i to all parties and the ideal execution terminates. Otherwise, the execution proceeds to the next step.

Trusted party sends output to adversary: At this point the trusted party computes $f_1(x', y')$ and $f_2(x', y')$ and sends $f_i(x', y')$ to party P_i (i.e., it sends the corrupted party its output).

Adversary instructs trusted party to continue or halt: \mathcal{A} sends either continue or abort_i to the trusted party. If it sends continue , the trusted party sends $f_j(x', y')$ to party P_j (where P_j is the honest party). Otherwise, if \mathcal{A} sends abort_i , the trusted party sends abort_i to party P_j .

Outputs: The honest party always outputs the output value it obtained from the trusted party. The corrupted party outputs nothing. The adversary \mathcal{A} outputs any arbitrary (probabilistic polynomial-time computable) function of the initial input of the corrupted party, the auxiliary input z , and the value $f_i(x', y')$ obtained from the trusted party.

Let $f : \{0, 1\}^* \times \{0, 1\}^* \rightarrow \{0, 1\}^* \times \{0, 1\}^*$ be a two-party functionality, where $f = (f_1, f_2)$, let \mathcal{A} be a non-uniform probabilistic polynomial-time machine, and let $i \in \{1, 2\}$ be the index of the corrupted party. Then, the ideal execution of f on inputs (x, y) , auxiliary input z to \mathcal{A} and security parameter n , denoted by $\text{IDEAL}_{f, \mathcal{A}(z), i}(x, y, n)$, is defined as the output pair of the honest party 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 two-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 party, and may follow an arbitrary polynomial-time strategy. In contrast, the honest party follows the instructions of π .

Let f be as above and let π be a two-party protocol for computing f . Furthermore, let \mathcal{A} be a non-uniform probabilistic polynomial-time machine and let $i \in \{1, 2\}$ be the index of the corrupted party. Then, the **real execution** of π on inputs (x, y) , auxiliary input z to \mathcal{A} and security parameter n , denoted by $\text{REAL}_{\pi, \mathcal{A}(z), i}(x, y, n)$, is defined as the output pair of the honest party 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.

Definition 2.3.1 (secure two-party 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 $i \in \{1, 2\}$,*

$$\{\text{IDEAL}_{f, \mathcal{S}(z), i}(x, y, n)\}_{x, y, z, n} \stackrel{c}{=} \{\text{REAL}_{\pi, \mathcal{A}(z), i}(x, y, n)\}_{x, y, z, n}$$

where $x, y \in \{0, 1\}^*$ under the constraint that $|x| = |y|$, $z \in \{0, 1\}^*$ and $n \in \mathbb{N}$.

The above definition assumes that the parties (and adversary) know the input lengths (this can be seen from the requirement that $|x| = |y|$ is balanced and so all the inputs in the vector of inputs are of the same length). We remark that some restriction on the input lengths is unavoidable because, as in the case of encryption, to some extent such information is always leaked.

2.3.2 Extension to Reactive Functionalities

Until now we have considered the secure computation of simple functionalities that compute a single pair of outputs from a single pair of inputs. However, not all computations are of this type. Rather, many computations have multiple rounds of inputs and outputs. Furthermore, the input of a party in a given round may depend on its output from previous rounds, and the outputs of that round may depend on the inputs provided by the parties in some or all of the previous rounds. A classic example of this is electronic poker. In this game, in the first phase cards are dealt to the players. Based on these cards, bets are made and cards possibly thrown and dealt. The important

thing to notice is that in each round human decisions must be made based on the current status. Thus, new *inputs* are provided in each round (e.g., how much to bet and what cards to throw), and these inputs are based on the current *output* (in this case, the output is the player's current hand and the cards previously played). A more cryptographic example of a multi-phase functionality is that of a commitment scheme. Such a scheme has a distinct commitment and decommitment phase. Thus, it cannot be cast as a standard functionality mapping inputs to outputs.

In the context of secure computation, multi-phase computations are typically called **reactive functionalities**. Such functionalities can be modeled as a series of functions (f^1, f^2, \dots) where each function receives some state information and two new inputs. That is, the input to function f^j consists of the inputs (x_j, y_j) of the parties in this phase, along with a state input σ_{j-1} output by f^{j-1} . Then, the output of f^j is defined to be a pair of outputs $f_1^j(x_j, y_j, \sigma_{j-1})$ for P_1 and $f_2^j(x_j, y_j, \sigma_{j-1})$ for P_2 , and a state string σ_j to be input into f^{j+1} . We stress that the parties receive only their private outputs, and in particular do *not* receive any of the state information; in the ideal model this is stored by the trusted party. Although the above definition is intuitively clear, a simpler formulation is to define a reactive functionality via a multi-phase probabilistic polynomial-time Turing machine that receives inputs and generates outputs (this is simpler because it is not necessary to explicitly define the state at every stage). The trusted party then runs this machine upon each new pair of inputs it receives and sends the generated outputs. In this formulation, the state information is kept internally by the Turing machine, and not explicitly by the trusted party. We remark that the formal ideal model remains the same, except that the trusted party runs a reactive functionality (i.e., reactive Turing machine) instead of a single function. In addition, once the corrupted party sends `aborti`, the ideal execution stops, and no additional phases are run.

2.3.3 Malicious Versus Semi-honest Adversaries

At first sight, it seems that any protocol that is secure in the presence of malicious adversaries is also secure in the presence of semi-honest adversaries. This is because a semi-honest adversary is just a “special case” of a malicious adversary who faithfully follows the protocol specification. Although this is what we would expect, it turns out to be *false* [45]. This anomaly is due to the fact that although a real semi-honest adversary is indeed a special case of a real malicious adversary, this is not true of the respective adversaries in the ideal model. Specifically, the adversary in the ideal model for malicious adversaries is allowed to change its input, whereas the adversary in the ideal model for semi-honest adversary is not. Thus, the adversary/simulator for the case of malicious adversaries has more power than the adversary/simulator

for the case of semi-honest adversaries. As such, it may be possible to simulate a protocol in the malicious model, but not in the semi-honest model. We now present two examples of protocols where this occurs.

Example 1 – secure AND. Consider the case of two parties computing the *binary AND* function $f(x, y) = x \wedge y$, where only party P_2 receives output. Note first that if party P_2 uses input 1, then by the output received it can fully determine party P_1 's input (if the output is 0 then P_1 had input 0, and otherwise it had input 1). In contrast, if party P_2 uses input 0 then it learns nothing about P_1 's input, because the output equals 0 irrespective of the value of P_1 's input. The result of this observation is that in the ideal model, an adversary corrupting P_2 can always learn P_1 's exact input by sending the trusted party the input value 1. Thus, P_1 's input is always revealed. In contrast, in the ideal model with a semi-honest adversary, P_1 's input is only revealed if the corrupted party has input 1; otherwise, the adversary learns nothing whatsoever about P_1 's input. We use the above observations to construct a protocol that securely computes the binary AND function in the presence of malicious adversaries, but is not secure in the presence of semi-honest adversaries; see Protocol 2.3.2.

