# The Round Complexity of Secure Protocols

by
Phillip Rogaway

A.B., Computer Science
University of California, Berkeley
(1985)

S.M., Electrical Engineering and Computer Science
Massachusetts Institute of Technology
(1988)

Submitted to the Department of
Electrical Engineering and Computer Science
in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy

June 1991



Signature of Author________________________________
Department of Electrical Engineering and Computer Science — Date

Certified by________________________________
Thesis Supervisor — Silvio Micali

Accepted by________________________________
Chair, Departmental Committee on Graduate Students — Arthur Smith

# The Round Complexity of Secure Protocols

by
Phillip Rogaway

*Submitted on April 22, 1991 to the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology in partial fulfillment of the requirements for the degree of Doctor of Philosophy.*

## Abstract

Assume we have a network of three or more players, each player in possession of some private input. The players want to compute some function of these private inputs, but in a way which protects the privacy of each participant's contribution. Not all of the players can be trusted to do as they are instructed. The resources the players are given to accomplish their goal are *communication*—the ability to privately send messages to one another, or to broadcast messages to the community as a whole—and *local computation*.

Many insightful protocols have been proposed for solving this problem of *multiparty secure function evaluation*. Building on Yao's protocol for the case of two players [Ya86], Goldreich, Micali and Wigderson [GMW87] offered the first general protocol for this problem, and they provided the paradigm on which a large body of successive work was based.

Despite enormous progress, research on secure function evaluation has suffered from some serious shortcomings. First, though many protocols have been devised for solving the problem, what, exactly, these protocols accomplish has not been fully understood. In fact, no rigorously specified and generally accepted definitions have been proposed in this field. Second, protocols for multiparty secure function evaluation could be extremely inefficient, the main cause being that they required an unbounded (and usually large) number of communication rounds.

We address both of these points, carefully crafting definitions which satisfactorily deal with the myriad of issues lurking here, and offering a new protocol for multiparty secure function evaluation—one which categorically improves the complexity requirements for this task. The new protocol completely divorces the computational complexity of the function being collaboratively computed from the round complexity of the protocol that evaluates it. Using this approach, we show that a rigorously-specified and extremely strong notion of secure function evaluation can be achieved by a protocol which requires only a fixed constant number of rounds of interaction. This result assumes only the existence of a one-way function and that the majority of the participants to the protocol behave correctly.

`Thesis Supervisor:` Silvio Micali.
`Title:` Professor of Electrical Engineering and Computer Science.
`Keywords:` computer security, cryptography, distributed computing, secure protocols.

# Acknowledgements

# Table of Contents

Chapter 1

# Introduction

This thesis is concerned with doing correct computation in ways that preserve people's privacy. We begin with some motivating examples.

## 1.1 Examples of Protocol Problems

*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?

*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?

*Oblivious transfer problem* (Rabin, [Ra81]). Is it possible for Alice to send to Bob a composite number $n$ in such a way that, half the time, Bob gets *just* $n$, while, the other half of the time, Bob gets $n$ together with a factorization of $n$? Alice should have no idea which of these two possibilities has occurred.

*Mental poker problem* (Shamir, Rivest and Adleman, [SRA81]). A group of cryptographers want to play a game of poker—*over the telephone.* The stakes are high, so Alice becomes uneasy when Bob begins, "Alright, Alice, I've just shuffled a deck of cards—really I have—and now I'm dealing you your hand. You've got a 3$\heartsuit$, a 10$\spadesuit$, a 7$\diamondsuit$, a J$\clubsuit$, and a 5$\clubsuit$..."

*Digital voting problem* (Benaloh and Fisher, [BF85]). Is it possible for a group of computer users to hold a fair election on a computer network? The election should enforce the "one-man, one-vote" constraint, and should respect the privacy of each participant's vote, revealing only the correctly computed tally.

*Office assignment problem* (Disgruntled staff, [MIT]). The $n$ graduate students are thrown into disarray when suddenly told that they should pair *themselves* off into $n/2$ two-person offices. (Assume $n$ is even.) Each student $i$ has a list of ${}^{i}\mathit{like}_{j}$-values specifying his willingness to share an office with each other student $j$. The students want their preferences taken into account, despite the fact that it would be more than a bit tactless for students to publish these lists! The students agree that the quality of a room assignment $M$, considered as a partition of the students into $n/2$ two-elements subsets, is reasonably well measured by $w_M = \sum_{\{i,j\}\in M} \min\left\{{}^{i}\mathit{like}_{j}, {}^{j}\mathit{like}_{i}\right\}$. Now *all* the students want to do is to collaboratively compute an $M$ that maximizes the value $w_M$ while simultaneously respecting the privacy of their individual ${}^{i}\mathit{like}_{j}$ values ...

