LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-511

SECURE COMPUTATION
(Preliminary Report)

Silvio Micali
Phillip Rogaway

August 1991

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

This blank page was inserted to preserve pagination.

Secure Computation

(Preliminary Report)

Silvio Micali **Phillip Rogaway**

August 9, 1991

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
USA

Abstract

We define what it means for a network of communicating players to *securely* compute a function of privately held inputs. Intuitively, we wish to *correctly* compute its value in a manner which protects the *privacy* of each player's contribution, even though a powerful *adversary* may endeavor to disrupt this enterprise.

This highly general and desirable goal has been around a long time, inspiring a large body protocols, definitions, and ideas, starting with Yao [1982, 1986] and Goldreich, Micali and Wigderson [1987]. But all the while, it had resisted a full and satisfactory formulation.

Our definition is built on several new ideas. Among them:

- *Blending* privacy and correctness in a deeper manner, using a *special type of simulator* designed for the purpose.
- Requiring *adversarial awareness*—capturing the idea that the adversary should know, in a very strong sense, certain information associated to the execution of a protocol.
- Imitating an *ideal evaluation* in a *run-by-run* manner, our definition depending more on individual executions than on global properties of ensembles.

Among the noteworthy and desirable properties of our definition is the *reducibility* of secure protocols, which we believe to be a cornerstone in a matu[illegible] theory of secure computation.

An earlier draft of this paper was presented and distributed at a DIMACS workshop on cryptography, October 1990. The paper continues to evolve; this is a preliminary report on our work.

Contents

1 Introduction

The last decade has witnessed the rise of *secure computation* as a new and exciting mathematical subject. This is the study of communication protocols allowing several parties to perform a correct computation on some inputs that are and should be kept private. As a simple example, the parties want to compute the tally of some privately held votes. This new discipline is extremely subtle, involving in novel ways fundamental concepts such as probabilism, information, and complexity theory.

In the making of a new science, finding the correct definitions can be one of the most difficult tasks: from relatively few examples, one should handle cases that have not yet arisen and reach the highest possible level of generality. It is the purpose of this paper to identify the *right* notion of secure computation and prove the *right* fundamental properties about it.

In the last few years, cryptography has been very successful in identifying its basic objectives, properly defining them, and successfully solving them. Secure encryption, secure pseudorandom generation, secure digital signatures, and zero-knowledge proofs—concepts that appeared forever elusive—have all found successful formalizations and solutions. But in contrast to these successes, and despite many beautiful and fundamental ideas that preceded us, not even a satisfactory *definition* of a secure protocol has been proposed so far. This is not surprising, since protocols are an extremely complex object: by defining security for encryption, signatures, and pseudorandom generation, one is defining properties of algorithms; but to properly define protocol security, one needs instead to define properties of the *interaction* of several algorithms, some of which may be deliberately designed to *disrupt* the joint computation in various ways. The intricacy of this scenario has often encouraged researchers to work either with definitions of security tailored to the problem at hand; or to consider broad definitions, but restricted to specific computational tasks; or to work with only intuitive notions in mind.

Lack of universally accepted definitions can only create confusion and mistakes, and it is only by reaching an *exact* understanding of what we can expect from a secure protocol can we safely rely on them and further develop them. By developing the right notion of secure computation we will spare cryptography the increasing dangers of confusion, error, and misunderstanding. Powerful computer networks are already in place and the possibility of using them for new and wonderful tasks will largely depend on how successful this development will be. The goal is definitely worth the effort, for its potential applications and from a purely intellectual point of view. Lets us begin!

1.1 Secure-Computation Problems

What is secure computation about? Informally, it consists of finding a communication *protocol* that allows a group of *players* to accomplish a special type of *task*, despite the fact that some of them may try to sabotage this enterprise. This said, we now explain terms. Let's start with the easy ones.

Players (also called *processors* or *parties* for variation of discourse) can be thought as people, each possessing a personal computer, and capable of exchanging messages. A *protocol* is a set of instructions for the players to follow for sending these messages. The rules of the game are as follows: (1) in executing a protocol, some of the participants may be *bad*, thereby disregarding their instructions and cooperating to disrupt the joint effort; (2) no trusted device or external entity is available; (3) every good party can perform private computation (i.e., computation unmonitored by the bad players).

What is a secure protocol supposed to accomplish? We start by looking at a few archetypal examples. Since our aim is to exemplify various issues and key desiderata that may inspire us

to properly define secure computation, in the following list we credit the one who first *posed* the problem.

1. THE MILLIONAIRES PROBLEM (Yao, [Ya82a]). Two millionaires wish to find out who is richer, though neither is willing to reveal the extent of his fortune. Can they carry out a conversation which identifies the richer millionaire, but doesn't divulge additional information about either's wealth?

2. THE DIGITAL VOTING PROBLEM (Chaum, [Ch81]). Is it possible for a group of computer users to hold a secret-ballot election on a computer network?

3. THE INDEPENDENT ANNOUNCEMENT PROBLEM (Chor, Goldwasser, Micali, and Awerbuch [CGMA85]). A group of players want to exchange messages so as to announce their secret values independently. That is, what the bad players announce cannot be chosen based on the values of the good players.

4. THE COIN FLIPPING PROBLEM (Blum, [Bl82]). How can Alice and Bob, speaking to one another over the telephone, agree on a random, unbiased coin flip—even if one of them cheats to try to produce a coin flip of a certain outcome?

5. THE OBLIVIOUS TRANSFER PROBLEM (Rabin, [Ra81]). Is it possible for Alice to send to Bob a message m in such a way that (i) half the time, Bob gets m; (ii) the other half of the time, Bob gets nothing; and (iii) Alice never knows which of the two events has occurred?

6. THE MENTAL POKER PROBLEM (Shamir, Rivest and Adleman, [SRA81]). Can a group of players properly shuffle and deal a deck of cards over the phone?

1.2 Privacy and Correctness

Even the above short list illustrates the enormous variety of types of goals for secure protocols. There may be two parties or many. The output of a protocol may be a single value known to all players (as in digital voting), or to only one of the players (as in an oblivious transfer), or it may be a private value for each player (as in mental poker). The output may depend on the players' initial state deterministically (as in the first three problems), or probabilistically (as in the last three problems).

What do such heterogeneous problems have in common, then? Essentially, that the joint computation should both be *private* and *correct*: while preserving the privacy of individually held data, the joint computation manages to correctly perform some computational task based on this data. Correctness and privacy may seem to be conflicting requirements, and capturing in the most general sense what simultaneously meeting them means (within our rules of the game) is quite difficult. As we shall see, to obtain a satisfactory notion of security, privacy and correctness should not be handled independently (like in all prior work), but need to be *blended* in the proper way.

1.3 Prior and Related Definitions

Y-EVALUATION. Distilling a common thread in many prior examples of secure computation, Yao proposed the following general problem [Ya82a]. Assume we have n parties, $1, \ldots, n$. Each party i has a *private* input, x_i, known only to him. The parties want to compute a given function f on their own inputs while maintaining the *privacy* of these inputs. In other words, they want to compute $y = f(x_1, \ldots, x_n)$ without revealing to any player more about the private inputs than the

output itself implicitly reveals. If the function is vector-valued, $\vec{y} = f(x_1, \ldots, x_n)$, where $\vec{y}$ has n components, it is desired that every party i privately learn the i-th component of $\vec{y}$.

Yao also proposed a notion for what it means for a protocol to solve the above problem. Roughly said, his formalization attempts to capture the idea that the worst the bad players can do to disrupt a computation is to choose alternative inputs for themselves, or quit in the middle of the computation. We will refer to this notion of security as *Y-evaluation*. Subsequently [Ya86], Yao strengthened his notion of a Y-evaluation so as to incorporate some *fairness* constraint. A fair protocol is one in which there is very little advantage to be gained by quitting in the middle. That is, the protocol takes care that, at each point during the execution, the "informational gap" among the players is small. The study of fair protocols was started earlier by Luby, Micali, and Rackoff [LMR83] and progressed with the contributions of [Ya86, BG89, GL90].

GMW-GAMES. A more general notion for security has been introduced by Goldreich, Micali and Wigderson [GMW87]. They consider secure protocols as implementations of abstract, but computable, games of partial information. Informally, ingredients of such an n-player game are an arbitrary set of *states*, a set of *moves* (functions from states to states), a set of *knowledge functions* (defined on the states), and a vector-valued *outcome function* (defined on the states) whose range values have as many components as there are players. The players wish to start the game by probabilistically selecting an initial state globally unknown to every one. Then the players take turns in making moves. When it is the turn of player i, a portion $K(S)$ of the current global state S must be privately revealed to him; here K denotes the proper knowledge function for this stage of the game. Based on this private information, player i secretly selects a move μ, thereby the new, secret, state must become $\mu(S)$. At the end of the game, each player privately learns his own component of the outcome function evaluated at the final state.

[GMW87] envisioned a notion of security which would mimic the abstract game in a "virtual" manner: states are virtually selected, moves act on virtual states, and so on. We will refer to their notion as *GMW-games*. Again putting aside how this can be achieved, let us point out an additional aspect of their notion. Namely, in a GMW-game, bad players cannot disrupt the computation *at all* by quitting early. (This condition can indeed be enforced whenever the majority of the players are honest.)

OTHER PRIOR WORK. Several noteworthy variants of Y-evaluation and GMW-games have been proposed, with varying degrees of explicitness and care. These definitional ideas include, most notably, the work of Galil, Haber and Yung [GHY87], Chaum, Damgård and van de Graff [CDG87], Ben-Or, Goldwasser and Wigderson [BGW88], Chaum, Crépeau and Damgård [CCD88], Kilian [Ki89], and Crépeau [Cr90].

CONCURRENT WORK. Early in our effort we told our initial ideas (like merging privacy and correctness) to Beaver, who later pursued his own ones in [Be91a, Be91b].

Later, we collaborated with Kilian in developing definitions for secure function evaluation. This collaboration was enjoyable and profitable. Its fruits are described in [KMR90], and will be discussed in Section 3.7

Concurrently with the effort of [KMR90], Goldwasser and Levin [GL90] independently proposed an interesting approach to defining secure function evaluation.

1.4 Critique of These Definitions

Definitions cannot, of course, be "wrong" in an absolute sense; but we feel that all previous ones were either vague, or not sufficiently general, or considered "secure" protocols that should have not

been called such at a closer analysis. We thus cheerfully decided to clarify the intuitive notion of secure computation. Little we knew that we had taken up a two-year commitment!

Let us very briefly critique some of the mentioned work.

Y-EVALUATION. Though Yao should be credited for presenting his notion with great detail[1], in our opinion his ideas do not fully capture the fundamental intuition of secure computation, leading to several difficulties.

Blind input correlation. One of these difficulties we call "blind input correlation." Namely, a two-party Y-evaluation, while preventing a bad player from directly learning the good player's input, might allow him to choose his own input so to be correlated with the good player's. For example, it is not ruled out that, whenever the good player starts with secret input x, the bad player, without finding out what x was, can force the output to be, say, $y = f(x, x)$ (or $y = f(x, -x)$, or ...). In some context, this correlation may result in (what would be considered by most people) a loss of security.[2]

Privacy/correctness separation. Y-evaluation considers privacy and correctness as individual constraints and, though with a different terminology, considers secure a protocol that is *both* private and correct. Indeed, privacy and correctness are the fundamental aspects of secure computation, but, as we point out in the discussion of Chapter 3, the logical connective "and" *does not* blend them adequately together. The negative effect of this definitional approach is thus considering secure some protocols that are not such, at least according to our intuition.

GMW-GAMES. GMW-games are most general and powerful, and they are endowed with very strong intuitive appeal. The weakness of [GMW87] is in not providing full help in turning this intuition into a successful formalization. Indeed, several wrong choices could still be made from the level their definition was left at. Additionally, their notion was being developed for protocols possessing a very particular structure. Namely, the notion of security was tailored for protocols that would first perform an initial committal phase, and would then perform computation on these committed inputs. While their algorithmic structure proved to be very successful (indeed, all subsequent protocols for secure computation share it), it should not be embedded in the definition of our goal. (Indeed, the tendency of defining security as a property of a specific protocol —or class of protocols— is wide-spread in the prior literature.)

CONCURRENT WORK. Concurrent work does not suffer from this drawback, but has other shortcomings. Beaver [Be91a] does not blend the various goals for a secure protocol, but treats them as separate requirements, as Yao first did and incurring in the same type of difficulties. (This same author has presented a new paper on this same subject to *Crypto '91*, concurrently with ours, but, not having yet seen it, we cannot comment on it.) As for the work of Goldwasser and Levin, we believe that, as we point out in the discussion of Section 4, in a computationally bounded setting, it may be difficult to design protocols proven secure according to their definitions.

[1] This precision, however, was made heavier from lacking more modern constructs for discussing these issues, like the notion of a simulator developed by Goldwasser, Micali and Rackoff [GMR89].

[2] Let's see what its effects might be on some of the discussed secure-computation problems. Consider solving the digital voting problem by Y-evaluating the tally function; for simplicity, let there be only two candidates to vote for. Then the bad players —though ignoring the electoral intentions of a given good player— might succeed in voting as a block for the opposite candidate. Should we consider this a secure election? We believe not. Similarly, trying to solve the independent announcement problem by Y-evaluating the "concatenation" function (i.e., the function that, on inputs $x_1, \ldots, x_n$, returns the single value $x_1 \# \ldots \# x_n$), a bad player might always succeed in announcing the same value as a given good player. Indeed, a poor case of independence!

1.5 Our Definitions

Our notion of *secure computation* solves the above difficulties, and other ones as well. We plan to build up our definition in two stages. First, as our alternative to Y-evaluation, we define *secure function evaluation*, to which this paper is devoted. Our notion is quite powerful and expressive; for instance, the first three secure-computation problems of Subsection 1.1 are straightforwardly solvable by securely evaluating the proper function. After developing secure function evaluation, we hint how this notion can be successfully extended to that of a *secure game*, our way of fully specifying GMW-games. (The present paper is already quite long, and a different one should be devoted to a detailed treatment of this extension. Also, the present treatment restricts its definitions to there being three or more players.) Secure games capture, in our opinion, the very notion of secure computation. Quite reassuringly, all six problems of Subsection 1.1 are straightforwardly solved by "playing" a secure game.

The basic intuition behind secure function evaluation is the same one put forward, in quite a different language, by [GMW87]. In essence, two scenarios are considered: an *ideal* one in which there is a trusted party helping in the function evaluation, and a *realistic* one in which the trusted party is simulated by running a protocol. Security is a property of a realistic evaluation, and consists of achieving "indistinguishability" from the corresponding ideal evaluation. While remaining quite informal, let us at least be less succinct.

IDEAL FUNCTION EVALUATION. In an ideal function evaluation of a vector-valued function f there is an external *trusted party* to whom the participants privately give their respective secret inputs. The trusted entity will not divulge the received inputs; rather, it will correctly evaluate function f on them, and will privately hand component i of the result to party i. An *adversary* can interfere with this evaluation as follows. At the very beginning, before any party has given his own input to the trusted party, the adversary can corrupt a first player and learn his private input. After this, and possibly based on what she has just learned, the adversary may corrupt a second player and learn this input, too. This continues until the adversary is satisfied. At this point, the adversary chooses alternative, *fake* inputs for the corrupted players, and all parties give the their inputs to the trusted authority—the uncorrupted players giving their initial inputs, and the corrupted players giving their new, fake inputs. When the proper, individual outputs have been returned by the trusted party, the adversary learns the value of the output of every corrupted player. The adversary can still corrupt, one by one, additional players, learning both their inputs and outputs when she does so.

It should be noticed that in such an ideal evaluation the adversary not only learns the inputs of the players she corrupts, but by choosing properly their substitutes, she may learn from the final result quite a bit about the other inputs as well.

IDEAL VS. SECURE FUNCTION EVALUATION. While the notion of an ideal function evaluation has been essentially defined above, formalizing the notion of a secure function evaluation is much more complex; we will do it in the next few sections. Here let us just give its basic intuition. To begin with, there no longer is any trusted party; the players will instead try to simulate one by means of a protocol. The adversary can still corrupt any t players, but this time a corruption will be much more "rewarding." For not only will she learn the private inputs of corrupted players, but also their current computational state in the protocol (and a bit of additional information as well). She will receive all future message addressed to them and she will get control of what messages they are responsible for sending out.

Given the greater power of the adversary in this new setting, it is intuitively clear that a protocol for function evaluation cannot perform "better" than an ideal function evaluation; but it can do

much worse! Roughly said, a secure function evaluation consists of simulating *every important aspect* of an ideal function evaluation, to the *maximum extent possible*, so that a secure protocol does *not* perform "significantly worse" than the ideal protocol.

Setting the stage of security in terms of the above indistinguishability is an important insight of the [GMW87] work. We also follow their approach, but our notion of indistinguishability is a bit more liberal —that is, as we shall see, it crucially allows a bit more of interaction. The main difference with their work, though, and the real difficulty of ours, is not so much in the spirit of the solution as in properly realizing what "maximum possible" should mean, and identifying what are the important features, implicit (and perhaps hidden) in an ideal function evaluation that should (and can!) be mimicked in a secure function evaluation.

1.6 Key Features of Our Definitions

Let us now highlight the key features of our definitions. These we distinguish as *choices* and *properties*. The former are key technical ideas of our notion of security. The latter are key desiderata: each one is a *condition-sine-qua-non* for calling a protocol secure. Notice, though, that we do not force these necessary conditions into our definitions in an artificial manner; rather, we derive them naturally as consequences (hence the name "properties") of our notion of security, and thus of our technical choices.

Key Choices

BLENDING PRIVACY AND CORRECTNESS. Secure protocols are more than just correct and private. Simply requiring simultaneous meeting of these two requirements leads to several "embarrassing" situations. In a secure protocol, correctness and privacy are *blended* a deeper manner. In particular, privacy —a meaningful notion all by itself— is taken to mean that the protocol admits a certain type of *simulator*, and correctness is a concept which we define *through the same simulator* proving the protocol private. This merging of privacy and correctness avoids calling secure protocols those which clearly are not, and is a main contribution of this paper, as well as one of our first achievements in the course of this research. We are pleased to see that our idea has been adopted by other researchers in the area.

ADVERSARIAL AWARENESS. In the ideal evaluation of f, not only is there a notion of what inputs the adversary has substituted for the original ones of the corrupted players, but also that she is *aware* of what these substituted inputs are! Similarly, she is *aware* of the outputs handed by the trusted party to the corrupted players. We realize that this is a crucial aspect of ideal evaluation, and thus one that should be preserved as closely as possible by secure evaluation. Indeed, *adversarial awareness* is an essential ingredient in obtaining the crucial *reducibility* property discussed below.

TIGHT MIMICRY. Our definition of a secure protocol mimics the ideal evaluation of a function f in a very "tight" manner. (For those familiar with the earlier definition of zero knowledge, the definition of security relies less on "global" properties of ensembles of executions and more on *individual* executions.) In each particular run of a secure protocol for evaluating f, one may "put a finger on" *what inputs* the adversary has effectively substituted for the original ones of the corrupted players; *when* in the computation this has happened; *what* the adversary and the players get back from the joint computation, and *when* this happens; and these values are guaranteed (almost certainly) to be exactly what they should be — based on f, the inputs the adversary has effectively substituted, and the inputs the good players originally had.

Key Properties

MODEL INDEPENDENCE. As we said a secure protocol is one that "properly" replaces the trusted external party of ideal evaluation with exchanging messages. There are, however, several possibilities for exchanging messages. For instance, each pair of participants may be linked by a secure communication channel (i.e., the adversary cannot hear messages exchanged by uncorrupted people); alternatively, each pair of players may have a dedicated, but insecure, communication channel; else, the only possible communication may consist of broadcasting messages, and so on. Though for ease of presentation we develop our notion of security with respect to a *particular*, underlying communication model, our notion of security is essentially *independent* of the underlying communication model. Indeed, we prove that the existence of secure protocol in one model of communication entails their existence in all other "reasonable" models. This proof is highly constructive: we show that, for any two rich-enough communication models, there exists a "compiler" that, given a protocol secure in the first model, generates a protocol for the same task which is secure in the second. (Interestingly, the proof, though often considered folklore, turned out to be quite difficult!)

SYNTACTIC INDEPENDENCE. Our definition of security is independent of the "syntax" in which a protocol is written. Designing a secure protocol is easier if one adopts a syntactic structure à la [GMW87]; that is, having the parties first execute a "committal phase" in which they pin down their inputs while keeping them still secret[3] and then they compute on these committed inputs. A different type of syntactic help consists of assuming that a primitive for securely computing some simple function is given, and then reducing to it the secure computation of more complex functions. This also simplifies the design of secure computation protocols (and actually presupposes that secure protocols enjoy the reducibility property we discuss below.) While it is alright to use a right syntax to lessen the difficulty of designing secure protocols, we insisted that our definition of security be independent of any specific syntactic restriction for a protocol to be called secure. (Of course, though we insist at remaining at an intuitive level here, being secure is itself an enormous restriction for a protocol, but of a different nature.)

INPUT INDEPENDENCE. There is yet a third, and crucial, type of independence property. In the ideal evaluation of a function f, the fake values the adversary chooses are completely *independent* of the values held by *uncorrupted* players. Of course, a secure protocol should have this property too, as closely as is possible. (It is a rather subtle matter how to state this goal satisfactorily.) After the proper definitions are in place, we show that our definition of security captures independence in an *extremely strong sense*—not by "adding it in" as a desired goal, but, rather, by it being a *provable property* of any secure protocol.

REDUCIBILITY. Reducibility has always been a key desideratum. Let us describe this goal. Suppose you have designed a secure protocol for some complicated task—computing some function f, say. In an effort to make more manageable your job as protocol designer, you assumed in designing the protocol that you had some *primitive*, g, in hand. You proved that your protocol P^g for computing f was secure in a "special" model of computation — one in which an "ideal" evaluation of the primitive g was provided "for free." One would want that you obtain a secure protocol for f by inserting the code of a secure protocol for g wherever it is necessary in P^g that g be computed. This key goal for any "good" definition of security is surprisingly difficult to obtain, *particularly* if one adopts the most natural and innocent-looking notion of adversarial awareness (namely, the result of applying a fixed algorithm to the adversary's entire computational state). Our notion of

[3]This secret commitment is called verifiable secret sharing, a notion introduced by [CGMA85]. For a precise definition and worked out example see [FM90]

security ensures reducibility; in fact, this has been one of the main driving forces in shaping our definitions.

1.7 How to Read This Paper

GENERAL ORGANIZATION. The paper is divided up into five chapters. After this (1) introduction, one is (2) a chapter describing protocols, then (3) one describing secure protocols, (4) a chapter on complexity issues and the computational security, and, finally, (5) a chapter on properties of secure protocols.

This release is a preliminary report. Though chapters (1)–(3) are getting close to their final form, chapters (4) and (5) are rather incomplete and contain historical baggage in their notation, outlook, and proofs.

SIMPLIFYING ASSUMPTIONS. In order to simplify the presentation of our notion of security, we initially makes a number of simplifying assumptions. In particular, as our development begins we initially assume the following:

1. First, that the players and our adversaries have no past and no destiny but to try to compute some given function. Thus each party brings into the collaborative computation of a function f *only* his input x_i, and the adversary brings into the collaborative computation no special information whatsoever. This disregard for players' and the adversary's *history* is rectified in Section 3.5.

2. We assume that there are only a *fixed* number of parties participating in the protocol. This restriction is relaxed in Section 3.6.1.

3. We assume that the goal of these players is to collaboratively compute some particular *finite function.* A finite function is a map from strings of some fixed length ℓ to strings of some fixed length, l. This restriction is relaxed in Section 3.6.2.

4. We assume that a secure protocol for evaluating a function takes some *particular* number of rounds, R. This number may be thought of as a constant, or, more generally, as a polynomial-time computable function of the "common input." A protocol so restricted is called a *fixed-round* protocol. In Section 3.6.3 we relax the restriction that our protocols are fixed-round protocols.

5. We initially take our goal to be an *information-theoretic standard of security.* What is more, hough there are complexity-issues which should be dealt with *even for information-theoretic security,* we ignore them. Complexity-theoretic extensions and modifications to our definitions are described in Chapter 4.

Because of some of these, the notion of security initially developed (Definition 3.10) fails to have some of the properties one would expect of a secure protocol, and it fails to illustrate the full power of our adversaries. Only after accommodating the first three of these extensions can we maintain that a reasonable notion has been achieved (Definition 3.15). Nonetheless, the "stripped-down" notion is meaningful, and describing it first and then specifying how the definition is modified highlights the importance of a variety of issues.

In this paper we aim at achieving *solely* what we believe should be the absolute minimum notion of security. This should encompass "input independence" and "reducibility." Indeed these are properties both *desirable* and *achieved* by our definition, as we show in Chapter 5. Protocols

have many important angles to them. Among these are "economicity of resources" and "fairness," whose investigation is already under way. We do not deal with these and other more sophisticated notions in our paper. To avoid that the field would outpace its foundations, we believe that a firmer understanding of basic security should come first.

Those who believe that what we consider an absolute minimum is actually "too much" (e.g., those who think that security should be defined without guaranteeing reducibility, say) may find in section 3.7 more general notions outlined.

2 Protocols and Adversaries

In this section we describe the execution of *protocols*—programs which are run by a collection of communicating players. Specifying how a protocol runs involves describing not only what happens when all the players perform flawlessly, but in describing, too, the manner in which they may *fail* to perform their specified instructions, and what effect this has on the execution.

First we describe how a fault-less network operates. Afterwards, we consider the possibility of some players becoming "bad"—that is, of deviating from their prescribed instructions. Several questions immediately arise concerning the capabilities of these bad players.

ADVERSARIES. First among them is: *how powerful* should we let these bad players be? In some scenarios the only natural way for a processor to deviate from a protocol is by ceasing all communications, such as in the case of a computer "crash." Alternatively, processors may start sending messages "at random," corresponding, perhaps, to having some short-circuited register. If people are behind their processors, it is safer to consider more "malicious" deviations. This possibility, clearly subsuming the previous ones, is the one we focus on. Our goal is in fact reaching the strongest, natural notion of security, so that a protocol satisfying our definitions may be *safely* and *easily* used in any natural context. We thus allow bad players to deviate from their prescribed instructions in any way—the only constraint being that the number of bad players may be limited to a certain fraction of the total. (We will later consider the case in which bad players are limited to being able to perform "reasonable" computational tasks, as well.) We also allow bad players to secretly cooperate with each other. Actually, to guarantee their "perfect cooperation," we envisage a *single* agent, the *adversary,* who, during an execution of a protocol, may corrupt and control players.

An equally important question is: *when* can the adversary corrupt players? One possibility is to consider a *static* adversary, one who can choose and corrupt a subset of the players only at the start of a protocol. Since any *real* adversary may be expected to make an effort to corrupt those players whose corruption would be most beneficial to her, a better possibility is to consider a *dynamic* adversary, one capable of corrupting players at arbitrary points during the execution of a protocol, based on the information acquired from previous corruptions. Such an adversary is provably more powerful than a static one (see Section 3.6.5), and security with respect to a dynamic adversary is both harder to achieve and harder to properly define. The adversary we consider is dynamic.

COMMUNICATION MODEL. Assuming such a strong adversary makes our definition of security much more meaningful. On the other hand, given that secure protocols are non-trivial to *find*, we at least make them easier to *write* by providing a generous communication model. In essence, we allow each player both to privately talk to any other player (that is, to speak over a "private communication line"), and to talk "aloud," so that all players will hear what is said and know who has said it (that is, to speak over a dedicated communication line that is monitored by all the other players).

While our notions and theorems are stated in this chosen chosen communication model, we emphasize that, as with all important notions, secure computation is essentially *independent* of the particulars of the selected model. Indeed, for the notion of security we develop, essentially a syntactical replacement of adversaries and networks in our definition by respective, weaker versions of them yields the corresponding, right notion of security in the modified framework.

2.1 Protocols

2.1.1 Informal description

THE GENERAL PICTURE. *Protocols* are the instructions which *players* follow in order to accomplish some task. A collection of players connected by some communication mechanism is said to form a *network*. Each player of a network has a string, called his *computational state*, and the effect of the protocol is in updating these strings. At the beginning of a protocol this string should encode a player's *private input*.

Computational states are updated either as a result of local computation, or as a result of interacting with other players by exchanging messages. As we shall see in detail, the ordering of these events is tightly controlled. In fact, our protocols operate in *rounds*. A round consists of having all players perform their local computation and then sending messages to all other players. Once all these messages have been delivered to the proper players, the next round begins. This process repeats a fixed number of times.

Intuitively, local computation consists of applying a probabilistic function to a player's computational state. Still, we find it convenient to distinguish between probabilistic and deterministic aspects involved in local computation. Namely, a player computes locally in two ways. First, he can replace his own computational state by its image under some (deterministic) function; in fact, a protocol is regarded as being *precisely* this function. Also, a player may append to his computational state a uniformly distributed random bit—a process called "flipping a coin." This models that fact that a computing device with suitable hardware can obtain "random" bits. Such random bits are crucial for achieving security in our sense. Which of these two activities a player wishes to perform is indicated in his computational state, according to some fixed conventions. Alternatively, the computational state may indicate that the player's local computation is over for the current round, and all his messages are ready to be sent out.

Interaction with other players occurs once all the players finish their local computation for the current round. At this time, each player has a computational state which determines his *broadcast message*, which is a string to be delivered to everyone, as well as a set of *private messages*, each of which is to be delivered to a specified recipient. As we shall specify shortly, these messages will be delivered by appending to each player's state the various messages he is entitled to, making it clear what was received from whom.

ADDITIONAL CHOICES. While notions such as rounds are central to the idea of a synchronous protocol and would likely be reflected in any formalization of the execution of a protocol, various other choices in our formalization reflect a more individual taste. For instance, while protocols, players, and networks are all ideas with strong intuitive meaning, we *only formalize protocols*. Players and networks are thought of as "virtual" objects, not directly represented in the formalism and primarily useful for the language of discourse they provide. Intuitively, any "computing device" can play the role of a player, and any "communication mechanism" can play the role of the network. But a protocol is mathematical object, natural to define and independent of any particular computing device or communication mechanism designed to execute it.

Another personal choice is our formalization of a protocol as a *single program*. Conceptually, each player i runs his *own* program, P_i, directing the evolution in time of his computational state. However, we consider each player as running the same program, P. Such a program is made specific to the player who runs it simply by specifying a player's identity in his initial computational state. This makes a protocol a finite-size object instructing each player how to behave, however many of them there are. Clearly this viewpoint entails no loss of generality when the number of players is fixed—you could always imagine that a protocol P begins by saying, "if you are player i,

execute P_i"—and the viewpoint ensures, when the number of players is not regarded as fixed, that a protocol always has a finite description.

To describe protocols more formally, we have drawn a distinction between the computation which a protocol P directs, and the other, "syntactic" components needed to compactly specify how a protocol runs. These *syntactic functions* direct whether it is time to apply the function P, or time to flip a coin, or time to end the round's activities; and they specify what messages a player means to broadcast and which ones he intends to deliver privately to whom.

To facilitate discussing complexity issues later, a protocol P is regarded as a function not only of a player's computational state, but also of a separate string called the *common input*, which is known by everyone.

2.1.2 Formal description

NOTATION. An ***alphabet*** is a finite set. We will only use the alphabet $\{0,1\}$. A *string* is a finite sequence of characters from some alphabet, and an ***infinite string*** is an infinite sequence of characters over some alphabet. If x and y are strings, xy denotes their concatenation. A *language* is a set of strings.

We define $\{0,1\}^n$ as the language of strings of length n, and $\{0,1\}^* = \bigcup_n \{0,1\}^n$ as the set of all strings. We let $\{0,1\}^\infty$ denote the set of infinite binary strings. The empty string (i.e., the length-0 sequence of characters) is denoted by Λ. If we write $x \in \{0,1\}^*$ where x is apparently *not* composed of characters of the alphabet $\{0,1\}$ (e.g., $x = 11*0$), then it is understood that the string x is encoded over $\{0,1\}$ in some natural manner.

The set of nonnegative integers is denoted $\mathbf{N} = \{0,1,2,\ldots\}$. If a and b are integers, $a \leq b$, we let $[a..b]$ denote the set of integers between a and b, inclusive. By $[a..\infty]$ we denote the set of all integers greater than or equal to a, together with a point "∞".

DEFINITION OF A PROTOCOL. We reiterate that we are first describing a protocol for a fixed number of players; later we discuss protocols for arbitrary numbers of players. We also note that the phrases beneath the underbraces should be considered as suggestive terminology, not part of any formal definition.

Definition 2.1 *An R-round, n-party protocol is a Turing-computable function*

$$P\colon \underbrace{\{0,1\}^*}_{\text{common input}} \times \underbrace{\{0,1\}^*}_{\text{current state}} \to \underbrace{\{0,1\}^*}_{\text{new state}}.$$

A protocol syntactic function is any of the following Turing-computable functions:

- a next-action *function*, $\mathcal{N}$: $\underbrace{\{0,1\}^*}_{\text{current state}} \to \underbrace{\{\text{compute}, \text{flip-coin}, \text{round-done}\}}_{\text{action to take}}$
- a broadcast-message *function*, M: $\underbrace{\{0,1\}^*}_{\text{current state}} \to \underbrace{\{0,1\}^*}_{\text{broadcast message}}$
- a private-message *function*, m: $\underbrace{\{0,1\}^*}_{\text{current state}} \times \underbrace{[1..n]}_{\text{identity of recipient}} \to \underbrace{\{0,1\}^*}_{\text{private message}}$

NOTATION. In place of $m(s_i, j)$ we write $m_j(s_i)$.