PROTOCOL 2.3.2 (A Protocol for Binary AND)

- **Input:** P_1 has an input bit x and P_2 has an input bit y .
- **Output:** The binary value $x \wedge y$ for P_2 only.
- **The protocol:**
 1. P_1 sends P_2 its input bit x .
 2. P_2 outputs the bit $x \wedge y$.

We have the following claims:

Claim 2.3.3 *Protocol 2.3.2 securely computes the binary AND function in the presence of malicious adversaries.*

Proof. We separately consider the case where P_1 is corrupted and the case where P_2 is corrupted. If P_1 is corrupted, then the simulator \mathcal{S} receives from \mathcal{A} the bit that it sends to P_2 in the protocol. This bit fully determines the input of P_1 to the function and so \mathcal{S} just sends it to the trusted party, thereby completing the simulation. In the case where P_2 is corrupted, \mathcal{S} sends input 1 to the trusted party and receives back an output bit b . By the observation above, b is the input of the honest P_1 in the ideal model. Thus, the simulator \mathcal{S} just hands \mathcal{A} the bit $x = b$ as the value that \mathcal{A} expects to receive from the honest P_1 in a real execution. It is immediate that the simulation here is perfect. ■

We stress that the above works because P_2 is the only party to receive output. If P_1 also were to receive output, then \mathcal{S} 's simulation in the case of a

corrupted P_2 would not work. In order to see this, consider an adversary who corrupts P_2 , uses input $y = 0$ and outputs its view in the protocol, including the bit x that it receives from P_1 . In this case, \mathcal{S} cannot send $y = 1$ to the trusted party because P_1 's output would not be correctly distributed. Thus, it must send $y = 0$, in which case the view that it generates for \mathcal{A} cannot always be correct because it does not know the input bit x of P_1 .

Claim 2.3.4 *Protocol 2.3.2 does not securely compute the binary AND function in the presence of semi-honest adversaries.*

Proof. Consider the simulator S_2 that is guaranteed to exist for the case where P_2 is corrupted; see (2.2) in Section 2.2. Then, S_2 is given y and $x \wedge y$ and must generate the view of P_2 in the computation. However, this view contains the value x that P_1 sends to P_2 in the protocol. Now, if $y = 0$ and x is random, then there is no way that S_2 can guess the value of x with probability greater than $1/2$. We conclude that the protocol is not secure in the presence of semi-honest adversaries. ■

Example 2 – set union. Another example where this arises is the problem of set union over a large domain where only one party receives output. Specifically, consider the function $f(X, Y) = (\lambda, X \cup Y)$ where $X, Y \subseteq \{0, 1\}^n$ are sets of the same size, and λ denotes the “empty” output. We claim that the protocol where P_1 sends its set X to P_2 is secure in the presence of malicious adversaries. This follows for the exact same reasons as above because a corrupted P_2 in the malicious model can replace its input set Y with a set Y' of the same size, but containing random values. Since the sets contain values of length n , it follows that the probability that $X \cap Y \neq \phi$ is negligible. Thus, the output that P_2 receives completely reveals the input of P_1 . In contrast, if a corrupted party cannot change its input, then when $X \cap Y \neq \phi$ the elements that are common to both sets are hidden. Specifically, if five elements are common to both sets, then P_2 knows that there are five common elements, but does not have any idea as to which are common. Thus, for the same reasons as above, the protocol is not secure in the presence of semi-honest adversaries. Once again, we stress that this works when only one party receives output; in the case where both parties receive output, securely computing this functionality is highly non-trivial.

Discussion. It is our opinion that the above phenomenon should not be viewed as an “annoying technicality”. Rather it points to a problem in the definitions that needs to be considered. Our position is that it would be better to define semi-honest adversaries as adversaries that *are allowed to change their input* before the computation starts (e.g., by rewriting the value on their input tape), and once the computation begins must behave in a semi-honest fashion as before. Conceptually, this makes sense because parties are allowed to choose their own input and this is not adversarial behavior. In addition, this model better facilitates the “compilation” of protocols that are secure in the semi-honest model into protocols that are secure in the malicious model.

Indeed, in order to prove the security of the protocol of [35], and specifically the compilation of a protocol for the semi-honest model into one that is secure in the presence of malicious adversaries, Goldreich introduces the notion of **augmented semi-honest behavior**, which is exactly as described above; see Definition 7.4.24 in Section 7.4.4.1 of [30]. We stress that all protocols presented in this book that are secure in the presence of semi-honest adversaries are also secure in the presence of *augmented semi-honest adversaries*. Furthermore, as stated in the following proposition, security in the malicious model implies security in the augmented semi-honest model, as one would expect.

Proposition 2.3.5 *Let π be a protocol that securely computes a functionality f in the presence of malicious adversaries. Then π securely computes f in the presence of augmented semi-honest adversaries.*

Proof. Let π be a protocol that securely computes f in the presence of malicious adversaries. Let \mathcal{A} be an augmented semi-honest real adversary and let \mathcal{S} be the simulator for \mathcal{A} that is guaranteed to exist by the security of π (for every malicious \mathcal{A} there exists such an \mathcal{S} , and in particular for an augmented semi-honest \mathcal{A}). We construct a simulator \mathcal{S}' for the augmented semi-honest setting, by simply having \mathcal{S}' run \mathcal{S} . However, in order for this to work, we have to show that \mathcal{S}' can do everything that \mathcal{S} can do. In the malicious ideal model, \mathcal{S} can choose whatever input it wishes for the corrupted party; since \mathcal{S}' is *augmented* semi-honest, it too can modify the input. In addition, \mathcal{S} can cause the honest party to output **abort**. However, \mathcal{S}' *cannot* do this. Nevertheless, this is not a problem because when \mathcal{S} is the simulator for an augmented semi-honest \mathcal{A} it can cause the honest party to output **abort** with at most negligible probability. In order to see this, note that when two honest parties run the protocol, neither outputs **abort** with non-negligible probability. Thus, when an honest party runs together with an augmented semi-honest adversary, it too outputs **abort** with at most negligible probability. This is due to the fact that the distribution over the messages it receives in both cases is identical (because a semi-honest real adversary follows the protocol instructions just like an honest party). This implies that the simulator for the malicious case, when applied to an augmented semi-honest real adversary, causes an abort with at most negligible probability. Thus, the augmented semi-honest simulator can run the simulator for the malicious case, as required. ■

Given the above, it is our position that the definition of *augmented semi-honest adversaries* is the “right way” of modeling semi-honest behavior. As such, it would have been more appropriate to use this definition from scratch. However, we chose to remain with the standard definition of semi-honest adversaries for historical reasons.