## 1.2 Secure Protocols

The problems above illustrate several variations in the goals one may have for carrying out a collaborative computation in a privacy-preserving manner. The following possibilities should be singled out: There may be two parties or there may be many. The result of the computation might be a single value, known to all players, or it might be a private value for each player. What is being collaboratively computed may be a computationally simple task or a computationally intensive one. What is being collaboratively computed may depend deterministically on the each player's initial state, or it may depend only probabilistically on this. And the problem being solved might naturally be considered as a function, or—following the notion of Goldreich, Micali and Wigderson [GMW87]—it may be something which is not naturally considered as computing a function, a more "game-like" problem.[1]

The general notion of a *secure protocol* encompasses all of these possibilities. Each participant is willing to divulge *some* information pertaining to his own initial state in exchange for learning some information influenced by other participants' initial states. A player's willingness to participate in the protocol is based on the *promise* that the protocol will not only correctly compute what it is supposed to compute, but it will also protect the privacy of each player's own contribution. We shall take seriously the notion that the job of the cryptographer is both *to make good on this promise,* and *to precisely elucidate its meaning.* We will do this in the general setting which we now describe.

[1] [GMW87] uses the phrase "playing a mental game" to describe the problem of implementing on a communication network an $n$-party game of partial information—without recourse to a trusted party. This is the problem of allowing a progression in time of states of a system, certain aspects of each state known to various players, other aspects of the current state known to no one. In this viewpoint, computing a function by executing a Turing machine computation on inputs held by the various players is just one simple type of game that a group of players may wish to play.

## 1.3 Secure Function Evaluation

Of the various "types" of secure protocols exemplified by our sample problems, we will limit our discussion in this thesis to *multiparty secure function evaluations.* Informally, we have $n \geq 3$ parties, $1, \ldots, n$. Each party $i$ has a *private* input, $x_i$, known only to him. The parties want to *correctly* evaluate some function $f$ on their private inputs—that is, to compute $y = f(x_1, \ldots, x_n)$—while maintaining the privacy of their own individual inputs. That is, they want to compute $y$ without revealing more about their inputs than this value implicitly reveals. The players task of securely computing the function is made particularly difficult by the presence of *bad* players, who may do their utmost to compromise a good player's privacy or disrupt the correctness of the collaborative computation.

An enormous variety of computational tasks that one might wish to perform in a privacy-preserving manner can naturally be expressed as multiparty function evaluations. Still, we note that a multiparty secure function evaluation specializes the general notion of a secure protocol in two ways: by insisting that there be *three or more* players; and by demanding that what the players are trying to do is to compute some (deterministic) *function.* From our initial list of examples, the digital voting problem and the office assignment problem are multiparty secure function evaluations, while the remaining problems are not.

There is good reason to limit our discussions to multiparty secure function evaluations. As to our insistence on there being three or more players, it turns out that what is achievable in the two-party case and what is achievable in the multiparty case are qualitatively very different. In fact, for achieving the strongest notion of security, one *needs* to have three or more participants.[2] As to excluding "game-like" computations, this simplifies our exposition, distilling the "core" of the problem of secure protocols without the extraneous complications.

## 1.4 The Accomplishments of a Decade

It is one of the major triumphs of modern cryptography that the idea of performing secure function evaluation has not only been conceived, but, also, protocols have been offered for this ambitious goal. Let us review some of the most important advances.

TWO PARTY COMPUTATION. The consideration of the millionaires problem, the coin flipping problem, the oblivious transfer problem, the mental poker problem, and other *specific*

[2]Intuitively, when two parties communicate with one another over a clear communication channel, everything each player sends out to the other player is known by every player in the system. This "symmetry" means that there is no "partial knowledge" that can be exploited for the design of clever protocols, and, consequently, severely limits what can be securely computed by two parties communicating over a clear channel.

computational problems led to Yao's recognizing that there is a *general* problem to be solved here—the problem of secure function evaluation for two or many players [Ya82a].