2.1.3 Remarks

RECOVERABILITY OF COIN TOSSES. As we have said, when a player flips a coin the outcome is appended to his computational state. This process can be repeated as many times as it is deemed sufficient. Then the protocol function P is applied to this augmented computational state, a new state results, and the coin flips may or may not be recoverable it. Some protocols "record" their coin tosses in the new state, others do not (though some information about these coin tosses may still be inferable from the new computational state).

THE SYNTACTIC FUNCTIONS. These specify how a protocol P directs the updating of player's computational state, effecting the computational states of other players in the process. As such, we might have considered a protocol to be the five-tuple $(P, \mathcal{N}, M, m)$ consisting of the protocol algorithm proper and the collection of syntactic functions. However, we choose instead to take the syntactic functions to be *fixed* functions, good for any protocol. This way, for example, protocols can more easily be composed with one another: there is no danger that two protocols employ different conventions on how processors communicate, say.

Though particular maps, we have not actually specified the syntactic functions since the conventions chosen for defining them are not important. Each function should specify its range value in a natural and simple manner from its domain value. For example, with "⟨", "⟩", "(", ")", "∗", and "," all being formal symbols, we might say that if computational state s_i does not contain the symbol "∗" but it does contain one and only one occurrence of a substring $\langle 1^j \rangle$ then $\mathcal{N}(s_i) =$ **flip-coin** if $j = 2$, $\mathcal{N}(s_i) =$ **round-done** if $j = 3$, $\mathcal{N}(s_i) =$ **protocol-done** if $j = 4$; in all other cases, $\mathcal{N}(s_i) =$ **compute**; if s_i contains one and only one occurrence of a substring (μ), then $M(s_i) = \mu$; otherwise, $M(s_i) = \Lambda$; if s_i contains one and only one occurrence of a substring $(1^j, \mu_j)$, for $j \in [1..n]$, then $m_j(s_i) = \mu_j$; otherwise; $m_j(s_i) = \Lambda$.

2.2 Executing Protocols in the Absence of an Adversary

2.2.1 Informal description

The syntactic goal of executing a protocol is transforming the players' initial computational states to a set of final computational states. Besides the common input and the initial computational states, this transformation depends on the particular coin flips made by each player. This naturally leads to the notion of a *player configuration*, which records both a player's computational state and his "future coin tosses." That is, each player, at the beginning of the protocol, has associated to him an infinite string of bits, each selected with uniform probability. When the player flips a coin, he reads and strips off the first bit in this sequence.

In absence of an adversary, the common input and the players' initial configurations completely determine the execution of a protocol. In the formalism below, we recursively express how these configurations progress from round to round. Here is the general idea.

Each player i begins round r in some configuration $\bar{C}_i^r$, and ends with some configuration C_i^r. This transformation is not done in one step. Rather, by flipping coins and computing, the player goes through a sequence of *micro-rounds*, $0, 1, 2, \ldots$. (The micro-round ρ is written as a second superscript to the quantity of interest.) In some cases there may be an infinite sequences of micro-rounds without completing the round (as when, being very unlucky, a player tries to select one out of three possibilities with uniform probability). Normally, however, there will be a finite number ϱ_{ir} of micro-rounds before the next-action function evaluates to "round-done." Accordingly, player i will begin round r in some configuration C_i^{r0} $(= \bar{C}_i^r)$, and end it in some configuration $C_i^{r\varrho_{ir}}$ $(= C_i^r)$. For every player i, $C_i^{(r+1)0}$ is determined from all of the configurations $\{C_j^{r\varrho_{jr}}\}$ by evaluating the

broadcast-message function and private-message function on them, and appending to each $C_i^{r\varrho_{ir}}$ its proper message subset.

2.2.2 Formal description

NOTATION. When a symbol denotes a string or an infinite string, use of the same symbol with a subscript denotes the indicated character. This convention holds even if the symbol denoting the string already bears a subscript. For example, if r_i is a string or an infinite string, then r_{i1} is the first character of r_i.

Two other notes. Symbols such as $\{\#, \natural, *, \bullet, \blacksquare\}$, which appear here or elsewhere, are all just formal punctuation symbols. Also, the character r, below, is somewhat overworked: with a subscript i, it indicates player i's random string; as a superscript, it indicates the round number. This should cause no confusion.

PLAYER CONFIGURATIONS. These should capture enough information to allow a convenient recursive formalization of the execution of a protocol, as player configurations evolve from microround-to-microround and round-to-round.

Definition 2.2 *A* **player configuration** $(s_i,\ r_i)$ *is an element of* $\underbrace{\{0,1\}^*}_{\substack{\text{player's}\\ \text{state}}} \times \underbrace{\{0,1\}^\infty}_{\substack{\text{future}\\ \text{coins}}}$.

CONFIGURATION SEQUENCES. An R-round, n-party protocol P generates from a *common input*, c, and n *initial player configurations*, $(C_1^{00}, \ldots, C_n^{00})$, a sequence of configurations $\{C_i^{r\rho}\colon\ i \in [1..n], r \in [0..R], \rho \in \mathbf{N}\}$, which we now describe. Sometimes, one or more of these configurations may fail to be defined by the recurrences below. If this happens, the protocol is said to have *diverged*.

Fix the notation

$$C_i^{r\rho} \ = \ (s_i^{r\rho},\ r_i^{r\rho})$$

for player configurations, and let ϱ_{ir} denote the least number such that $\mathcal{N}(s_i^{r\varrho_{ir}}) = \textsf{round-done}$ (if such a number exists). Define

$$\begin{aligned} M_i^r &= M(s_i^{r\varrho_{ir}}) \quad \text{and} \\ m_{ij}^r &= m_j(s_i^{r\varrho_{ir}}) \end{aligned}$$

for what processor i broadcasts in round r and sends to processor j in round r, respectively. Let

$$\mathcal{M}_i^r \ = \ M_1^r * \cdots * M_n^r * m_{1i}^r * \cdots * m_{ni}^r$$

be the ***actual messages*** (public and private) sent to processor i in round r. (These will not be received until the beginning of round $r+1$.) Define $C_i^{0\rho} = C_i^{10} = C_i^{00}$ for all $\rho \in [1..\infty)$ and $i \in [1..n]$. Then, for $r \geq 1$, the players' configurations progress as follows:

$$C_i^{r(\rho+1)} \ = \ \begin{cases} (P(c, s_i^{r\rho}),\ r_i^{r\rho}), & \text{if } \mathcal{N}(s_i^{r\rho}) = \textsf{compute} \\ (s_i^{r\rho} * r_{i1}^{r\rho},\ r_{i2}^{r\rho} r_{i3}^{r\rho} \cdots) & \text{if } \mathcal{N}(s_i^{r\rho}) = \textsf{flip-coin} \\ C_i^{r\rho} & \text{if } \mathcal{N}(s_i^{r\rho}) = \textsf{round-done} \end{cases}$$

$$C_i^{(r+1)0} \ = \ \begin{cases} (s_i^{r\varrho_{ir}} * \mathcal{M}_i^r,\ r_i^{r\varrho_{ir}}) & \text{if } r > 0 \\ C_i^{r\varrho_{ir}} & \text{if } r = 0 \end{cases}$$

EXECUTING PROTOCOLS. An *initial configuration of the network* is identified by a tuple $(\vec{x}, c, \vec{r})$ specifying the player's *private inputs* $x_1, \ldots, x_n \in \{0,1\}^*$, the *common input* $c \in \{0,1\}^*$, and the *players' coins* $r_1, \ldots, r_n \in \{0,1\}^\infty$. An initial configuration of the network determines an *execution* of the n-party protocol P. This execution is the sequence of configurations $\{C_i^{r\rho}\}$ generated by P when the initial configuration of party i is taken to be $C_i^{00} = (x_i \sharp i,\ r_i)$, player i's coins are r_i, and the common input is c.

The set of executions of P with common input c, private inputs $x_1, \ldots, x_n$, and all possible coin sequences $r_1, \ldots, r_n$ enjoys a probability measure by endowing each execution with the measure induced by taking each bit of each r_i to be selected uniformly and independently. In the absence of an adversary, *executing* a protocol P with common input c and private inputs $x_1, \ldots, x_n$, means sampling according to this probability distribution.

ADMISSIBLE PROTOCOLS. As we have said, a protocol may diverge (when a player repeatedly flips coins and applies the protocol function P without ever ending a round). We are not interested in protocols for which this happens.[4] A protocol is called *admissible* if, for any common input and any initial player configuration, it does not diverge. *From now on, all protocols are assumed to be admissible.* Forcing a protocol to flip at most a fixed number of coins per round (a number which depends only on the common input, say) is certainly a way to guarantee its admissibility.

PLAYER'S VIEW. The computational state of a player need only record what he requires to continue his, and therefore prior computational states might or might not be inferable from the current computational state. It is useful to introduce *view*, a notion formalizing this "history" of prior computational states.

In a protocol execution, the *round-r view* of player i is the string $h_i^r = (s_i^{r0}, s_i^{r1}, \ldots, s_i^{r\ell_{ir}})$. His view *through* round r is defined as $H_i^r = (c,\ h_i^0, h_i^1, \ldots, h_i^r)$, while the *view* of player i from an R-round protocol is H_i^R.

2.2.3 Remarks

MORE ON COIN TOSSES. Whether or not a protocol keeps explicit record of the coins tossed (see remark 2.1.3), it is still meaningful to speak about notions such as "the coins flipped by the first player during a particular round in an execution of a protocol."

SIMULATING ROUNDS. The organization of the computation in rounds, though purely conceptual, can be simulated in the real world without relying on perfect synchrony. For instance, in a computer network one may not be able to guarantee simultaneous delivery of messages nor access to a global clock. Nonetheless, it may be reasonable to assume that every message sent will arrive within five minutes of when it was sent, that the local computation of every processor can be completed within half an hour, and that each processor's clock knows the time of day within five minutes. In this case, the round structure can be simulated by instructing every processor to send its messages on the (local) hour, and start its local computation, using all the newly received messages, at a quarter past the hour.

As can be appreciated such a simulation, rounds are often an expensive resource. Since the time allocated for simulating each round should be sufficiently generous, so to be sure that all messages will be received and all local computation done, it is important to design protocols that use few rounds.

[4] Alternatively, we could say "we are not interested in protocols for which this happens with positive probability." This relaxation would slightly complicate the statement of some later definitions, but is otherwise immaterial.

THE VOID ROUND. We have established the convention that a protocol begins with the players executing a "dummy" round, round 0. During this round, "nothing happens." The presence of the void round facilitates bringing the adversary into the model of computation in a manner which allows for clean protocol composition.

2.3 Adversaries

2.3.1 Informal description

An adversary is a device capable of interacting with the participants of a network in a prescribed way. For convenience of discourse, we will adopt gender conventions: a player, "he," and the adversary, "she."[5]

The options of an adversary are greater than those of a player. Like the players, she can compute, flip coins, send and receive messages. Her additional feature is her ability to *corrupt* players in the execution of a protocol. Though we postpone, for a bit, the details of her interaction with the players, it is necessary to understand right off that once a player i is corrupted, it is the adversary who will receive messages sent to i and send messages on his behalf.

Our aim for now is to describe the adversary as a "gadget." Like the players, this gadget has associated to it a collection of syntactic functions. The adversary's next-action function $\tilde{\mathcal{N}}$ indicates what action she wishes to perform next; her broadcast-message function $\tilde{M}$ indicates what she will broadcast on behalf of corrupted players; and her private-message function $\tilde{m}$ indicates what message she will send privately to each player, on behalf of each corrupted processor.

2.3.2 Formal description

Definition 2.3 *An* adversary for an n-party protocol *is a function*

$$A\colon \underbrace{\{0,1\}^*}_{\substack{\text{common}\\ \text{input}}} \times \underbrace{\{0,1\}^*}_{\substack{\text{current}\\ \text{state}}} \to \underbrace{\{0,1\}^*}_{\substack{\text{new}\\ \text{state}}}.$$

An adversary syntactic function *is any of the following functions:*

- a next-action *function,* $\tilde{\mathcal{N}}$: $\underbrace{\{0,1\}^*}_{\substack{\text{current}\\ \text{state}}} \to \{\textsf{compute}, \textsf{flip-coin}, \textsf{corrupt}_1, \ldots, \textsf{corrupt}_n, \textsf{round-done}\}$
- a broadcast-message *function,* $\tilde{M}$: $\underbrace{\{0,1\}^*}_{\substack{\text{current}\\ \text{state}}} \times \underbrace{[1..n]}_{\substack{\text{corrupted}\\ \text{player}}} \to \underbrace{\{0,1\}^*}_{\substack{\text{broadcast}\\ \text{message}}}$
- a private-message *function,* $\tilde{m}$: $\underbrace{\{0,1\}^*}_{\substack{\text{current}\\ \text{state}}} \times \underbrace{[1..n]}_{\substack{\text{corrupted}\\ \text{player}}} \times \underbrace{[1..n]}_{\substack{\text{identity of}\\ \text{recipient}}} \to \underbrace{\{0,1\}^*}_{\substack{\text{private}\\ \text{message}}}$

NOTATION. In place of $\tilde{M}(s_A, i)$ and $\tilde{m}(s_A, i, j)$ we will write $\tilde{M}_i(s_A)$ and $\tilde{m}_{ij}(s_A)$, respectively.

[5] Justification for gender conventions judiciously relegated to an unpublished manuscript.

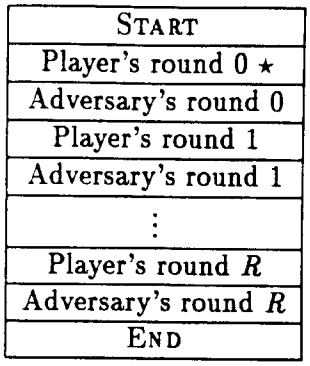

Figure 1: *An R-round execution of a protocol with the adversary. The player's 0^{th} round is a round in which there is not activity. Such rounds will be marked with a "$\star$".)*

2.3.3 Remarks

Again, we do not explicitly specify the syntactic functions since the particular conventions selected for them is irrelevant to further discussion. All that is important is that each function specifies its range value in a natural and simple manner from its domain value, as with the syntactic functions associated to the players of a network.

As with a protocol, the first component in A's domain is the common input. Though this could be considered as an unnecessary component—it could be encoded in the adversary's initial computational state—we make a separate argument of it to facilitate, later on, dealing with computational complexity.

Note that, while a protocol must at least be computable, no similar assumption is made of an adversary; an adversary which is, say, an arbitrary function, with no finite description, is a perfectly good adversary. Possible restrictions on the power of an adversary will be investigated in Chapter 5.

2.4 Executing Protocols in the Presence of an Adversary

2.4.1 Informal description

ALTERNATING ACTIVITY. Think of an adversary A as an abstract machine interacting with the participants of a network in a prescribed way. This way entails the players and the adversary alternating periods of activity, as suggested by Figure 1.

In the beginning, all of the players are *good*, and so they remain unless the adversary *corrupts* them. While the players compute, the adversary is quiescent. But once all of the still-good players have finished their local computation, they effectively go to sleep, and A is awakened. She is active for a while—computing locally and possibly corrupting additional players—until she has finished, for the time being, and goes to sleep. Then the players execute again. The players and the adversary continue alternating periods of execution in this way, until the players complete their last round, R. At that point, the adversary is given one last round of activity, and then the protocol execution is said to be over.

AN ADVERSARY ROUND. When an adversary begins a round of her activity, she is given certain information. Namely, she is handed the messages that still-good players have just broadcasted, plus the messages the still-good players just transmitted to already-corrupted players.

After some local computation, the adversary may choose to corrupt some players. This she can do in a *dynamic* way. When corrupting a player i in round r, the adversary learns some

information about him, his *exposed state*, σ_i^r, which will be described momentarily. Then, based on this information, still within the same round, the adversary can corrupt, one at a time and exactly as before, additional players, until she does not want to corrupt any more of them. At this point, the adversary composes all out-going messages from the bad players to the good, and it is these messages (together with the messages sent by good players) which will be delivered to the players in the next round.

In our formalization, an adversary round is divided into a sequence of micro-rounds, just as player rounds were. As her micro-rounds progress, the adversary computes, flips coins, and corrupts players. Though we find it convenient to adopt a formalism in which the players execute first, this is a round in which there is no activity—so really it is the adversary who is both the first and the last agent to be active in a protocol execution.

EXPOSED STATE. As we said, the exposed state is whatever information the adversary learns when corrupting a player. If a player i is corrupted in adversary round r, his exposed state will always consist of all of the messages addressed to him in round r (which would have been delivered to him at the beginning of players' round $r+1$), together with some information, $E(H_i^r)$, about his current view. Since we are now aiming at defining secure computation with respect to the most powerful natural adversary, through out this chapter we let the *exposed state function* E be the identity. That is, by corrupting a player, the adversary gets, over the most recent messages addressed to him, the view of his execution of the protocol up to that point (thus, all prior messages, coin tosses, computational states etc.)

The exposed-state function, as we shall see in the next discussion section, is essential in pinning down the underlying model of computation and communication, and different exposed state functions will be considered in later chapters.

2.4.2 Formal description

We now describe how an n-party protocol P executes in the presence of an adversary A.

ADVERSARY CONFIGURATIONS. These should capture enough information to allow a convenient recursive formulation of how a protocol runs in the presence of an adversary, as player and adversary configurations progress from round to round and micro-round to micro-round.

Definition 2.4 *An* **adversary configuration** (s_A, r_A, κ_A) *is an element of*

$$\underbrace{\{0,1\}^*}_{\substack{\text{adversary's}\\ \text{state}}} \times \underbrace{\{0,1\}^\infty}_{\substack{\text{future}\\ \text{coins}}} \times \underbrace{2^{[1..n]}}_{\substack{\text{corrupted}\\ \text{players}}} .$$

CONFIGURATION SEQUENCES. Let A be an adversary for an n-party protocol, and let P be such a protocol. We describe how, from a common input c, any n initial player configurations, $(C_1^{00}, \ldots, C_n^{00})$, any initial adversary configuration, $C_A^{00} = (s_A, r_A, \kappa_A)$, and any exposed-state function E: $\{0,1\}^* \to \{0,1\}^*$, protocol P and adversary A generate a sequences of player configurations, $\{C_i^{r\rho} : i \in [1..n], r \in [0..R], \rho \in \mathbf{N}\}$, and a sequence of adversary configurations, $\{C_A^{r\rho} : r \in [0..R], \rho \in \mathbf{N}\}$. Sometimes, one or more of these configurations may fail to be defined by the recurrences below. If this happens, the protocol is said to have *diverged.*

Fix the notation

$$\begin{aligned} C_i^{r\rho} &= (s_i^{r\rho}, r_i^{r\rho}) \quad \text{and} \\ C_A^{r\rho} &= (s_A^{r\rho}, r_A^{r\rho}, \kappa_A^{r\rho}) \end{aligned}$$

for player and adversary configurations.

Let ϱ_{ir} denote the least number such that $\mathcal{N}(s_i^{\varrho_{ir}}) = \textsf{round-done}$ (if such a number exists), and let ϱ_{Ar} denote the least number such that $\tilde{\mathcal{N}}(s_A^{\varrho_{Ar}}) = \textsf{round-done}$ (if such a number exists). For clarity, we sometimes *omit* designating the micro-round to indicate the *final* micro-round of the indicated player or adversary round. For example, we may write s_i^r, C_i^r, s_A^r, κ_A^r, and C_A^r, in place of $s_i^{r\varrho_{ir}}$, $C_i^{r\varrho_{ir}}$, $s_A^{r\varrho_{Ar}}$, $\kappa_A^{r\varrho_{Ar}}$, and $C_A^{r\varrho_{Ar}}$, respectively.

As before, define $C_i^{0\rho} = C_i^{10} = C_i^{00}$ for $\rho \in [1..\infty)$ and $i \in [1..n]$. The players' configurations progress as before,

$$C_i^{r(\rho+1)} = \begin{cases} (P(c, s_i^{r\rho}),\ r_i^{r\rho}), & \text{if } \mathcal{N}(s_i^{r\rho}) = \textsf{compute} \\ (s_i^{r\rho} * r_{i1}^{r\rho},\ r_{i2}^{r\rho} r_{i3}^{r\rho} \cdots) & \text{if } \mathcal{N}(s_i^{r\rho}) = \textsf{flip-coin} \\ C_i^{r\rho} & \text{if } \mathcal{N}(s_i^{r\rho}) = \textsf{round-done} \end{cases}$$

$$C_i^{(r+1)0} = \begin{cases} (s_i^{r\varrho_{ir}} * \mathcal{M}_i^r,\ r_i^{r\varrho_{ir}}) & \text{if } r > 0 \\ C_i^{r\varrho_{ir}} & \text{if } r = 0 \end{cases}$$

the only difference being that the set of *actual messages* delivered to player i from round r, $\mathcal{M}_i^r$, will be defined differently. The adversary's sequence of configurations progresses as follows:

$$C_A^{r(\rho+1)} = \begin{cases} (A(c, s_A^{r\rho}),\ r_A^{r\rho},\ \kappa_A^{r\rho}) & \text{if } \tilde{\mathcal{N}}(s_A^{r\rho}) = \textsf{compute} \\ (s_A^{r\rho} * r_{A1}^{r\rho},\ r_{A2}^{r\rho} r_{A3}^{r\rho} \cdots,\ \kappa_A^{r\rho}) & \text{if } \tilde{\mathcal{N}}(s_A^{r\rho}) = \textsf{flip-coin} \\ (s_A^{r\rho} * \sigma_i^r,\ r_A^{r\rho},\ \kappa_A^{r\rho} \cup \{i\}) & \text{if } \tilde{\mathcal{N}}(s_A^{r\rho}) = \textsf{corrupt}_i \\ C_A^{r\rho} & \text{if } \tilde{\mathcal{N}}(s_A^{r\rho}) = \textsf{round-done} \end{cases}$$

$$C_A^{(r+1)0} = (s_A^{r\varrho_{Ar}} * \dot{\mathcal{M}}_A^{r+1},\ r_A^{r\varrho_{Ar}},\ \kappa_A^{r\varrho_{Ar}})$$

where σ_i^r is the *exposed state* of processor i in round r, and $\dot{\mathcal{M}}_A^{r+1}$ are the *intended messages* sent by good players and intercepted by the adversary at the start of her round $r+1$. These strings, and $\mathcal{M}_i^r$, are defined below.

$$\mathcal{M}_i^r = M_1^r * \cdots * M_n^r * m_{1i}^r * \cdots * m_{ni}^r \qquad \textit{"Actual round-r messages visible to i"}$$

where

$$M_i^r = \begin{cases} M(s_i^r) & \text{if } i \notin \kappa_A^r \\ \tilde{M}_i(s_A^r) & \text{if } i \in \kappa_A^r \end{cases} \qquad \textit{"Actual round-r broadcast from i"}$$

$$m_{ij}^r = \begin{cases} m_j(s_i^r) & \text{if } i \notin \kappa_A^r \\ \tilde{m}_{ij}(s_A^r) & \text{if } i \in \kappa_A^r \end{cases} \qquad \textit{"Actual round-r message from i to j"}$$

The intended messages prepared by good players during round r are given by

$$\bar{M}_i^r = \begin{cases} M(s_i^r) & \text{if } i \notin \kappa_A^{r-1} \\ \Lambda & \text{otherwise} \end{cases} \qquad \textit{"Intended round-r broadcast prepared by good player i"}$$

$$\bar{m}_{ij}^r = \begin{cases} m_j(s_i^r) & \text{if } i \notin \kappa_A^{r-1} \\ \Lambda & \text{otherwise} \end{cases} \qquad \textit{"Intended round-r message for j prepared by good player i"}$$

(Messages other than $\bar{M}_i^r$ and $\bar{m}_{ij}^r$ may be delivered to processor j in round $r+1$ because of the action of the adversary.) The exposed state of processor i which A will get hold of if she corrupts him in round r is given by

$$\sigma_i^r = \begin{cases} s_i^r * x_i & \text{if } r = 0 \\ s_i^r * x_i * \bar{m}_{1i}^r * \cdots * \bar{m}_{ni}^r * E(H_i^r) & \text{if } r > 0 \end{cases} \qquad \textit{"The exposed state of processor } i \textit{ at round } r\textit{."}$$

for some fixed function E. (The definition of H_i^r is unchanged: $H_i^r = (c, h_i^0, h_i^1, \ldots, h_i^r)$, where $h_i^r = (s_i^{r0}, s_i^{r1}, \ldots, s_i^{r\varrho_r})$.) Finally, the incoming messages to the adversary at the beginning of her round r are given by

$$\dot{\mathcal{M}}_A^r = \bar{M}_1^r * \cdots * \bar{M}_n^r * \dot{m}_{11}^r * \cdots * \dot{m}_{1n}^r * \cdots * \dot{m}_{n1}^r * \cdots * \dot{m}_{nn}^r \qquad \textit{"Incoming messages to the adversary at the start of her round } r\textit{"}$$

where

$$\dot{m}_{ij}^r = \begin{cases} \bar{m}_{ij}^r & \text{if } j \in \kappa_A^{r-1} \\ \Lambda & \text{otherwise} \end{cases}$$

We add the stipulation to the above definitions that $M_i^r = m_{ij}^r = \bar{M}_i^r = \bar{m}_{ij}^r = \Lambda$ for $r = 0$.

EXECUTING PROTOCOLS. An *initial configuration of the network* is identified by a tuple $(\vec{x}, c, \vec{r}, r_A)$ specifying the players' *private inputs* $x_1, \ldots, x_n \in \{0,1\}^*$, the *common input* $c \in \{0,1\}^*$, the *players' coins* $r_1, \ldots, r_n \in \{0,1\}^\infty$, and the *adversary's coins* $r_A \in \{0,1\}^\infty$. For a given protocol P and adversary A, an initial configuration of the network determines an *execution* of P in the presence of A: this is the sequence of configurations $\{C_i^{r\varrho}, C_A^{r\varrho}\}$ generated by P and A when the initial configuration of party i is taken to be $C_i^{00} = (x_i \sharp i, r_i)$, the initial configuration of A is $(\kappa_A, r_A, \kappa_A)$, for $\kappa_A = \emptyset$, player i's coins are r_i, the adversary's coins are r_A, and the common input is c.

The set of executions of P with common input c, private inputs $x_1, \ldots, x_n$, and all possible coin sequences $r_1, \ldots, r_n$ and r_A becomes a probability space by endowing each execution with the measure induced by taking each bit of r_A and each bit of each r_i to be selected uniformly and independently. In the presence of an adversary, *executing* a protocol P with common input c and private inputs $x_1, \ldots, x_n$ means sampling from this probability space.

ADMISSIBLE ADVERSARIES. As we have said, all our protocols are admissible; thus no adversary can force a good player to compute forever by sending him messages in some tricky way. On the other hand, the adversary can gain nothing if she herself computes forever—except for messing up our recursions! We say that an adversary *diverges* if she goes through an infinite sequence of computational states, her next-action function never indicating round-done. An adversary A is *admissible* if for any common input, any initial adversary state, any set of messages received, and any set of exposed states learned, adversary A does not diverge. *From now on, all adversaries are assumed to be admissible.* Thus, when a protocol interacts with an adversary, it does not diverge.

ADVERSARIES WITH BOUNDED CHARISMA. If an adversary corrupts all the players, then nothing can be said about their behavior in her presence. We thus favor less captivating adversaries.

Definition 2.5 *Let $t \in \mathbf{N}$ and let A be an adversary for an R-round protocol. We say that A is a* t-adversary *if A corrupts at most t players: that is, $|\kappa_A^R| \leq t$ in any execution of P with A.*

For our purposes, it is equally acceptable to strengthen the constraint above and demand that a t-adversary always corrupt *exactly* t players.

2.4.3 Remarks

2.5 Discussion

THE EXPOSED STATE. As we said, the exposed state represents all the adversary learns when she corrupts a player. What should this be? Certainly the adversary is entitled to the player's computational state in that instant; indeed, a player must keep this information around for continuing executing the protocol. Also, she is entitled to learn the player's private input; indeed, we aim at precisely capturing the situation of an ideal function evaluation, and she gets this information in that scenario. Additionally, we let the adversary learn all the messages addressed to a player which were sent in the same round in which she corrupted him; indeed, when rounds are simulated in an asynchronous communication model, the adversary might be lucky and get these messages early enough to use this information in making her decisions in the current simulated round. Thus, seeking a strong enough model to capture any "real" underlying network, we assume the adversary is always lucky.

Besides these three components, we let the exposed state, σ_i^r, also contain a piece, $E(H_i^r)$, of player i's view, H_i^r, at the end of round r. Since this specified portion of the player's view may not be recoverable from his computational state, why give it to the adversary? As with "prompt" message delivery to the adversary, our reason stems from modeling reality robustly. To avoid being unduly specific, we have rightly chosen a formalism that is silent about the nature of players and adversaries. In reality, though, an adversary may be able to exploit the peculiarities of the agents that perform local computation. For instance, if players are people, it may not be so easy to instruct them to forget their past and resist interrogation! Alternatively, the way in which coins are flipped may allow the adversary to recover all past coin tosses once a player is corrupted. In no case, however, can the adversary learn more about a player, by corrupting him, than his view. The exposed-state function E thus constitutes a "knob" to control precisely what additional information can be derived by an "actual" adversary corrupting an "actual" player.

CHOOSING THE EXPOSED STATE. Though our notion of secure computation is applicable for all exposed-state functions E, we suggest that E, at the minimum, divulge the current-round view of a corrupted player. (This can be formalized by saying that $E(H_i^r) = h_i^r * E'(H_i^r)$ for some function E'.) We define the *standard model* to be the one in which E gives *exactly* the current-round view, $E(H_i^r) = h_i^r$. We have two excellent reasons for favoring the standard model.

- *Independence from asynchrony models.* Achieving security in the standard model ensures in an elegant way that the adversary cannot exploit any asynchrony present in the underlying real communication system. In fact, the standard model eliminates the burden of formalizing and choosing among infinitely many and often incomparable models for an asynchronous adversary.[6] It is because we let the adversary receive the current-round view of corrupted players that we can neatly alternate player and adversary rounds.

[6] Without receiving the current-round view of a corrupted player, it may be beneficial for the adversary to corrupt a player in the *middle* of his round of computation, since computing, in practice, "decreases" information. For example, one may consider an adversary who can corrupt a chosen player i at chosen micro-round ρ. (This capability, which is subsumed by the standard model, is undesirable since it is not model independent; in fact, it depends in a crucial way on how much computation can be performed within a micro-round.) Another model allows the adversary to corrupt players only at the end of their rounds, like we do, still granting her private inputs, current computational states, and messages intended for corrupted players in the current round, but nothing else (i.e., our model with $E = \Lambda$). As this model is incomparable with the one just mentioned, which is better to adopt? "Below" the standard model lie myriads of specialized and not-too-interesting models.

- *Independence from communication mechanisms.* If public-key cryptography is possible, then secure computation in the standard model is achievable not only in the rich type of networks we describe, but even in ones where communication is only possible via broadcasting, or only possible via separate communication lines between every pair of players, even if these lines can be monitored by the adversary, as discussed in section XX. In fact, it can be achieved in any "reasonable" communication mechanism — a statement necessarily part theorem and part thesis. Here, by "reasonable" we essentially mean that the adversary cannot disrupt the communication between two good players, a property without which there is no secure computation.

For the reasons given above, and for concreteness, we assume the standard model, unless stated otherwise.

Another noteworthy exposed-state function is the identity map, $E(H_i^r) = H_i^r$, which defines the *complete-history model.* The importance of this model is not only due to the fact that it gives rise to the worst adversarial scenario, but also to the fact that security with respect to it is achievable under at least one communication mechanism —in fact, as explained in section YY, the one adopted in this paper.

3 Secure Protocols

As we said in the introduction, we define a secure protocol for computing a function to be one that mimics an ideal evaluation, in all of its aspects, as closely as possible. Thus, in this section, we recall what an ideal evaluation is, establishing a convenient notation for the rest of our enterprise; we highlight its important aspects; we define the sense in which a protocol can imitate them; and, finally, we set just *how close* this imitation should be. This last point is both important and delicate. Demanding "perfect imitation" is both easier and counterproductive: for the notion of secure computation to be of any use, *it must allow the existence of secure protocols*, at least in some reasonable models of communication.

The notion of secure computation is inherently complex, even disregarding —as we do in this chapter— computational complexity issues and other, complicating issues. We thus decided to break its presentation in smaller and more manageable pieces of some independent interest. Accordingly, we provide separate definitions of both correctness and privacy, and then show how to properly interleave them to yield the notion of security. While we hope that this will improve the readability of our paper, we ask the reader to keep in mind that it is secure computation which is our real aim! Thus, while meaningful alternatives to our definitions of correctness (alone) and privacy (alone) can be found, we strongly believe that any satisfactory definition for what it means for a protocol, in the presence of a sufficiently powerful adversary, to closely mimic an ideal function evaluation, will be essentially similar to ours.

We do, though, encourage further work if felt necessary, and thus do explicitly highlight what we found to be the key ingredients and concerns for putting together any meaningful definition of secure computation.

3.1 Ideal Evaluation

VECTOR NOTATION. As usual, we denote a vector by a letter topped with an arrow symbol, "$\vec{\ }$". The same letter without the arrow but with a subscript denotes the indicated component. For example, if $\vec{x}$ is a vector, x_i is its i^{th} component. An n-vector is a vector with n components.

Fix n, and let $T \subseteq [1..n]$. Then we write $\overline{T}$ for $[1..n] - T$. If $\vec{x}$ is an n-vector and $T \subseteq [1..n]$, the *tagged vector* $x_T = \{(i, x_i)\colon i \in T\}$. (That is, x_T keeps track of the indices as well as the values.) To each two n-vectors $\vec{x}$ and $\vec{x}'$ and subset $T \subseteq [1..n]$, we associate the "shuffled" vector $x_{\overline{T}} \cup x'_T$, which is the n-vector $\vec{y}$ where $y_i = x_i$ if $i \in T$ and $y_i = x'_i$ otherwise. For $f\colon (\{0,1\}^*)^n \to (\{0,1\}^*)^n$ and $T \subseteq \{1, \ldots, n\}$, $f_T(\vec{x})$ is defined as $f_T(\vec{x}) = (f(\vec{x}))_T$, and $f_i(\vec{x})$ is defined as $f_i(\vec{x}) = (f(\vec{x}))_i$ —thus $f = (f_1, \ldots, f_n)$.