2.4 Security in the Presence of Covert Adversaries

2.4.1 Motivation

In this chapter, we present a relatively 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, the 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. The definition of security here is based on the ideal/real simulation paradigm (as in the definition in Section 2.3), 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 this notion. 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 . 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. This notion is therefore called **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 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 $\epsilon = 1$, the definition can be used to capture a requirement that cheating parties are always caught.

Formalizing the notion. The standard definition of security (see Definition 2.3.1) 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 this one, we begin with a somewhat naive implementation of the notion, and show its shortcomings:

1. *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). 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 manages to double the probability of 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 wrongdoing. 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 adversaries may be willing to risk being caught with more than negligible probability, say 10^{-6} . With such an adversary, the proposed 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.

2. *Second attempt.* To solve the aforementioned problems, we first require that the protocol itself be responsible for detecting cheating. Specifically, in the case where 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 happen 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 adversary that is not “covert” (as defined in the first attempt) 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.

3. *Final attempt.* 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 2.3.1), or that the honest parties 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 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 such, we use the terminology **covert adversaries** to refer to malicious adversaries that do not wish to be caught cheating.

The above intuitive notion can be interpreted in a number of ways. We present the main formulation here. The definition works by modifying the ideal model so that the ideal-model adversary (i.e., 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 tosses coins and with probability ϵ announces to the honest parties that cheating has taken place (by sending the message **corrupted_i**, where party P_i is the corrupted party that sent the **cheat** instruction). In contrast, with probability $1 - \epsilon$, the trusted party sends the honest party’s input to the adversary, and in addition lets

the adversary fix the output of the honest party. We stress that in this case the trusted party does not announce that cheating has taken place, and so the adversary gets off scot-free. Observe that if the trusted party announces that cheating has taken place, then the adversary learns absolutely nothing. This is a strong guarantee because when the adversary attempts to cheat, it must take the risk of being caught and gaining nothing.

2.4.2 The Actual Definition

We begin by presenting the modified ideal model. In this model, we add new instructions that the adversary can send to the trusted party. Recall that in the standard ideal model, the adversary can send a special `aborti` message to the trusted party, in which case the honest party receives `aborti` as output. In the ideal model for covert adversaries, the adversary can send the following additional special instructions:

- *Special input corrupted_i*: If the ideal-model adversary sends `corruptedi` instead of an input, the trusted party sends `corruptedi` to the honest party 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 `cheati` instead of an input, 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 `corruptedi` to the honest party. If it was not detected, the trusted party hands the adversary the honest party’s input and gives the ideal-model adversary the ability to set the output of the honest party to whatever value it wishes. Thus, a `cheati` input is used to model a protocol execution in which the real-model adversary decides to cheat. However, as required, this cheating is guaranteed to be detected with probability at least ϵ . Note that if the cheat attempt is not detected then the adversary is given “full cheat capability”, including the ability to determine the honest party’s output.

The idea behind the 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 party’s input and the output).

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

Inputs: Let x denote the input of party P_1 , and let y denote the input of party P_2 . The adversary \mathcal{A} also has an auxiliary input z .

Send inputs to trusted party: The honest party P_j sends its received input to the trusted party. The corrupted party P_i , controlled by \mathcal{A} , may abort (by replacing the input with a special **abort_i** or **corrupted_i** message), send its 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 input value of P_i and the auxiliary input z . Denote the pair of inputs sent to the trusted party by (x', y') .

Abort options: If a corrupted party sends **abort_i** to the trusted party as its input, then the trusted party sends **abort_i** to the honest party and halts. If a corrupted party sends **corrupted_i** to the trusted party as its input, then the trusted party sends **corrupted_i** to the honest party and halts.

Attempted cheat option: If a corrupted party sends **cheat_i** to the trusted party as its input, then the trusted party works as follows:

1. With probability ϵ , the trusted party sends **corrupted_i** to the adversary and the honest party.
2. With probability $1 - \epsilon$, the trusted party sends **undetected** to the adversary along with the honest party's input. Following this, the adversary sends the trusted party an output value τ of its choice for the honest party. The trusted party then sends τ to P_j as its output (where P_j is the honest party).

If the adversary sent **cheat_i**, then the ideal execution ends at this point. Otherwise, the ideal execution continues below.

Trusted party sends output to adversary: At this point the trusted party computes $f_1(x', y')$ and $f_2(x', y')$ and sends $f_i(x', y')$ to P_i (i.e., it sends the corrupted party its output).

Adversary instructs trusted party to continue or halt: After receiving its output, the adversary sends either **continue** or **abort_i** to the trusted party. If the trusted party receives **continue** then it sends $f_j(x', y')$ to the honest party P_j . Otherwise, if it receives **abort_i**, it sends **abort_i** to the honest party P_j .

Outputs: The honest party always outputs the output value it obtained from the trusted party. The corrupted party outputs nothing. The adversary \mathcal{A} outputs any arbitrary (probabilistic polynomial-time computable) function of the initial inputs of the corrupted party, the auxiliary input z , and the value $f_i(x', y')$ obtained from the trusted party.

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

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. The other type of cheating in this ideal model is more serious for two reasons: first, the ramifications of the cheating are greater (the adversary may learn the honest party’s input and may be able to determine its output), 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 obviously of the honest party’s input 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 2.4.1 (security – strong explicit cheat formulation [3]): *Let f and π be as in Definition 2.2.1, and let $\epsilon : \mathbb{N} \rightarrow [0, 1]$ be a function. 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 $i \in \{1, 2\}$:*

$$\left\{ \text{IDEALSC}_{f, \mathcal{S}(z), i}^{\epsilon}(x, y, n) \right\}_{x, y, z, n} \stackrel{c}{=} \left\{ \text{REAL}_{\pi, \mathcal{A}(z), i}(x, y, n) \right\}_{x, y, z, n}$$

where $x, y, z \in \{0, 1\}^*$ under the constraint that $|x| = |y|$, and $n \in \mathbb{N}$.

2.4.3 Cheating and Aborting

It is important to note that in the above definition, a party that halts midway through the computation may be considered a “cheat” (this is used in an inherent way when constructing 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 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. In order to do this, we introduce the notion of “non-halting detection accuracy” so that if a party halts early, but otherwise does not deviate from the protocol specification, then it is not considered cheating. We formalize this by considering **fail-stop adversaries** who act semi-honestly except that they may halt early. Formally:

Definition 2.4.2 *Let π be a two-party protocol, let \mathcal{A} be an adversary, and let i be the index of the corrupted party. The honest party P_j is said to detect cheating in π if its output in π is corrupted_i ; this event is denoted by $\text{OUTPUT}_j(\text{REAL}_{\pi, \mathcal{A}(z), i}(x, y, n)) = \text{corrupted}_i$. The protocol π is called **non-halting detection accurate** if for every fail-stop adversary \mathcal{A} , the probability that P_j detects cheating in π is negligible.*

Definition 2.4.1 can then be modified by requiring that π be non-halting detection accurate. We remark that although this strengthening is clearly desirable, it may also be prohibitive. Nevertheless, as we will see in Chapter 5, it is possible to efficiently obtain this stronger guarantee for the case of general protocols.