The task of devising a protocol to securely evaluate an arbitrary function seemed enormous. But in 1986, Yao proposed a protocol for exactly that, for the special case of $n = 2$ parties, under a specific cryptographic assumption (that factoring is hard). Yao's protocol made important use of the oblivious transfer primitive [Ra81], and it inaugurated the "garbled circuit" approach, which will be crucial for us.

Secure two-party computation has been investigated extensively by Kilian [Ki89]. He offers a protocol for secure two-party computation and proves precise claims about its properties. Instead of making a number theoretic assumption, Kilian achieves his results under a model of computation which supports a simple abstract primitive—oblivious transfer.

In a two-party computation, if one of the parties stops speaking before the protocol is specified to end, we would like that he does not for his silence earn an unfair information advantage over the other party. Yao first brought to light this consideration [Ya86]. Strengthening what a two-party computation should accomplish to be called "secure," Goldwasser and Levin investigate just how fair a two-party computation can be, and they show how to achieve such a high standard of fairness [GL90]. Their ideas are not only applicable to two-party computation, but to multiparty computation when half or more of the players are bad.

Multiparty computation. In 1987, Goldreich, Micali and Wigderson proposed a protocol for solving the problem of multiparty secure function evaluation, and problems even more general than that [GMW87]. Their protocol was designed to overcome the influence of bad players, as long as they were in the minority. Significantly, it provided the paradigm on which successive solutions were based, a paradigm which is described Section 3.1.

The protocol of Goldreich, Micali and Wigderson, as with Yao's protocol, requires a complexity-theoretic assumption (a trapdoor permutation, say). The assumption is used in several places: to provide for private communication between pairs of players; to permit oblivious transfer between pairs of players; and to implement the "garbled circuit" computation of Yao's two-party protocol. In 1988, several workers managed to banish the complexity assumption of the [GMW87] protocol by positing a richer communication model, in which private communication was provided for by the model itself. Under this richer model of computation, Ben-Or, Goldwasser and Wigderson [BGW88], and Chaum, Crépeau and Damgård [CCD88] proposed multiparty protocols which made no cryptographic assumptions and were designed to overcome the influence of bad players, as long as they constituted fewer than a third of all of the players.

Besides achieving error-free secure distributed computation, the protocol of [BGW88] demonstrated the power of the arithmetization of Boolean computation—the power of working over a finite field, instead of the Boolean domain, and exploiting its algebraic structure. It inaugurated the use of error correcting codes in this context. The arithmetization of

Boolean computation has subsequently proven important in a variety of contexts.

Extending the protocols above. The fault tolerance of the [BGW88] and [CCD88] protocols was subsequently improved.[3] Through the development of a new verifiable secret sharing scheme, Rabin and Ben-Or [RB89] managed to match the fault-tolerance of the [GMW87] protocol under the communication model providing both broadcast and pairwise private communication.

The paper of Galil, Haber and Yung [GHY87], among other contributions, made several improvements to the [GMW87] protocol, one idea from which we will make use of; see Section 3.1.

Work which made the protocols above possible. A large number of ideas had to be in place before the protocols above could be conceived. These ideas include the notion of secret sharing and verifiable secret sharing [Sh79, CGMA85], oblivious transfer [Ra81], probabilistic encryption [GM84], zero-knowledge proofs [GMR85, GMW86], and the slow revelation of a secret [Bl82, LMR83, BG89].

## 1.5 Limitations of Earlier Work

Limitations on definitions. The papers of the last subsection describe protocols. It is clear that these protocols have some remarkable properties. Of course the various authors have worked to describe what these properties *are*, offering definitions, to varying degrees of explicitness. But, in our view, no one succeeded in devising satisfactory definitions for the general problem at hand.

Not having satisfactory definitions for secure protocols is a major problem. Cryptographers have seen, in contexts like digital signatures, how a lack of well-planned definitions can lead them astray.[4] Without good definitions, there are no proofs, there can be no full understanding of the problem, and, eventually, no one understands what anything really means.

For secure function evaluation, definitions must be crafted with extreme care. Otherwise, they are likely to admit as "secure protocols" some protocols which we would like not to have this status; or they may exclude as being "secure" protocols which ought to be called secure; or protocols which achieve the definitions may fail to provably have properties one would expect a secure protocol to enjoy; or the definitions may not capture the appropriate intuition; or they may be too complicated to understand; and so forth.