THE IDEAL SCENARIO. Let $\vec{x} \in (\{0,1\}^*)^n$ be a vector and $f : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$ be a function. Then the *ideal evaluation* of $f(\vec{x})$ in the presence of an adversary proceeds as follows:

(1) the adversary *dynamically* (one at a time) corrupts some set $T \subseteq [1..n]$ of the players;

(2) she then chooses some *fake* inputs, x'_T, for them;

(3) each uncorrupted player i now receives $f_i(x'_T \cup x_{\overline{T}})$ (from a "trusted party," one sometimes imagines), while the adversary receives $f_T(x'_T \cup x_{\overline{T}})$;

(4) finally, the adversary may dynamically corrupt some additional players. When she corrupts a player j, she learns $(x_j,\ f_j(x'_T \cup x_{\overline{T}}))$.

3.2 Correctness (without privacy)

So far, we have discussed executing a protocol in purely syntactic terms. In fact, we never even discussed the possibility of protocols having an output! Since the ultimate goal of our protocols is to securely evaluate a function, the time has come to add a bit of semantics to our setting.

In this section we put forward a definition of "correctness without privacy." That is, we provide a way to "rationalize" the execution of some protocols and distinguish those whose "net result" consists of correctly computing, in a *weak sense*, a function f.

The weakness of the proposed definition comes exactly from our wish to ignore "privacy" for now. In fact, we believe that any computation in the presence of an adversary, to be robust, needs also to be a private one. Since, however, we will define security as a strengthening of the following notion of ***weak computation***, all the quantities and concepts introduced here will be relevant to our future effort. Among these concepts is that of *awareness*, which will be particularly crucial.

3.2.1 Awareness and Correctness

There are, of course, several ways in which correctness without privacy may be intended. Since we aim here at making a step towards defining secure computation, the one we want to capture corresponds to an *ideal evaluation without privacy*; namely, an ideal evaluation in which the adversary knows, in advance, the initial inputs of all players.

First, without much trouble, we can enrich protocols with a new syntactic function: an *output* function, defined on the final computational state,

$$o: \underbrace{\{0,1\}^*}_{\substack{\text{round-}R\\ \text{state}}} \rightarrow \underbrace{\{0,1\}^*}_{\substack{\text{output}\\ \text{value}}}.$$

Armed of this small syntactical change, we now discuss a few definitions of "correctness without privacy." Fortunately, we have little use of formalization here, since all considered definitions will be rejected, except for the last one which will be greatly expanded in later subsections. To simplify things further, we shall here be content of considering solely the case in which, whenever n players —whose initial inputs are, respectively, $x_1, \ldots, x_n$— want to evaluate a function $f = (f_1, \ldots, f_n)$, the adversary always corrupts a fixed set T of them at the very beginning. This way $\overline{T}$ will represent the set of good players at any step of a protocol execution.

ELEMENTARY CORRECTNESS. A protocol whose only guarantee is that, upon termination, each good player i outputs f_i evaluated at "some" inputs, does not capture the desired an ideal evaluation without privacy. In the latter one, in fact, "these inputs" are the same for all players; moreover, the ones corresponding to good players coincide with the initial ones. We thus move on to consider the following

Informal Definition: *A protocol computes a function $f = (f_1, \ldots, f_n)$ with* **elementary correctness** *if, at the end of each execution, there exist values x'_T such that every good player i outputs $f_i(x'_T \cup x_{\overline{T}})$.*

With proper language changes, elementary correctness corresponds to the original correctness constraint of [Ya82a], and to that of many other researchers as well. At a first glance, it may appear to be exactly what we want. In fact, in the execution of any protocol, nothing prevents the adversary from corrupting a set of players T at the very beginning, substituting their initial inputs with x'_T, and then letting them follow the protocol scrupulously. If the protocol is assumed to compute a function in any meaningful sense, we should expect in this unavoidable circumstance that each

good player i ought then to output $f_i(x'_T \cup x_{\overline{T}})$. Since any reasonable definition should not demand what cannot be obtained, elementary correctness appears totally adequate in modeling an ideal evaluation without privacy.

Unfortunately, things are more subtle, and some other important aspect has not been properly captured.

AWARENESS. The definition of elementary correctness only requires that, in each execution, there exists some values x'_T such that the good players will each output his component of f evaluated at their initial inputs "shuffled in" with fake inputs x'_T. In an ideal evaluation without privacy, instead, not only the corresponding x'_T values exist, but the adversary is *aware* of them, since she actually hands them to the trusted party. This awareness aspect is an integral part of the ideal scenario, that is, one that we *demand* to be captured. Why such insistence? At first glance this awareness may appear something purely "psychological" and extraneous to a purely mathematical setting. Quite to the contrary, it is instead *full of technical consequences*, not the least of which being the much desired and needed *reducibility* property of secure computation. At this point of the paper, however, we are not yet in a position of of elevating the level of our discussion and justify this key choice of ours, and thus take a more formalistic approach:

> *Since the adversary is aware of which inputs she substitutes in an ideal evaluation, we demand that she be aware of them in a correct-without-privacy computation (and a secure computation), as well.*

Though there may be some latitude in the formalization of adversarial awareness, we strongly believe it to be a central concern for secure computation. We wish to describe our notion of security without initially entangling the reader in the subtleties arising from this formalization. Thus, for now, we adopt a simple but restrictive notion of awareness, leaving to subsequent sections both relaxing it and discussing it from a complexity point of view.

> *We say that the adversary is aware of some quantity associated to the execution of the protocol if that quantity can be computed by evaluating a specified function on the adversary traffic.*

The *adversary traffic* simply consists of all the messages she sends and receives via the corrupted processors, the exposed state of the processors she corrupts, and the common input. These are quantities immediately available to her, and thus it is easy for her, if she so wants, to evaluate a specified function on them, thus yielding a simple and natural notion of awareness. (Further illustration of the rationale behind this initial choice of awareness may be found in the next discussion subsection.)

To incorporate awareness, the definition of correctness without privacy will make use of an *adversary input* function, $\mathcal{AI}$. This function may at first be defined as mapping the adversary's final traffic to a "fake" input, x'_T, indicating the value which the adversary regards herself as having substituted into the collaborative computation.

At a second thought, rather than defining $\mathcal{AI}$ on the adversary's final view, we define it on any intermediate adversary view as well, and enlarge its range to contain the distinguishing value not-yet—indicating that the adversary's activity has not yet resulted in her entering any substituted values into the joint computation. This way, while executing a correct-without-privacy protocol, by evaluating $\mathcal{AI}$ on her growing traffic, the adversary can —if she so wants— be aware not only of which values she has substituted, but also *when* this substitution has taken place—something

which is certainly present in an ideal evaluation, with or without privacy. We call this *temporal awareness.*

To keep things meaningful, we require a certain "monotonicity" property from $\mathcal{AI}$: namely, if some intermediate adversary view takes on a value $x'_T \neq$ not-yet, it will keep on assuming the value x'_T on any subsequent view. (Else, the interpretation that the adversary knows when the input substitution occurred would fail to hold.) Moreover, when evaluated on the final adversary view, $\mathcal{AI}$ should take a value different from not-yet. (Else, the interpretation that the adversary does eventually enter some fake values into the ideal evaluation would fail to hold.)

Finally, the "fake" input the adversary substitutes is not the only thing she is aware of in the ideal evaluation. She is also aware of the private *outputs* of the corrupted players. Thus our formalization of correctness-without-privacy will make use of an *adversary output* function, $\mathcal{AO}$. It maps the final traffic to a tagged vector y_T, indicating the output "attributable" to each corrupted player $i \in T$.

We are thus led to the following

Informal Definition: *A protocol* awarely computes *the function f if there exist awareness functions $\mathcal{AI}$ and $\mathcal{AO}$ such that, upon termination, each good player i outputs $f_i(x'_T \cup x_{\overline{T}})$, $x'_T =$ $\mathcal{AI}$*(adversary traffic)*, and (what the adversary "can compute as bad player outputs" =)* $\mathcal{AO}$*(adversary traffic)* $= y_T = f_T(x'_T \cup x_{\overline{T}})$.

WEAK COMPUTATION. Have we finally captured correctness without privacy? Not yet: there a "mismatch." In an ideal evaluation without privacy, the adversary could substitute the inputs of the corrupted players with a *given* set of fake values w'_T, and, after this, she is entitled to become aware of the values $f_T(w'_T \cup x_{\overline{T}})$. Unfortunately, there is not yet any guarantee that, in an execution of a correct-without-privacy protocol for f, the adversary may become aware of the result of evaluating the good inputs shuffled with a *different* set of fake ones, even though she executes honestly on behalf of each corrupted player. That is, assume after corrupting the players in T at the very beginning, the adversary sends messages on their behalf as if they were honest and had w'_T as the set of their initial inputs. By doing so she generates some specific traffic, *traffic*, for herself. However, our formalization so far does not guarantee that $\mathcal{AI}(\textit{traffic}) = w'_T$. This does not affect the output of good players at all, since, whether or not $\mathcal{AI}(\textit{traffic}) = w'_T$, each good player $g \in \overline{T}$ will output f_g evaluated at the good inputs shuffled with w'_T.[7] What can, instead, be said about the "output" of the adversary? Only that in the ideal scenario it will be $y^I_T = f_T(w'_T \cup x_{\overline{T}})$ while in the protocol scenario will be $y^P_T = f_T(\mathcal{AI}(\textit{traffic}) \cup x_{\overline{T}})$. Since there is no guarantee, so far, that $\mathcal{AI}(\textit{traffic})$ equals w'_T, there is also no guarantee that y^I_T equals y^P_T. This "mismatch" would result in the ***non-composability*** of correct-without-privacy protocols! This may not be too bad *per se*, since correctness without privacy is for us only an intermediate step, but the notion of security would be affected the same way.

To prevent this mismatch, a correct protocol implicitly defines a *local input function*, $\mathcal{I}$. This function, evaluated at the message traffic of an individual player, yields the input of that particular player into the joint computation. Naturally, it must be the case that for each good player, this evaluation yields his initial input. The output of a correct protocol, both for the good players and the adversary, is correctly correlated with these local inputs. Awareness is guaranteed by having the projection of $\mathcal{I}$ onto the corrupted players coincides with the result of applying $\mathcal{AI}$.

[7] This is so because, to a player in $\overline{T}$, the situation in which all players remained good and the ones in subset T happened to have w'_T as their set of initial inputs, and the situation in which the players in T were corrupted, were given w'_T as a set of fake inputs, but otherwise followed faithfully the protocol, are *totally identical.* Thus, given that in the first situation each good player $g \in \overline{T}$ would output $f_T(w'_T \cup x_{\overline{T}})$, the same will happen in the second situation.

Let us try to explain a bit. The adversary cannot compute the input that a good player contributes to the computation via the public function $\mathcal{AI}$, since she does not have all the messages necessary to evaluate it. On the other hand, by evaluating $\mathcal{AI}$ on her traffic, she can be aware of the inputs of the corrupted players. It should be noticed that this latter requirement is not, in general, a trivial one! In fact, the input of a corrupted player is determined by evaluating $\mathcal{I}$ on his local traffic. When dealing with a dynamic adversary, capable of corruptiong players in the middle of an execution, this traffic is composed of two parts: a "good sub-traffic" containing all the messages sent and received when the was good, and a "bad sub-traffic" containing all of his messages after corruption. The adversary certainly knows the second sub-traffic, but to compute what input a bad player has really entered the computation one needs either to also know the good sub-traffic, or to have some other quantity that suffices for her to be aware of her corrupted players' inputs. In the setting considered in our first three sections, the adversary will explicitly have access to the good sub-traffic. (In fact, we are defining security with respect to an adversary with the biggest, natural advantage. Accordingly, upon corrupting a player, she obtains, as his exposed state, his full view; thus, in particular, his good sub-view.) In general, in less adversarial models, the protocol must guarantee adversarial awareness through the function $\mathcal{AI}$. The resulting notion of security is what we call *weak computation*. A synopsis of it is offered by the following

Informal Definition: *A protocol* weakly computes *the function f if, upon termination, each good player i outputs $f_i(x'_T \cup x_{\overline{T}})$, where $x'_T = \mathcal{AI}$(adversary traffic), and the adversary "can compute as bad player outputs" $y_T = f_T(x'_T \cup x_{\overline{T}})$, where $y_T = \mathcal{AO}$(adversary traffic). Bad player i's "input contribution" depends on his local traffic (and if this traffic pattern arises while i is good, it specifies his initial input.)*

The next subsections of this section are devoted at properly formalizing this notion.

3.2.2 View and traffic

Using the notation developed in the last section, we associate to the execution of an R-round protocol with an adversary some important quantities: an individual *player's view*, the *adversary's view*,, an individual *player's traffic*, and the *adversary's traffic*.

A PLAYER'S VIEW. A player's view encodes "everything that happens to him"; We already defined this quantity, but let us recall the definition here. For i be a player, his *round-r view* is the string $h_i^r = (s_i^{r0}, s_i^{r1}, \ldots, s_i^{r\ell_{ir}})$, his *view through round-r* is $H_i^r = (c,\ h_i^0, h_i^1, \ldots, h_i^r)$, and his *final view* is H_i^R.

THE ADVERSARY'S VIEW. The adversary's view encodes "everything that happens to the adversary". More formally, her *round-r view* is the string $h_A^r = (s_A^{r0}, s_A^{r1}, \ldots, s_A^{r\ell_{Ar}})$, her *view through round-r* is $H_A^r = (c, h_A^0, h_A^1, \ldots, h_A^r)$, and her *final view* is H_A^R.

PLAYER TRAFFIC. Player i's traffic records all the messages into or out of player i—whether or not player i is corrupted. To define this, recall that M_i^r is the message broadcast by i at the end of round-r, and m_{ij}^r is the message sent from i to j at the end of round-r. Then *player i's round-r traffic* is the string

$$t_i^r \ = \ (M_1^r * M_2^r * \cdots * M_n^r * m_{1i}^r * m_{2i}^r * \cdots * m_{ni}^r,\ m_{i1}^r * m_{i2}^r * \cdots * m_{in}^r),$$

his traffic through round-r is $T_i^r = (c,\ t_i^0, t_i^1, \ldots, t_i^r)$, and *player i's traffic* is T_i^R.

ADVERSARY TRAFFIC. The adversary traffic records all of the exchanges between the adversary and the uncorrupted players: everything the adversary "gets" from good players (the messages they broadcast, the messages they send to corrupted players, and the information the adversary learns when one of these good players is corrupted), together with the information that the adversary "gives" to good players (the messages the adversary broadcasts on behalf of corrupted players, and the messages the adversary sends out along private channels to uncorrupted players on behalf of corrupted players). To define the adversary's traffic, recall (from Page 23) the quantity $\dot{\mathcal{M}}_A^r$ for the incoming messages to the adversary at the beginning of her round-r. Similarly, we introduce a notation for the out-going messages from the adversary at the end of her round-r. It is

$$\ddot{\mathcal{M}}_A^r = \ddot{M}_1^r * \cdots * \ddot{M}^r * \ddot{m}_{11}^r * \cdot \ddot{m}_{1n}^r * \cdots * \ddot{m}_{n1}^r * \cdot \ddot{m}_{nn}^r \qquad \textit{"Round-}r\textit{ messages sent out by bad players."}$$

where

$$\ddot{M}_i^r = \begin{cases} \tilde{M}_i(s_A^r) & \text{if } i \in \kappa_A^r \\ \Lambda & \text{otherwise} \end{cases} \qquad \textit{"Round-}r\textit{ messages prepared by corrupted players"}$$

$$\ddot{m}_{ij}^r = \begin{cases} \tilde{m}_{ij}(s_A^r) & \text{if } i \in \kappa_A^r \text{ and } j \notin \kappa_A^{r\ell_{A^r}} \\ \Lambda & \text{otherwise} \end{cases} \qquad \textit{"Round-}r\textit{ messages from bad players to good players"}$$

We add in the proviso that $\ddot{M}_i^r = \ddot{m}_{ij}^r = \Lambda$ for $r = 0$. Now, consider a round-r in which some number m of players are corrupted, in sequence: $i_1, \ldots, i_m$. The *round-r adversary traffic* is the string

$$t_A^r = (\dot{\mathcal{M}}_A^r, (i_1, \sigma_{i_1}^r), \ldots, (i_m, \sigma_{i_m}^r), \ddot{\mathcal{M}}_A^r)$$

containing the messages received by the adversary at the beginning of the round; the corruption requests, in order, and exposed state given to the adversary as a result of these requests; and the messages composed and sent out by the adversary at the end of her round of activity. The adversary traffic through round-r is $T_A^r = (c, t_A^0, t_A^1, \ldots, t_A^r)$, and the *adversary traffic* is T_A^R.

NOTATION. Fix a R-round, n-party protocol, P, and an adversary, A. Then for any initial configuration $(\vec{x}, c, \vec{r}, r_A)$ there is an induced execution of P with A, and, consequently, for any player i and round r, there is an associated player i view through round r, player i traffic through round r, adversary view through round r, and adversary traffic through round r. These are denoted by $H_i^{A,P,r}(\vec{x}, c, \vec{r}, r_A)$, $T_i^{A,P,r}(\vec{x}, c, \vec{r}, r_A)$, $H_A^{A,P,r}(\vec{x}, c, \vec{r}, r_A)$, and $T_A^{A,P,r}(\vec{x}, c, \vec{r}, r_A)$. When A and P are understood, we omit them as superscripts.

3.2.3 Adversary input and output

Definition 3.1 *An* **adversary input function** *for an R-round protocol is a map*

$$\mathcal{AI}: \underbrace{\{0,1\}^*}_{\substack{\text{common}\\ \text{input}}} \times \underbrace{\{0,1\}^*}_{\substack{\text{adversary}\\ \text{traffic}}} \to \underbrace{2^{[1..n]\times\{0,1\}^\ell}}_{\substack{\text{fake}\\ \text{inputs}}} \cup \{\textsf{not-yet}\}$$

such that for any $c \in \{0,1\}^*$ and any well-formed sequence of traffic values $T_A^0, \ldots, T_A^R$, indicating corrupted players $\kappa_A^0 \subseteq \cdots \subseteq \kappa_A^R$, there is some $r < R$ for which

- $\mathcal{AI}(c, T_A^r) = \textsf{not-yet}$ *for all $\bar{r} < r$; and,*

- *For some tagged vector* x'_T, $T = \kappa_A^r$ *and* $\mathcal{AI}(c, T_A^{\bar{r}}) = x'_T$ *for all* $\bar{r} \geq r$.

An **adversary output function** *for an* R*-round protocol is a map*

$$\mathcal{AO}\colon \underbrace{\{0,1\}^*}_{\substack{\text{common}\\ \text{input}}} \times \underbrace{\{0,1\}^*}_{\substack{\text{final}\\ \text{traffic}}} \to \underbrace{2^{[1..n]\times\{0,1\}^l}}_{\substack{\text{deserved}\\ \text{output}}}$$

such that for any $c \in \{0,1\}^*$ *and any well-formed sequence of traffic values* $T_A^0, \ldots, T_A^R$, *indicating corrupted players* $\kappa_A^0 \subseteq \cdots \subseteq \kappa_A^R$,

- *For some tagged vector* x'_T, $T = \kappa_A^R$ *and* $\mathcal{AO}(c, T_A^R) = x'_T$

NOTATION. If P is an n-party, R-round protocol and A is an adversary, then any initial configuration $(\vec{x}, c, \vec{r}, r_A)$ determines an execution of P with A, and hence a final adversary traffic T_A^R and an *adversary committal* $\mathcal{AI}^{A,P}(\vec{x}, c, \vec{r}, r_A) \stackrel{\text{def}}{=} \mathcal{AI}(c, T_A^R)$. When A and P are understood, we omit them as superscripts.

Similarly, protocol P, adversary A, and initial configuration $(\vec{x}, c, \vec{r}, r_A)$ determine an *adversary output* $\mathcal{AO}^{A,P}(\vec{x}, c, \vec{r}, r_A) \stackrel{\text{def}}{=} \mathcal{AI}(c, T_A^R)$. When A and P are understood, we omit them as superscripts.

3.2.4 Network input and output

NETWORK INPUT. Fix a protocol P and an adversary A. It is natural to extend $\mathcal{AI}(\vec{x}, c, \vec{r}, r_A)$ to an n-vector by including, along with the substituted inputs of corrupted processors, the original inputs of the processors which were *uncorrupted* at the time that $\mathcal{AI}$ specified a substituted value. Thus we define the *network input*, $\overline{\mathcal{AI}}(\vec{x}, c, \vec{r}, r_A)$, as $x'_T \cup x_{\overline{T}}$, where $x'_T = \mathcal{AI}(\vec{x}, c, \vec{r}, r_A)$. Intuitively, the network input is the good players' original inputs "shuffled in" with the substituted inputs the adversary regards herself as entering into the collaborative computation; it specifies, for the indicated execution, what the *network* has entered into the joint computation on this run.

NETWORK OUTPUT. Fix a protocol P and an adversary A. It is natural to extend $\mathcal{AO}(\vec{x}, c, \vec{r}, r_A)$ to an n-vector by including, along with the adversarially-perceived outputs of corrupted players, the private output of the players which were *uncorrupted* at the protocol's conclusion. Thus we define the *network output*, $\overline{\mathcal{AO}}(\vec{x}, c, \vec{r}, r_A)$, as $y'_T \cup y_{\overline{T}}$, where $y'_T = \mathcal{AO}(\vec{x}, c, \vec{r}, r_A)$ and $y_{\overline{T}}$ is defined by $y_i = o(s_i^R)$, for $i \in \overline{T}$. Intuitively, the network output consists of the good players' outputs "shuffled in" with the outputs the adversary regards herself as having been given as a result of the collaborative computation; it specifies, for the indicated execution, what the good players *did* compute as output values, and what the adversary *could* compute on behalf of the corrupted players.

3.2.5 Local input

DEFINITION OF A LOCAL INPUT. Local input functions for protocols have a bit too much structure to conveniently define them in one swoop. So we break the definition in two; first, as a syntactic object:

Definition 3.2 *A* **local input function** *is a map*

$$\mathcal{I}\colon \underbrace{[1..n]}_{\text{player}} \times \underbrace{\{0,1\}^*}_{\substack{\text{common}\\ \text{input}}} \times \underbrace{\{0,1\}^*}_{\substack{\text{player's}\\ \text{traffic}}} \to \underbrace{\{0,1\}^\ell}_{\substack{\text{committed}\\ \text{input}}} \cup \{\text{not-yet}\}$$

We denote $\mathcal{I}(i, c, T)$ by $\mathcal{I}_i(c, T)$.

Now fix an n-party, R-round protocol, P, and a t-adversary, A. Any initial configuration $(\vec{x}, c, \vec{r}, r_A)$ determines an execution, and hence a collection of player traffic values, $\{T_i^r : i \in [1..n], r \in [0..R]\}$, and a collection of players corrupted during this execution, $\{\kappa_A^r\}$. We define the following quantities associated with the specified execution:

- *(The round-r input of player i.)* $\mathcal{I}_i^r(\vec{x}, c, \vec{r}, r_A) = \mathcal{I}_i(c, T_i^r)$
- *(The round-r input of the network.)* $\mathcal{I}^r(\vec{x}, c, \vec{r}, r_A) = (\mathcal{I}_1(c, T_1^r), \ldots, \mathcal{I}_n(c, T_n^r))$
- *(The committed input of player i.)* $\mathcal{I}_i(\vec{x}, c, \vec{r}, r_A) = \mathcal{I}_i^R(c, T_i^r)$
- *(The local inputs.)* $\mathcal{I}(\vec{x}, c, \vec{r}, r_A) = (\mathcal{I}_1(c, T_1^R), \ldots, \mathcal{I}_n(c, T_n^R))$
- *(The bad players at round-r.)* $\mathrm{Bad}^r(\vec{x}, c, \vec{r}, r_A) = \kappa_A^r$
- *(The good players at round-r.)* $\mathrm{Good}^r(\vec{x}, c, \vec{r}, r_A) = [1..n] - \kappa_A^r$

For $\mathcal{I}$ to be a local input function that is "ok" with respect to a protocol P, we need the following additional structure:

Definition 3.3 *Fix $t \in \mathbf{N}$ and a R-round, n-party protocol, P. We say that an $\mathcal{I}$ is a* t-local input function for P *if for every t-adversary A the following conditions hold:*

- *(Monotone committal.)* $\mathcal{I}_i^r(\vec{x}, c, \vec{r}, r_A) = x_i \in \{0,1\}^\ell \Rightarrow \mathcal{I}_i^{r+1}(\vec{x}, c, \vec{r}, r_A) = x_i$
- *(Simultaneous committal.)* $\mathcal{I}_i^r(\vec{x}, c, \vec{r}, r_A) = \textsf{not-yet} \Rightarrow \mathcal{I}_j^r(\vec{x}, c, \vec{r}, r_A) = \textsf{not-yet}$
- *(Eventual committal.)* $\mathcal{I}_1(\vec{x}, c, \vec{r}, r_A) \in \{0,1\}^\ell$
- *(Meaningful committal.)* $g \in \mathrm{Good}^r(\vec{x}, c, \vec{r}, r_A) \Rightarrow \mathcal{I}_g^r(\vec{x}, c, \vec{r}, r_A) \in \{\textsf{not-yet}, x_i\}$

ADMISSIBLE ADVERSARY INPUT FUNCTION. An adversary input function is admissible if it coincides with a local input function. More formally,

Definition 3.4 *Fix an n-party, R-round protocol, P. An adversary input function for P is* t-admissible *if there exists a t-local input function $\mathcal{I}$ for P such that for any t-adversary A and initial configuration $(\vec{x}, c, \vec{r}, r_A)$, any round r and player $i \in \mathrm{Bad}^r(\vec{x}, c, \vec{r}, r_A)$,*

- $\mathcal{AI}(c, T_A^r) = \textsf{not-yet} \Rightarrow \mathcal{I}(c, T_i) = \textsf{not-yet}$, *and*
- $\mathcal{AI}(c, T_A^r) = x'_T \Rightarrow \mathcal{I}(c, T_i) = x'_i$ *for all $i \in T$,*

where $T_i^r = T_i^r(\vec{x}, c, \vec{r}, r_A)$ and $T_A^r = T_A^r(\vec{x}, c, \vec{r}, r_A)$.

3.2.6 Weak correctness

We now describe what it means for a protocol to *weakly* compute a finite function.

Definition 3.5 *Let $n, \ell, l, t \in \mathbf{N}$, let f: $(\{0,1\}^\ell)^n \to (\{0,1\}^l)^n$ be a function, and let P be a fixed-round, n-party protocol. We say that P* perfectly t-weakly computes *f if there is a t-admissible adversary input function $\mathcal{AI}$ and an adversary output function $\mathcal{AO}$ such that for every t-adversary A,*

$$\overline{\mathcal{AO}}(\vec{x}, c, \vec{r}, r_A) \;=\; f(\overline{\mathcal{AI}}(\vec{x}, c, \vec{r}, r_A)).$$

3.2.7 Remarks

EXISTENCE OF WEAKLY CORRECT PROTOCOLS.

TRAFFIC *vs.* VIEW. In subsequent sections, we both relax our notion of awareness and discuss its complexity. We wish, however, to briefly discuss here the rationale for letting it depend on the traffic. Let's concentrate on adversarial awareness, since similar points can be made on player's awareness. Why not, for instance, allow adversarial awareness to arbitrary depend on the full adversary view? (Here, by *adversary view* we mean the adversary traffic plus all the coins she has tossed; alternatively, it is the sequence of her computational states.) There are at least two excellent answers.

- On the philosophical side, this view is meant to capture all that affected her computation, not necessarily what she is aware of. For instance, the adversary might have a source of entropy for randomizing her state, without having explicit access to her underlying coin tosses. More generally, if the adversary is a person, it may be unrealistic to say that she knows a full description of her own state, so as to be able to evaluate a function on it. By defining adversarial awareness as a function of her view, we would be making an unnecessary assumption about the nature of an adversary, and we would ultimately obtain a notion of security adequate only if the adversary is —say— a Turing machine. It is, instead, incontrovertible that the adversary, independently of her nature, has immediate access to (and thus is aware of) her traffic. Thus, ignoring complexity concerns, computability on the traffic is a simple and natural notion of awareness.

- A more technical answer is that adopting too generous a notion of awareness (like computability on the adversary's full view) seems to entail, at a very close examination, loosing crucial properties of secure computation—such as reducibility.

Thus, while for some agents, awareness can depend on additional quantities, both weak-computation and secure protocols make sure that it only depends on the traffic. This way, they guarantee the independence of their respective notions from the nature of the particular agents.

3.3 Privacy (without correctness)

3.3.1 Motivation

A secure computation of a function f should mimic—as closely as possible—its ideal evaluation; that is, roughly, it should be exactly as private and exactly as correct. Though we will ultimately insist that these two issues be handled *together* in a protocol we call secure, privacy by itself is meaningful and interesting notion. In this section, we describe a notion of "privacy leaking f," motivated by the ideal evaluation.

Privacy is a measure of how much the adversary learns. Of course, in any protocol correctly computing f the adversary can learn at least the *ideal information*—the information available in the ideal evaluation. (In fact, she can dynamically corrupt a few players during her round 0 using the same strategy she would use in the ideal evaluation; give them the same fake inputs she would give them in the ideal scenario; then let them run as in the original protocol; then corrupt, during her final round, additional players, as she would in the ideal evaluation. This way, upon termination, the adversary will learn all the information available to her in the ideal evaluation.) In any correct protocol, however, the adversary sees additional information: the adversary traffic generated by the protocol and, more generally, her own view. Thus a secure protocol for evaluating f should ensure

that this view does not contain more than the ideal information. In fact, we want it to contain *precisely* the ideal information. We express that a secure protocol is exactly as private as an ideal protocol by saying that:

> (i) (THE PROTOCOL IS AT LEAST AS PRIVATE AS THE IDEAL PROTOCOL.) *There must exist an agent which, knowing only the ideal information, manages to interact with the adversary so as to generate for her what is essentially the same view as she gets when interacting with the real network of players.*
>
> (ii) (THE PROTOCOL IS AT MOST AS PRIVATE AS THE IDEAL PROTOCOL.) *From the information obtained in executing a protocol, the adversary should be able to "extract" the ideal information.*

The principal tools for properly expressing the above ideas are *simulators* and *awareness.* The later we have already seen, while the former is the agent that generates the right view for the adversary. It does this with the aid of an *ideal-evaluation oracle,* an abstraction which models possession of the ideal information.

Once we have in hand a notion of how an adversary interacts with a simulator, the notion of privacy is close at hand: all we have to say is the the adversary–network interactions are "just like" the adversary–player interaction. The crux, of course, is in saying what "just like" means—but this task is made easy for us by now well-known notions in modern cryptography.

3.3.2 Ideal-evaluation oracles

An *ideal-evaluation oracle* is a special, history-dependent oracle. For any n-vector $\vec{x}$ and function $f : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$, the ideal-evaluation oracle $\mathcal{O}_f(\vec{x})$ responds to two types of queries:

- A **component query** is an integer $i \in [1..n]$. How a component query is answered depends on whether or not a valid function query has been made. If a valid function query has not yet been made, then the component query i is answered by x_i. If a valid function query x'_T has been made and $\vec{y} = f(x'_T \cup x_{\overline{T}})$, then component query i is answered by (x_i, y_i). An invalid component query is answered by the empty string.
- A **function query** is a tagged vector x'_T. A function query x'_T is valid if it is the first query of this type and T consists precisely of the component queries made so far. If x'_T is a valid function query and $\vec{y} = f(x'_T \cup x_{\overline{T}})$, then the function query is answered by y_T. An invalid function query is answered by the empty string.

Clearly, if $\vec{x}$ represents the players' private inputs and f is the function that they want to collaboratively compute, then having an $\mathcal{O}_f(\vec{x})$ oracle precisely captures possessing the ideal information: what an adversary can learn in the ideal evaluation of $f(\vec{x})$ by corrupting t players she can also learn by making t component to an $\mathcal{O}_f(\vec{x})$ oracle, and vice versa.

3.3.3 Simulators

As suggested by Figure 2, an adversary interacts with a simulator with "mechanics" similar to her interacting with a network. (In fact, the interesting simulators are those which cause these interactions to be indistinguishable to the adversary.) For example, since an adversary expects to see messages broadcast by uncorrupted players, a simulator is equipped to provide such messages; since the adversary can corrupt a player and learn his exposed state, a simulator is equipped to specify it.

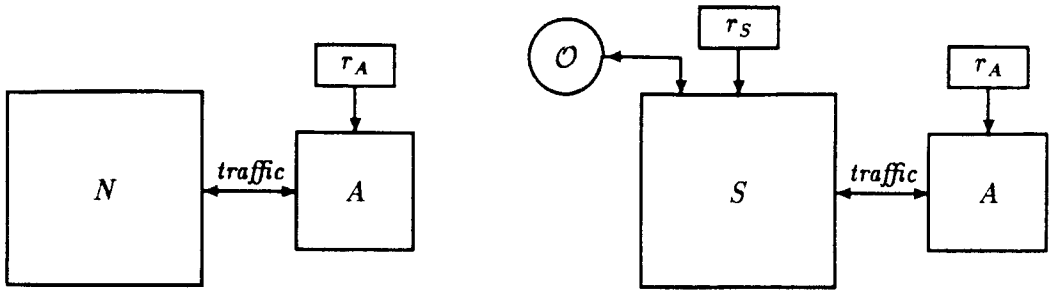

Figure 2: *A simulator S creates a "virtual world" and allows the adversary A to act in this world, as though S itself were a network, N. To accomplish this, a simulator is provided an ideal-evaluation oracle $\mathcal{O}_f(\vec{x})$. Above, r_A are the adversary's coins, and r_S are the simulator's coins.*

To meaningfully interact with an adversary, a simulator has access to an ideal-evaluation oracle, $\mathcal{O}_f(\vec{x})$. It is crucial that the simulator's access to this oracle and its interaction with the adversary be properly coordinated—else, one does not obtain a meaningful notion of security. We specify this coordination below:

- When, *and only when,* the adversary A with whom the simulator S interacts corrupts a player i, the simulator makes a component query of i.
- The strategy employed by the simulator to come up with its function query consists of evaluating an adversary input function, $\mathcal{AI}$.
- The simulator makes this function query immediately following the completion of an adversary round.