2.4.4 Relations Between Security Models

In order to better understand the definition of security in the presence of covert adversaries, 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 2.4.3 *Let π be a protocol that securely computes some functionality f with abort in the presence of malicious adversaries, as in Definition 2.3.1. Then, π securely computes f in the presence of covert adversaries with ϵ -deterrent, for every $0 \leq \epsilon \leq 1$.*

This proposition follows from the simple observation that according to Definition 2.3.1, there exists a simulator that always succeeds in its simulation. Thus, the same simulator works here (there is simply no need to ever send a cheat input).

Next, we consider the relation between covert and semi-honest adversaries. As we have discussed in Section 2.3.3, 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. This same argument holds for covert adversaries and we therefore consider *augmented semi-honest* adversaries. We have the following:

Proposition 2.4.4 *Let π be a protocol that securely computes some functionality f in the presence of covert adversaries with ϵ -deterrent, for $\epsilon(n) \geq 1/\text{poly}(n)$. Then, π securely computes f in the presence of augmented semi-honest adversaries.*

Proof. Let π securely compute f in the presence of covert adversaries with ϵ -deterrent, where $\epsilon(n) \geq 1/\text{poly}(n)$. The first observation is that since honest parties cannot send `abort`, `corrupted` or `cheat` instructions in the ideal model, it holds that when both parties are honest in a real execution of π , the values `abort` and `corrupted` appear in the output with only negligible probability.

Consider now the case of an augmented semi-honest adversary \mathcal{A} that controls one of the parties, and let \mathcal{S} be the simulator for \mathcal{A} . We claim that \mathcal{S} sends `abort`, `corrupted` or `cheat` in the ideal model with at most negligible probability. This is due to the fact that the output of the honest party in an execution with \mathcal{A} is indistinguishable from its output when both parties are honest (because the distribution over the messages received by the honest party in both executions is identical). In particular, the honest party in an execution with \mathcal{A} outputs `abort` or `corrupted` with at most negligible probability. Now, in the ideal setting, an honest party outputs `abort` or `corrupted` whenever \mathcal{S} sends `abort` or `corrupted` (and so it can send these with only negligible probability). Furthermore, an honest party outputs `corrupted` with probability ϵ whenever \mathcal{S} sends `cheat`. Since $\epsilon \geq 1/\text{poly}(n)$, it follows that \mathcal{S} can send `cheat` also with only negligible probability. We therefore have that the ideal model with such an \mathcal{S} is the *standard* ideal model (with no cheating possibility), and the augmented semi-honest simulator can just run \mathcal{S} . We stress that this only holds for the augmented semi-honest case, because \mathcal{S} may change the corrupted party's inputs (we have no control over \mathcal{S}) and so the semi-honest simulator can only run \mathcal{S} if it too can change the corrupted party's inputs. ■

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 $1/\text{poly}(n) \leq \epsilon(n) \leq 1 - 1/\text{poly}(n)$ then it can be shown that Definition 2.4.1 is strictly different to both the semi-honest and malicious models (this is not difficult to see and so details are omitted). However, as we show below, when $\epsilon(n) = 1 - \mu(n)$, Definition 2.4.1 is equivalent to security in the presence of malicious adversaries (Definition 2.3.1).

Stated differently, the following proposition shows that the definition of security for covert adversaries “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 malicious model an honest party never outputs `corruptedi`, whereas this can occur in the setting of covert adversaries 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 corrupted_i in π , then it outputs abort_i instead in π' . We have the following:

Proposition 2.4.5 *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 2.4.1 if and only if π' securely computes f with abort in the presence of malicious adversaries.*

Proof. The fact that security in the presence of malicious adversaries implies security in the presence of covert adversaries has already been proven in Proposition 2.4.3 (observe that Proposition 2.4.3 holds for all ϵ , including ϵ that is negligibly close to 1). We now prove that security in the presence of covert adversaries under Definition 2.4.1 with ϵ that is negligibly close to 1 implies security in the presence of malicious adversaries. This holds because if the ideal adversary does not send cheat_i then the ideal execution is the same as in the regular ideal model. Furthermore, if it does send cheat_i , it is caught cheating with probability that is negligibly close to 1 and so the protocol is aborted. Recall that by Definition 2.4.1, when the adversary is caught cheating it learns nothing and so the effect is the same as an abort in the regular ideal model (technically, the honest party has to change its output from corrupted_i to abort_i as discussed above, but this makes no difference). We conclude that when ϵ is negligibly close to 1, sending cheat_i is the same as sending abort_i and so the security is the same as in the presence of malicious adversaries. ■

2.5 Restricted Versus General Functionalities

In this section, we show that it often suffices to construct a secure protocol for a restricted type of functionality, and the result to general functionalities can be automatically derived. This is most relevant for *general constructions* that are based on a circuit that computes the functionality in question. As we will see, in these cases the cost of considering the restricted types of functionalities considered here is inconsequential. For the sake of clarity, our general constructions will therefore all be for restricted functionalities of the types defined below.

The claims in this section are all quite straightforward. We therefore present the material somewhat informally, and leave formal claims and proofs as an exercise to the reader.

2.5.1 Deterministic Functionalities

The general definition considers probabilistic functionalities where the output $f(x, y)$ is a random variable. A classic example of a probabilistic functionality is that of coin-tossing. For example, one could define $f(1^n, 1^n)$ to be a uniformly distributed string of length n .

We show that it suffices to consider deterministic functionalities when constructing general protocols for secure computation. Specifically, we show that given a protocol for securely computing any deterministic functionality, it is possible to construct a secure protocol for computing any probabilistic functionality. Let $f = (f_1, f_2)$ be a two-party probabilistic functionality. We denote by $f(x, y; w)$ the output of f upon inputs x and y , and random tape w (the fact that f is probabilistic means that it has a uniformly distributed random tape). Next, define a deterministic functionality $g((x, r), (y, s)) = f(x, y; r \oplus s)$, where (x, r) is P_1 's input and (y, s) is P_2 's input, and assume that we have a secure protocol π' for computing f' . We now present a secure protocol π for computing f that uses π' for computing f' . Upon respective inputs $x, y \in \{0, 1\}^n$, parties P_1 and P_2 choose uniformly distributed strings $r \leftarrow_R \{0, 1\}^{q(n)}$ and $s \leftarrow_R \{0, 1\}^{q(n)}$, respectively, where $q(n)$ is an upper bound on the number of random bits used to compute f . They then invoke the protocol π' for securely computing f' in order to both obtain $f'((x, r), (y, s)) = f(x, y; r \oplus s)$. The fact that this yields a secure protocol for computing f follows from the fact that as long as either r or s is uniformly distributed, the resulting $w = r \oplus s$ is also uniformly distributed. This reduction holds for the case of semi-honest, malicious and covert adversaries.