[3] For *error-free* secure computation, the fault tolerance achieved by the [BGW88] protocol is, in fact, already optimal.

[4] Central contributions in getting *this* notion straight were due to Diffie and Hellman [DH76], Goldwasser, Micali and Yao [GMY83], and, finally, to Goldwasser, Micali, and Rivest [GMR88]. The notion of digital signatures is now well understood.

Thus —as I see it— a large body of beautiful notions and protocols has sat atop rather murky foundations. As a consequence of the lack of agreement on definitions, the protocols which were offered were to a large extent offered without proof that they accomplished any rigorously-specified set of goals.

LIMITATIONS ON THE PROTOCOLS. There was another problem as well: all of the multiparty protocols which had been devised were computationally infeasible. To a large extent, this was because each minute step of the computation the players were interested in carrying out would manifest itself as additional communication rounds between the players. This is a serious problem, because in a network of communicating parties, such back-and-forth communication is usually the most costly resource.

## 1.6 Contributions of this Thesis

This thesis makes three contributions in the design and understanding of secure protocols.

• First, we define the notion of secure function evaluation, in detail and with care never before attempted. The definitions given in this thesis are for secure function evaluation under the communication model that players may speak privately to one another in pairs, or they may broadcast messages to everyone. More general notions in the same spirit as ours will be described in [MR91]. It should be emphasized again that achieving good definitions in this domain is a very tricky matter; it is here that the most delicate issues arise, and, indeed, the definitions given here is a part of definitional work nearly two years in the making.

• Second, we offer a new protocol for multiparty secure function evaluation. This protocol has a major advantage over all previous protocols: it runs *fast*, in the sense of requiring little back-and-forth communication among the players. In fact, the protocol uses just a (fixed) constant number of rounds. This independence of the round complexity of the protocol from the computational complexity of the underlying function being evaluated is in sharp contrast with previous protocols, which used a number of rounds which grew directly with the circuit complexity of the function being evaluated. Not only does reducing the rounds to a constant amount to overcoming the main barrier to making secure protocols practical, but it also provides a key insight about the problem itself. Additionally, the new protocol is easier to analyze and make provable assertions about than previous complexity-theoretic proposals.

• Third, we prove that the formal notion of secure computation described is in fact achieved by the protocol presented, under the sole assumption of the existence of a one-way function, and that the majority of the players behave honestly. The later assumption is not really a limitation, but a necessary consequence of the "strong" notion of security which our definitions are designed to capture.[5] To prove our protocol secure, we assume the correctness of some previous work on secure multiparty function evaluation. What exactly is assumed and what is achieved is stated precisely as Theorem 4.1.1, and Theorems 4.3.1 and 4.4.1, respectively.

Taken together, this research puts secure function evaluation on much firmer footing, and provides direction for the continued development of the area.

## 1.7 Related Work

Concurrent with the definitional work of Kilian, Micali and Rogaway [KMR90], Goldwasser and Levin independently proposed interesting definitions for secure function evaluation [GL90]. It is early to assess how their definitions compare with ours.

Early in our research we shared definitional ideas with Beaver, who later pursued his own ones in [Be91].

## 1.8 Organization and History of Thesis Results.

ORGANIZATION OF THESIS RESULTS. Paralleling the three contributions of Section 1.6, the notion of security is described in Chapter 2; the constant-round protocol is given in Chapter 3; and the proof that it is indeed a secure protocol is given in Chapter 4.

PUBLICATION HISTORY OF THESIS RESULTS. The definitions of Chapter 2 are a special case of notions developed by Micali and Rogaway. An extensive paper investigating these notions is currently undergoing revision [MR91].

At an earlier stage of this research, Micali and Rogaway collaborated with Kilian in developing definitions for secure function evaluation. The fruits of this collaboration are described in [KMR90]. The definitions offered here are more stringent than those of [KMR90], attempting to capture elements of the "ideal evaluation" of a function which this earlier work did not attempt to capture. (See Section 2.4.1 for an explanation of the ideal evaluation of a function $f$.)

[5] Intuitively, when half or more of the players may cheat, certain functions can be computed "more and more correctly" only as you spend "more and more time trying to compute them." There is never a point in time in which the function is computed "totally correctly." Since we want a strong notion of security, in which our protocols should stop, at some fixed point in time, with everyone knowing exactly what they should know, we are forced to accept that there should be fewer than half faulty players.