Let us briefly justify these choices. The first requirement highlights that it would be unacceptable if an adversary could, say, corrupt players 1, 3 and 4, and the simulator, in response, would ask the private inputs of players 2, 3 and 7; were this allowed even simple serial composition properties would not hold for the resulting notion of privacy. The second constraint is at the heart of our *blending* of privacy and correctness; it guarantees that in a computation with the *network*, the adversary is forced to "know" the value x'_T that she has effective entered into the joint computation. This *same* value will be used to define correctness. The last assumption concretizes the intuition that the adversary can enter a value x'_T into the collaborative computation only by sending out messages, not by corrupting some player.

HISTORY. Before proceeding further, we would like to mention, at least briefly, a history of the ideas of this subsection. The notion of using a simulator for proving and, more importantly, for defining that the amount of knowledge learned in an interaction is bounded, is due to Goldwasser, Micali and Rackoff [GMR89]. Our notion of a simulator, however, is, *by necessity,* more demanding than the one developed in their context. In particular, while our simulator must *interact* with an adversary in a manner similar to an adversary interacting with a network—implicitly assembling a transcript associated to this interaction which is precisely the adversary's view—simulators in the sense of [GMR89] may lay down an *arbitrary* transcript intended to resemble the adversary view. Thus we distinguish our simulators as being "on-line."

In fact, on-line simulatability was already recognized as possessing a useful structure. Kilian [Ki89] discusses interactions of this form, and they are explicitly required in the work of Crépeau and Micali [Cr90].

We note that on-line simulatability is essentially unrelated to the notion of black-box simulatability, an idea first investigated by Oren [Or87]. Black-box simulatability concerns the manner in which a simulator may depend on the adversary it is intended to simulate, while on-line simulatability concerns the mechanism in which "simulated" adversary transcripts are assembled. (In fact, we have not yet discussed the extent to which our simulators may depend on the adversary with whom they speak.)

3.3.4 Formal description

We now define what a simulator is and how it runs with an adversary A, using a particular oracle $\mathcal{O}_f(\vec{x})$ and adversary input function $\mathcal{AI}$.

Let us briefly explain why a simulator, as an agent, looks different from players and adversaries. In describing protocols and adversaries, we insisted on a fine level of granularity. This was necessary because, for a player, any single chunk of probabilistic computation significantly alters his internal state of knowledge, and may make it more profitable or less profitable to corrupt him; for the adversary, we needed to pin-point her ability to corrupt several players in sequence within a round, and to specify exactly the view she gets in the execution of a protocol. On the other hand, for a simulator it is important to tightly coordinate its interaction with the oracle and the adversary, as we have already specified, but capturing precisely the simulator's internal state at each point in time is less crucial. Thus for a simulator we can get by with a "coarser" level of granularity. In fact, we do not keep track of its individual coin tosses: a simulator moves from computational state to computational state in complete knowledge of its infinite sequence of coins. Since these coin tosses remain fixed (i.e., are never explicitly "stripped off" as they were with the players and the adversary), a simulator's configuration may be regarded as coinciding with its computational state.

Though one would do, we associate to a simulator two syntactic function, σ and μ. When applied to the simulator's computational state "at the right time," these specify what the simulator *sends* to the adversary. Specifically, when the adversary corrupts a player i, she expects her computational state to be augmented by the exposed state σ_i^r of the newly corrupted player. Imitating this, the simulator evaluates σ on its computational state and the result of this is appended to the adversary's computational state. Likewise, at the beginning of an adversary round, the adversary expects to have her incoming messages $\dot{\mathcal{M}}_A^r$ augment her computational state. Imitating this, the simulator evaluates μ on its computational state and the result of this is appended to the adversary's computational state.

Definition 3.6 *A* **simulator** S *is a function*

$$S : \underbrace{\{0,1\}^*}_{\substack{\text{common}\\ \text{input}}} \times \underbrace{\{0,1\}^*}_{\substack{\text{current}\\ \text{state}}} \times \underbrace{\{0,1\}^\infty}_{\substack{\text{simulator}\\ \text{coins}}} \to \underbrace{\{0,1\}^*}_{\substack{\text{new}\\ \text{state}}} .$$

A **simulator syntactic function** *is either of the following functions:*

- **a simulated exposed state** *function* σ: $\underbrace{\{0,1\}^*}_{\substack{\text{current}\\ \text{state}}} \to \underbrace{\{0,1\}^*}_{\substack{\text{simulated}\\ \text{exposed state}}}$

- a simulated incoming-messages *function* μ: $\underbrace{\{0,1\}^*}_{\text{current state}} \rightarrow \underbrace{\{0,1\}^*}_{\text{simulated incoming messages}}$

DISCUSSION. As before, the simulator syntactic functions are fixed maps, each determining its range point in a natural, easily encoded manner from its domain points. As usual, we omit specifying these functions.

CONFIGURATION SEQUENCES. Let f: $(\{0,1\}^\ell)^n \rightarrow (\{0,1\}^\ell)^n$ be a finite function, let A be an adversary for an R-round, n-party protocol, and let $\mathcal{AI}$ be an adversary input function. If S is a simulator then running adversary A with simulator S, oracle $\Omega_f(\vec{x})$, adversary input $\mathcal{AI}$, common input c, initial adversary state s_A, adversary coins r_A, initial simulator state s_S, simulator coins r_S, and initial corruptions κ_A, is a way to map the initial adversary configuration, $C_A^{00} = (s_A, r_A, \kappa_A)$, to a final adversary configuration, $C_A^R = C_A^{R\varrho_{AR}}$.

To describe this mapping, fix the notation that $C_S^{r\rho} = s_S^{r\rho}$, let ϱ_{Ar} be as before, and let the adversary configurations progress according to the following recurrences:

$$C_A^{r(\rho+1)} = \begin{cases} (A(c, s_A^{r\rho}),\ r_A^{r\rho},\ \kappa_A^{r\rho}) & \text{if } \tilde{\mathcal{N}}(s_A^{r\rho}) = \textsf{compute} \\ (s_A^{r\rho} * r_{A1}^{r\rho},\ r_{A2}^{r\rho} r_{A3}^{r\rho} \cdots,\ \kappa_A^{r\rho}) & \text{if } \tilde{\mathcal{N}}(s_A^{r\rho}) = \textsf{flip-coin} \\ (s_A^{r\rho} * \sigma(s_S^{r(\rho+1)}),\ r_A^{r\rho},\ \kappa_A^{r\rho} \cup \{i\}) & \text{if } \tilde{\mathcal{N}}(s_A^{r\rho}) = \textsf{corrupt}_i \\ C_A^{r\rho} & \text{if } \tilde{\mathcal{N}}(s_A^{r\rho}) = \textsf{round-done} \end{cases}$$

$$C_A^{(r+1)0} = (s_A^{r\varrho_{Ar}} * \mu(s_S^{(r+1)0}),\ r_A^{r\varrho_{Ar}},\ \kappa_A^{r\varrho_{Ar}})$$

while the simulator's computational state progresses according to

$$s_S^{r(\rho+1)} = \begin{cases} S(c,\ s_S^{r\rho} * \Omega^r(i),\ r_S) & \text{if } \mathcal{N}(s_A^{r\rho}) = \textsf{corrupt}_i \\ s_S^{r\rho} & \text{otherwise} \end{cases}$$

$$s_S^{(r+1)0} = S(c,\ s_S^{r\varrho_{Ar}} * \Omega(\mathcal{AI}(c, T_A^r)))$$

where, to specify the behavior of the oracle, we set

$$\Omega^r(i) = \begin{cases} x_i & \text{if } \mathcal{AI}(c, T_A^r) = \textsf{not-yet} \\ (x_i,\ y_i) & \text{if } \mathcal{AI}(c, T_A^r) = x'_T \text{ and } \vec{y} = f(x'_T \cup x_{\overline{T}}) \end{cases}$$

$$\Omega(x'_T) = y_T \text{ where } \vec{y} = f(x'_T \cup x_{\overline{T}})$$

and $\Omega(\textsf{not-yet}) = \Lambda$. As usual, T_A^r denotes the traffic to the adversary through round r.

EXECUTING AN ADVERSARY WITH A SIMULATOR. Consider a fixed-round protocol, P, having adversary input function $\mathcal{AI}$. An *execution* of an adversary A with a simulator S, ideal-evaluation oracle $\Omega_f(\vec{x})$, adversary input function $\mathcal{AI}$, common input c, adversary coins r_A, simulator coins r_S, and initial corruptions $\kappa_A = \emptyset$, is the collection of configurations $\{C_A^{r\rho},\ s_A^{r\rho}\}$ generated, as above,

when the initial computational state of the simulator is $c*\kappa_A$ and the initial computational state of the adversary is κ_A.

REMARK. The fact that the simulator computes its function query by evaluating an awareness function imposes a restriction on the way the simulator computes the function query. The awareness property not withstanding, the function query might also depend on the coin tosses of the simulator, which are not part of the adversary traffic. By binding together the function query and the adversary input function $\mathcal{AI}$, we intertwine privacy and awareness in a significant way.

The $\mathcal{AO}$ function, instead, poses no restrictions on the simulating algorithm. However, suppose that the adversary A has oracle $\Omega_f(\vec{x})$ and the adversary and simulator S interact leading to an R-round traffic T_A^R. In this interaction, suppose the simulator asked a function query x'_T, as indicated by $\mathcal{AI}$, and the players of set $\mathcal{T}$ were corrupted at termination. Then, in light of (ii) on Page 35, one would expect that

$$\mathcal{AO}(T_A^R) = f_T(x'_T \cup x_{\overline{T}}).$$

In fact, we might have demanded this relationship between S, $\mathcal{AI}$, and $\mathcal{AO}$ (for any adversary A), dismissing most uninteresting cases. But it is not necessary to do so; the assertion above will be true automatically (except perhaps a negligible fraction of the time) in a protocol we call "secure."

3.3.5 Ensembles

Ensembles are collections of probability measures; they provide a convenient abstraction for discussing asymptotics in many cryptographic scenarios. Our ensembles bear two indices, which are treated differently. Since one of these will always be a *security parameter,* we being by giving some intuition on what one is.

SECURITY PARAMETER. Very informally, a computation achieves *perfect* security if its execution imparts to an adversary absolutely *no* information beyond what is divulged in the corresponding ideal computation.

Unfortunately, for most reasonable cryptographic scenarios and for most interesting tasks, the goal of achieving perfect security is not obtainable. Thus we must be less demanding, allowing our protocols to fall short of perfect security in a well-specified, quantizable way. A *security parameter* is used to measure how far a protocol is from the ideal scenario. The larger its value, the more secure the resulting executions, and the closer to the abstraction one is achieving. Before they run a protocol, players agree to use an adequately large security parameter to satisfy their needs.

Since our notion of a secure computation allows "imperfect" security, it explicitly depends on a security parameter, k. We establish the convention that this security parameter is presented to the players and the adversary on the common input, written in unary. In fact, for now we fix the convention that the common input c contains only the security parameter: $c = 1^k$.

DEFINITION OF AN ENSEMBLE Our notion of an ensemble is somewhat more general than that of [GM84]. We define it below.

Definition 3.7 *If $\mathcal{L} = \{L_k \subseteq \{0,1\}^*\colon k \in \mathbf{N}\}$ is a family of languages, then an* **ensemble** *E (over $\mathcal{L}$) is a collection of probability measures on $\{0,1\}^*$, one for each $(k,\omega) \in \mathbf{N} \times L_k$. The argument k is called the* **index** *of the ensemble E, the argument ω is called the* **parameter** *of E, and $\mathcal{L}$ is called the* **parameter set** *of E.*

As the index k is always drawn from $\mathbf{N}$, we never specify it, writing $E = \{E_k(\omega)\colon \omega \in L_k\}$ to indicate an ensemble over $\mathcal{L} = \{L_k\}$. When the parameter set of an ensemble is understood and

there is no danger of confusion, we refer to an ensemble by writing the symbol "$\mathcal{E}$" in front of its "generic element"—that is, we simply write $\mathcal{E}E_k(\omega)$ instead of $\{E_k(\omega)\colon\ \omega \in L_k\}$.

The above notion of an ensemble applies only to distributions on strings. However, the notion is trivially extended to any other domain whose elements can be canonically encoded as strings. We will thus speak of ensembles on other domains, where it is understood that there is, implicitly, a fixed encoding of domain points into strings.

AN EXAMPLE OF AN ENSEMBLE. A *probabilistic encryption algorithm* is an algorithm E that takes as input a *security parameter*, k, a *message*, m, and a sequence of *random coins*, r, and produces as output a *cyphertext*, y, which is an encryption of m. The algorithm E produces its output in time polynomial in the security parameter. Thus a probabilistic encryption algorithm can only encrypt messages m whose length is bounded by some polynomial, k^c, in the security parameter.

Fixing the security parameter, k, a message $m \in \{0,1\}^{k^c}$, and taking each bit of the random coins r to be selected uniformly at random, the algorithm E induces a distribution $E_k(m)$ on space of encryptions of m under security parameter k. Letting $\mathcal{L} = \{L_k = \{0,1\}^{k^c}\}$, these distributions form an ensemble $\mathcal{E}E_k(m)$ over $\mathcal{L}$.

ENSEMBLES FOR SECURE FUNCTION EVALUATION In defining secure computation, two ensembles will be of importance to us. One is the ensemble of views the adversary sees when she interacts with the *network*, while the other is the ensemble of views she sees when interacting with the *simulator*.

The ensemble induced by a network. Consider an adversary, A, interacting with a network running an R-round, n-party protocol, P. When $\vec{x} \in (\{0,1\}^\ell)^n$ is the vector of player inputs and the security parameter is k, so $c = 1^k$, then for each pair $(\vec{r}, r_A)$ of player random strings and adversary random strings there is an associated adversary view. If $\vec{r}$ and r_A are allowed to vary, then there is an induced distribution on these adversary views, which we denote by $A\text{-VIEW}_k^P(\vec{x})$. If $\vec{x}$ and k are allowed to vary as well, we obtain an ensemble

$$\mathcal{E}\,A\text{-VIEW}_k^P(\vec{x})$$

over $\mathcal{L} = \{L_k = (\{0,1\}^\ell)^n\}$.

The ensemble induced by a simulator. Let $f\colon\ (\{0,1\}^\ell)^n \to (\{0,1\}^l)^n$, and consider an adversary, A, interacting with a simulator $S = (\mathcal{S}, \mathcal{AI})$ for an R-round protocol. When $\vec{x} \in (\{0,1\}^\ell)^n$, the security parameter is k, and the simulator S equipped with an $\mathcal{O}_f(\vec{x})$-oracle, then for each pair (r_A, r_S) of adversary and simulator random strings there is an associated adversary view. If r_A and r_S are allowed to vary, then there is an induced distribution on these adversary views, which we denote by $A\text{-VIEW}_k^{S,f}(\vec{x})$. If $\vec{x}$ and k are allowed to vary as well, we obtain an ensemble

$$\mathcal{E}\,A\text{-VIEW}_k^{S,\mathcal{AI},f}(\vec{x})$$

over $\mathcal{L} = \{L_k = (\{0,1\}^\ell)^n\}$.

3.3.6 Indistinguishability

A central notion in this paper is *indistinguishability*, as introduced by [GM84] in the context of encryption. (The notion has also proven crucial for a complexity-theoretic treatment of pseudorandom generation [Ya82b] and of zero-knowledge proofs [GMR89].) Essentially, it captures the idea that two families of distributions can be (asymptotically) so close as to be considered the same. To

say this exactly requires some specialized language—the notion of a *distinguisher* and a *probability ensemble.* The reader who wishes a bit more discussion of these notions may consult [GMR89] (pages 191–193).

NEGLIGIBILITY. We will say that a function $\epsilon : \mathbf{N} \to \mathbf{R}$ is *negligible* if it is nonnegative and vanishes faster than the inverse of any polynomial: for any $c > 0$ there exists a $K \in \mathbf{N}$ such that $\epsilon(k) \leq k^{-c}$ for all $k \geq K$. A function $\epsilon(k)$ which is not negligible is called *nonnegligible.*

DEFINITION OF INDISTINGUISHABILITY. We now specify what it means for two ensembles to be so close as to be considered insignificantly different.

Definition 3.8 *Ensembles E and E' are* **statistically indistinguishable**, *written $E \simeq E'$, if they are over the same parameter set $\mathcal{L} = \{L_k\}$, and*

$$\epsilon(k) = \sup_{\omega \in L_k} \sum_{x \in \{0,1\}^*} \left| \mathrm{Prob}_{E_k(\omega)}[x] - \mathrm{Prob}_{E'_k(\omega)}[x] \right|$$

is negligible.

When we wish to emphasize the parameter set $\mathcal{L}$ of indistinguishable ensembles E and E' we write $E \underset{\mathcal{L}}{\simeq} E'$.

3.3.7 Privacy

We give a rather stringent definition of privacy below.

Definition 3.9 *Let $n, \ell, l, t \in \mathbf{N}$, let f: $(\{0,1\}^\ell)^n \to (\{0,1\}^l)^n$ be a function, and let P be a fixed-round, n-party protocol. We say that P is* **statistically t-private leaking** *f if there is a simulator S,, and an adversary input function input function $\mathcal{AI}$, such that for any t-adversary A,*

$$\mathcal{E}A\text{-VIEW}_k^P(\vec{x}) \underset{\mathcal{L}}{\simeq} \mathcal{E}A\text{-VIEW}_k^{S,\mathcal{AI},f}(\vec{x}).$$

3.3.8 Remarks

ABSOLUTE PRIVACY. A special case of a private protocol leaking f is an *absolutely* private protocol: such a protocol is one which is private leaking "absolutely nothing"—the constant map $f = \Lambda$, say. Privacy alone, and absolute privacy in particular, are useful notions; for instance, the COMMIT protocol of a verifiable secret sharing should be absolutely private. (See [CGMA85] for introducing the notion of a verifiable secret sharing, [FM90] for a protocol and crisp definition, and [RB89] for another important protocol.)

$\forall\exists$ vs. $\exists\forall$.

3.4 Security (without history)

As we shall see, a secure protocol is somewhat more than one which meets the previously given privacy and correctness constraints. In this section, we define our most basic definition of security. Since it ignores the possible "history" of players and the adversary the notion is called "security from scratch."

3.4.1 Security from scratch

We define what it means for a protocol P to securely evaluate a finite function f, from scratch, ignoring issues of computational complexity.

Definition 3.10 *Let $n, \ell, l, t \in \mathbf{N}$, let f: $(\{0,1\}^\ell)^n \to (\{0,1\}^l)^n$ be a function, and let P be a fixed-round, n-party protocol. We say that P* statistically t-securely computes f from scratch, ignoring complexity, *if there is a simulator S, a t-admissible adversary input function $\mathcal{AI}$, and an adversary output function $\mathcal{AO}$, such that for any t-adversary A,*

- PRIVACY: *(The adversary's view when she talks to the network is indistinguishable from her view when she talks to the simulator.)*

$$\mathcal{E}A\text{-VIEW}_k^P(\vec{x}) \stackrel{\mathcal{C}}{\simeq} \mathcal{E}A\text{-VIEW}_k^{S,\mathcal{AI},f}(\vec{x}).$$

- CORRECTNESS: *(The network output almost always coincides with f evaluated at the network input.) There is a negligible function $\epsilon(k)$ such that for all $\vec{x} \in (\{0,1\}^\ell)^n$,*

$$\text{Prob}\left[\overline{\mathcal{AO}}(\vec{x}, 1^k, \vec{r}, r_A) \neq f(\overline{\mathcal{AI}}(\vec{x}, 1^k, \vec{r}, r_A))\right] \leq \epsilon(k),$$

where the probability is taken over all possible $\vec{r}$ and r_A.

The privacy constraint asserts indistinguishability of ensembles over $\mathcal{C} = \{L_k = (\{0,1\}^\ell)^n\}$. The correctness constraint says that almost never does a *good* player i compute something other than $f_i(\vec{x}')$, or a bad player j not "know" $f_j(\vec{x}')$, where $\vec{x}'$ is the network input.

3.4.2 Remarks

EXPLANATION. Definition 3.10 can be explained as follows. For this explanation, we regard simulation as a game the adversary plays "in her head," the purpose of which is to construct some view.

- *Privacy.* When the adversary talks to the network, the players' inputs being $\vec{x}$, what she sees is something which she can generate for herself, given just an ideal-evaluation oracle, $\Omega_f(\vec{x})$. In fact, with the aid of such an oracle, the adversary can simulate, in her subconscious, a *virtual network*, interacting with which is just like interacting with the actual network.

- *Correctness.* When the adversary interacts with this virtual network, on each execution she must make an output query to the oracle. What this output query is, and when she makes it, is determined by a function $\mathcal{AI}$. Now, if the adversary speaks to an *actual* network in lieu of the virtual network, the same function $\mathcal{AI}$ *still* indicates what the "output query is", and "when it would be made." Of course, there is really no output query being made—but we take the value of $\mathcal{AI}$ as defining what the adversary has *committed to,* and when.

  Correctness says that if the adversary has committed to a value x'_T, and the initial input was $\vec{x}$, then, almost certainly, each good player i will compute $f_i(x'_T \cup x_{\overline{T}})$. Additionally, the adversary can compute a value $f_j(x'_T \cup x_{\overline{T}})$ on behalf of any bad player j. She does this simply by evaluating the adversary output function, $\mathcal{AO}$.

ADVERSARIAL AWARENESS. We emphasize that not only is there a notion of *what* the adversary has substituted in for the input values of the bad players, and *when* she has done it, but, what is more, the adversary is *aware* of what this substituted value is, and when this substitution took place. This is called *input awareness.* One could, instead, have defined the adversary's input in a way which did not require such awareness—but doing this would represent a failure to mimic an important aspect of the ideal evaluation.

Besides input awareness, the adversary is aware of her share of the output—the output which bad players are entitled to, even though they are bad. This is called *output awareness.*

Adversarial awareness—meaning both input awareness and output awareness—are essential ingredients in obtaining the *reducibility* result discussed in Section 5.5.

WHY ON-LINE SIMULATABILITY?

BLENDING PRIVACY AND CORRECTNESS. We might have made separate definitions of privacy and correctness, and then said that a secure protocol is one simultaneously meeting both of them. Instead of this, our definition *blends* privacy and correctness in a deeper manner. Given that this is done implicitly in Definition 3.10 we explicitly say it here: the simulator—crucial in defining privacy—specifies the adversary input function, $\mathcal{AI}$, and therefore the network input function it determines, $\overline{\mathcal{AI}}$. The correctness constraint depends on this network input, and thus on the notion of privacy and the simulator which establishes it. In this way, we "weave" correctness into privacy, the same simulator used for defining privacy being used to define correctness, too.

DECOUPLED PRIVACY AND CORRECTNESS. Since our method of blending of privacy and correctness is a somewhat delicate admixture of these objectives, it is important to explain what is wrong with *decoupled security*—saying that a secure protocol is one simultaneously meeting disjoint requirements for privacy and correctness. In fact, decoupled security can lead to various "embarrassing" problems can result! Let us be more concrete, even if somewhat informal.

For concreteness, when we say "privacy alone" we mean the notion of t-private leaking f, as given by Definition 3.9; and when we say "correctness alone" we mean the notion of t weakly computing f, as given by Definition de-weakcorrecntess. Pleasantly, however, our criticism of decoupled notions of security is immune to variants in definitional choices for privacy and correctnes. We emphasize, for decoupled security, the function $\mathcal{AI}^C$ used for correctness need not be linked or related in any manner to the function $\mathcal{AI}^P$ used for privacy.

Decoupled security. What is wrong with decoupled security? An example from our work with Kilian is useful for answering this question [KMR90].

Suppose there are three players, each player i having a private input $x_i \in \{0,1\}^\ell$. Let $x_i[1{:}\ell-1]$ denote x_i stripped of its last bit. Consider the following function f that the players may wish to jointly compute. For each $i \in \{1,2,3\}$, define

$$f_i(x_1,x_2,x_3) = \begin{cases} x_1[1{:}\ell-1] & \text{if the last bit of } x_2 \text{ is a 0, and} \\ x_2[1{:}\ell-1] & \text{if the last bit of } x_2 \text{ is a 1.} \end{cases}$$

That is, each player is to learn the *same* string, either the private input of the first player or second player, as determined by the parity of the second player's input.

Consider the following protocol, P_1.

PROTOCOL P_1:

Step 1. *Player 1 announces his input,* x_1.

Step 2. *Player 2 announces his output,* y_2.

Players 1 and 3 take y_2 *this as their output, too.*

We claim that *protocol P_1 is 1-private leaking f*. Very roughly, the argument is as follows. The main concern is about Player 2; for simplicity, consider the case that the adversary corrupts him right off. Now privacy requires the existence of a simulator $S = (\mathcal{S}, \mathcal{AI})$. But if we define the adversary input function as being identically 0, then the adversary learns x_1 right away from its ideal-evaluation oracle. Using this value, the simulator can easily interact with the adversary so as to give to her a view from the right distribution.

Likewise, *protocol P_1 1-weakly computes f*. Very roughly, the argument is as follows. The main concern is about Player 2; for simplicity, consider the case that the adversary corrupts him right off. If we define the adversary input as consisting of $y_2 1$ once it is announced in Step 2, and we define the output function as being y_2, then the protocol computes a network output which is f applied to the network inputs.

What went wrong? Clearly protocol P_1 is not a "good" protocol. For in the ideal evaluation of f, a bad Player 2 has two disjoint options: he can *either* learn Player 1's input (by choosing an even fake input), *or* he can control Player 1's output (by choosing the desired odd input). In protocol P_1, Player 2 can can *both* see Player 1's input *and* control Player 1's output.

Input independence. A first explanation for this failure might be that the protocol P_1 was bad because it failed to achieve *input independence*: in particular, a "fake" value of x_2 chosen by a bad Player 2 might depend on the private input of Player x_1. For he who shares our point of view —that a secure evaluation should mimic as closely as possible an ideal evaluation— input independence must be either an explicit requirement of a definition of security, or (as in our case) a consequence of it; for independence *exists* in an ideal evaluation, as the the adversary has no idea of the inputs of uncorrupted players when she chooses her substitutes for them.

Realizing that a formalization of privacy and correctness, taken together, might not imply independence, several researchers have considered independence as an additional requirement to add along.

Security $\neq$ privacy + correctness + input independence. Though formalizing input independence in a robust and satisfying way is not trivial, we shall not need to in order to make our point: that disjointly requiring privacy, correctness, and independence is still inadequate to properly capture the idea of security. Let us demonstrate this by modifying our counterexample.

Suppose, once again, that there are three players, each player i having a private input $x_i \in \{0,1\}^\ell$. Consider the following function g that they may wish to jointly compute. For each $i \in \{1,2,3\}$, define

$$g_i(x_1, x_2, x_3) = \begin{cases} 1x_1 & \text{if the last bit of } x_2 \text{ is a 0, and} \\ 2x_2 & \text{if the last bit of } x_2 \text{ is a 1.} \end{cases}$$

That is, each player is to learn the *same* string, either that belonging to the first player or second player, as determined by the parity of the second player's input. The string which is released is tagged by the identity of the one who contributed it.

We sketch a protocol P_2 below. Though somewhat artificial, other, "natural" protocols might suffer from the same defect which this one highlights.

Our protocol makes use of a *committal* protocol. Roughly, this is a way to pin down an input while simultaneously keeping it secret. At a later stage, a *decommittal* protocol can be used to globally reveal exactly what was committed during the committal stage. References for this notion of a committal and decommittal protocol (the ingredients of a verifiable secret sharing) appear in the last footnote.

PROTOCOL P_2:

Step 1. *Each player i commits to his private input, x_i.*
Step 2. *The last bit of x_2 is decommitted.*
Step 3. *If it is a* 0, *x_1 is decommitted.*
If it is a 1, *x_2 is decommitted.*
Each party i outputs $g_i(x_1, x_2, x_3)$.
Step 4. *Player* 1 *tells Player* 2 *his input, x_1.*

Though we shall not formally argue it, protocol P_2 is private leaking g, correctly computes g, and achieves input independence. Still, the protocol is certainly wrong. The "mistake" is the inclusion of Step 4, which gives Player 2 information he does not deserve when x_2 ends in a 1.

The fundamental problem illustrated by this example is that privacy and correctness can be *competing* requirements, and a decoupled definition of security does not respect the possible interaction of these two competing goals.

MODEL OF COMPUTATION. Our definition implicitly fixes the model of computation to be that described—for example, the exposed state function, E, gives the current-round view (that is, "the standard model"). Other models of computation can be specified, each "automatically" giving rise to its own notion of security. For example, information-theoretic security, from scratch, in the complete-history model, is exactly as Definition 3.10 gives, except that the exposed-history function, E, implicit in the definition, is the identity map.

3.5 Incorporating History

We now drop the first simplifying assumption listed in Section 1.7. Namely, we wish to take account of the fact that the players and the adversary may have a *history* dating back before the execution of the current protocol. Because of this history, a player may initially possess more information than just his private input, and an adversary may initially possess information correlated to the players' inputs. We must capture the intuition that even in the presence of such extra information, the adversary can no more compromise a secure protocol than she can compromise the ideal evaluation which takes account of the extra information. For example, even if the adversary already knows the private inputs of all the odd-numbered players, and even if each even-numbered player $i < n-2$ already knows the private input of player $i+2$, still, a protocol should leak nothing more than it is "supposed to" under these circumstances.

A HISTORY OF HISTORY. Concern that protocols be robust against such additional information in the hands of its agents has its roots in the *auxiliary input* provided in the definition of a zero-knowledge proof systems. Auxiliary input is more than a "trick" to permit basic composition properties: it is an addition which mirrors the refined intuition.

At the very beginning, the need for auxiliary input in a robust formalization of zero-knowledge was unnoticed by its creators, Goldwasser, Micali and Rackoff. They quickly discovered this issue included it in their paper setting forth the notion of zero-knowledge [GMR89]. Independently, Tompa and Woll [TW87] and Oren [Or87] realized that a good definition of zero-knowledge needs to have auxiliary input.

MODELING HISTORY. Everyone needs a sense of history. Indeed, the *adversary* may initially possess certain information, and each *player* may have information associated to him at a protocol's inception—this information becoming available to the adversary when she corrupts him. Mirroring

this, history is modeled as *advice* to the adversary, and as *auxiliary input* for each player. We remark that in our context, "auxiliary input" is a new technical term, different from the usage mentioned above, in the context of zero-knowledge proof systems.

In the upcoming subsections, we first expand our explication of the auxiliary inputs. Then we explain adversary advice, and formally specify how to run a protocol in the presence of an adversary when the players have auxiliary inputs and the adversary has advice. In Section 3.5.3 we update our definition of security, giving a more refined notion for information-theoretic secure function evaluation than was Definition 3.10.

3.5.1 Auxiliary inputs

Underlying our notion of security from scratch is the idea that each player i comes into a collaborative computation having no past and with but one destiny—to compute some $f_i(x_1, \ldots, x_n)$. With this in mind, we modeled the initial state of a player i as containing only his private input and his identity, $s_i^{00} = x_i \sharp i$. We said that when the adversary corrupts player i in a round r she learns an exposed state σ_i^r containing some piece of i's history and the messages he is about to receive.

To be more general, corrupting a player should be more rewarding. When the adversary corrupts a player i, she is handed some arbitrary information a_i associated to him. This information might represent the history of that player before the execution began; or, it might be information irrelevant to the current task at hand (as in the "saved state" in a subroutine call) not modeled by the player's computational state.

PROTOCOLS RUNNING WITH ADVERSARIES. To incorporate this change into our model, to each player i we associate not only his private input, x_i, but also his *auxiliary input*, a_i. A player i cannot see his auxiliary input—only the adversary, when she corrupts him, is given it. By denying a player access to his auxiliary input we reflect the intuition that this is information which a good player does not need to use for performing the current computational task.

Executing a protocol in the presence of an adversary now requires specifying the private input $\vec{x}$, the auxiliary input $\vec{a}$, and the common input c. For simplicity, all auxiliary inputs are taken to be of the *same* length, which we denote by m. It is convenient to demand that this value appear, in unary, on the common input: say $c = 1^k \# 1^m$.

When a processor i is corrupted by the adversary in round r, the exposed state she receives is re-defined as

$$\sigma_i^r = \begin{cases} s_i^{r\infty} * x_i * a_i & \text{if } r = 0 \\ s_i^{r\infty} * x_i * a_i * \bar{m}_{1i}^r * \cdots * \bar{m}_{ni}^r * E(H_i^r) & \text{if } r > 0 \end{cases} \qquad \text{"The exposed state of processor } i \text{ at round } r. \text{"}$$

where, recall, $\bar{m}_{ji}^r$ is the round-r message prepared by j for i, and E is the exposed-state function. We re-define the configuration of a player to include the his auxiliary input; thus a player configuration is an element $(s_i,\ r_i,\ a_i)$ of

$$\underbrace{\{0,1\}^*}_{\substack{\text{player's} \\ \text{state}}} \times \underbrace{\{0,1\}^\infty}_{\substack{\text{future} \\ \text{coins}}} \times \underbrace{\{0,1\}^\infty}_{\substack{\text{future} \\ \text{advice}}}$$

SIMULATORS RUNNING WITH ADVERSARIES. To devise a history-sensitive notion of security we must take account of auxiliary inputs in our notion of the ideal evaluation. Namely, in the ideal evaluation of f on input $\vec{x}$ and auxiliary input $\vec{a}$, when the adversary corrupts a player i she learns

(x_i, a_i) if she has not yet substituted in her fake inputs. If she has substituted in these fake inputs and they were x'_T, then the adversary receives $((x_i, a_i),\ y_i)$, where $\vec{y} = f(x'_T \cup x_{\overline{T}})$.