Observe that in the case of general protocols that can be used for securely computing any functionality, the complexity of the protocol for computing f' is typically the same as for computing f . This is due to the fact that the complexity of these protocols is related to the size of the circuit computing the functionality, and the size of the circuit computing f' is of the same order as the size of the circuit computing f . The only difference is that the circuit for f' has $q(n)$ additional exclusive-or gates, where $q(n)$ is the length of f 's random tape.

2.5.2 Single-Output Functionalities

In the general definition of secure two-party computation, both parties receive output and these outputs may be *different*. However, it is often far simpler to assume that only party P_2 receives output; we call such a functionality *single-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 $f(x, y)$ 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 $f_1(x, y)$ and party P_2 receives $f_2(x, y)$. For simplicity, we will assume that the length of the output of $f_1(x, y)$ 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. We show this reduction separately for semi-honest and malicious adversaries, as the semi-honest reduction is more efficient than the malicious one.

Semi-honest adversaries. Let $f = (f_1, f_2)$ be an arbitrary probabilistic polynomial-time computable functionality and define the single-output functionality f' as follows: $f'((x, r), (y, s)) = (f_1(x, y) \oplus r \parallel f_2(x, y) \oplus s)$ where $a \parallel b$ denotes the concatenation of a with b . Now, given a secure protocol π' for computing the single-output functionality f' where P_2 only receives the output, it is possible to securely compute the functionality $f = (f_1, f_2)$ as follows. Upon respective inputs $x, y \in \{0, 1\}^n$, parties P_1 and P_2 choose uniformly distributed strings $r \leftarrow_R \{0, 1\}^{q(n)}$ and $s \leftarrow_R \{0, 1\}^{q(n)}$, respectively, where $q(n)$ is an upper bound on the output length of f on inputs of length n . They then invoke the protocol π' for securely computing f' in order for P_2 to obtain $f'((x, r), (y, s))$; denote the first half of this output by v and the second half by w . Upon receiving (v, w) , party P_2 sends v to P_1 , which then computes $v \oplus r$ and obtains $f_1(x, y)$. In addition, party P_2 computes $w \oplus s$ and obtains $f_2(x, y)$. It is easy to see that the resulting protocol securely computes f . This is due to the fact that r completely obscures $f_1(x, y)$ from P_2 . Thus, neither party learns more than its own input. (In fact, the strings $f_1(x, y) \oplus r$ and $f_2(x, y) \oplus s$ are uniformly distributed and so are easily simulated.)

As in the case of probabilistic versus deterministic functionalities, the size of the circuit computing f' is of the same order as the size of the circuit computing f . The only difference is that f' has one additional exclusive-or gate for every circuit-output wire.

Malicious adversaries. Let $f = (f_1, f_2)$ be as above; we construct a protocol in which P_1 receives $f_1(x, y)$ and P_2 receives $f_2(x, y)$ that is secure in the presence of malicious adversaries. As a building block we use a protocol for computing any efficient functionality, with security for malicious adversaries, with the limitation that only P_2 receives output. As in the semi-honest case, P_2 will also receive P_1 's output in encrypted format, and will then hand it to P_1 after the protocol concludes. However, a problem arises in that P_2 can modify the output that P_1 receives (recall that the adversary may be malicious here). In order to prevent this, we add message authentication to the encrypted output.

Let $r, a, b \leftarrow_R \{0, 1\}^n$ be randomly chosen strings. Then, in addition to x , 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:

$$g((r, a, b, x), y) = (\alpha, \beta, f_2(x, y))$$

where $\alpha = r + f_1(x, y)$, $\beta = a \cdot \alpha + b$, 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 whether $\beta = a \cdot \alpha + b$; 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 $f_1(x, y)$, 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. Formally, proving the security of this transformation requires a modular composition theorem; this is discussed in Section 2.7 below.

As is the case for the previous reductions above, the circuit for computing g is only mildly larger than that for computing f . Thus, the modification above 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 the protocol below).

2.5.3 Non-reactive Functionalities

As described in Section 2.3.2, a reactive functionality is one where the computation is carried out over multiple phases, and the parties may choose their inputs in later phases based on the outputs that they have already received. Recall that such a reactive functionality can be viewed as a series of functionalities (f^1, f^2, \dots) such that the input to f^j is the tuple (x_j, y_j, σ_{j-1}) and the output includes the parties' outputs and state information σ_j ; see Section 2.3.2 for more details. We denote by f_1^j and f_2^j the corresponding outputs of parties P_1 and P_2 from f^j , and by σ_j the state output from f^j .

In this section, we show that it is possible to securely compute any reactive functionality given a general protocol for computing non-reactive functionalities. The basic idea behind the reduction is the same as for same-output functionalities (for the semi-honest case) and single-output functionalities (for the malicious case). Specifically, the parties receive the same output as usual, but also receive random shares of the state at each stage; i.e., one party receives a random pad and the other receives the state encrypted by this pad. This ensures that neither party learns the internal state of the reactive functionality. Observe that although this suffices for the semi-honest case, it does not suffice for the case of malicious adversaries, which may modify the values that they are supposed to input. Thus, for the malicious case, we also add a message authentication tag to prevent any party from modifying the share of the state received in the previous stage.

Semi-honest adversaries. Let (f^1, f^2, \dots) be the series of functionalities defining the reactive functionality. First, we define a series of functionalities (g^1, g^2, \dots) such that

$$\begin{aligned} g^j &((x_j, \sigma_{j-1}^1), (y_j, \sigma_{j-1}^2)) \\ &= \left(\left(f_1^j(x_j, y_j, \sigma_{j-1}^1 \oplus \sigma_{j-1}^2), \sigma_j^1 \right), \left(f_2^j(x_j, y_j, \sigma_{j-1}^1 \oplus \sigma_{j-1}^2), \sigma_j^2 \right) \right) \end{aligned}$$

where σ_j^1 and σ_j^2 are uniformly distributed strings under the constraint that $\sigma_j^1 \oplus \sigma_j^2 = \sigma_j$ (the state after the j th stage). That is, g^j receives input (x_j, σ_{j-1}^1) from P_1 and input (y_j, σ_{j-1}^2) from P_2 and then computes f^j on inputs (x_j, y_j, σ_{j-1}) where $\sigma_{j-1} = \sigma_{j-1}^1 \oplus \sigma_{j-1}^2$. In words, g^j receives the parties' inputs to the phase, together with a *sharing* of the state from the previous round. Functionality g^j then outputs the phase outputs to each party, and a sharing of the state from this round of computation.