The first protocol to show how time spent interacting could be traded for increased correctness was due to Luby, Micali and Rackoff [LMR83], in the context of the coin flipping problem. Indeed it is *necessary* to pay for correctness with time for the coin flipping problem, as demonstrated by Cleve [Cl85]. Goldwasser and Levin, following Beaver and Goldwasser, have investigated just how strong a notion of security *is* achievable when there is a dishonest majority [BG89, GL90].

The constant-round protocol of Chapter 3 was developed jointly with Micali. Having heard from Beaver that he too had developed these same results, we thought it fit to produce a jointly authored proceedings version. However, written documentation subsequently provided to us by Beaver was only mildly related to our constant round-protocol [Be88b]. What we describe in Chapter 3 is a revised and simplified version of the protocol in [BMR90].

The contents of Chapter 4—the proof of security of the constant-round protocol—has not appeared elsewhere.

Chapter 2

# The Notion of Secure Computation

To a cryptographer, security *means* defeating an adversary. The stronger the adversary that can be defeated, the more secure the cryptosystem. Thus cryptographers try to dream up nastier and nastier adversaries, and then prove (sometimes under various assumptions) that these very strong adversaries are harmless nonetheless.

In each context, one must carefully define *what* the adversary can do, and *in what sense* the adversary is rendered powerless.

In this chapter, we carefully do this, in the context of secure function evaluation. We begin with a brief overview of our goal.

## 2.1 Overview

For secure function evaluation, achieving security means overcoming the influence of those who would try to compromise a player's privacy or disrupt the integrity of the collaborative computation. The stronger this *adversary*, the more difficult to overcome the effect of her activities, and the more meaningful the resulting notion of security. Specifying what a protocol we call secure must accomplish consists of specifying the abilities of the protocol's participants, specifying the abilities of the adversary, and saying in what sense the adversary is rendered harmless.

Powerful adversaries. The adversary we will consider will be extremely strong—the strongest "reasonable" adversary we can postulate. Roughly, the adversary is able to *corrupt* players at any time she wishes. When a player is corrupted, the adversary learns the state of the player at the time at which he was corrupted, and all future computation the corrupted player was responsible for is now controlled by the adversary. Effectively, the player has been turned into the adversary's loyal agent. To give our adversaries even more power, we assert that even though our protocols are *synchronous* (communication occurs in fixed increments of *rounds*), the adversary is granted a certain amount of *asynchrony* in her ability to corrupt players.[1] The only restrictions placed on the adversary is that there may be a bound on the number of players whom she is able to corrupt, and (for complexity-theoretic security) her local computation time must be "reasonable."

Regarding an adversary as a single agent, rather than many, makes the resulting notion of an adversary *stronger*, effectively permitting a maximal degree of surreptitious cooperation among the maliciously faulty participants to a protocol.

Defeating an adversary. To develop the notion of what it should mean to defeat such an adversary, we imagine an *ideal* protocol, in which computation is carried out by some external, trusted party. Privacy and correctness are non-issues in this scenario because the *model* provides for correct and private computation. Defeating an adversary ought to mean that the computation carried out by the protocol under attack by the adversary mimics the computation by the ideal protocol under attack by the adversary *as closely as possible.* In other words, a secure protocol succeeds in simulating the existence of an external trusted party, while actually trusting no one.

Making this precise entails specifying *in what sense* the computation of a secure protocol is "just like" the computation of the function by a trusted party. A secure protocol should be "just as private" and "just as correct" as with a trusted party. To define privacy, we choose a simulation-based viewpoint, motivated by Goldwasser, Micali and Rackoff's ideas developed in the context of zero-knowledge proof systems [GMR85]. Basically, a protocol is deemed private if for any adversary, what she learns when executing with a protocol is nothing but a sample point of a distribution which she is *entitled* to sample. Correctness is then "interwoven" into the notion of privacy: the simulator existentially guaranteed for privacy is the principal object through which correctness is defined. We do this in a manner which preserves the idea present in the ideal protocol of sending a value off to the external trusted party, and then each player—even the bad players—getting (the right) value back from this agent.