The changes to the rules in the ideal evaluation are reflected in the definition of the ideal-evaluation oracle. First we change the notation to $\mathcal{O}_f(\vec{x}, \vec{a})$, explicitly indicating the dependency of the oracle on the auxiliary input. The oracle behaves as follows: to a component query of i, if the valid function query has not yet been made then the oracle responds with (x_i, a_i); while if the valid function query x'_T has already been made and $\vec{y} = f(x'_T \cup x_{\overline{T}})$, the oracle responds $((x_i, a_i),\ y_i)$.

The notion of simulatability is unchanged, except, of course, that the oracle given to the simulator is now an $O_f(\vec{x}, \vec{a})$-oracle, and the execution of a simulator with an adversary depends on the auxiliary input $\vec{a}$.

3.5.2 Adversary advice

The adversary's advice, denoted a_A, can be thought of as information the adversary has come to know, somehow, before the protocol begins running. As shall be clear when we formalize security, the adversary's advice may depend in an arbitrary way on the common input, the players' inputs, and their auxiliary inputs. But it cannot depend on the random coin sequences which are issued to the players.

(When we describe the complexity-theoretic notion of security in Section 5, the adversary's advice will play an additional role: it will model *nonuniformity* which may be available to the adversary. See Section 4.3 for some discussion of this.)

We model the adversary's advice as an infinite string, which the adversary consumes bit-by-bit, in exactly the same manner as she consumed her random coin tosses. When a bit from a_A is read, it vanishes from a_A and augments the adversary's computational state.

To implement these ideas, we modify an adversary's next-action function to allow her to get a single bit of advice from a_A. The adversary is still permitted to perform all the actions she was previously permitted. Thus the adversary's next-action function is now a syntactic function with the following domain and (expanded) range:

$$\tilde{\mathcal{N}} : \underbrace{\{0,1\}^*}_{\substack{\text{current}\\ \text{state}}} \to \{\mathsf{compute}, \mathsf{flip\text{-}coin}, \mathsf{get\text{-}advice}, \mathsf{corrupt}_1, \ldots, \mathsf{corrupt}_n, \mathsf{round\text{-}done}\}.$$

We must also augment the notion of an adversary configuration, adding a component for the unconsumed portion of her advice. Thus an adversary configuration $(s_A, r_A, a_A, \kappa_A)$ is an element of the set

$$\underbrace{\{0,1\}^*}_{\substack{\text{adversary's}\\ \text{state}}} \times \underbrace{\{0,1\}^\infty}_{\substack{\text{future}\\ \text{coins}}} \times \underbrace{\{0,1\}^\infty}_{\substack{\text{future}\\ \text{advice}}} \times \underbrace{2^{[1..n]}}_{\substack{\text{corrupted}\\ \text{players}}}.$$

CONFIGURATION SEQUENCES. Let A be an adversary for an n-party protocol, and let P be such a protocol. We describe how, from any common input c, player inputs $\vec{x}$, initial player configurations $(C_1^{00}, \ldots, C_n^{00})$, initial adversary configuration $C_A^{00} = (s_A, r_A, a_A, \kappa_A)$, and exposed-state function E, protocol P and adversary A generate a sequence of player configurations, $\{C_i^{r\rho}\colon\ i \in [1..n], r \in [0..R], \rho \in \mathbf{N}\}$, and a sequence of adversary configurations, $\{C_A^{r\rho}\colon\ r \in [0..R], \rho \in \mathbf{N}\}$.

With all notation as before, the players' computational states progress according to

$$C_i^{r(\rho+1)} = \begin{cases} (P(c, s_i^{r\rho}),\ r_i^{r\rho},\ a_i^{r\rho}) & \text{if } \mathcal{N}(s_i^{r\rho}) = \textsf{compute} \\ (s_i^{r\rho} * r_{i1}^{r\rho},\ r_{i2}^{r\rho} r_{i3}^{r\rho} \cdots,\ a_i^{r\rho}) & \text{if } \mathcal{N}(s_i^{r\rho}) = \textsf{flip-coin} \\ C_i^{r\rho} & \text{if } \mathcal{N}(s_i^{r\rho}) = \textsf{round-done} \end{cases}$$

$$C_i^{(r+1)0} = (s_i^{r\varrho_{ir}} * \mathcal{M}_i^r,\ r_i^{r\varrho_{ir}},\ a_i^{r\varrho_{ir}})$$

while the adversary's sequence of configuration is given by

$$C_A^{r(\rho+1)} = \begin{cases} (A(c, s_A^{r\rho}),\ r_A^{r\rho},\ a_A^{r\rho},\ \kappa_A^{r\rho}) & \text{if } \tilde{\mathcal{N}}(s_A^{r\rho}) = \textsf{compute} \\ (s_A^{r\rho} * r_{A1}^{r\rho},\ r_{A2}^{r\rho} r_{A3}^{r\rho} \cdots,\ a_A^{r\rho},\ \kappa_A^{r\rho}) & \text{if } \tilde{\mathcal{N}}(s_A^{r\rho}) = \textsf{flip-coin} \\ (s_A^{r\rho} * a_{A1}^{r\rho},\ r_A^{r\rho},\ a_{A2}^{r\rho} a_{A3}^{r\rho} \cdots,\ \kappa_A^{r\rho}) & \text{if } \tilde{\mathcal{N}}(s_A^{r\rho}) = \textsf{flip-coin} \\ (s_A^{r\rho} * \sigma_i^r,\ r_A^{r\rho},\ a_A^{r\rho},\ \kappa_A^{r\rho} \cup \{i\}) & \text{if } \tilde{\mathcal{N}}(s_A^{r\rho}) = \textsf{corrupt}_i \\ C_A^{r\rho} & \text{if } \tilde{\mathcal{N}}(s_A^{r\rho}) = \textsf{round-done} \end{cases}$$

$$C_A^{(r+1)0} = \begin{cases} (s_A^{r\varrho_{Ar}} * \dot{\mathcal{M}}_A^{r+1},\ r_A^{r\varrho_{Ar}},\ a_A^{r\varrho_{Ar}},\ \kappa_A^{r\varrho_{Ar}}) & \text{if } r > 0 \\ C_A^{r\varrho_{ir}} & \text{if } r = 0 \end{cases}$$

with the exposed state σ_i^r as specified on Page 46.

EXECUTING PROTOCOLS. An *initial configuration of the network* is now identified by a tuple $(\vec{x}, \vec{a}, a_A, c,\ \vec{r}, r_A)$, specifying the players' *private inputs* $\vec{x} \in (\{0,1\}^\ell)^n$, their *auxiliary input* $\vec{a} \in (\{0,1\}^m)^n$, the *adversary's advice* $a_A \in \{0,1\}^\infty$, the *common input* $c = 1^k \# 1^m$, the *players' coins* $\vec{r}$, and the *adversary's coins* r_A. For a given protocol P and adversary A, an initial configuration of the network determines an *execution* of P in the presence of A. This execution is the sequence of configurations $\{C_i^{r\rho}, C_A^{r\rho}\}$ generate by P and A when the initial configuration of party i is taken to be $C_i^{00} = (x_i \| i,\ r_i)$, the initial configuration of A is $(\kappa_A,\ r_A,\ a_A\ \kappa_A)$, for $\kappa_A = \emptyset$, player i's coins are r_i, the adversary's coins are r_A, the common input is c, and the players' inputs are $\vec{x}$.

3.5.3 Security (including history)

VALID INITIAL CONFIGURATIONS. Not all tuples $(\vec{x}, \vec{a}, a_A, c,\ \vec{r}, r_A)$ denote valid initial configurations. In fact, under the conventions we have specified, the only initial configurations allowed are those having their first four components given by $\bigcup_k L_k$, where

$$L_k = \bigcup_{n \geq 3,\ \ell, l, m \geq 0} \{(\vec{x}, \vec{a}, a_A, c) :\ \vec{x} \in (\{0,1\}^\ell)^n,\ \vec{a} \in (\{0,1\}^m)^n,\ a_A \in \{0,1\}^\infty, \text{ and } c = 1^k \# 1^m\}.$$

Here, L_k denotes the valid tuples $(\vec{x}, \vec{a}, a_A, c)$ having security parameter k.

ENSEMBLES FOR SECURE FUNCTION EVALUATION. Given a fixed-round, n-party protocol P and an adversary A, an initial network configuration $(\vec{x}, \vec{a}, a_A, c,\ \vec{r}, r_A)$ determines the an execution,

and a corresponding adversary's view. If $\vec{r}$ and r_A are allowed to vary, then there is an induced distribution on these adversary views, which we denote by $A\text{-VIEW}_k^P(\vec{x}, \vec{a}, a_A, c)$. If $(\vec{x}, \vec{a}, a_A, c)$ are allowed to vary as well, then we obtain an ensemble $\mathcal{E}A\text{-VIEW}_k^P(\vec{x}, \vec{a}, a_A, c)$ over $\mathcal{L} = \{L_k\}$.

Similarly, fix a function $f : (\{0,1\}^t)^n \to (\{0,1\}^l)^n$, an adversary, A, and a simulator, S, using an adversary input function $\mathcal{AI}$. Then each point $(\vec{x}, \vec{a}, a_A, c) \in L'_k$, taken together with a set of adversary and simulator coins (r_A, r_S), determine an associated adversary view. If r_A and r_S are allowed to vary, then there is an induced distribution on these views, which we denote by $A\text{-VIEW}_k^{S,f}(\vec{x}, \vec{a}, a_A, c)$. If $(\vec{x}, \vec{a}, a_A, c)$ are allowed to vary as well, then we obtain an ensemble $\mathcal{E}A\text{-VIEW}_k^{S,\mathcal{AI},f}(\vec{x}, \vec{a}, a_A, c)$ over $\mathcal{L} = \{L_k\}$.

NETWORK INPUT AND OUTPUT. For a given adversary A and protocol P, The adversary input and output formerly gave rise to functions $\mathcal{AI}(\vec{x}, c, \vec{r}, r_A)$ and $\mathcal{AO}(\vec{x}, c, \vec{r}, r_A)$. Since our initial configurations are now somewhat richer, the awareness functions now give rise to functions $\mathcal{AI}(\vec{x}, \vec{a}, a_A, c, \vec{r}, r_A)$ and $\mathcal{AO}(\vec{x}, \vec{a}, a_A, c, \vec{r}, r_A)$, specifying the adversary's input and output that results from the given initial configuration. These function, in turn, give rise to functions $\overline{\mathcal{AI}}(\vec{x}, \vec{a}, a_A, c, \vec{r}, r_A)$ and $\overline{\mathcal{AO}}(\vec{x}, \vec{a}, a_A, c, \vec{r}, r_A)$, exactly as before.

SECURE FUNCTION EVALUATION. Using the language just introduced, we are now ready to define what it means for a protocol P to securely evaluate a finite function f, ignoring issues of computational complexity.

Definition 3.11 *Let $f : (\{0,1\}^t)^n \to (\{0,1\}^l)^n$ be a function, and let P be an fixed-round, n-party protocol. We say that P* **statistically t-securely computes f, ignoring complexity,** *if there is a simulator S, a t-admissible adversary input function $\mathcal{AI}$, and and an adversary output function $\mathcal{AO}$, such that for any t-adversary A,*

- *Privacy:* $\mathcal{E}A\text{-VIEW}_k^P(\vec{x}, \vec{a}, a_A, c) \underset{\mathcal{L}}{\simeq} \mathcal{E}A\text{-VIEW}_k^{S,\mathcal{AI},f}(\vec{x}, \vec{a}, a_A, c)$.
- *Correctness: There is a negligible function $\epsilon(k)$ such that for all $(\vec{x}, \vec{a}, a_A, c) \in L_k$,*

$$\text{Prob}\left[\overline{\mathcal{AO}}(\vec{x}, \vec{a}, a_A, c, \vec{r}, r_A) \neq f(\overline{\mathcal{AI}}(\vec{x}, \vec{a}, a_A, c, \vec{r}, r_A))\right] \leq \epsilon(k),$$

where the probability is taken over all possible $\vec{r}$ and r_A.

3.5.4 Remarks

ON-LINE ADVICE. We remark that more general notions of adversary advice are possible than the one we have described here. For example, one could regard the adversary as being entitled to obtain information in installments, new information becoming available to her at each round and with each corruption of a player. The information she is entitled to in a given round or following a given corruption would be a string computable by an explicitly given, efficiently computable probabilistic map on the adversary's current computational state, the common input c, the input vector $\vec{x}$, and the auxiliary input $\vec{a}$. Though somewhat unintuitive, when embedded in the notion of a secure protocol, demanding robustness against such *on-line advice* appears to be a stronger notion of security. Though on-line advice may be necessary for a theory of secure protocols robust against running multiple protocols concurrently, we have chosen to stick with the simpler notion of an infinite adversary advice string, and finite auxiliary strings for each player, for reasons of ease of explication.

3.6 Incorporating Variability

We now drop the second, third, and fourth simplifying assumption listed in Section 1.7. Namely, we no longer imagine that the number of players be fixed, that our functions be finite functions, and that a protocol run in a number of rounds $R = R(c)$ which is independent of the random choices made during its execution.

After describing each relaxation in separate subsections, we give a more refined definition of secure function evaluation than was Definition 3.11. It still ignores issues of computational complexity, but does not imagine that protocols are run, "from scratch," by some fixed number of players, in some fixed number of rounds, their goal being to evaluate a fixed finite function.

3.6.1 Variable number of players

GENERAL PROTOCOLS. To properly talk about general multiparty protocols we must relax the requirement that a protocol is tailored for one particular number of players. Thus we consider a (general) protocol P to be a *family* of n-party protocols, $P = \{P_n\}$. However, recall that we demanded that a protocol at least be reasonable to *describe* (that is, Turing-computable); in passing to general protocols, we would like not to lose this property. In fact, any "reasonable" protocol should have a description that is efficiently computable knowing the number of players involved. We demand this in our definition below.

Definition 3.12 *A* protocol P *is a polynomial-time computable function that maps a number* 1^n *to a standard encoding of an* n*-party protocol* P_n.

(A *nonuniform protocol* fails to meet the "polynomial time" constraint above.) We suppress the subscript n when considering an n-party protocol P_n, using P to denote either a general protocol, or the particular n-party protocol that it specifies, when n is understood.

Now that the number of players may vary, we will always specify it on the common input. We briefly postpone describing its syntax.

GENERAL ADVERSARIES. To talk about a general protocol under attack by an adversary we must suitably relax our notion of an adversary, too. In particular, we consider an adversary to be a collection of agents, the appropriate agent selected according to the number of players. As there was no requirement on the computability of the adversary's strategy, there is no computability requirement in determining the collection of functions which constitute an adversary.

Definition 3.13 *An* adversary A *is a function that maps a number* 1^n *to an* n*-party adversary* A_n.

Once again, we suppress the subscript n when considering an adversary for an n-party protocol A_n, using A to denote either a general adversary, or the n-party adversary A_n that it specifies, when n is understood.

3.6.2 Function families

WHY FUNCTION FAMILIES? Though the secure evaluation of finite functions is not a trivial matter, it is not general enough. For example, dealing exclusively with such functions does not let us discuss "a secure computation for the mean salary"—since the number of players and how big their salaries might be has not yet been specified. To deal with such computational tasks, and to realize the full power of a dynamic adversary, we consider secure computations of *families* of finite functions. Our formalization of these objects constitutes one way to deal with infinite domains (rather than

$(\{0,1\}^\ell)^n$) while insisting, in any given execution, that the potential domain be finite. See the discussion at the end of this chapter for an explanation of why this is necessary.

FUNCTION FAMILIES. Recall that a finite function is a map $f_{n\ell l} : (\{0,1\}^\ell)^n \to (\{0,1\}^l)^n$, for some fixed values of n, ℓ, and l. Our notion of distributively computing "arbitrary" functions is to use the common input to specify what finite function the current protocol is supposed to compute. In fact, a *function family* f is precisely a way to interpret each common input $c = 1^k\#1^n\#1^\ell\#1^l\#1^m\#C$ as a map $f_c : (\{0,1\}^\ell)^n \to (\{0,1\}^l)^n$. That is,

Definition 3.14 *A* **function family** f *is a map from strings to finite functions such that when* $c = 1^k\#1^n\#1^\ell\#1^l\#1^m\#C$, *for some* $C \in \{0,1\}^*$, *the finite function* $f_c \stackrel{\text{def}}{=} f(c)$ *is a map* $f_c : (\{0,1\}^\ell)^n \to (\{0,1\}^l)^n$.

This viewpoint of what a function family is has two pleasant features. First, it expunges nonuniformity issues from consideration: if you want to compute some function, it should be described on the common input, the complexity of producing such common inputs being irrelevant. Second, the viewpoint emphasizes that the *encoding* of a function is highly relevant to its computation: for example, it may be possible to securely compute a function under one encoding, but impossible under some other.

As a consequence, the claim that a protocol securely computes a function family f must explicitly fix *how* each common inputs c encodes its finite functions f_c. On the other hand, a definition of the secure computation of f need not fix how common inputs encode functions; all that is important is the relationship between the behavior of the protocol when the common input is c to the function f_c.

UNIVERSAL PROTOCOLS. Suppose the function family f is as follows: when the common input is $c = 1^k\#1^n\#1^\ell\#1^l\#1^m\#C$, and C encodes a Boolean circuit from $(\{0,1\}^\ell)^n$ to $(\{0,1\}^l)^n$, then f_c denotes the mapping described by this circuit. If C does not properly encode such a circuit, then f_c is some particular constant map.

In this case, if there is a secure protocol to compute f then it is *universal,* in the sense that any family of finite functions has a natural encoding as a family of circuits. Other encodings are equally good and equally universal: C might be taken as a description of a Turing machine program, say.

The existence of universal protocols will be discussed in Section XXX.

3.6.3 Variable number of rounds

In the ideal evaluation of a function f, one imagines that the computation takes a fixed amount of time, and therefore the players and the adversary know when the computation is over. Though we do not regard taking a fixed amount of time as being an important feature of the ideal evaluation, we do regard *termination awareness* as an important feature of it.

Therefore, we relax the constraint that the number of rounds be fixed at some value R, depending, if at all, only on the common input. Instead, we allow protocols to take a number of rounds which may vary with the random choices made in the execution.

Since we wish to capture that all the participants should "know" when the computation is over, we might as well require that the computation be *jointly terminating*—that is, all good players halt within the space of a single player round. But this requirement alone would not be quite enough—because the adversary, too, should know when termination occurs. We can ensure that she does very simply, by demanding that termination be a function of the traffic of broadcasts.

We describe below, without undo formalism, the changes used to implement these ideas.

REQUIRED CHANGES. We leave unchanged the notion of a protocol and adversary, but add a single player interaction function: a terminated function, τ. In defining the execution of a protocol in the presence of the adversary, τ is applied to the vector of broadcast messages computed at the completion of a player round. When it evaluates to 1, the last round of player activity has just been completed, and the protocol terminates after the following round of adversary activity. At the end of this round, τ must still evaluate to 1, regardless of what the adversary did.

A protocol *diverges* if it never terminates, due either to an infinite sequence of micro-rounds, or to τ's failure to signal termination. An *admissible* protocol never diverge, regardless of starting configuration. A protocol for secure function evaluation must be admissible.

An R-round execution is one which terminates at round R. Adversary views include only the sequence of computational states up until termination. In defining the adversary output and the network output, the adversary output function, $\mathcal{AO}$, is applied to the adversary's final traffic, while the players' output function, o, is applied to each player's final computational state.

In defining the interaction of an an adversary with a simulator, the simulator and the adversary together determine the broadcasts. The interaction function τ can be applied to these broadcasts. Therefore, termination is an equally meaningful condition applied to adversary-simulator interactions as it was applied to adversary-network interactions, and quantities like the adversary view produced from conversations with the simulator are well-defined. An adversary-simulator interaction *diverges* if it does not terminate. We require that a simulator for secure function evaluation never diverges, regardless of the adversary with whom it interacts.

WHEN BROADCAST IS NOT AVAILABLE. The notion above must be modified to be meaningful if broadcast is not permitted. In this case, a secure protocol must come equipped with a function τ which, applied to player i's traffic T_i^r, evaluates to either protocol-done or protocol-not-done. Applied to the adversary's traffic T_A^r, it also evaluates in this range. In an execution of a protocol with an adversary, these functions all evaluate to protocol-not-done—until the round at which they all evaluate to protocol-done. The function τ can then be used as above.

In the absence of broadcast, what is outlined above might be considered too much to expect, because of the impossibility of achieving constant-round secure protocols which it would entail. To avoid this, we can relax the definition further, and require only *approximate-termination awareness*: namely, each player, and the adversary as well, must know when the protocol has terminated *only to within a certain number of rounds, δ*. This amount can be considered as a fixed polynomial of parameters specified by c, say. In an execution of a protocol with an adversary, these functions all evaluate to protocol-not-done and then to protocol-done—except for a "window" of up to δ rounds, during which some may evaluate to protocol-not-done and others to protocol-done. The value of τ can now be used as above.

3.6.4 Security (including history and variability)

VALID INITIAL CONFIGURATIONS. Not all tuples $(\vec{x}, \vec{a}, a_A, c, \vec{r}, r_A)$ denote valid initial configurations. In fact, under the conventions we have specified, the only initial configurations allowed are those having their first four components given by $\bigcup_k L_k$, where

$$L_k = \bigcup_{n\geq 3,\ \ell,l,m\geq 0} \{(\vec{x}, \vec{a}, a_A, c) :\ \vec{x} \in (\{0,1\}^\ell)^n,\ \vec{a} \in (\{0,1\}^m)^n,\ a_A \in \{0,1\}^\infty, \text{ and } c = 1^k\#1^n\#1^\ell\#1^l\#1^m\#C, \text{ for } C \in \{0,1\}^*\}.$$

Here, L_k denotes the valid tuples $(\vec{x}, \vec{a}, a_A, c)$ having security parameter k.

ENSEMBLES AND AWARENESS FUNCTIONS. For any protocol P, adversary A, function family f, and simulator S, we have associated ensembles

$$\mathcal{E}A\text{-VIEW}_k^P(\vec{x},\vec{a},a_A,c) \quad \text{and} \quad \mathcal{E}A\text{-VIEW}_k^{S,f}(\vec{x},\vec{a},a_A,c)$$

over $\mathcal{L} = \{L_k\}$, defined exactly mirroring the definitions of the corresponding ensembles of Section 3.5.3.

For any protocol P and adversary A, each initial network configuration $(\vec{x},\vec{a},a_A,c,\ \vec{r},r_A)$ determines $\mathcal{AI}(\vec{x},\vec{a},a_A,c,\ \vec{r},r_A)$ and $\mathcal{AO}(\vec{x},\vec{a},a_A,c,\ \vec{r},r_A)$, exactly as before. These, in turn, give rise to function $\overline{\mathcal{AI}}(\vec{x},\vec{a},a_A,c,\ \vec{r},r_A)$ and $\overline{\mathcal{AO}}(\vec{x},\vec{a},a_A,c,\ \vec{r},r_A)$.

SECURE FUNCTION EVALUATION. We now revise our definition of secure function evaluation, still ignoring complexity issues but taking account of the extensions from this and the previous section.

Definition 3.15 *Let $f = \{f_c\}$ be a family of finite functions, and let P be a protocol. We say that P* **statistically t-securely computes** *f,* **ignoring complexity,** *if there is a simulator S, a t-admissible adversary input function $\mathcal{AI}$, and an adversary output function $\mathcal{AO}$, such that for any t-adversary A,*

- *Privacy:* $\mathcal{E}A\text{-VIEW}_k^P(\vec{x},\vec{a},a_A,c) \underset{\mathcal{C}}{\simeq} \mathcal{E}A\text{-VIEW}_k^{S,\mathcal{AI},f}(\vec{x},\vec{a},a_A,c)$.
- *Correctness: There is a negligible function $\epsilon(k)$ such that for all $(\vec{x},\vec{a},a_A,c) \in L_k$,*

$$\text{Prob}\left[\overline{\mathcal{AO}}(\vec{x},\vec{a},a_A,c,\ \vec{r},r_A) \neq f(\overline{\mathcal{AI}}(\vec{x},\vec{a},a_A,c,\ \vec{r},r_A))\right] \leq \epsilon(k),$$

where the probability is taken over all possible $\vec{r}$ and r_A.

3.6.5 Remarks

VARIABLE NUMBERS OF PLAYERS. Varying the number of players is more than a nicety to neatly handle functions on larger domains: it is a necessary relaxation to our notions if we are to realize the true power of our adversaries.

Roughly said, for any *fixed* number of players, security with respect to the class of "powerful" (*dynamic*) adversaries we have described is equivalent to security with respect to the class of "weak" (*static*) adversaries who behave as follows: they choose the set of players they will corrupt at the beginning of the protocol, ***before*** any players are corrupted; then they corrupt exactly this set of players. The equivalence of static and dynamic adversaries when the number of players is fixed shall be proven in an expanded version of this paper.

On the other hand, when n is allowed to vary the situation is exactly the reverse. Namely, there exist protocols which are (provably) secure with respect to a static adversary, but (provably) insecure with respect to a dynamic adversary. In an expanded version of this paper, a protocol having this property will be described.

FUNCTION FAMILIES. Similarly, dealing with families of functions is important to the notion of security. Informally, there exists a protocol P and a function family $f = \{f_c\}$ having the property that for any c, the protocol P when run on an initial configuration having common input c securely computes the finite function f_c. All the same, the protocol does *not* securely compute f. An example of this phenomenon will be described in an expanded version of this paper.

TERMINATION AWARENESS. The requirement that players and their adversaries "know" when a computation is over is necessary, for example, to make a sensible notion of serial composition. It is necessary for our reducibility theorem, as well. Of course, a meaningful notion of security can still be given which does not demand this property.

KNOWING THE INPUT LENGTHS. Notice that, whatever our computational task, all players know how long other players inputs are. Though we need not be quite so demanding, it is essential that the players at least know a *bound* on the length of the longest possible private input. As long as such a bound is known, we can take all inputs to be of the same length, as a matter of convenience; but if such a bound is not known, and one is unwilling to reveal this information in the course of a "secure" protocol, then problem of secure computation changes markedly, and many computational tasks become impossible. (See [CGK90] for work on the problem within this more restricted framework.) The cause for this, intuitively, is that for some functions f, a communication protocol to distributively evaluate them *must* divulge not only the function value itself, but also a bound on the length of its inputs: for one thing a player i learns when interacting with another player j is the *number of bits* which were sent out by j and received by i, and, for some functions, this number of bits will have to be as many as j's input is long.

As a consequence, to arrive at a generally obtainable notion of security, one has the choice between developing a notion of a secure function evaluation which "leaks nothing except the function value and a bound on the input length," or, alternatively, one that "leaks nothing beyond the function value, where a bound on input lengths is known by everyone *ipso facto*." We have adopted the second viewpoint.

3.7 Discussion

Discuss the following points:

Doing nothing is not a secure computation of the constant function. Explain why it shouldn't be.

Weaker notions of security, as in [KMR90].

More "direct" mimicking of ideal evaluation, as in [GL90].

Simulators in our context vs. zero-knowledge: why the differences?

4 Complexity

This chapter —particularly Section 4.4— is to be substantially revised.

This section describes a relaxed definition of security, called *computational security.* Though an information-theoretically secure protocol is necessarily computationally secure, the reverse need not hold, as computational security only captures immunity against those t-adversaries which are limited to performing *reasonable* computational tasks.

The broad strokes for relaxing our notion of security are straightforward, so we sketch them now. We say that reasonable adversaries are those which run in polynomial time, and a computationally secure protocol only considers the injurious effects such agents. As expected, a computationally-secure protocol must be efficiently simulatable given the appropriate ideal-evaluation oracle. But instead of saying that interacting with this simulator produces nearly an *identical* view as interacting with the network, we permit the distribution on views to be quite different, as long as these distributions *appear* the same to observers limited to efficient computation. Computationally-secure protocols must achieve the same correctness condition as information-theoretically secure protocols, but they do this with respect to efficiently computable awareness functions, $\mathcal{AI}$ and $\mathcal{AO}$.

The main subtlety in defining computational security lurks in this last sentence: we must modify our adversary input and output functions very carefully, or else we do not arrive at a model-independent notion of security which preserves our reducibility theorem. The new notions for awareness functions are described in Section 4.4.

After having defined this more liberal notion of security, we also define a less liberal one, called *perfect security.* Though perfect security, like perfect zero-knowledge, is more cumbersome to work with than its statistical counterpart, it is always good to understand what perfect security is and when it can be achieved.

4.1 Polynomiality of Algorithms

We follow well-entrenched tradition, regarding deterministic functions as computationally tractable exactly when they can be computed by Turing machines running in polynomial time.

It is convenient to speak of computable functions of several inputs, some of which may be *infinite* strings. For example, a probabilistic encryption function is such an object (page 40). For functions on possibly infinite strings, we explain the meaning of "polynomial time" below.

A Turing machine is a device with some number of *input tapes,* a single *output tape,* and a *finite control.* Roughly said, a Turing machine M computes a function $\mathcal{M}(x_1, \ldots, x_i)$ if, when input tape i is initialized to x_i, the execution of M always halts with output tape containing $\mathcal{M}(x_1, \ldots, x_i)$. Notice that the function $\mathcal{M}$ may be defined on any fixed number of finite *or* infinite strings.

For Turing machine M to compute the function $\mathcal{M}$ in polynomial time, we require M compute $\mathcal{M}$, halting after a number of finite-state transitions bounded above by some fixed polynomial in the length of M's *first* argument, x_1.

Recall that we have made the common input the first argument to n-party protocols, adversaries, and simulators. So, in all these cases, polynomiality means polynomial in $|c|$.

4.2 The Complexity of Protocols and Their Adversaries

MICRO-ROUNDS. Let $e = \{C_i^{r\rho}, C_A^{r\rho}\}$ be an execution of an n-party protocol, terminating at round R. The number of *player micro-rounds* for this execution is the number of micro-rounds

utilized by *uncorrupted* processors: that is, it is

$$\sum_{i=1}^{n}\sum_{r=1}^{R} \varrho'_{ir},$$

where

$$\varrho'_{ir} = \begin{cases} 0 & \text{if } \exists \bar{r} < r \text{ such that } i \in \kappa_A^{\bar{r}\infty} \\ \varrho_{ir} & \text{otherwise} \end{cases}$$

The *player micro-round complexity* is a function of the common input, c. It gives the worst-case number of player micro-rounds when the common input is c, and is "∞" if no such number exists. By "worst case" we mean the maximum over all executions of the protocol interacting with *any* adversary.

Similarly, the number of *adversary micro-rounds* for an execution e is

$$\sum_{r=0}^{R} \varrho_{Ar}.$$

The *adversary micro-round complexity* is a function of the common input, c. It gives the worst-case number of adversary micro-rounds when the common input is c, and is "∞" if no such number exists. Here, "worst case" gives the maximum over all executions of the adversary interacting with *any* protocol.

ROUNDS. The *round complexity* of a protocol P is a function of the common input, c. It gives the least value R such that when protocol P is run in the presence of any adversary A and the common input is c, the execution necessarily terminates within R rounds. The round complexity is "∞" if no such number exists.

COMMUNICATION BITS. The *communication complexity* of a protocol P is a function of the common input, c. It gives the least value μ such that when protocol P is run in the presence of any adversary A and the common input is c, the total number of bits sent out by uncorrupted processors is at most μ. The communication complexity is "∞" if no such number exists. The number of communication bits for a specific R-round execution is defined as $\sum_{i=1}^{n}\sum_{r=1}^{R}(|\bar{M}_i^r| + \sum_{j=1}^{n}|\bar{m}_{ij}^r|)$, using the definitions from Page 22. This expression counts all of the bits sent by processors before they are corrupted.

POLYNOMIALITY OF PROTOCOLS AND ADVERSARIES. A protocol is *polynomial time* if each micro-round can be computed quickly, and there are not too many of them. More formally:

Definition 4.1 *A protocol P is* polynomial time *if there is a polynomial Q such that*

- *For any n, the n-party protocol $P_n(c, s)$ evaluates within $Q(|c|)$-time, where c is the common input.*
- *For each n, the player micro-round complexity of P_n is bounded above by $Q(|c|)$.*

Analogously, an adversary A is *polynomial time* if she is easily described, each micro-round is easy to compute, and there are not too many of them. More formally:

Definition 4.2 *An adversary A is* polynomial-time *if there is a polynomial Q such that*

- *For any n, the encoding of A_n is computed by A in time bounded above by $Q(n)$.*

- *For any n and common input c, the adversary micro-round complexity of $A_n(c,s)$ is bounded above by $Q(|c|)$.*
- *For any n, the algorithm $A_n(c,s)$ evaluates within $Q(|c|)$-time.*

Note that, under our definitions, a protocol can be polynomial time even though it may interact with an adversary who "floods" the network with a superpolynomial number of communication bits.

4.3 The Complexity of Distinguishing Ensembles

ALTERNATIVE DEFINITION OF INDISTINGUISHABILITY. We being by rephrasing our definition of indistinguishability in a way which generalizes better for handling issues of computational complexity. This notion is based on the idea of a *distinguisher*, the formalization of a "judge" who votes to decide among two competing alternatives.