Malicious adversaries. As we have mentioned, the solution for semi-honest adversaries does not suffice when considering malicious adversaries because nothing prevents the adversary from modifying its share of the state. This is solved by also having party P_1 receive a MAC (message authentication code) key k_1 and a MAC tag $t_j^1 = \text{MAC}_{k_2}(\sigma_j^1)$, where the keys k_1, k_2 are chosen randomly in the phase computation and σ_j^1 is the share of the current state that P_1 holds. Likewise, P_2 receives k_2 and $t_j^2 = \text{MAC}_{k_1}(\sigma_j^2)$. Then, the functionality in the $(j + 1)$ th phase receives the parties' phase-inputs, shares σ_j^1 and σ_j^2 of σ_j , keys k_1, k_2 , and MAC-tags t_j^1, t_j^2 . The functionality checks the keys and MACs and if the verification succeeds, it carries out the phase computation with the inputs and given state. By the security of the MAC, a malicious adversary is unable to change the current state, except with negligible probability.

2.6 Non-simulation-Based Definitions

2.6.1 Privacy Only

The definition of security that follows the ideal/real simulation paradigm provides strong security guarantees. In particular, it guarantees privacy, correctness, independence of inputs and more. However, in some settings, it may be sufficient to guarantee privacy only. We warn that this is not so simple and in many cases it is difficult to separate privacy from correctness and independence of inputs. For example, consider a function f with the property that for every y there exists a x_y such that $f(x_y, y) = y$. Now, if party P_1 can somehow make its input x depend on P_2 's input (something which is not possible when independence of inputs is guaranteed), then it may be able to

always set $x = x_y$ and learn P_2 's input in entirety. (We stress that although this sounds far fetched, such attacks are actually sometimes possible.)

Another difficulty that arises when defining privacy is that it typically depends very much on the function being computed. Intuitively, we would like to require that if two different inputs result in the same output, then no adversarial party can tell which of the two inputs the other party used. In other words, we would like to require that for every adversarial P_1 and input x , party P_1 cannot distinguish whether P_2 used y or y' when the output is $f(x, y)$ and it holds that $f(x, y) = f(x, y')$. However, such a formulation suffers from a number of problems. First, if f is 1-1 no privacy guarantees are provided at all, even if it is hard to invert. Second, the formulation suffers from the exact problem described above. Namely, if it is possible for P_1 to implicitly choose $x = x_y$ based on y (say by modifying a commitment to y that it receives from P_2) so that $f(x_y, y)$ reveals more information about y than “the average x ”, then privacy is also breached. Finally, we remark that (sequential) composition theorems, like those of Section 2.7, are not known for protocols that achieve privacy only. Thus, it is non-trivial to use protocols that achieve privacy only as subprotocols when solving large protocol problems.

Despite the above problems, it is still sometimes possible to provide a workable definition of privacy that provides non-trivial security guarantees and is of interest. Due to the difficulty in providing a general definition, we will present a definition for one specific function in order to demonstrate how such definitions look. For this purpose, we consider the oblivious transfer function. Recall that in this function, there is a sender S with a pair of input strings (x_0, x_1) and a receiver R with an input bit σ . The output of the function is nothing to the sender and the string x_σ for the receiver. Thus, a secure oblivious transfer protocol has the property that the sender learns nothing about σ while the receiver learns at most one of the strings x_0, x_1 . Unfortunately, defining privacy here without resorting to the ideal model is very non-trivial. Specifically, it is easy to define privacy in the presence of a malicious sender S^* ; we just say that S^* cannot distinguish the case where R has input 0 from the case where it has input 1. However, it is more difficult to define privacy in the presence of a malicious receiver R^* because it does learn something. A naive approach to defining this says that for some bit b it holds that R^* knows nothing about x_b . However, this value of b may depend on the messages sent during the oblivious transfer and so cannot be fixed ahead of time (see the discussion above regarding independence of inputs).

Fortunately, for the case of *two-message* oblivious transfer (where the receiver sends one message and the sender replies with a single message) it is possible to formally define this. The following definition of security for oblivious transfer is based on [42] and states that replacing one of x_0 and x_1 with some other x should go unnoticed by the receiver. The question of which of x_0, x_1 to replace causes a problem which is solved in the case of a two-message protocol by fixing the first message; see below. (In the definition below we use the following notation: for a two-party protocol with parties S and R ,

we denote by $\text{VIEW}_S(S(1^n, a), R(1^n, b))$ the view of S in an execution where it has input a , R has input b , and the security parameter is n . Likewise, we denote the view of R by $\text{VIEW}_R(S(1^n, a), R(1^n, b))$.

Definition 2.6.1 *A two-message two-party probabilistic polynomial-time protocol (S, R) is said to be a private oblivious transfer if the following holds:*

- **NON-TRIVIALITY:** *If S and R follow the protocol then after an execution in which S has for input any pair of strings $x_0, x_1 \in \{0, 1\}^*$, and R has for input any bit $\sigma \in \{0, 1\}$, the output of R is x_σ .*
- **PRIVACY IN THE CASE OF A MALICIOUS S^* :** *For every non-uniform probabilistic polynomial-time S^* and every auxiliary input $z \in \{0, 1\}^*$, it holds that*

$$\{\text{VIEW}_{S^*}(S^*(1^n, z), R(1^n, 0))\}_{n \in \mathbb{N}} \stackrel{c}{=} \{\text{VIEW}_{S^*}(S^*(1^n, z), R(1^n, 1))\}_{n \in \mathbb{N}}.$$

- **PRIVACY IN THE CASE OF A MALICIOUS R^* :** *For every non-uniform deterministic polynomial-time receiver R^* , every auxiliary input $z \in \{0, 1\}^*$, and every triple of inputs $x_0, x_1, x \in \{0, 1\}^*$ such that $|x_0| = |x_1| = |x|$ it holds that either:*

$$\{\text{VIEW}_{R^*}(S(1^n, (x_0, x_1)); R^*(1^n, z))\}_{n \in \mathbb{N}} \stackrel{c}{=} \{\text{VIEW}_{R^*}(S(1^n, (x_0, x)); R^*(1^n, z))\}_{n \in \mathbb{N}}$$

or

$$\{\text{VIEW}_{R^*}(S(1^n, (x_0, x_1)); R^*(1^n, z))\}_{n \in \mathbb{N}} \stackrel{c}{=} \{\text{VIEW}_{R^*}(S(1^n, (x, x_1)); R^*(1^n, z))\}_{n \in \mathbb{N}}.$$

The way to view the above definition of privacy in the case of a malicious R^* is that R^* 's first message, denoted by $R^*(1^n, z)$, fully determines whether it should receive x_0 or x_1 . If it determines for example that it should receive x_0 , then its view (i.e., the distribution over S 's reply) when S 's input is (x_0, x_1) is indistinguishable from its view when S 's input is (x_0, x) . Clearly this implies that R^* cannot learn anything about x_1 when it receives x_0 and vice versa. In addition, note that since R^* sends its message before receiving anything from S , and since this message fully determines R^* 's input, we have that the problem of independence of inputs discussed above does not arise.