A *distinguisher* D^a is a $\{0,1\}$-valued function D on some number of *input strings*, $x_1,\ldots,x_i$, an infinite *advice string*, a, and an infinite sequence of *random bits*, r_D. By $\mathbf{E}D^a(x_1,\ldots,x_i)$ we denote the expected value of D, over its random bits r_D, when the inputs are $x_1,\ldots,x_i$ and the advice is a. This is precisely the probability that D evaluates to 1. The following definition of indistinguishability is easily proven equivalent to Definition 3.8.

Definition 4.3 *Ensembles E and E' are* **statistically indistinguishable**, *written $E \simeq E'$, if they are over the same parameter set $\mathcal{L} = \{L_k\}$, and for every distinguisher D^a,*

$$\epsilon(k) = \sup_{\omega \in L_k} \left|\mathbf{E}D^a(1^k, E_k(\omega), \omega) - \mathbf{E}D^a(1^k, E'_k(\omega), \omega)\right|$$

is negligible.

COMPUTATIONAL INDISTINGUISHABILITY. We now specify what it means for two ensembles to be indistinguishable to an observer with bounded computational resources, thereby relaxing the notion of statistical indistinguishability of Definition 4.3.

Recall that a distinguisher D^a is a $\{0,1\}$-valued function D on a set of inputs, $x_1,\ldots,x_n$, an infinite advice string, a, and an infinite string of coin tosses, r_D. We say that a distinguisher D^a is *polynomial-time* if D is a polynomial-time algorithm. Computational indistinguishability demands negligible bias in distinguishing ensembles based on such resource-bounded algorithms:

Definition 4.4 *Ensembles E and E' are* **computationally indistinguishable**, *written $E \approx E'$, if they are over the same parameter set $\mathcal{L} = \{L_k\}$, and for every polynomial-time distinguisher D^a*

$$\epsilon(k) = \sup_{\omega \in L_k} \left|\mathbf{E}D^a(1^k, E_k(\omega), \omega) - \mathbf{E}D^a(1^k, E'_k(\omega), \omega)\right|$$

is negligible.

When we wish to emphasize the parameter set $\mathcal{L} = \{L_k\}$ which parameterizes the ensembles E and E' we write $E \underset{\mathcal{L}}{\approx} E'$.

4.4 The Complexity of Awareness

In this subsection, we discuss what it means for a computationally bounded adversary to *know* something about an interaction. The intuition we are trying to capture is that the adversary knows some information if she could compute it "on the side," without much extra expenditure of effort and without knowing much about herself at all.

Our notion of awareness is very demanding, for a variety of reasons. First: because we wish to make *minimal assumptions* about what an adversary, as a devise, actually is and can do. For example, we certainly do not wish to assume that an adversary has a description of herself as a Turing machine, for example, or that she might "know" something only if she knows exactly her own nature: if we take our anthropomorphization seriously and our adversary actually *is* a person, such an assumption is highly questionable! For the same rationale, our adversary should not be able to "reset herself," "see her coin flips," or figure out what she would have done in the distant past or distant future. Second: our notion of awareness is strong in order to facilitate dealing with complexity concerns at very fine level, if we choose to. As an example, if an adversary interacts with an agent using just linear computational complexity, but she "knows" some information in the sense that she could compute it using some cubic-time algorithm, then it might not, in fact, "know" the information in a very real sense—her computing it may simply be impractical. Third: we must adopt a rather stringent notion of awareness or we no longer know how to prove our reducibility theorem.

The idea that an interaction might establish that an agent "knows something" was put forward by Goldwasser, Micali and Rackoff [GMR89] and formalized by Feige, Fiat and Shamir [FFS87], and Tompa and Woll [TW87]. However, our notion of awareness differs from ideas to be found there. For one thing, the goal is different: we are discussing proofs of knowledge of a function associated with the interaction itself, as opposed to proofs of knowledge of a witness to a boolean predicate, say. Apart from this, we wish to be more demanding than in previous work, for the reasons given above.

4.4.1 Informal description

We are interested in knowledge associated *to an interaction*—principally, the knowledge an *adversary* possesses when she interacts. We thus make the following definition.

Definition 4.5 *An* (adversary) awareness function $\mathcal{F}$ *is a map*

$$\mathcal{F}\colon \underbrace{\{0,1\}^*}_{\substack{\text{common}\\ \text{input}}} \times \underbrace{\{0,1\}^*}_{\substack{\text{adversary}\\ \text{traffic}}} \to \underbrace{\{0,1\}^*}_{\substack{\text{value}\\ \text{known}}}.$$

If $\mathcal{F}$ is not as above because its range is not $\{0,1\}^*$, we still call $\mathcal{F}$ an awareness function, as long as its range has a natural encoding into $\{0,1\}^*$. As an example, the adversary input and output functions were awareness functions.

An adversary A interacts with an *agent* B who send messages to A and receives messages from her. We have described in detail the interaction of an adversary with two types of agents: protocols and simulators. In defining awareness, we may imagine an adversary A interacting with an arbitrary agent—but for simplicity we will jump back, in a bit, and take this agent to be a network running a protocol P. The nature of the interaction between A and B may depend on some parameters—in fact, the interactions we care about are probabilistic functions of $\omega = (\vec{x}, \vec{a}, a_A, c)$.

An interaction of an adversary A with an agent B will be denoted $A \leftrightarrow B$. To emphasize some parameter ω on which this interaction depends, we write $A \overset{\omega}{\leftrightarrow} B$.

ALGORITHM $F^A(c;\ \alpha_1, \beta_1, \ldots, \alpha_s)$:

The algorithm is given c and does some computation.
for $i \leftarrow 1$ **to** $s-1$:
    until the algorithm is satisfied:
        It is given $\alpha_1, \beta_1, \ldots, \alpha_i$.
        It must compute a sample $\beta_i' \leftarrow (A \overset{\omega}{\leftrightarrow} B)[\alpha_1, \beta_1, \ldots, \alpha_i]$.
        The algorithm is provided a sample from $(A \overset{\omega}{\leftrightarrow} B)[\alpha_1, \beta_1, \ldots, \alpha_i, \beta_i']$.
        It does some more computation.
It outputs a value, y.

Figure 3: *The structure of a probabilistic algorithm F for extracting knowledge from an agent A about traffic T. The algorithm is given input c and is "trying" to compute $\mathcal{F}(c,T)$. The traffic $T = (\alpha_1, \beta_1, \ldots, \alpha_s)$*

An $A \overset{\omega}{\leftrightarrow} B$ interaction can be described as follows: A sends to B a string α_1 (consisting of a corruption request $\mathbf{corrupt}_i$ or a vector of messages on behalf of currently corrupted player M); then B responds with a string β_1 (consisting, presumably, of an exposed state or a vector of messages on behalf of good players, respectively); then A sends another string, α_2; then B responds with a string, β_2; and so forth, until A sends some last string, α_s, to the agent B.

Now, for any ω, not only is there is a distribution on possible $A \overset{\omega}{\leftrightarrow} B$ interactions, but, fixing an initial segment $\alpha_1, \beta_1, \ldots, \alpha_i$, there is a distribution $(A \overset{\omega}{\leftrightarrow} B)[\alpha_1, \beta_1, \ldots, \alpha_i]$ on possible next messages β_i returned by B. Likewise, there is a distribution on any $(A \overset{\omega}{\leftrightarrow} B)[\alpha_1, \beta_1, \ldots, \beta_i]$

What does it mean for an adversary awareness function $\mathcal{F}$ to be *efficiently computable* (with respect to the agent B)? Roughly, there must be *fixed* probabilistic polynomial time algorithm F (which one might imagine "embedding" in A) which is almost always able to compute $\mathcal{F}$ "on the side," as A interacts with B. The algorithm F does this by playing the following simple game. As the conversation between A and B progresses, after A has already engaged in some partial conversation τ and has just heard a message β from B, the algorithm F may choose to concoct some *different* message β'—but one drawn from the *same* distribution as the last message that B just sent. Algorithm F may then use A as a "black box"to figure out what message A would respond on the conversation in which the last message was β' instead of β. (Each time A is so invoked, independent random coins are used for it.) Algorithm F may run this experiment several times, getting various sample replacement messages $\beta', \beta'', \ldots$. However, this is the only manner under which F may experiment with the black-box A. At the end, F outputs it "guess" of $\mathcal{F}(c,T)$.

Basically, we say that an adversary A is *efficiently aware* of an awareness function $\mathcal{F}$ arising from interactions with B under the condition that $\mathcal{F}(c,T)$ can be computed by a *probabilistic polynomial-time alternative sampling algorithm*—which is a algorithm having the structure described in Figure 3.

Definition 4.6 *Fix $t \in \mathbf{N}$. An adversary awareness function $\mathcal{F}$ for interacting with a protocol P is* **efficiently computable** *(with respect to t) if there exists a probabilistic polynomial time online alternative sampling algorithm F such that for any polynomial-time t-adversary A, for some negligible function $\epsilon(k)$, for any $(\vec{x}, \vec{a}, a_A, c) \in L_k$, with probability at least $1 - \epsilon(k)$ the following even happens: if T is a transcript drawn from by allowing A and P to interact with a configuration with $(\vec{x}, \vec{a}, a_A, c)$, then $F^A(c,T)$ outputs $\mathcal{F}(c,T)$.*

For a computational notion of security, not only do we allow allow the adversary input and output functions to depend on the adversary A in this "controlled" but efficiently computable way, but

we allow the same of the simulator. That is, recall that, in our previous definitions of security, the simulator S had no information whatsoever about the adversary with whom it was interacting (apart from that which could be inferred by the messages this adversary sent out). Now, we allow the simulator to compute its messages to A by allowing S to be a

Namely, the simulator S, has an "oracle" for the adversary A with which it interacts, considered as a probabilistic algorithm that S to be a probabilistic polynomial-time on-line alternative sampling algorithm. We call such a simulator an *efficient simulator.*

4.4.2 Formal description

4.4.3 Remarks

4.5 Computational Security

SECURE FUNCTION EVALUATION. We now give our notion of computational security.

Definition 4.7 *Let $t \in \mathbf{N}$, let $f = \{f_c\}$ be a family of finite functions, and let P be a protocol. We say that P* computationally t-securely computes *f if there is an efficient simulator S using an efficiently-computable t-admissible adversary input function $\mathcal{AI}$, and an adversary output function $\mathcal{AO}$, such that for any t-adversary A,*

- *Privacy:* $\quad \mathcal{E}A\text{-VIEW}_k^P(\vec{x}, \vec{a}, a_A, c) \underset{\mathcal{C}}{\approx} \mathcal{E}A\text{-VIEW}_k^{S^A, \mathcal{AI}, I}(\vec{x}, \vec{a}, a_A, c).$
- *Correctness: There is a negligible function $\epsilon(k)$ such that for all $(\vec{x}, \vec{a}, a_A, c) \in L_k$,*

$$\text{Prob}\left[\overline{\mathcal{AO}}(\vec{x}, \vec{a}, a_A, c, \vec{r}, r_A) \neq f(\overline{\mathcal{AI}}(\vec{x}, \vec{a}, a_A, c, \vec{r}, r_A))\right] \leq \epsilon(k),$$

where the probability is taken over all possible $\vec{r}$ and r_A.

4.6 Statistical Security

4.7 Perfect Security

PROBABILISTIC ALGORITHMS. A probabilistic algorithm M is a Turing machine which takes some number of inputs, $x_1, \ldots, x_i$, and an infinite sequence of "coin flips," r. It is required that for any set of inputs, M halts with probability 1, where the probability is taken over all choices of r. Thus, fixing any set of inputs $x_1, \ldots, x_i$, probabilistic algorithm M induces a distribution on outputs, $M(x_1, \ldots, x_i)$.

There is no widespread agreement for a robust and generally adequate notion of an *efficient* probabilistic algorithm. Requiring that a probabilistic algorithm run in fixed polynomial time is usually adequate, but in some cases this is too restrictive. For example, simulators for perfect zero-knowledge proofs often do not run in fixed polynomial time, but we still want to demand that they be "efficient." The same problem arises in our context: a perfect notion of security requires something more generous than fixed polynomial time simulation.

When fixed polynomial time is inadequate, it is common to relax the notion to *expected* polynomial time. This is often a poor solution, for a variety of reasons:

- First, expected polynomial-time is not a very robust notion. For example, the composition of expected polynomial time algorithms is not necessarily expected polynomial time. This lack of robustness leads, for example, to an unaesthetic and conceptually questionable asymmetry in the definition of perfect zero-knowledge: allowing simulators to be expected polynomial-time while verifiers are required to be fixed polynomial time.

- Second, expected polynomial time can be too liberal a notion. For example, for any constant c, an expected polynomial time algorithm might take longer than n^c-time for a non-negligible fraction of the possible random strings. Arguably, such behavior should be considered unacceptable in an algorithm termed "efficient."

- Finally, expected polynomial time can be an insufficiently liberal notion. For example, an algorithm which is linear-time —apart from the fact that it diverges with negligible probability— is a pretty efficient algorithm! But it is not expected polynomial time.

To address such issues, we say that a probabilistic algorithm $M(x_1, \ldots, x_i, r)$ is *polynomial-time* if there exists a fixed polynomial Q and a negligible function ϵ such that for any $(x_1, \ldots, x_i)$, with probability at least $1 - \epsilon(|x_1|)$, algorithm M halts within $Q(|x_1|)$-steps. That is, M almost always halts in fixed polynomial time.

We note that the class of probabilistic polynomial time algorithms is closed under composition; that such algorithms are robust against changing their behavior on a negligible fraction of random coin sequences; and that such algorithms would seem to be adequate for most simulation purposes.

4.8 Discussion

NONUNIFORMITY. The notion we have defined for computational indistinguishability is a *nonuniform* notion—possibly, the ensembles appear different to the resource-bounded judge *only by virtue of the infinite advice string, a.* Nonuniform notions of indistinguishability have more commonly been defined by polynomial-size circuit families. We find the phrasing above more convenient, because of it more natural to allow an infinite string to be an input to algorithm than to a circuit.

Uniform notions of computational indistinguishability—where the distinguisher does not have benefit of the advice a—are also possible. In fact, *all notions and results of this paper have analogs in the uniform model, and it is not difficult to make the appropriate extensions.* For economy of notions, we choose to describe only the nonuniform notion of security. This means that not only are the adversaries we consider restricted to *nonuniform* efficient computation, but, mirroring this, the the underlying notion of security is nonuniform as well.

WHY PREFER NONUNIFORMITY? In our context, nonuniform notions of security have several advantages over their uniform counterparts. Most importantly, since a cryptosystem is generally run for a *particular* choice of security parameter, one would be unhappy with a protocol which was only secure against uniform adversaries: a sufficiently committed attacker would mount an attack that would break the cryptosystem *itself*, a much worse break than just breaking a particular *usage* of the cryptosystem. Secondly, proofs are frequently simpler or more natural in the nonuniform model. Third, existence results on secure protocols talk about how an arbitrary circuit can be used to specify a protocol for securely evaluating it; thus there is already nonuniformity present in the families of circuits which might be evaluated.

5 Properties of Secure Protocols

This chapter is to be substantially revised.

This section describes some of the basic properties which secure protocols enjoy. After some basic results on indistinguishability in Section 5.1, we begin, in Section 5.2, with the observation that in a secure protocol for computing f, when run in the presence of an adversary that is only "nosy" (that is, as she corrupts players, she continues to perform computation according to the protocol), each player i almost certainly outputs $f_i(\vec{x})$. In Section 5.4 we show that, in a secure protocol, what the adversary commits to is essentially *independent* of input values held by good players, as it was with the abstraction which a secure protocol endeavors to imitate. In Section 5.3 we show that the level of generality necessary for designing secure protocols for doing vector-valued secure function evaluation is already present in the seemingly simpler task of doing string-valued secure computation; in particular, vector-valued secure function evaluation can be implemented on top of string-valued secure function evaluation in a "generic" and efficient manner. Finally — and most importantly— in Section 5.5 we show that modular design of cryptographic protocols is possible, in the sense that if you design a secure protocol for computing some function g, you may thereafter treat g as a "primitive" operation available to you in designing other secure protocols. This *reducibility* property is the center of our notion of security.

5.1 Preliminaries

We state without proof some basic properties of indistinguishability.

Proposition 5.1 *If two ensembles are perfectly indistinguishable, they are statistically indistinguishable. If two ensembles are statistically indistinguishable, they are computationally indistinguishable.* ■

Proposition 5.2 *Computational indistinguishability of ensembles forms an equivalence relation. So does statistical indistinguishability and perfect indistinguishability.* ■

The following definition and lemma capture the notion that if two ensembles are computationally indistinguishable, then the ensembles appear different even when sampled by selecting the parameter according to an arbitrary distribution. The same assertion holds for statistical for perfect indistinguishability.

Definition 5.3 *Let A and B be ensembles, and let $I = \{I_k\colon k \in \mathbf{N}\}$ a family of probability measures on $\{0,1\}^*$. We say that A and B are* statistically indistinguishable over *I if A and B are ensembles over the same parameter set, $\mathcal{L} = \{L_k\}$, and support $I_k \subseteq L_k$ for all k, and for every distinguisher D^a,*

$$\epsilon(k) = |\mathbf{E}_{\omega \leftarrow I_k} ED^a(1^k, A_k(\omega), \omega) - \mathbf{E}_{\omega \leftarrow I_k} ED^a(1^k, B_k(\omega), \omega)|$$

is negligible. They are computationally indistinguishable over $\mathcal{L}$ if the same assertion holds for every polynomial time distinguisher D^a.

Lemma 5.4 *Let A and B be indistinguishable over $\mathcal{L} = \{L_k\}$. Then for any family of probability measures $I = \{I_k\}$, where support $I_k \subseteq L_k$ for all k, A and B are indistinguishable over I.* ■

We next note that computational indistinguishability coincides with statistical indistinguishability in one natural setting—when the support of the ensemble grows very slowly (or not at all) in the index of the ensemble. Such ensembles arise when, for example, a predicate is applied to an ensemble, and the image is considered an ensemble once again.

Lemma 5.5 *Let A be an ensemble over $\mathcal{L} = \{L_k\}$ such that, for some constant c,* support $A_k(\omega) \subseteq \{0,1\}^{c \lg k}$ *for all $k \in \mathbf{N}$, $\omega \in L_k$. Then A and B are computationally indistinguishable if and only if they are statistically indistinguishable.* ■

Last of all, we state a technical assumption under which a particular "composition" lemma holds on indistinguishable ensembles.

Lemma 5.6 *Suppose* $\mathcal{E}A_k(x) \underset{\mathcal{L}}{\approx} \mathcal{E}A'_k(x)$, support $A_k(x) \subseteq L'_k$, support $A'_k(x) \subseteq L'_k$, $\mathcal{E}B_k(x,y) \underset{\mathcal{L}\times\mathcal{L}'}{\approx} B'_k(x,y)$, *with $B_k(x,y)$ and $B'_k(x,y)$ are polynomial-time samplable. Then for any constants $c_k \in L_k$, $\mathcal{E}B_k(c_k, A_k(x)) \underset{\mathcal{L}}{\approx} \mathcal{E}B'_k(c_k, A_k(x))$.* ■

5.2 Correctness in the Presence of Passive Adversaries

A particularly sedate type of adversary is one that is "honest-but-curious"; such an adversary, when she corrupts a player, continues to execute the protocol faithfully on behalf of that player. (As always, such an adversary may choose whom to corrupt using an adaptive strategy.) As expected, when a secure protocol is run under attack by such an adversary, each player almost always computes the correct function value, as determined by the initial input vector. We state and argue this in the full paper.

Theorem 5.7 *Suppose P t-securely computes a function $f = \{f_c\}$, and let A be an honest-but-curious t-adversary for P. Then, when P is run in the presence of A, for some negligible function $\epsilon(k)$, for a fraction of at least $1 - \epsilon(k)$ of all $(\vec{x}, \vec{a}, a_A, c) \in L_k$ each player i outputs $((f_c)(\vec{x}))_i$.*

Proof: Consider P being run in the presence of the "void adversary," A_0, who corrupts no players whatsoever. Let S be the simulator establishing P's security. Since the adversary input function $\mathcal{AI}$ provides a value x'_T where the set T consists precisely of the currently corrupted players, and since the set of currently corrupted players must be void, we know that $\mathcal{AI}$ is identically equal to $\emptyset$. Consequently, the network input $\overline{\mathcal{AI}}$ is identical to the input vector $\vec{x}$, and the correctness constraint for P run in the presence of adversary A_0 says that, for some negligible function $\epsilon(k)$, for all $(\vec{x}, \vec{a}, a_A, c) \in L_k(f)$, with probability at least $1 - \epsilon(k)$ each processor i will output $(f_c(\vec{x}))_i$.

Now consider P being run in the presence of the honest-but-curious t-adversary A. Because this adversary executes the protocol P faithfully on behalf of each corrupted player, we know that the distribution on outputs of processors when running under attack by the void adversary A_0 is *identical* to the distribution on outputs of processors when under attack by A. Thus the conclusion regarding the distribution on outputs for P under attack by A_0 holds equally well for P under attack by A. ■

5.3 Completeness of String-Valued Computation

The following theorem justifies, in some instances, restricting attention to the computation of string-valued function families. For its statement, we let f' be the function family of string-valued functions, encoded as boolean circuits, and we let f be the function family of vector-valued functions, encoded as boolean circuits. More specifically, we let the function family $f' = \{f'_{c'}\}$ be described as follows: for a common input c', a function $f_{c'}$: $(\{0,1\}^{\ell_{c'}})^{n_{c'}} \to \{0,1\}^{l_{c'}}$ is described by a boolean circuit $C_{c'}$ over bounded fan-in gates and unbounded fan-in XOR gates. Each function $f_{c'}$ can also be interpreted as a vector-valued function simply by replicating its output for each player. The function family $f = \{f_c\}$ is the one which maps any common input c to a function f_c: $(\{0,1\}^{\ell_c})^{n_c} \to (\{0,1\}^{l})^{n_c}$ described by a boolean circuit C_c over bounded fan-in gates and unbounded fan-in XOR gates.

Theorem 5.8 *Suppose that there is an $O(\mathbf{depth}(C_c))$-round protocol P that t-securely computes f. Then there is an $O(\mathbf{depth}(C'_c))$-round protocol P' that t-securely computes f', where*

Proof: This proof is a remnant of notation past. It requires substantial updating, and is left here only to give some idea of the argument involved.

We must describe a protocol $P = \{P_{n,\ell}\}$ and show that it securely computes $f = \{C_{n,\ell}\}$.

The basic idea is very simple: P is derived from a protocol $\hat{P}$ for some deterministic function $\hat{f}$ related to f, as follows: each party i inputs into the computation of $\hat{f}$ a random *pad* p_i which is XOR'ed with his "own" piece of the output. The is done so that the part of the output which "belongs" to a good player is not meaningful to the adversary, but it is still easy for the good player to interpret what his private output is.

We now proceed to make this idea precise, and prove that it works.

Definition of $\hat{f}$.

Define the string-valued function $\hat{f} = \{\hat{C}_{n,\hat{\ell}} : (\{0,1\}^{\hat{\ell}})^n \to \{0,1\}^{\hat{\ell}}\}$ according to

$$\begin{aligned}
&\hat{C}_{n,\hat{\ell}}(x_1 p_1 01^{\ell_1}0^{\gamma_1}, \ldots, x_n p_n 01^{\ell_n}0^{\gamma_n}) \\
&\quad = (p_1 \oplus C_{n,\ell}(\vec{x}))[1:\ell] \cdots (p_n \oplus C_{n,\ell}(\vec{x}))[(n-1)\ell + 1 : n\ell]\ 0^g \\
&\quad = p \oplus C_{n,\ell}(\vec{x})\ 0^g,
\end{aligned}$$

where

ℓ is the value that occurs most among $\{\ell_1, \ldots, \ell_n\}$,
$|x_i| = |p_i| = \ell$,
$g = \hat{\ell} - n\ell$, and
$p = \vec{p} = p_1 \cdots p_n = (p_1, \ldots, p_n)$.

Note that we have assumed an encoding of $\hat{C}_{n,\ell}$ for which $y = C_{n,\ell}(\vec{x})$ specifies, consecutively, the outputs of each of the players. Also, we do not distinguish vectors, such as $(p_1, \ldots, p_n)$, from the corresponding string $p_1 \ldots p_n$.

For values $\hat{C}_{n,\hat{\ell}}$ which are not well-defined by the expression above (for example, there is no unique ℓ, $\hat{\ell}$ is too small, etc.), assert that $\hat{C}_{n,\hat{\ell}} = 0^{\hat{\ell}}$.

We remark that $\hat{f} = \{\hat{C}_{n,\hat{\ell}}\}$ has polynomial-size circuits; the only observation needed is that $\hat{C}_{n,\hat{\ell}}$ is defined in terms of fewer than $\hat{\ell}$ different circuits $\{C_{n,\ell}\}_{\ell < \hat{\ell}}$, each having size at most $\mathbf{poly}(n\ell) \leq \mathbf{poly}(n\hat{\ell})$.

Definition of the protocol P.

Let $\hat{P}$ be a t-secure protocol for $\hat{f}$. Protocol $P_{n,\ell}$ is as follows: each player i, on input $x_i \in \{0,1\}^{\ell}$, flips coins to determine a random string p_i,

$$p_i \leftarrow \{0,1\}^{\ell}. \tag{1}$$

Player i defines

$$\hat{\ell} = \max\{3\ell + 1, n\ell\}, \text{ and} \tag{2}$$
$$\gamma = \hat{\ell} - (3\ell + 1). \tag{3}$$

Player i runs $\hat{P}_{n,\hat{\ell},i}$ on input

$$x_i p_i 01^{\ell}0^{\gamma}. \tag{4}$$

However, instead of outputting the string y which $\hat{P}$ specifies to output, player i outputs

$$y_i = y[(i-1)\ell + 1 : i\ell] \oplus p_i. \tag{5}$$

Outline.

We must show that P t-securely computes f. Let A be a polynomially-bounded adversary. We must construct an on-line simulator S for A. Let $\hat{S}$ be the on-line simulator showing that A does not defeat $\hat{P}$, and let $\hat{\chi}$ be the associated committal function for A attacking $\hat{P}$. The simulator S for P is made by modifying $\hat{S}$. As the most apparent obstacle, note that $\hat{S}$ has access to an $O_t(\vec{x}\vec{p}\vec{s}; \hat{f})$-oracle, whereas S has access to an $O_t(\vec{x}; f)$-oracle.

The simulator S will behave like $\hat{S}$, with a few exceptions. First, its queries to its $O_t(\vec{x}\vec{p}\vec{s}; \hat{f})$-oracle will be answered by an oracle $\bar{\Omega}$, which the simulator S itself will provide, using access only to its $O_t(\vec{x}; f)$-oracle. Secondly, simulator S, instead of producing transcripts with "output strings" y in the simulated views, will produces transcripts in which the string y is replaced by $y_i = y[(i-1)\ell + 1 : i\ell] \oplus p_i$, where p_i is taken from the simulator's output. Last of all, instead of producing transcripts in which $x_i p_i s_i$ appears as a player's input, S will produce transcripts in which x_i appears as the player's input, and p_i prefixes the player's random coins.

To prove that the simulator S "works," we will establish the following sequence of implications, for protocols, simulators, and oracles which we shall shortly define.

$$\mathrm{VIEW}^k_{A(a),\hat{P}(\vec{x}\vec{p}\vec{s})} \approx \hat{S}^{O_t(\vec{x}\vec{p}\vec{s};\hat{f})}_{A(a)} \Rightarrow \tag{6}$$

$$\mathrm{VIEW}^k_{A(a),\bar{P}(\vec{x})} \approx \bar{S}^{\bar{\Omega}(\vec{x};\hat{f})}_{A(a)} \Rightarrow \tag{7}$$

$$= \bar{S}^{\bar{\Omega}(\vec{x};f)}_{A(a)} \Rightarrow \tag{8}$$

$$\mathrm{VIEW}^k_{A(a),P(\vec{x})} \approx \dot{S}^{\bar{\Omega}(\vec{x};f)}_{A(a)} \tag{9}$$

$$= S^{O_t(\vec{x};f)}_{A(a)} \tag{10}$$

The Protocol $\bar{P}$.

We begin by defining a probabilistic analog of $\hat{P}$, the protocol $\bar{P}$. Each player i, executing $\bar{P}_{n,\ell,i}$ on input $x_i \in \{0,1\}^\ell$, begins by flipping coins to determine p_i according to Equation 1. Player i defines $\hat{\ell}$ and γ according to Equations 2 and 3, sets $s_i = 01^\ell 0^\gamma$, and runs $\hat{P}_{n,\hat{\ell},i}$ on input $x_i p_i s_i$. Player i outputs the string y which $\hat{P}$ computes.

In other words, P is identical to $\bar{P}$, except that P outputs the string $y_i = y[(i-1)\ell + 1 : i\ell] \oplus p_i$, whereas $\bar{P}$ just outputs y.

The oracle $\bar{\Omega}$.

We next consider a probabilistic analog $\bar{\Omega}$, to the oracle $O_t(\vec{x}\vec{p}\vec{s}; \hat{f})$. Oracle $\bar{\Omega}$ behaves as follows: it selects random

$$\vec{p} \leftarrow (\{0,1\}^\ell)^n, \tag{11}$$

defines $\hat{\ell}$ and γ according to Equations 2 and 3, and sets

$$\vec{s} = (01^\ell 0^\gamma)^n. \tag{12}$$

The oracle then behaves like an $O_t(\vec{x}\vec{p}\vec{s}; \hat{f})$ oracle. In particular, to a component query of i preceding the output query, $\bar{\Omega}$ responds with $x_i p_i s_i$; to an output query of $\vec{x}'_T \vec{p}'_T \vec{s}'_T$, $\bar{\Omega}$ responds with $y = \bar{f}(\vec{x}'_T \vec{p}'_T \vec{s}'_T \cup \vec{x}_{\overline{T}} \vec{p}_{\overline{T}} \vec{s}_{\overline{T}})$; and to subsequent component queries i, $\bar{\Omega}$ responds $(x_i p_i s_i,\ y_i)$.

The simulator $\bar{S}$.

The simulator $\bar{S}$ is identical to $\hat{S}$, apart from the encoding of transcripts. Namely, suppose, that $\hat{S}$, when interacting with its oracle, produces a transcript τ. The transcript τ specifies (among other things), for each corrupted processor i, i's private input $x_i p_i s_i$ and a portion of i's random tape, R_i. Simulator $\hat{S}$ outputs the same transcript τ, except that each player i's input tape is taken to contain *only* x_i, whereas player i's random tape is replaced by $p_i R_i$. This can be considered as applying a simple function h to the transcript τ that $\hat{S}$ computes.

Equation 7 holds.

We wish to show that Equation 7 holds,

$$\mathrm{VIEW}^k_{A(a),\bar{P}(\vec{x})} \approx \bar{S}^{\bar{\Omega}(\vec{x};\hat{f})}_{A(a)},$$

asserting the computational indistinguishability of $(\vec{x}, a)$-parameterized ensembles. What we know is that Equation 6 holds,

$$\mathrm{VIEW}^k_{A(a),\hat{P}(\vec{x}\vec{p}\vec{s})} \approx \hat{S}^{O_t(\vec{x}\vec{p}\vec{s};\hat{f})}_{A(a)},$$

asserting the computational indistinguishability of $(\vec{x}\vec{p}\vec{s}, a)$-parameterized ensembles.

The argument closely parallels Claim ?? in the proof of the main theorem of this thesis (page ??). Equation 6 means that for any polynomial-size family of circuits $\{D_k\}$,

$$\epsilon(k) = \max_{\vec{x}\vec{p}\vec{s},a} \left| \mathbf{E} D_k(\mathrm{VIEW}^k_{A(a),\hat{P}(\vec{x}\vec{p}\vec{s})}) - \mathbf{E} D_k(\hat{S}^{O_t(\vec{x}\vec{p}\vec{s};\hat{f})}_{A(a)}) \right|$$

is negligible. Let

$$\epsilon_{\vec{p}}(k) = \max_{\vec{x},a} \left| \mathbf{E} D_k(\mathrm{VIEW}^k_{A(a),\hat{P}(\vec{x}\vec{p}\vec{s})}) - \mathbf{E} D_k(\hat{S}^{O_t(\vec{x}\vec{p}\vec{s};\hat{f})}_{A(a)}), \right|$$

where $\vec{p} \in \{0,1\}^{n\ell}$ and $\vec{s}$ is given by Equation 12. Then the weighted sum

$$\epsilon'(k) = \max_{\vec{x},a} \left| 2^{-n\ell} \sum_{\vec{p} \in (\{0,1\}^\ell)^n} \epsilon_{\vec{p}}(k) \right|$$

is negligible. Now we can bound this from above by

$$\begin{aligned}
\epsilon'(k) &= \max_{\vec{x},a} 2^{-n\ell} \sum_{p \in (\{0,1\}^\ell)^n} \left| \mathbf{E} D_k(\mathrm{VIEW}^k_{A(a),\hat{P}(\vec{x}\vec{p}\vec{s})}) - \mathbf{E} D_k(\hat{S}^{O_t(\vec{x}\vec{p}\vec{s};\hat{f})}_{A(a)}), \right| \\
&\geq \left| \max_{\vec{x},a} 2^{-n\ell} \sum_{p \in (\{0,1\}^\ell)^n} \mathbf{E} D_k(\mathrm{VIEW}^k_{A(a),\hat{P}(\vec{x}\vec{p}\vec{s})}) - 2^{-n\ell} \sum_{p \in (\{0,1\}^\ell)^n} \mathbf{E} D_k(\hat{S}^{O_t(\vec{x}\vec{p}\vec{s};\hat{f})}_{A(a)}) \right| \\
&= \epsilon''(k),
\end{aligned}$$

using the triangle inequality: $\sum |A_i - B_i| \geq |\sum (A_i - B_i)| = |\sum A_i - \sum B_i|$; thus $\epsilon''(k)$ is negligible.