Note that when defining the privacy in the case of a malicious R^* we chose to focus on a *deterministic* polynomial-time receiver R^* . This is necessary in order to fully define the message $R^*(z)$ for any given z , which in turn fully defines the string x_b that $R^*(z)$ does *not* learn. By making R^* non-uniform, we have that this does not weaken the adversary (since R^* 's advice tape can hold its “best coins”). We remark that generalizing this definition to protocols that have more than two messages is non-trivial. Specifically, the problem of independence of inputs described above becomes difficult again when more than two messages are sent.

The above example demonstrates that it is possible to define “privacy only” for secure computation. However, it also demonstrates that this task

can be *very difficult*. In particular, we *do not know of a satisfactory definition* of privacy for oblivious transfer with more than two rounds. In general, one can say that when a party does not receive output, it is easy to formalize privacy because it learns nothing. However, when a party does receive output, defining privacy without resorting to the ideal model is problematic (and often it is not at all clear how it can be achieved).

We conclude with one important remark regarding “privacy-only” definitions. As we have mentioned, an important property of security definitions is a composition theorem that guarantees certain behavior when the secure protocol is used as a subprotocol in another larger protocol. No such general composition theorems are known for definitions that follow the privacy-only approach. As such, this approach has a significant disadvantage.

2.6.2 One-Sided Simulatability

Another approach to providing weaker, yet meaningful, security guarantees is that of *one-sided simulation*. This notion is helpful when only one party receives output while the other learns nothing. As discussed above in Section 2.6.1, when a party should learn nothing (i.e., when it has no output), it is easy to define privacy via indistinguishability as for encryption. Specifically, it suffices to require that the party learning nothing is not able to distinguish between *any* two inputs of the other party.¹ In contrast, the difficulty of defining privacy appropriately for the party that does receive output is overcome by requiring full simulation in this case. That is, consider a protocol/functionality where P_2 receives output while P_1 learns nothing. Then, in the case where P_1 is corrupted we require that it not be able to learn anything about P_2 ’s input and formalize this via indistinguishability. However, in the case where P_2 is corrupted, we require the existence of a simulator that can fully simulate its view, as in the definition of Section 2.3. This is helpful because it enables us to provide a general definition of security for problems of this type where only one party receives output. Furthermore, it turns out that in many cases, it *is* possible to achieve high efficiency when “one-sided simulation” is sufficient and “full simulation” is not required.

It is important to note that this is a *relaxed level of security* and does not achieve everything we want. For example, a corrupted P_1 may be able to make its input depend on the other party’s input, and may also be able to cause the output to be distributed incorrectly. Thus, this notion is not suitable for all protocol problems. Such compromises seem inevitable given the current state of the art, where highly-efficient protocols that provide full simulation-based security in the presence of malicious adversaries seem very

¹ Note that this only makes sense when the party receives no output. Otherwise, if it does receive output, then the other party’s input has influence over that output and so it is unreasonable to say that it is impossible to distinguish between any two inputs.

hard to construct. We stress that P_2 cannot carry out any attacks, because full simulation is guaranteed in the case where it is corrupted.

The definition. Let f be a function $f : \{0, 1\}^* \times \{0, 1\}^* \rightarrow \{0, 1\}^*$ with only a single output which is designated for P_2 . Let $\text{REAL}_{\pi, \mathcal{A}(z), i}(x, y, n)$ denote the outputs of the honest party and the adversary \mathcal{A} (controlling party P_i) after a real execution of protocol π , where P_1 has input x , P_2 has input y , \mathcal{A} has auxiliary input z , and the security parameter is n . Let $\text{IDEAL}_{f, \mathcal{S}(z), i}(x, y, n)$ be the analogous distribution in an ideal execution with a trusted party that computes f for the parties and hands the output to P_2 only. Finally, let $\text{VIEW}_{\pi, \mathcal{A}(z), i}^{\mathcal{A}}(x, y, n)$ denote the view of the adversary after a real execution of π as above. Then, we have the following definition:

Definition 2.6.2 *Let f be a functionality where only P_2 receives output. We say that a protocol π securely computes f with one-sided simulation if the following holds:*

1. *For every non-uniform PPT adversary \mathcal{A} controlling P_2 in the real model, there exists a non-uniform PPT adversary \mathcal{S} for the ideal model, such that*

$$\{\text{REAL}_{\pi, \mathcal{A}(z), 2}(x, y, n)\}_{x, y, z, n} \stackrel{c}{=} \{\text{IDEAL}_{f, \mathcal{S}(z), 2}(x, y, n)\}_{x, y, z, n}$$

where $n \in \mathbb{N}$, $x, y, z \in \{0, 1\}^$ and $|x| = |y|$.*

2. *For every non-uniform PPT adversary \mathcal{A} controlling P_1 ;*

$$\{\text{VIEW}_{\pi, \mathcal{A}(z), 1}^{\mathcal{A}}(x, y, n)\}_{x, y, y', z, n} \stackrel{c}{=} \{\text{VIEW}_{\pi, \mathcal{A}(z), 1}^{\mathcal{A}}(x, y', n)\}_{x, y, y', z, n} \quad (2.3)$$

where $n \in \mathbb{N}$, $x, y, y', z \in \{0, 1\}^$ and $|x| = |y| = |y'|$.*

Note that the ensembles in (2.3) are indexed by two different inputs y and y' for P_2 . The requirement is that \mathcal{A} cannot distinguish between the cases where P_2 used the first input y and the second input y' .

2.7 Sequential Composition – Simulation-Based Definitions

A protocol that is secure under *sequential* composition maintains its security when run multiple times, as long as the executions are run sequentially (meaning that each execution concludes before the next execution begins). Sequential composition theorems are theorems that state “if a protocol is secure in the stand-alone model under definition X, then it remains secure under sequential composition”. Thus, we are interested in proving protocols secure under Definitions 2.2.1, 2.3.1 and 2.4.1 (for semi-honest, malicious and covert adversaries), and immediately deriving their security under sequential

composition. This is important for two reasons. First, sequential composition constitutes a security goal within itself as security is guaranteed even when parties run many executions, albeit sequentially. Second, sequential composition theorems are useful tools that help in writing proofs of security. Specifically, when constructing a protocol that is made up of a number of secure subprotocols, it is possible to analyze the security of the overall protocol in a modular way, because the composition theorems tell us that the subprotocols remain secure in this setting.

We do not present proofs of the sequential composition theorems for the semi-honest and malicious cases as these already appear in [32]; see Sections 7.3.1 and 7.4.2 respectively. However, we do present a formal *statement* of the theorems as we will use them in our proofs of security of the protocols. In addition, we provide a proof of sequential composition for the case of covert adversaries.

Modular sequential composition. The basic idea behind the formulation of the modular sequential composition theorems is to show 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 using 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 with real executions of a secure protocol computing this functionality, the protocol remains secure. We begin by presenting the “hybrid model” where parties communicate by sending regular messages to each other (as in the real model) but also have access to a trusted party (as in the ideal model).

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 $f_1, \dots, f_{p(n)}$. 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 $f_1, \dots, f_{p(n)}$ in that order). In addition, if a functionality f_i is *reactive* (meaning that it contains multiple stages like a commitment functionality which has a commit and reveal stage), then no messages are sent by the parties directly to each other from the time that the first message is sent to f_i to the time that all stages of f_i have concluded. We stress that the honest party sends its input to the trusted party in the same round and does not send other messages until it receives its output (this is because we consider *sequential composition* here). Of course, the trusted

party may be used a number of times throughout the execution if π . However, each use 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, the trusted party computing $f_1, \dots, f_{p(n)}$ behaves as in Section 2.3 when malicious adversaries are being considered, and as in Section 2.4 for covert adversaries.

Sequential composition – malicious adversaries. Let $f_1, \dots, f_{p(n)}$ be probabilistic polynomial-time functionalities and let π be a two-party protocol that uses ideal calls to a trusted party computing $f_1, \dots, f_{p(n)}$. Furthermore, let \mathcal{A} be a non-uniform probabilistic polynomial-time machine and let i be the index of the corrupted party. Then, the $f_1, \dots, f_{p(n)}$ -hybrid execution of π on inputs (x, y) , auxiliary input z to \mathcal{A} and security parameter n , denoted $\text{HYBRID}_{\pi, \mathcal{A}(z), i}^{f_1, \dots, f_{p(n)}}(x, y, n)$, 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 $f_1, \dots, f_{p(n)}$.

Let $\rho_1, \dots, \rho_{p(n)}$ be a series of protocols (as we will see ρ_i takes the place of f_i in π). We assume that each ρ_i has a fixed number rounds that is the same for all parties. Consider the real protocol $\pi^{\rho_1, \dots, \rho_{p(n)}}$ that is defined as follows. All standard messages of π are unchanged. When a party is instructed to send an ideal message α to the trusted party to compute f_j , it begins a real execution of ρ_j with input α instead. When this execution of ρ_j concludes with output y , the party continues with π as if y were the output received by the trusted party for f_j (i.e., as if it were running in the hybrid model).

The composition theorem states that if $\rho_1, \dots, \rho_{p(n)}$ securely compute $f_1, \dots, f_{p(n)}$ respectively, and π securely computes some functionality g in the $f_1, \dots, f_{p(n)}$ -hybrid model, then $\pi^{\rho_1, \dots, \rho_{p(n)}}$ securely computes g (in the real model). As discussed above, the hybrid model that we consider here is where the protocols are run sequentially. Thus, the fact that sequential composition only is considered is implicit in the theorem, via the reference to the hybrid model.

Theorem 2.7.1 (modular sequential composition – malicious): *Let $p(n)$ be a polynomial, let $f_1, \dots, f_{p(n)}$ be two-party probabilistic polynomial-time functionalities and let $\rho_1, \dots, \rho_{p(n)}$ be protocols such that each ρ_i securely computes f_i in the presence of malicious adversaries. Let g be a two-party functionality and let π be a protocol that securely computes g in the $f_1, \dots, f_{p(n)}$ -hybrid model in the presence of malicious adversaries. Then, $\pi^{\rho_1, \dots, \rho_{p(n)}}$ securely computes g in the presence of malicious adversaries.*

Sequential composition – covert adversaries. Let $f_1, \dots, f_{p(n)}$, π and $\rho_1, \dots, \rho_{p(n)}$ be as above. Furthermore, define $\pi^{\rho_1, \dots, \rho_{p(n)}}$ exactly as in the malicious model. Note, however, that in the covert model, a party may receive

corrupted_k as output from ρ_j . In this case, as with any other output, it behaves as instructed in π (corrupted_k may be received as output both when the real ρ_j is run and when the trusted party is used to compute f_j because we consider the covert ideal model here). The covert ideal model IDEALSC depends on the deterrent factor because this determines the probability with which the trusted party sends corrupted or undetected . Therefore, we refer to the (f, ϵ) -hybrid model as one where the trusted party computes f and uses the given ϵ . When considering protocols $\rho_1, \dots, \rho_{p(n)}$ we will refer to the $(f_1, \epsilon_1), \dots, (f_{p(n)}, \epsilon_{p(n)})$ -hybrid model, meaning that the trusted party uses ϵ_i when computing f_i (and the ϵ values may all be different). We have the following:

Theorem 2.7.2 *Let $p(n)$ be a polynomial, let $f_1, \dots, f_{p(n)}$ be two-party probabilistic polynomial-time functionalities and let $\rho_1, \dots, \rho_{p(n)}$ be protocols such that each ρ_i securely computes f_i in the presence of covert adversaries with deterrent ϵ_i . Let g be a two-party functionality and let π be a protocol that securely computes g in the $(f_1, \epsilon_1), \dots, (f_{p(n)}, \epsilon_{p(n)})$ -hybrid model in the presence of covert adversaries with ϵ -deterrent. Then, $\pi^{\rho_1, \dots, \rho_{p(n)}}$ securely computes g in the presence of covert adversaries with ϵ -deterrent.*

Proof (sketch). Theorem 2.7.2 can be derived as an almost immediate corollary from the composition theorem of [11, 32] in the following way. First, define a special functionality interface that follows the instructions of the trusted party in Definition 2.4.1. That is, define a *reactive functionality* (see Section 2.3.2) 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 corrupted_i to the honest party and with probability $1 - \epsilon_\ell$ gives the adversary the honest party's input and lets it choose its output). Next, consider the standard ideal model of Definition 2.3.1 with functionalities of the above form. It is easy to see that a protocol securely computes some functionality f under Definition 2.4.1 *if and only if* it securely computes the appropriately defined reactive functionality under Definition 2.3.1. This suffices because the composition theorem of [11, 32] can be applied to Definition 2.3.1, yielding the result. ■

Observe that in Theorem 2.7.2 the protocols $\rho_1, \dots, \rho_{p(n)}$ 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 $\epsilon_1, \dots, \epsilon_{p(n)}$ of the protocols $\rho_1, \dots, \rho_{p(n)}$, respectively.



<http://www.springer.com/978-3-642-14302-1>

Efficient Secure Two-Party Protocols
Techniques and Constructions

Hazay, C.; Lindell, Y.

2010, XIII, 263 p., Hardcover

ISBN: 978-3-642-14302-1