With h the map specified earlier (taking a transcript τ with input $x_i p_i s_i$ and random tape R_i to a transcript with input x_i and random tape $p_i R_i$), we conclude that $\epsilon'''(k)$ is negligible, where

$$\begin{aligned}
\epsilon'''(k) &= \left| \max_{\vec{x},a} 2^{-n\ell} \sum_{p \in (\{0,1\}^\ell)^n} \mathbf{E} D_k(h(\mathrm{VIEW}^k_{A(a),\hat{P}(\vec{x}\vec{p}\vec{s})})) - 2^{-n\ell} \sum_{p \in (\{0,1\}^\ell)^n} \mathbf{E} D_k(h(\hat{S}^{O_t(\vec{x}\vec{p}\vec{s};\hat{f})}_{A(a)})) \right| \\
&= \left| \max_{\vec{x},a} \mathbf{E} D_k(\mathrm{VIEW}^k_{A(a),\bar{P}(\vec{x})}) - \mathbf{E} D_k(\dot{S}^{\bar{\Omega}(\vec{x};\hat{f})}_{A(a)}) \right|
\end{aligned}$$

by the definitions of $\hat{P}$, $\bar{P}$, and $\bar{\Omega}$. This being negligible is precisely what is is needed to establish the claim.

An alternative way to look at $\bar{\Omega}$.

Reviewing what we have defined, the oracle $\bar{\Omega}$ can be described as follows: it chooses a random pad $\vec{p}$, selects $\vec{s}$ as we have specified, and then, to a component query of i preceding the output query, $\bar{\Omega}$ responds $\vec{x}_i\vec{p}_i\vec{s}_i$; to an output query of $\vec{x}'_T\vec{p}'_T\vec{s}'_T$, $\bar{\Omega}$ responds with $y \oplus (\vec{p}_{\overline{T}} \cup \vec{p}'_T)$, where $y \leftarrow f(\vec{x}'_T \cup \vec{x}_{\overline{T}})$; and to subsequent output queries i, $\bar{\Omega}$ responds with $(x_i r_i p_i s_i,\ p_i \oplus y_i)$.

There is another way to describe the behavior of $\bar{\Omega}$, which is identical. Namely, $\bar{\Omega}$ chooses random $\vec{r}$ and $\vec{p}$ as above, selects $\vec{s}$ as we have specified, and flips coins to select $\vec{w} \leftarrow (\{0,1\}^\ell)^n$. Then, to a component query of i preceding the output query, $\bar{\Omega}$ responds $\vec{x}_i\vec{p}_i\vec{s}_i$; to an output query of $\vec{x}'_T\vec{p}'_T\vec{s}'_T$, $\bar{\Omega}$ selects $y \leftarrow f(\vec{x}'_T \cup \vec{x}_{\overline{T}})$ and responds with $\vec{w}_{\overline{T}} \cup \vec{p}_T \oplus y_T$; and to subsequent output queries i, $\bar{\Omega}$ responds with $(x_i\ w_i\ s_i,\ w_i \oplus y_i)$.

The advantage of this description of $\bar{\Omega}$ is that now it is apparent that $\bar{\Omega}$ can be simulated using only the aid of an $O_i(\vec{x}; f)$ oracle; we thus change the arguments to this oracle to $\bar{\Omega}(\vec{x}; f)$.

Protocol P is private.

Consider the simulator $\dot{S}$ which is identical to $\bar{S}$ except that in any transcript τ that $\bar{S}$ would output in which a player had a private output of y and a random string prefixed by p_i, $\dot{S}$ outputs the same transcript, but with player i having private output of y_i given by Equation 5. This can be considered as applying a simple function, h, to the transcripts τ that $\bar{S}$ would output, and outputting $h(\tau)$ instead of outputting τ. Applying h to both sides of Equation 7, the definitions of $\bar{P}$, P, $\bar{S}$ and $\dot{S}$ gives that

$$\mathrm{VIEW}^k_{A(a),P(\vec{x})} \approx \dot{S}^{\bar{\Omega}(\vec{x};f)}_{A(a)}.$$

Oracle S is defined to behave like $\dot{S}$, except that it simulates the behavior of $\bar{\Omega}$ with the aid of an $O_i(\vec{x}; f)$ oracle; we have already discussed why this is possible. By construction, then, the last assertion gives that

$$\mathrm{VIEW}^k_{A(a),P(\vec{x})} \approx S^{O_i(\vec{x};f)}_{A(a)},$$

as desired.

Protocol P is correct.

The committal function $\hat{\chi}_A$ was a well-defined committal function associated with simulator $\hat{S}$ for A attacking $\hat{P}$. By our definition of the simulator S, $\chi_A = \hat{\chi}_A$ remains a well-defined committal function associated with S, for A attacking P. We know that, almost certainly (over coin flips $\vec{r}$ and r_A, and uniformly, across $(\vec{x}\vec{p}\vec{s}, a)$), each good player running $\hat{P}$ computes $\hat{f}_i(\hat{\chi}(\vec{x}\vec{p}\vec{s}, \vec{r}, r_A, a, k))$. Whenever this event occurs, each good player i in the protocol P computes $\hat{f}_i(\hat{\chi}(\vec{x}\vec{p}\vec{s}, \vec{r}, r_A, a, k)) \oplus p_i = f_i(\chi(\vec{x}, \vec{p}\vec{r}, r_A, a, k))$. Almost certainly, then, each good player i computes $f_i(\chi(\vec{x}, \vec{r}, r_a, a, k))$.

■

5.4 Independence of Committed Inputs

This section demonstrates that what the adversary commits to in an execution of a secure protocol is essentially *independent* of the inputs held by currently uncorrupted players. Stating this theorem in a precise yet intuitive way is tricky — both because the adversary necessarily *can* commit values which are quite dependent on the values held by good player (but somehow they are not *meaningfully* correlated), and also because the number of players and the input length are *not* regarded as being fixed.

5.4.1 Minimum dependence on input distributions

Let $I = \{I_c\}$ and $I' = \{I'_c\}$ be families of distributions, each $c \in \bigcup L_k$ specifying distributions I_c and I'_c on tuples $(\vec{x}, \vec{a}, a_A, c) \in L_k$. We wish to investigate the adversary's limitations in guessing which of the two distributions the initial configuration is drawn from. To do this, consider the following two games:

Game 1 [Distinguish I and I' using a secure protocol]. A protocol P which t-securely computes $f = \{f_c\}$ is run in the presence of a slightly "special" polynomial-time adversary A. The speciality of the adversary consists of her being able to specify—via a network interaction function

$$Guess: \underbrace{\{0,1\}^*}_{\substack{\text{current}\\ \text{state}}} \rightarrow \underbrace{\{0,1\}^*}_{\substack{\text{guess of}\\ I \text{ or } I'}}$$

which gives A's "guess" as to whether the input configuration was drawn from I or from I'. When the adversary's committal becomes well defined (as specified by $\mathcal{AI}$) the bit specified by *Guess* is taken to be this guess.

To quantify how well the adversary A does in distinguishing I from I' during the execution of protocol P, define the *adversarial distinguishability* of these ensembles by

$$\begin{aligned} \textit{adv-dist}_{A,P,I,I'}(c) \;=\; & \big|\, \mathbf{E}_{(\vec{x},\vec{a},a_A,c)\leftarrow I_c}\; \textit{Guess}_{A,P}(\vec{x},\vec{a},a_A,c,\ \vec{r},r_A) - \\ & \quad \mathbf{E}_{(\vec{x},\vec{a},a_A,c)\leftarrow I'_c}\; \textit{Guess}_{A,P}(\vec{x},\vec{a},a_A,c,\ \vec{r},r_A) \,\big|, \end{aligned}$$

where $\textit{Guess}_{A,P}(\vec{x},\vec{a},a_A,c,\ \vec{r},r_A)$ denotes A's guess when the initial configuration is specified by $(\vec{x},\vec{a},a_A,c,\ \vec{r},r_A)$.

Game 2 [Distinguish I and I' using component queries]. Let D_a be a distinguisher having access to an ideal evaluation oracle $\Omega(\vec{x},\vec{a})$. (That is, the oracle only makes component queries, no output queries. There is, implicitly, some bound t on the number of permitted queries.) By $D_{\Omega(\vec{x},\vec{a})}(c,a_A)$, where $c = 1^k\#1^n\#1^l\#1^l\#1^m\#C_c$, we denote the distribution induced on D's output when its input is (c,a_A) and its oracle responds to up to t component queries i with (x_i,a_i).

To quantify how well D_a distinguishes between I and I', define the *inherent distinguishability* of these families of distributions by

$$\textit{inherent-dist}_{D_a,I,I'}(c) = \left|\mathbf{E}_{(\vec{x},\vec{a},a_A,c)\leftarrow I_k}\; D_a^{\Omega(\vec{x},\vec{a})}(c,a_A) \;-\; \mathbf{E}_{(\vec{x},\vec{a},a_A,c)\leftarrow I'_k}\; D_a^{\Omega(\vec{x},\vec{a})}(c,a_A)\right|.$$

The inherent distinguishability of ensembles I and I' can be considered as a lower-bound on the extent to which they can be distinguished by polynomial-time algorithms given the ability to adaptively "open" a bounded number of components of $(\vec{x},\vec{a})$.

One expression of independence of what the adversary is committing to lies in the fact that the adversary cannot win at Game 1—where she gets to run protocol P—significantly better than she could win at Game 2—where she does not bother to run the protocol at all, but just talks to an ideal evaluation oracle which can make at most t component queries, instead. More formally, we have the following theorem:

Theorem 5.9 *Let P is a t-secure protocol for some function family f. Then for any t-adversary A there exists a distinguisher D_a such that for all pairs of ensembles $I = \{I_c\}$ and $I' = \{I'_c\}$,*

$$\epsilon(k) = \sup_{c=1^k\#\cdots} |\textit{adv-dist}_{A,P,I,I'}(c) - \textit{inherent-dist}_{D,I,I'}(c)|$$

is negligible.

Proof sketch: Since P is a secure protocol for f, there exists a simulator S which, when given an $\Omega_f(\vec{x},\vec{a})$-oracle and interacting with an adversary A having advice a_A and sharing common input c with the network, provides to A a view computationally indistinguishable from that which the network provides on input $(\vec{x},\vec{a},a_A,c)$. The distinguisher D_a existentially guaranteed by the theorem is constructed from S by simulating S interacting with A—but halting this simulation at the point at which S attempts to make its output query. The output query is not made; instead, at this point D reads off A's guess *Guess*, outputs the bit which it specifies, and terminates. Note that the simulation—since it never actually made its output query—can be carried out with only an $\Omega_f(\vec{x},\vec{a})$ oracle.

To see that this algorithm D achieves the bound of Theorem 5.9, note that, by Proposition 5.4, the ensemble provided by S to A when the initial configuration is I-distributed is computationally indistinguishable over I to that provided to A by the network when the initial configuration is I-distributed. Similarly, the ensemble provided to A by S is computationally indistinguishable over I' to that provided by the network when the initial configuration is I'-distributed. The result follows by Lemma 5.5 and the transitivity of indistinguishability (Proposition 5.2). ■

REMARK. *Add in: The same theorem with f value given to both.*

5.4.2 Minimum dependence on good inputs

Let $I = \{I_c\}$ be a family of distributions, each I_c a distribution on tuples $(\vec{x},\vec{a},a_A,c)$. Let B be *any* predicate on vectors, $B : \bigcup_{n\geq 3,\ell\geq 0}(\{0,1\}^\ell)^n \to \{0,1\}$. (Predicate B need not even be computable, even for the case computationally bounded case of this theorem.) Consider the following two games:

Game 1′ [Guess B using a secure protocol]. A protocol P which securely computes $f = \{f_c\}$ is run in the presence of a polynomial-time t-adversary A. The adversary is again equipped with a network interaction function to allow it to specify a "guess" *Guess*. This time, *Guess* is interpreted as specifying a a "guess" by A as to the value $B(\vec{x})$, a guess which is specified when the committal is made by A. The input configuration is distributed according to I_c.

To quantify how well the adversary does in guessing the predicate B, define the *adversarial approximability* of B by A over I as

$$\textit{adv-approx}_{A,P,B,I}(c) = \text{Prob}[\textit{Guess}_{A,P}(\vec{x},\vec{a},a,c,\ \vec{r},r_A) = B(\vec{x})],$$

where the probability is taken over $(\vec{x},\vec{a},a_A,c)$ being distributed according to I_k (where $c = 1^k\cdots$), and the uniform distribution on coins $\vec{r}$ and r_A. Here $\textit{Guess}_{A,P}(\vec{x},\vec{a},a,c,\ \vec{r},r_A)$ denotes the guess that A makes (as specified by *Guess*) in the execution of protocol P when the initial configuration is given by $(\vec{x},\vec{a},a,c,\ \vec{r},r_A)$.

Game 2′ [Guess B using component queries]. Let G be a distinguisher having access to an ideal evaluation oracle $\Omega(\vec{x},\vec{a})$. By $G^{\Omega(\vec{x},\vec{a})}(c,a_A)$, where $c = 1^k\#1^n\#1^\ell\#1^l\#1^m\#C_c$, we denote the distribution induced on G's output its input is (c,a_A) and its oracle responds to up to t component queries i with (x_i,a_i).

It is G's job to guess the value of $B(\vec{x})$. To quantify how well D does this, define the *inherent approximability* of predicate B by

$$\textit{inherent-approx}_{G,B,I}(c) = \text{Prob}[G^{\Omega_f(\vec{x},\vec{a})}(c,a_A) = B(\vec{x})],$$

where the probability is taken over $(\vec{x},\vec{a},a_A,c)$ being I_k-distributed. The inherent approximability of predicate B over I can be considered as a lower bound on how accurately $B(\vec{x})$ can be guessed

by a polynomial-time algorithm when the initial configuration is I-distributed and the algorithms are given the ability to adaptively "open" a bounded number of components of $(\vec{x}, \vec{a})$.

One expression of independence of what the adversary is committing to lies in the fact that the adversary cannot win at Game 1′—where she guesses B based on a run of the protocol P—significantly better than she could win at Game 2′—where she guesses B by just talking to a t-bounded oracle. More formally, we have the following theorem:

Theorem 5.10 *Let P be a t-secure protocol for a function family f. Then for all polynomial-time t-adversaries A there exists a polynomial-time distinguisher G such that for all families of distributions $I = \{I_c\}$ and for all predicates B,*

$$\epsilon(k) = \sup_{c=1^k\ldots} \left| \textit{adv-approx}_{A,P,B,I}(c) - \textit{inherent-approx}_{G,B,I}(c) \right|$$

is negligible.

Proof sketch: The proof is similar to that of Theorem 5.9. As before, fixing an adversary A and letting S be the simulator establishing security, the guessing algorithm G is constructed from S by running the simulation of S interacting with A, but halting this simulation at the point at which S attempts to make its output query. The output query is not made; instead, G reads A's guess, outputs this bit, and terminates. That the distinguisher G approximates B nearly as well as the adversary approximates B when attacking the network running P and when the initial configuration is I-distributed follows along the same lines as Theorem 5.9. ■

5.5 Reducibility

Suppose you want to design a secure protocol for some complicated task—computing some function f, say. In an effort to make more manageable your job as protocol designer, you assume in designing this protocol that you have some *primitive*, g, at your disposal. (As an example, you might wish to design protocols assuming the ability to perform an oblivious transfer between any pair of players.) You prove that P^g securely computes f with respect to a "special" model of computation, one for which an ideal evaluation of the primitive g is provided.

Now suppose you have continued your work and designed a protocol P_g which securely computes g. It is secure in the "generic" model of computation.

One would hope that you obtain a secure protocol for f (in the "generic" model of computation) by inserting the code P_g wherever it is necessary in P^g that g be computed. Such a *reducibility* property offers the promise that we can not only design protocols top-down, but that, associated to this design scheme, is a modular proof of security. On the other hand, without reducibility, modular design of secure protocols may be impossible.

In this section, we show that our notion of security allows reducibility, establishing, after the necessary preliminaries, a powerful *reducibility theorem* in Section 5.5.5

Before launching into effort, we wish to emphasize that reducibility is not the same as *composability*. In fact, reducibility is more subtle and interesting than serial or parallel composition. These properties capture the sense in which, say, the concatenation of secure protocols remains a secure protocol, or the parallel execution of secure protocols remains a secure protocols; but reducibility captures the *sense* in which an ideal protocol can be replaced by a secure protocol. An explanation of this notion necessarily requires the construction of specific computational models and notions of security.

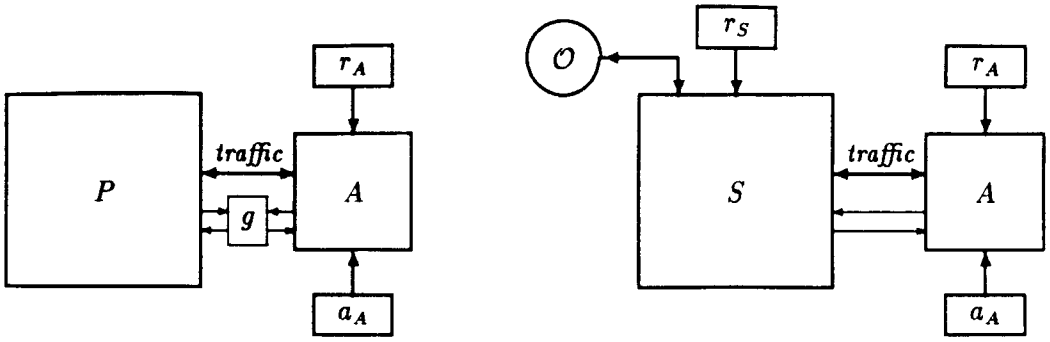

Figure 4: **Left:** *a protocol and an adversary, sharing the ability to do ideal g-evaluation. The traffic is defined to include the information coming into and going out of the adversary-side of the box labeled g.* **Right:** *the simulator providing all traffic to A — including the appearance of the box labelled g.*

Some of this section is rather formal. We comment that we have found it impossible to reason properly about reducibility when "intuitively clear" concepts like how to compose protocols are not formalized; thus, we must specify an *extremely explicit* model of computation—and one tailored to making it relatively facile to state and prove our theorems.

5.5.1 Issues in reducibility

Reducibility brings to light a variety of issues, including the need for devising special-purpose models of computation and their corresponding notions of security, the importance of input and output awareness, and the need for auxiliary inputs and adversary advice. We briefly discuss these issues in this subsection.

SPECIAL-PURPOSE MODELS. When a protocol is designed under the assumption that we provide a certain primitive "for free," there must be formal notions crafted to reflect the new scenario.

The abstraction this special-purpose model is meant to capture can be described as follows. The players are given a physical "box" to use to collaboratively compute g. At the beginning each round, good and bad players insert values along the input wires to this box. At the beginning of the *next* round, good and bad player get the appropriate output back along the output wires of this box. The box, of course, faithfully computes the function g applied to the vector of values coming in along its input wires.

We need to make this abstraction into a definition of *a protocol in the model of computation allowing ideal evaluation of g.* For simplicity, we imagine that the protocol makes just one call to the ideal evaluation of g, for the case where this occurs more often (or even every round) is then easily handled. Explicitly, at the end of some particular player round $\bar{r}$, each player is to have "written" in his computational state his input w_i to the ideal evaluation of g. Likewise, the adversary, at the end of her round $\bar{r}$, must have "written" in her computational state some tagged vector w'_T, where T is the set of currently-corrupted players. At the beginning of the player's next round, each good player i learns the $g_i(w_{\overline{T}} \cup w'_T)$, while, at the beginning of the adversary's next round, she is given $g_T(w_{\overline{T}} \cup w'_T)$. This model of computation is pictorially depicted on the left hand side of Figure 5, and it will be formalized in Section 5.5.3.

After understanding the revised model, we must describe what security means for it. The

notion is essentially the same as before: in particular, one demands a simulator that produces the appropriate views for the adversary. Of course, these views now include the output from the ideal evaluation of g. But the simulator is given no special abilities to produce these outputs. The notion of player and adversary traffic, though, is slightly amended to accommodate the new scenario, as we now discuss.

AWARENESS. Note that if the players have a physical "box" to use to collaboratively compute g, then the values w'_T that the adversary inserts along the "input wires" to this box should thus be considered as part of her traffic—something the adversary is definitely is aware of. Similarly, the values $g_T(w'_T \cup w_{\overline{T}})$ that the adversary gets back along the "output wires" of the box are again something she definitely knows, and should be considered as part of her traffic.

The construction of the simulator implicit in the claim that we have a secure protocol in the model of computation providing an ideal evaluation for g will, in general, need to have the simulator "see" the adversary's contribution w'_T: if the simulator were denied this information, it could not, in general carry out the simulation—as can be seen by considering the case where g is the identity function, $g_i(\vec{w}) = w_i$.

Furthermore, modeling the idea that the adversary can extract the output received from the box is important. Imagine the "box" being replaced by a protocol, and suppose an adversary derived from A^g—an adversary designed to attack the protocol with ideal g-evaluation—is to be set loose on the composite protocol. Even if the adversary promises to make no corruptions during the execution of the subprotocol, there is still no natural sense in which the adversary attacking the composite protocol can be derived from an adversary attacking the protocol with ideal g-evaluation—unless we mimic the possession of adversary output in the ideal evaluation of g. This would show up as a technical impediment to establishing reducibility, prohibiting the natural construction of a simulator.

AUXILIARY INPUTS AND ADVERSARY ADVICE. At the time at which an abstract protocol for g is invoked, the players are in some computational state. Most of this computational state is irrelevant to the computation of g. However, it might be highly correlated with other player's inputs to g. Were we not to have included auxiliary inputs in our formulation of security we would be unable to assume that the adversary, being granted this information associated to player i when she corrupts him, would admit simulatability. Similar statements apply to the adversary's advice.

5.5.2 Subroutine composition

For protocols designed for subroutine composition, we augment the notion of a protocol as follows: in addition to the four interaction functions associated with a protocol—η, M, m and o—we provide a *subroutine input function*

$$\iota\colon \underbrace{\{0,1\}^*}_{\substack{\text{current}\\ \text{state}}} \to \underbrace{\{0,1\}^\ell}_{\substack{\text{input to}\\ \text{subroutine}}}$$

To describe our reducibility theorem, it is necessary to specify the mechanics by which one protocol *calls* another. For simplicity, we will describe only the case in which a protocol P calls an R-round protocol Q, and it does this at some particular round $\bar{r}$.

The sequence of computational states that a player goes through is depicted in Figure 5. We establish the convention that there is no activity in the players' rounds 0, $\bar{r}+1$, and $\bar{r}+R+2$, and, consequently, there is no message delivery to the players in rounds 0, 1, $\bar{r}+1$, $\bar{r}+2$, $\bar{r}+R+2$, $\bar{r}+R+3$. Notice how saved state from the subroutine call is not considered part of the computational state

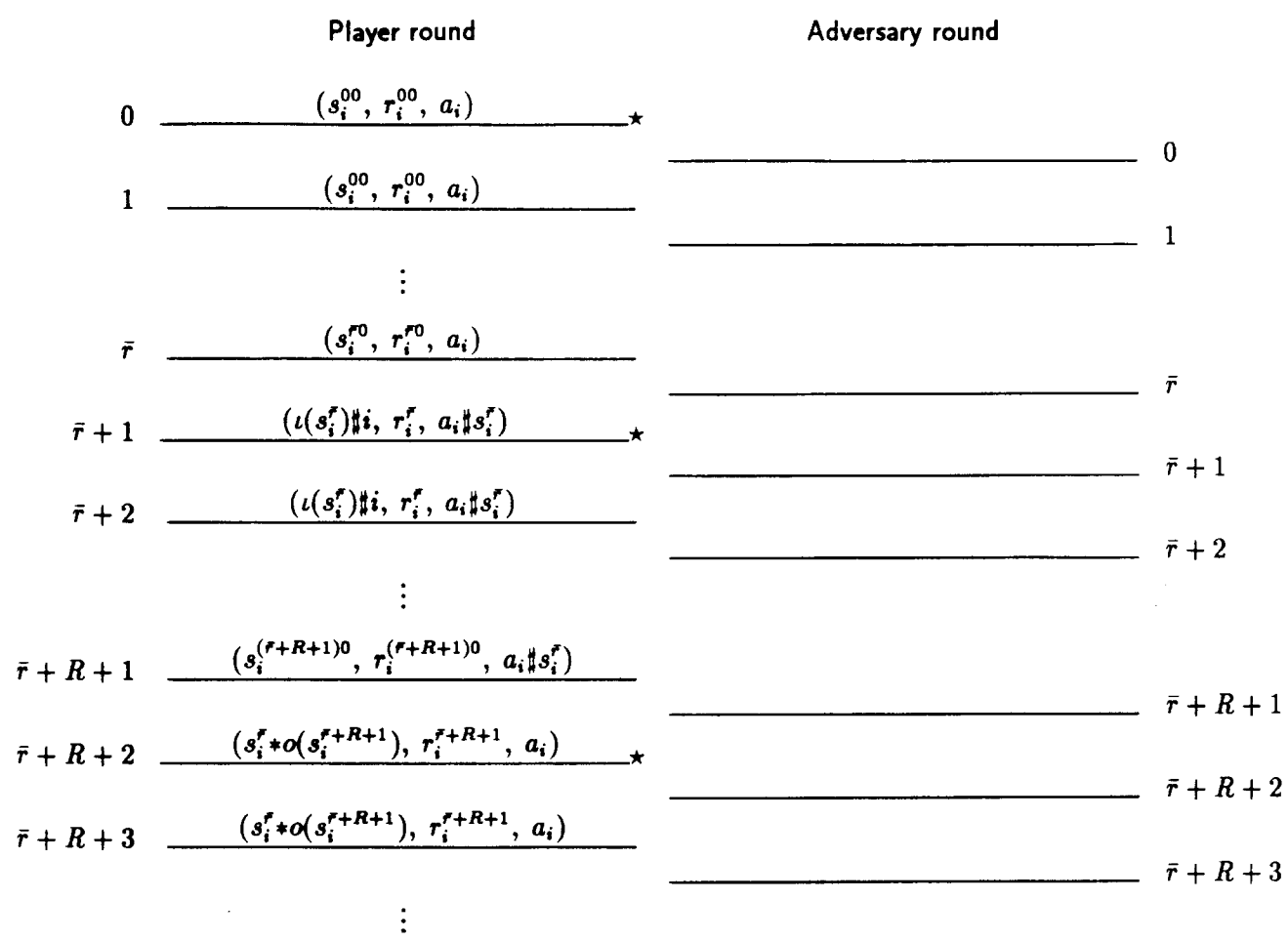

Figure 5: *The sequence of configurations at the beginning of player i macro-rounds in a protocol that calls an R-round protocol at time $\bar{r}$.*

of a player; this ensures that a protocol only depends on what it "should" depend on. Formally, the sequence of player configurations is defined as follows:

$$C_i^{r(\rho+1)} = \begin{cases} (P(c, s_i^{r\rho}),\ r_i^{r\rho},\ a_i^{r\rho}) & \text{if } \mathcal{N}(s_i^{r\rho}) = \textsf{compute and either } 1 \le r \le \bar{r} \text{ or } r \ge \bar{r}+R+3 \\ (Q(c, s_i^{r\rho}),\ r_i^{r\rho},\ a_i^{r\rho}) & \text{if } \mathcal{N}(s_i^{r\rho}) = \textsf{compute and } r \in [\bar{r}+2..\bar{r}+R+1] \\ (s_i^{r\rho} * r_{i1}^{r\rho},\ r_{i2}^{r\rho} r_{i3}^{r\rho} \cdots,\ a_i^{r\rho}) & \text{if } \mathcal{N}(s_i^{r\rho}) = \textsf{flip-coin and } r \notin \{0, \bar{r}+1, \bar{r}+R+2\} \\ C_i^{r\rho} & \text{otherwise} \end{cases}$$

$$C_i^{(r+1)0} = \begin{cases} (s_i^r * \mathcal{M}_i^r,\ r_i^r,\ a_i^r) & \text{if } \mathcal{N}(s_i^r) = \textsf{round-done} \text{ and} \\ & \quad r \notin \{0, \bar{r}, \bar{r}+1, \bar{r}+R+1, \bar{r}+R+2\} \\ (\iota(s_i^r) \sharp i,\ r_i^r,\ a_i^r \sharp s_i^r) & \text{if } r = \bar{r} \\ (s_i^{\bar{r}} * o(s_i^r),\ r_i^r,\ a_i^r) & \text{if } r = \bar{r}+R+1 \\ C_i^r & \text{otherwise} \end{cases}$$

while the adversary's sequence of configurations is given by:

$$C_A^{r(\rho+1)} = \begin{cases} (A(c, s_A^{r\rho}),\ r_A^{r\rho},\ a_A^{r\rho},\ \kappa_A^{r\rho}) & \text{if } \tilde{\mathcal{N}}(s_A^{r\rho}) = \textsf{compute} \\ (s_A^{r\rho} * r_{A1}^{r\rho},\ r_{A2}^{r\rho} r_{A3}^{r\rho} \cdots,\ a_A^{r\rho},\ \kappa_A^{r\rho}) & \text{if } \tilde{\mathcal{N}}(s_A^{r\rho}) = \textsf{flip-coin} \\ (s_A^{r\rho} * a_{A1}^{r\rho},\ r_A^{r\rho},\ a_{A2}^{r\rho} a_{A3}^{r\rho} \cdots,\ \kappa_A^{r\rho}) & \text{if } \tilde{\mathcal{N}}(s_A^{r\rho}) = \textsf{get-advice} \\ (s_A^{r\rho} * \sigma_i^r,\ r_A^{r\rho},\ a_A^{r\rho},\ \kappa_A^{r\rho} \cup \{i\}) & \text{if } \tilde{\mathcal{N}}(s_A^{r\rho}) = \textsf{corrupt}_i \\ C_A^{r\rho} & \text{otherwise} \end{cases}$$

$$C_A^{(r+1)0} = \begin{cases} (s_A^r * \dot{\mathcal{M}}_A^{r+1},\ r_A^r,\ a_A^r,\ \kappa_A^r) & \text{if } r \notin \{0, \bar{r}, \bar{r}+1, \bar{r}+R+1, \bar{r}+R+2\} \\ (s_A^r *,\ r_A^r,\ a_A^r,\ \kappa_A^r) & \text{otherwise} \end{cases}$$

where

$$\sigma_i^r = \begin{cases} s_i^r * x_i * a_i & \text{if } r \in \{0, 1, \bar{r}+1, \bar{r}+2, \bar{r}+R+1, \bar{r}+R+2\} \\ s_i^r * x_i * a_i * \bar{m}_{1i}^r * \cdots * \bar{m}_{ni}^r * E(H_i^r) & \text{otherwise} \end{cases}$$

and all other notation is as before. The player and adversary traffic have the same formal definition as given previously, except that we add the proviso that all messages ($M_i^r, \dot{m}_{ij}^r$, etc.) are defined to be the empty string, for rounds 0, $\bar{r}+1$, and $\bar{r}+R+2$.

REMARK. Notice that if the adversary corrupts a player i during the execution of a subprotocol, the adversary necessarily "learns" player i's input $\iota_i = \iota(s_i^{\bar{r}})$ into the subprotocol, since the saved state $s_i^{\bar{r}}$ was preserved and given to the adversary on corrupting i. The same preservation of player input values was built into the definition of executing protocols in the presence of an adversary—in the handing to the adversary on the corruption of player i the initial private input x_i belonging to that player.

5.5.3 Protocols with an ideal evaluation

Let $g = \{g_c : (\{0,1\}^t)^n \to (\{0,1\}^l)^n$. We consider an enriched model of computation in which the players can compute the function g "for free."

To define the running in the presence of an adversary a protocol P^g which assumes the ideal evaluation of a function g, we augment the notion of a protocol as we did for subroutine compositions,

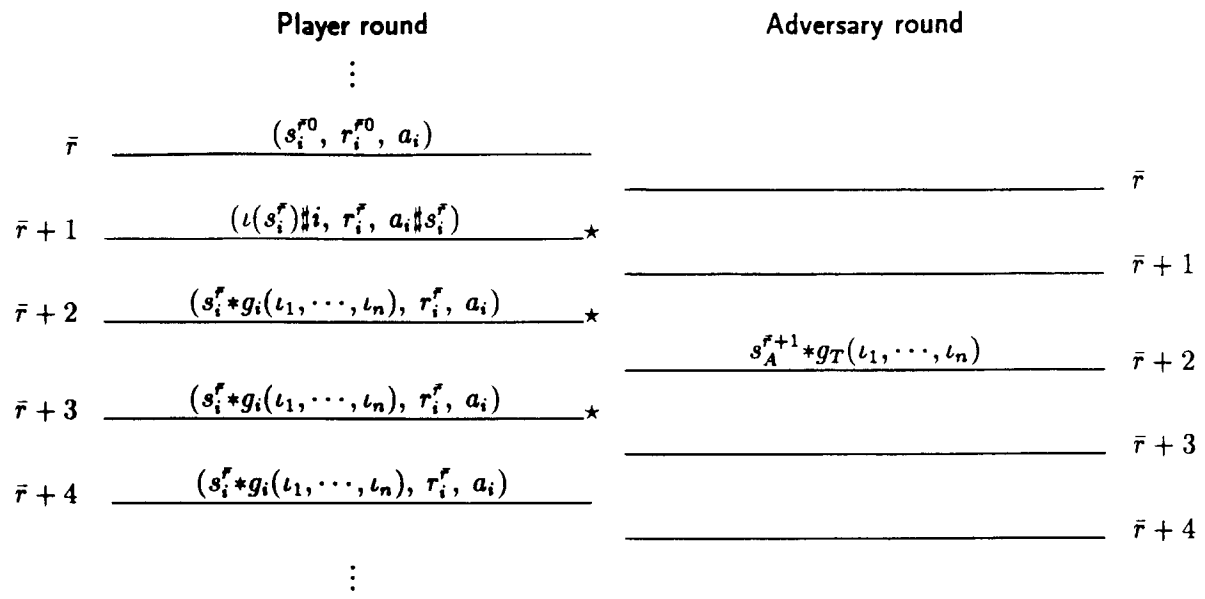

Figure 6: *The sequence of configurations at the beginning of each macro-round for a player i in a protocol in the model of computation providing ideal g-evaluation executed at round $\bar{r}$.*

by adding a function ι on player computational states,

$$\iota\colon \underbrace{\{0,1\}^*}_{\substack{\text{current}\\ \text{state}}} \to \underbrace{\{0,1\}^\ell}_{\substack{\text{input}\\ \text{to } g}}$$

Similarly, we augment the notion of an adversary by adding to the four interaction functions associated to an adversary a *g-input function*

$$\hat{\iota}\colon \underbrace{\{0,1\}^*}_{\substack{\text{current}\\ \text{state}}} \to \underbrace{2^{[1..n]\times\{0,1\}^\ell}}_{\substack{\text{input}\\ \text{to } g}}$$

The behavior of a protocol with a ideal g-evaluation is depicted in Figure 6. We establish the convention that there is no activity in rounds 0, $\bar{r}+1$, $\bar{r}+2$, or $\bar{r}+3$, and, consequently, no message delivery in rounds 0, 1, $\bar{r}+1$, $\bar{r}+2$, or $\bar{r}+3$, and $\bar{r}+4$. The sequence of player and adversary configurations is formally defined as follows:

$$C_i^{r(\rho+1)} = \begin{cases} (P(c, s_i^{r\rho}), r_i^{r\rho}, a_i^{r\rho}) & \text{if } \mathcal{N}(s_i^{r\rho}) = \textsf{compute} \text{ and } r \notin \{0, \bar{r}+1, \bar{r}+2, \bar{r}+3\} \\ (s_i^{r\rho} * r_{i1}^{r\rho}, r_{i2}^{r\rho} r_{i3}^{r\rho} \cdots, a_i^{r\rho}) & \text{if } \mathcal{N}(s_i^{r\rho}) = \textsf{flip-coin} \text{ and } r \notin \{0, \bar{r}+1, \bar{r}+2, \bar{r}+3\} \\ C_i^{r\rho} & \text{otherwise} \end{cases}$$

$$C_i^{(r+1)0} = \begin{cases} (s_i^r * \mathcal{M}_i^r,\ r_i^r,\ a_i^r) & \text{if } r \notin \{0, \bar{r}..\bar{r}+2\} \\ (\iota(s_i^r) \sharp i,\ r_i^r,\ a_i^r \sharp s_i^r) & \text{if } r = \bar{r} \\ (s_i^r * g_i(\iota_1, \cdots, \iota_n),\ r_i^r,\ a_i^r) & \text{if } r = \bar{r}+1 \\ C_i^r & \text{otherwise} \end{cases}$$

while the sequence of configurations the adversary goes through is given by

$$C_A^{r(\rho+1)} = \begin{cases} (A(c, s_A^{r\rho}),\ r_A^{r\rho},\ a_A^{r\rho},\ \kappa_A^{r\rho}) & \text{if } \tilde{\mathcal{N}}(s_A^{r\rho}) = \textsf{compute} \\ (s_A^{r\rho} * r_{A1}^{r\rho},\ r_{A2}^{r\rho} r_{A3}^{r\rho} \cdots,\ a_A^{r\rho},\ \kappa_A^{r\rho}) & \text{if } \tilde{\mathcal{N}}(s_A^{r\rho}) = \textsf{flip-coin} \\ (s_A^{r\rho} * a_{A1}^{r\rho},\ r_A^{r\rho},\ a_{A2}^{r\rho} a_{A3}^{r\rho} \cdots,\ \kappa_A^{r\rho}) & \text{if } \tilde{\mathcal{N}}(s_A^{r\rho}) = \textsf{get-advice} \\ (s_A^{r\rho} * \sigma_i^r,\ r_A^{r\rho},\ a_A^{r\rho},\ \kappa_A^{r\rho} \cup \{i\}) & \text{if } \tilde{\mathcal{N}}(s_A^{r\rho}) = \textsf{corrupt}_i \\ C_A^r & \text{otherwise} \end{cases}$$

$$C_A^{(r+1)0} = \begin{cases} (s_A^r * \dot{\mathcal{M}}_A^{r+1},\ r_A^r,\ a_A^{r\rho},\ \kappa_A^r) & \text{if } r \notin \{0, \bar{r}..\bar{r}+2\} \\ (s_A^r * g_T(\iota_1, \cdots, \iota_n),\ r_A^r,\ a_A^r,\ \kappa_A^r) & \text{if } r = \bar{r}+1 \\ (s_A^r *,\ r_A^r,\ a_A^r,\ \kappa_A^r) & \text{otherwise} \end{cases}$$

where

$$\iota_i = \begin{cases} \iota(s_i^{\bar{r}}) & \text{if } i \notin \kappa_A^{\bar{r}+1} \\ \tilde{\iota}(s_A^{\bar{r}+1}) & \text{if } i \in \kappa_A^{\bar{r}+1} \end{cases}$$

$$\iota_T = \{(i, \iota_i) \colon i \in \kappa_A^{\bar{r}+1}\}$$

and

$$\sigma_i^r = \begin{cases} s_i^r * x_i * a_i & \text{if } r \in \{0, 1, \bar{r}+1, \bar{r}+3, \bar{r}+4\} \\ s_i^r * x_i * a_i * g_i(\iota_1, \cdots, \iota_n) & \text{if } r = \bar{r}+2 \\ s_i^r * x_i * a_i * \bar{m}_{1i}^r * \cdots * \bar{m}_{ni}^r * E(H_i^r) & \text{otherwise} \end{cases}$$

Player i's round-r traffic is given by

$$t_i^r = \begin{cases} (M_1^r * M_2^r * \cdots * M_n^r * m_{1i}^r * m_{2i}^r * \cdots * m_{ni}^r,\ m_{i1}^r * m_{i2}^r * \cdots * m_{in}^r) & \text{if } r \neq \bar{r}+1 \\ (M_1^r * M_2^r * \cdots * M_n^r * m_{1i}^r * m_{2i}^r * \cdots * m_{ni}^r,\ m_{i1}^r * m_{i2}^r * \cdots * m_{in}^r,\ \iota_i) & \text{if } r = \bar{r}+1 \end{cases}$$

while the adversary's round-r traffic is given by

$$t_A^r = \begin{cases} (\dot{\mathcal{M}}_A^r,\ (i_1, \sigma_{i_1}^r), \ldots, (i_m, \sigma_{i_m}^r),\ \ddot{\mathcal{M}}_A^r) & \text{if } r \notin \{\bar{r}+1, \bar{r}+2\} \\ (\dot{\mathcal{M}}_A^r,\ (i_1, \sigma_{i_1}^r), \ldots, (i_m, \sigma_{i_m}^r),\ \ddot{\mathcal{M}}_A^r,\ \iota_T) & \text{if } r = \bar{r}+1 \\ (\dot{\mathcal{M}}_A^r,\ (i_1, \sigma_{i_1}^r), \ldots, (i_m, \sigma_{i_m}^r),\ \ddot{\mathcal{M}}_A^r,\ g_T(\iota_1, \cdots, \iota_n)) & \text{if } r = \bar{r}+2 \end{cases}$$

All messages, above, are defined to be the empty string for round $r \in \{0, \bar{r}+1, \bar{r}+2, \bar{r}+3\}$. The traffic *through* round-r is defined exactly as before.

REMARK. Notice how, in the formalization above, the traffic includes the values "input" by the adversary into the ideal evaluation of g, and the values "output" by this ideal evaluation are, in the following round, given back to the adversary.

5.5.4 Security in this model

The definition of security in the new model of computation is almost exactly what is inherited by using the new definition of how players and adversaries interact: *A protocol P t-securely computes a function f in the model of computation providing ideal g-evaluation if there exists a simulator S for it, a t-admissible adversary input function $\mathcal{AI}$, and an adversary output function $\mathcal{AO}$, such that ....* Note that the simulator is now responsible for providing the "simulated g-output," "$g_T(\iota_1, \ldots, \iota_n)$." Refer again to Figure 4 for a suggestive picture.

We said *almost* exactly because we add on a minor technical detail: *we forbid the f-function query to be made during the execution of the ideal g-evaluation.* More precisely, if $\mathcal{AI}(c, T_A^{\bar{r}}) =$ **not-yet**, then $\mathcal{AI}(c, T_A^{\bar{r}+3}) \neq$ **not-yet**, too.

This assumption can be justified in several ways. First, allowing an f-output query to be made by the simulator during the execution of the ideal g-evaluation goes against the intuition that we have properly reduced the computation of f to that of g. Second, we have chosen a formalization that allows no message delivery during the execution of the ideal g-evaluation—and it goes against the intuition (and the existence of the local input function) to say that a committal occurs at a moment when no messages are delivered. Finally, the assumption is necessary to prove our reducibility theorem.

5.5.5 The reducibility theorem

Under the assumption that g has a secure protocol, and when there is a protocol P^g that securely computes a function f in the model of computation which assumes ideal evaluation for g, we show that we can do without the enriched model of computation: by substituting into P^g the code for the protocol P_g that securely computes g, we get a secure protocol that securely computes f in the "generic" model of computation. Diagrammatically, we wish to do without the box labeled g that appear in Figure 4.

If P^g is a protocol in the model of computation that assumes the ideal evaluation of g at round $\bar{r}$, and if P_g is fixed-round protocol, then by P^{P_g} we denote the protocol that consists of P^g modified so as to call P_g in round $\bar{r}$; this is a protocol in the "standard" model of computation.

Theorem 5.11 *Let $t \in \mathbf{N}$, and let f: $(\{0,1\}^\ell)^n \to (\{0,1\}^l)^n$ be a function. Suppose there is a protocol P^g that t-securely computes f in the model of computation in which it can perform an ideal g-evaluation at some fixed round $\bar{r}$. Suppose also that there is an R-round protocol P_g that t-securely computes g. Then the protocol $P = P^{P_g}$, in which P^g calls P_g as a subroutine at round $\bar{r}$, is a t-secure protocol for computing f.*

Stronger statements of this theorem are possible, but this one was already difficult enough to state!

Proof sketch: This proof is an unfinished remnant from times past. It requires substantial updating, and is left here only to give some idea of the argument involved in the construction of the simulator.

Protocol P is the composite protocol consisting of P^g calling P_g at round $\bar{r}$, where P^g is the protocol that computes f in the model of computation assuming g-hardware active at round $\bar{r}$, and P_g is the R-round protocol computing g.

To show that P t-securely computes f, we must construct a simulator $S = (\mathcal{S}, \mathcal{AI}, \mathcal{AO})$ and show that the simulator "works" to establish privacy and correctness. Before beginning this, fix the following notation: that $S^g = (\mathcal{S}^g, \mathcal{AI}^g, \mathcal{AO}^g)$ is the simulator existentially guaranteed for P^g by its security, and $S_g = (\mathcal{S}_g, \mathcal{AI}_g, \mathcal{AO}_g)$ is the simulator existentially guaranteed for P_g by its security. We now specify, in turn, the functions $\mathcal{AI}$, $\mathcal{AO}$, and $\mathcal{S}$.

DEFINITION OF $\mathcal{AI}$. The adversary input function for P is derived from the adversary input function $\mathcal{AI}^g$ in a natural manner. Given traffic τ_A^r, if the traffic is so short that P_g has not yet been called, we apply $\mathcal{AI}^g$ directly to τ_A^r; if τ_A^r indicates that P_g is currently running, we return not-yet for $\mathcal{AI}(\tau_A^r)$, an action which will be justified by our assumption that a simulator for a protocol with g-hardware does *not* makes its f-output query during the simulation of the computation of g; and, finally, if τ_A^r indicates that P_g has already been completed, then we extract the portion of τ_A^r in which P_g was running, obtain values for $\mathcal{AI}_g$ and $\mathcal{AO}_g$ from this portion of the traffic, and, using these, construct a recoded version of τ_A^r to which $\mathcal{AI}^g$ is applied in order to compute $\mathcal{AI}$.

Making this more precise, for $r \geq \bar{r} + R + 2$, write

$$\begin{aligned} \tau_A^r \quad = \quad & c{*}\kappa_A{*}\tau_A^r[0..\bar{r}] \\ & \tau_A^r[\bar{r}+1..\bar{r}+R+1] \\ & \tau_A^r[\bar{r}+R+2..] \end{aligned}$$

where $\tau_A^r[0..\bar{r}]$ ends with the $(\bar{r}+1)^{th}$ "■"-character in τ_A^r; and $\tau_A^r[\bar{r}+1..\bar{r}+R+1]$ begins with the character following that and ends with the $(\bar{r}+R+2)^{nd}$ "■"-symbol in τ_A^r; and $\tau_A^r[\bar{r}+R+2..]$ begins with the character following that and includes all of the rest of the string τ_A^r. Let

$$\tau_g = c{*}\kappa_A'{*}\tau_A^r[\bar{r}+1..\bar{r}+R+1]$$

where

$$\kappa_A' = \kappa_A \cup \{i : \bullet i \bullet \text{ is a substring of } \tau_A^r[0..\bar{r}]\};$$

that is, κ_A' is the set of all processors that the traffic τ_A^r indicates were corrupted by time $(\bar{r},)$, and τ_g is the properly recoded traffic from the portion of the protocol in which P_g was being run.

Let ι_T be the non-void value $\iota_T = \mathcal{AI}_g(c, \tau_g[0..r'])$ associated to some prefix of τ_g, where $\tau_g[0..r']$ is the prefix of τ_g terminated by the $(r'+1)^{st}$ "■"-symbol. This is the "input" in the subprotocol, as extracted from the traffic. Let $\gamma_T = \mathcal{AO}_g(c, \tau_g)$; this is the "output" of the corrupted processors which the adversary could compute on their behalf.

Let $\{i_1, \ldots, i_\alpha\} = T$ be the sequence of processors, in order, corrupted by the adversary *before* the adversary's committal $\iota_T = \mathcal{AI}_g(c, \tau_g[0..r'])$ became defined, and let $\{i_{\alpha+1}, \cdots, i_\beta\} = T - T$ be the ordered sequence of processors corrupted *after* the adversary's committal ι_T became defined. Let $s_{i_j}^r$, $a_{i_j}^r$ and s_{i_j} be defined by the presence of the substring

$$\bullet i_j \bullet {*} s_{i_j}^{r_j} {*} \underbrace{s_{i_j}^r \sharp a_{i_j}^r \sharp s_{i_j}}$$

in τ_g. The "recoded traffic," $recode(\tau_A^r)$, is given by

$$recode(\tau_A^r) \quad = \quad c{*}\kappa_A{*}\tau_A^r[0..\bar{r}]$$

$$\bullet i_1 \bullet s^{\bar{r}}_{i_1} * a^{\bar{r}}_{i_1} * x_{i_1}$$
$$\vdots$$
$$\bullet i_\alpha \bullet s^{\bar{r}}_{i_\alpha} * a^{\bar{r}}_{i_\alpha} * x_{i_\alpha}$$
$$\iota_T \blacksquare$$
$$* \gamma_T$$
$$\bullet i_{\alpha+1} \bullet s^{\bar{r}}_{i_{\alpha+1}} * a^{\bar{r}}_{i_{\alpha+1}} * x_{i_{\alpha+1}} * \gamma_{i_{\alpha+1}}$$
$$\vdots$$
$$\bullet i_\beta \bullet s^{\bar{r}}_{i_\beta} * a^{\bar{r}}_{i_\beta} * x_{i_\beta} * \gamma_{i_\beta} \blacksquare$$
$$\tau^r_A[\bar{r} + R + 2..]$$

and, finally, define

$$\mathcal{AI}(\tau^r_A) = \begin{cases} \mathcal{AI}^g(\tau^r_A) & \text{if } r \leq \bar{r} \\ \textsf{not-yet} & \text{if } r \in [\bar{r}+1..\bar{r}+R+1] \\ \mathcal{AI}^g(\mathit{recode}(\tau^r_A)) & \text{if } r \geq \bar{r}+R+2 \end{cases}$$

DEFINITION OF $\mathcal{AO}$. In an r-round execution of an adversary with protocol P, the adversary output is defined by

$$\mathcal{AO}(\tau^r_A) = \mathcal{AO}^g(\mathit{recode}(\tau^r_A)).$$

DEFINITION OF S. The simulator S for P is produced by "properly combining" the simulators S^g and S_g. The main difficulty lies in properly dealing with the queries made by the simulator S_g, since S lacks the $\mathcal{O}(\vec{\iota}, \vec{\pi} * \vec{s}; g)$-oracle which S_g expects.

Simulator S will "simulate" the simulators S^g and S_g. To this end, it uses separate and independent coin flips r_{S^g} and r_{S_g}. The simulator S is defined as follows:

1. Rounds $0..\bar{r}$. Beginning with the simulator's round 0 and continuing until the simulators round $\bar{r}$, S behaves exactly as S^g behaves. That is, S *simulates* the behavior of S^g during these rounds, communicating with A exactly as S^g would communicate with A. At the end of this, the simulator S^g is in some computational state $s^r_{S^g}$, with some set of corrupted processors at this point κ^r_A.

2. Rounds $\bar{r}+1..\bar{r}+R+1$. Beginning with the simulator's round $\bar{r}+1$ and continuing until the completion of the simulator's round $\bar{r}+R+1$, simulator S runs the simulators S^g and S_g in the manner we now describe.

   Simulator S continues to interact with A, but now it uses the algorithm S_g to decide on its messages to A. Simulator S_g is initially in computational state $c * \kappa^r_A$. There is an immediate question that must be answered: how is the simulator S to answer S_g's oracle queries when it is not equipped with an $(\vec{\iota}, \vec{a} \sharp \vec{s}; g)$-oracle. We now describe how S does this:

   (a) When A corrupts a processor i and, consequently, S_g makes a *component query* of i, processor i has been corrupted, and so S "knows" a string $s^r_i * a_i \sharp s^r_i * x_i$. The algorithm S simulates S^g having just received a corruption of i and component query answered by $(\iota(s^r_i), a_i \sharp s^r_i)$. The simulator S_g outputs a string for A which S itself outputs for A.

(b) Similarly, S^g is told that processor i has just been corrupted, and S^g is given a component query response of $(x_i, \vec{a}_i)$. Simulator S^g prepares an outgoing message for the adversary with which it interacts, but this message is ignored by S.

(c) Suppose now that S_g make an output query of ι'_T. Then $\iota'_T\blacksquare$ is properly appended to the traffic τ^g between the simulator S^g and the adversary (indicating the end of round $\bar{r}+1$, as far as S^g is concerned), and S^g is allowed to run, determining a value $\gamma_T = \Phi(s_S^{(\bar{r}+1)0})$. The output query of S_g is now answered by γ_T.

(d) If, in the future, a processor i is corrupted, then S just learned a value $s_i^r a_i \sharp s_i^{\bar{r}} * x_i$. Simulator S^g is then executed with component query i answered by (x_i, a_i). The simulator S^g returns a value $s_i^r * a_i \sharp s_i^{\bar{r}} * \gamma_i$. The component query of S_g is answered by $((\iota(s_i^{\bar{r}}), a \sharp s_i^{\bar{r}}), \gamma_i)$.

(e) When round $\bar{r}+R+1$ has been completed, the traffic τ^g for the simulator S^g is updated by appending a "$\blacksquare$"-character, indicating—to S^g—the end of round $\bar{r}+2$. Simulator S^g is now in some computational state $s_S^{\bar{r}+2}$.

3. Rounds $[\bar{r}+R+2..\text{end}]$. During A's round $\bar{r}+R+2$ and continuing until termination, S behaves exactly as S^g dictates, beginning in state $s_S^{\bar{r}+2}$.

THE CONSTRUCTION WORKS. To argue correctness, we begin by constructing an adversary A^g—intended for attacking protocol P^g—and an adversary A_g—intended for attacking the protocol P_g. Of course these adversaries are constructed based on adversary A.

DEFINITION OF A_g. Adversary A_g is precisely adversary A, except that its advice is used to initialize its state. That is, the first thing A_g does is request some advice, the answer to which determines the initial state of A_g. Thereafter, A_g behaves exactly like A. The security of P_g

DEFINITION OF A^g. The adversary A^g behaves as follows:

1. Rounds $0..\bar{r}$. Between A^g's round 0 and A^g's round $\bar{r}$, inclusive, A^g behaves exactly as A behaves. That is, A^g *simulates* the behavior of A during these rounds, corrupting processors when A does, and communicating with the network with which it is running exactly as A would. At the end of this, the simulated adversary A is in some computational state $s_A^{\bar{r}}$.

2. Round $\bar{r}+1$. During A^g's round $\bar{r}+1$, A^g will "play a game in her head"—a game which we now describe. During this game, various processors will be corrupted, and the state of adversary A will be updated.

   In this game, the adversary A^g simulates the behavior of A—beginning in state s_A^r*—talking to the simulator S_g. The coins A^g uses in the simulation of S_g are distinct and uncorrelated to the coins A^g provides to the simulated adversary A. There is an immediate question that must be answered: how is adversary A^g to answer the simulator S_g's oracle queries when, in fact, A^g has no oracle? We now describe how A^g does this.

   (a) When, in the simulation, A corrupts a processor i and, consequently, S_g makes a *component query* of i, A^g actually *does* corrupt processor i. This results in A^g obtaining a string $c*\iota(s_i^{\bar{r}})*i*a_i*s_i$. The component query asked by S_g is then answered by $(c*\iota(s_i^{\bar{r}})*i,\ a_i*s_i^{\bar{r}})$.

   (b) When S_g makes an *output query* of ι_T, A^g encodes within its computational state the information required so that $\tilde{\iota}(s_{A^g}^{\bar{r}}) = \iota_T$ and $\tilde{\mathcal{N}}(s_{A^g}^{\bar{r}}) = \textbf{round-done}$. At this point in time, with the simulated adversary A in some computational state $s_A^{r'\rho'}$, A^g is done with its round $\bar{r}+1$ activities.

3. Round $\bar{r}+2$. At the beginning A^g's round $\bar{r}+2$, she has been presented a set of values γ_T. (If A^g is interacting with a network, $\gamma_T = g_T(\iota_1 \cdots \iota_n)$), where T is the set of currently corrupted processors and ι_i is given by Equation ??.) Adversary A^g uses γ_T to answer the oracle's output query, and the "mental game" A^g is playing using A (which is now in state $s_A^{r'\rho'}$) continues. This may result in additional corruptions by the simulated adversary A. When the simulated adversary A corrupts a processor i, resulting in S_g making a component query of i, A^g actually does corrupt processor i, obtaining a value $s_i^{\bar{r}} * \gamma_i * a_i$. The component query is then answered by $((c * \iota(s_i^r) * i,\ a_i * s_i^r),\ \gamma_i)$, and the simulation continues until A's round $\bar{r}+R+1$ has been completed. After that, with the simulated adversary A's in some computational state s_A^{r+R+1}, adversary A^g is done with her round $\bar{r}+2$, and enters round-done into her computational state.

4. Rounds $\bar{r}+3$..end. During A^g's round $\bar{r}+3$ and continuing until its termination, A^g continues to simulate the behavior of A (starting off in state $s_A^{r+R+1} *$—until A terminates. At this point, A^g terminates as well, outputting what A outputs.

■

5.6 Existence and Other Folklore

5.7 Discussion

Acknowledgments

In distilling our notion of secure computation we have benefitted highly from the beautiful insights of those who preceded us.

This work may not have come about without the earlier notions of secure function computation set forth by Yao [Ya82a, Ya86], and by Goldreich, Wigderson, and the first author [GMW87].

A more recent, fundamental source of inspiration was provided by the work of Kilian [Ki89] and the joint work of Kilian and the authors [KMR90]. Some of the knots solved here were first identified and/or untangled there.

An equally crucial role was played by the work of Claude Crépeau and the first author [Cr90], which also provided us with a wonderful source of examples that proved crucial for distilling our notion.

We have also gained plenty of crucial insights from Goldwasser's, Rackoff's, and the first author's work on the earlier, related notion of a zero-knowledge proof [GMR89].

Last but not least, we would like to thank the many friends with which we exchanged ideas about secure protocols for so many years.

I (the first author) was fortunate to have had Manuel Blum, Shafi Goldwasser, Oded Goldreich, Charles Rackoff, and Michael Fischer as traveling companions in very heroic times, when secure protocols were a totally unexplored and hostile territory. A bit more recently, my very special thanks go to my (cryptography) students Paul Feldman, Claude Crépeau, Phil Rogaway, Mihir Bellare, and Rafail Ostrovsky, from which I continue learning an enormous amount.

I (the second author) happily thank Mihir Bellare and Joe Kilian for many nice discussions on protocols and cryptography.

References

[Be91a] D. BEAVER, "Formal Definitions for Secure Distributed Protocols," in *Distributed Computing and Cryptography — Proceedings of a DIMACS Workshop*, October 1989. (Paper not presented at workshop but invited to appear in proceedings.)

[Be91b] D. BEAVER, "Foundations of Secure Interactive Computing," to appear in CRYPTO-91 Proceedings.

[BG89] D. BEAVER AND S. GOLDWASSER, "Multiparty Computations with Faulty Majority," *Proc. of the 30th FOCS* (1989), 468–473.

[BMR90] D. BEAVER, S. MICALI AND P. ROGAWAY, "The Round Complexity of Secure Protocols," *Proc. of the 22nd FOCS* (1990), 503-513.

[BF85] J. BENALOH (COHEN) AND M. FISCHER, "A Robust and Verifiable Cryptographically Secure Election Scheme," *Proc. of the 26th FOCS* (1985), 372–381.

[BGW88] M. BEN-OR, S. GOLDWASSER AND A. WIGDERSON, "Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation," *Proc. of the 20th STOC* (1988), 1–10.

[Bl82] M. BLUM, *Coin Flipping by Telephone, IEEE COMPCON*, (1982) 133–137.

[BM82] M. BLUM AND S. MICALI, "How to Generate Cryptographically Strong Sequences of Pseudo-Random Bits," *SIAM J. of Computing*, Vol. 13, No. 4, 1984, 850–864. Earlier version in *Proc. of the 23rd FOCS* (1982).

[Ch81] D. CHAUM, "Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms," *Comm. of the ACM* 24 (2) February 1981, 84–88.

[BCC88] G. BRASSARD, D. CHAUM AND C. CRÉPEAU, "Minimum disclosure proofs of knowledge," *Journal of Computer and System Sciences*, Vol. 37, No. 2, October 1988, 156–189.

[CCD88] D. CHAUM, C. CRÉPEAU AND I. DAMGÅRD, "Multiparty Unconditionally Secure Protocols," *Proc. of the 20th STOC* (1988), 11–19.

[CDG87] D. CHAUM, I. DAMGÅRD AND J. VAN DE GRAFF, "Multiparty Computations Ensuring the Privacy of Each Party's Input and Correctness of the Result," CRYPTO-87 Proceedings, 87–119.

[CG89] B. CHOR AND E. KUSHILEVITZ, "A Zero-One Law for Boolean Privacy," *Proc. of the 21st STOC* (1989), 62–72.

[CGK90] B. CHOR, M. GERÉB-GRAUS AND E. KUSHILEVITZ, "Private Computations over the Integers," *Proc. of the 31st FOCS* (1990), 325–344. Earlier version by Chor and Kushilevitz, "Sharing over Infinite Domains," CRYPTO-89 Proceedings, Springer-Verlag, 299–306.

[CGMA85] B. CHOR, O. GOLDWASSER, S. MICALI AND B. AWERBUCH, "Verifiable Secret Sharing and Achieving Simultaneity in the Presence of Faults," *Proc. of the 26th FOCS* (1985), 383-95.

[Cr90] C. CRÉPEAU, "Correct and Private Reductions Among Oblivious Transfers," MIT Ph.D. Thesis, February 1990.

[DLM] R. DEMILLO, N. LYNCH, AND M. MERITT, "Cryptographic Protocols," *Proc. of the 14th STOC* (1982) 383–400.

[DH76] W. DIFFIE AND M. HELLMAN, "New Directions in Cryptography," *IEEE Transactions on Information Theory*, **22**(6) (November 1976). 644-654.

[Ed65] J. Edmonds, "Paths, Trees, and Flowers," *Canadian J. of Mathematics*, 17:449–467, 1965.

[FFS87] U. FEIGE, A. FIAT, AND A. SHAMIR, "Zero Knowledge Proofs of Identity," *Proc. of the 19th STOC* (1987), 210–217.

[FS90] U. FEIGE AND A. SHAMIR, "Witness indistinguishability and witness hiding protocols," *Proc. of the 22nd STOC* (1990), 416–426.

[FS91] U. FEIGE AND A. SHAMIR, "On Expected Polynomial Time Simulation of Zero Knowledge Protocols," in *Distributed Computing and Cryptography — Proceedings of a DIMACS Workshop*, October 1989.

[Fe88] P. FELDMAN, "One Can Always Assume Private Channels," unpublished manuscript (1988).

[FM90] P. FELDMAN AND S. MICALI, "An Optimal Algorithms for Synchronous Byzantine Agreement," MIT/LCS Technical Report TM-425 (June 1990). Previous version in *Proc. of the 20th STOC* (1988), 148–161.

[GHY87] Z. GALIL, S. HABER AND M. YUNG, "Cryptographic Computation: Secure Fault-Tolerant Protocols and the Public-Key Model," CRYPTO-87 Proceedings, 135–155.

[Go89] O. GOLDREICH, "Foundations of Cryptography – Class Notes," Spring 1989, Technion University, Haifa, Israel.

[GL90] S. GOLDWASSER AND L. LEVIN, "Fair Computation of General Functions in Presence of Immoral Majority," CRYPTO-90 Proceedings, 75–84.

[GM84] S. GOLDWASSER AND S. MICALI, "Probabilistic Encryption," *Journal of Computer and System Sciences*, Vol. 28, No. 2 (1984), 270–299. Earlier version in *Proc. of the 14th STOC* (1982).

[GMR89] O. GOLDWASSER, S. MICALI, AND C. RACKOFF, "The Knowledge Complexity of Interactive Proof Systems," *SIAM J. of Comp.*, Vol. 18, No. 1, 186–208 (February 1989). Earlier version in *Proc. of the 17th STOC* (1985), 291–305.

[GMR88] S. GOLDWASSER, S. MICALI, AND R. RIVEST, "A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks," *SIAM Journal on Computing*, 17(2):281–308, April 1988.

[GMW87] O. GOLDREICH, S. MICALI AND A. WIGDERSON, "How to Play Any Mental Game," *Proc. of the 19th STOC* (1987), 218–229.

[GV87] O. GOLDREICH AND R. VAINISH, "How to Solve any Protocol Problem—An Efficiency Improvement," CRYPTO-87 Proceedings, 76–86.

[Ha88] S. HABER, "Multi-Party Cryptographic Computation: Techniques and Applications," Columbia University Ph.D. Thesis (1988).

[HU79] J. HOPCROFT AND J. ULLMAN, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

[Ki89] J. KILIAN, "Uses of Randomness in Algorithms and Protocols," MIT Ph.D. Thesis, April 1989.

[KMR90] J. KILIAN, S. MICALI, AND P. ROGAWAY, "The Notion of Secure Computation," manuscript, 1990.

[Le85] L. LEVIN, "One-Way Functions and Pseudorandom Generators," *Combinatorica*, Vol. 17, 1988, 357–363. Earlier version in *Proc. of the 17th STOC* (1985).

[LMR83] M. LUBY, S. MICALI AND C. RACKOFF, "How to Simultaneously Exchange a Secret Bit by Flipping a Symmetrically Biased Coin," *Proc of the 24th FOCS* (1983).

[Me83] M. MERITT, "Cryptographic Protocols." Georgia Institute of Technology Ph.D. Thesis, Feb. 1983.

[MRS88] S. MICALI, C. RACKOFF AND B. SLOAN, "The Notion of Security for Probabilistic Cryptosystems," *SIAM J. of Computing*, 17(2):412–26, April 1988.

[Or87] Y. OREN, "On the Cunning Power of Cheating Verifiers: Some Observations about Zero Knowledge Proofs," *Proc. of the 28th FOCS* (1987), 462–471.

[PSL80] M. PEASE, R. SHOSTAK AND L. LAMPORT, "Reaching Agreement in the Presence of Faults," *J. of the ACM* Vol. 27, No. 2, 1980.

[Ra81] M. RABIN, "How to Exchange Secrets by Oblivious Transfer," Technical Memo TR-81, Aiken Computation Laboratory, Harvard University, 1981.

[RB89] T. RABIN AND M. BEN-OR, "Verifiable Secret Sharing and Multiparty Protocols with Honest Majority," *Proc. of the 21st STOC* (1989), 73–85.

[SRA81] A. SHAMIR, R. RIVEST, AND L. ADLEMAN, "Mental Poker," in *Mathematical Gardener*, D. D. Klarner, editor, Wadsworth International (1981) pp 37–43,

[TW87] M. TOMPA AND H. WOLL, "Random Self-Reducibility and Zero Knowledge Interactive Proofs of Possession of Information," *Proc. of the 28th FOCS* (1987), 472–482.

[Ya82a] A. YAO, "Protocols for Secure Computation," *Proc. of the 23 FOCS* (1982), 160–164.

[Ya82b] A. YAO, "Theory and Applications of Trapdoor Functions," *Proc. of the 23 FOCS* (1982) 80–91.

[Ya86] A. YAO, "How to Generate and Exchange Secrets," *Proc. of the 27 FOCS* (1986).