Getting the notions right. In the context of secure protocols, "getting the notions right" is extremely delicate. Definitional issues are tricky both because of the inherent complexity of the setting (a distributed protocol under attack by a powerful adversary is a complicated object!), and because of the severity of the constraints one wants to put on the behavior of the protocol in the presence of the adversary (that is, one wants to ensure

[1] In particular, we permit *rushing*—the ability of the adversary to corrupt a player at the end of round $r$ and use this information to decide on additional players to corrupt *during round* $r$. Information gleaned from these players may in turn motivate additional corruptions, and so forth, until the adversary is done corrupting players for now. Then the outgoing messages are constructed and sent out on behalf of the corrupted players. All of these activities are completed before the beginning of round $r+1$.

the highest possible standard for correctness as well as for privacy). Too weak a notion of security and "secure protocols" begin to pop into existence that one would not like to call secure; too strong a notion and "secure protocols" virtually drop out of existence.

Besides achieving the right balance between definitions which are too strong and too weak, there are a host of other issues in the crafting of good definitions. For example, composability and reducibility properties are important. Uniformly being able to treat complexity-theoretic security and information-theoretic security is desirable. Simplicity is very important. Model independence. And there are many other concerns. Successful definitional work refines one's intuition about security, protocols, and adversaries, leading to substantially improved understanding.

## 2.2 Preliminaries

Before specifying what it means that our adversary "learns nothing," and that our protocol "computes something," we must introduce some basic language to talk about these things. Some of this is standard, but much has been tailored to our specific goal of defining secure computation.

### 2.2.1 Notation

SETS, STRINGS, LANGUAGES, AND FUNCTIONS. An *alphabet* is a finite set. Fix the alphabet $\Sigma = \{0,1\}$. For $b \in \Sigma$, we call $b$ a *bit* and we let $\overline{b}$ denote its complement. A *string* is a finite sequence of characters from some alphabet, and an *infinite string* is an infinite sequence of characters over some alphabet. A *language* is a set of strings.

If $A$ is a set, then $A^n$ denotes the set which is the $n$-wise Cartesian product of $A$ with itself. Thus $\Sigma^n$ is the language of strings of length $n$ over alphabet $\Sigma$, and we let $\Sigma^* = \bigcup_n \Sigma^n$ denote the set of all strings over $\Sigma$. We let $\Sigma^\omega$ denote the set of infinite strings over alphabet $\Sigma$. The empty string (i.e., the length-0 sequence of characters) is denoted by $\Lambda$. For $A$ a set, fix the convention that $A^0$ is the singleton language $\{\Lambda\}$. The notation $1^n$ represents $n$ written in unary. If we write $x \in \Sigma^*$ where $x$ is apparently *not* composed of characters of the alphabet $\Sigma$ (e.g., $x = 11\sharp 0\blacksquare$), then it is understood that the string $x$ is encoded over $\Sigma$ in some natural manner. If $A$ is a set, $2^A$ is the set of all subsets of $A$.

If $x$ and $y$ are strings, $xy$ denotes their concatenation. If $x = a_1 \cdots a_n$ is a string, $a_i \in \Sigma$, then $x[i:j]$ (where $i \leq j$) is the substring $a_i \cdots a_j$. If $x = a_1 \cdots a_n$ is a string, $a_i \in \Sigma$, then $\mathrm{lsb}(x) = a_n$. For $x, y \in \Sigma^*$, $|x| = |y|$, $x \oplus y$ is the bitwise exclusive-or (XOR) of these strings. The NAND operator on bits give the negation of their conjunct.

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 $r_i$.

The set of nonnegative integers is denoted $\mathbf{N} = \{0, 1, 2, \ldots\}$, and $\mathbf{R}$ is the set of real numbers. If $a$ and $b$ are integers, $a \le b$, 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 $a$, and by $[a..\infty]$ we denote the set of all integers greater than or equal to $a$, together with a point "$\infty$". This set is ordered in the natural way, with all numbers $n < \infty$. For $A$ and $B$ ordered sets, the set $A \times B$ is ordered according to $(a, b) < (a', b')$ if either $a < a'$ or $a = a'$ and $b < b'$.

If $A$ and $B$ are sets, $A - B$ denotes the elements of $A$ which are not in $B$.

For $f\colon\ A \to B$ a function, $f$ is automatically extended to a function on subsets of $A$ according to $f(X) = \{f(x)\colon\ x \in X\}$. For $f\colon\ A \to B$ a function, $f^{-1}(y) = \{x : f(x) = y\}$.

Vectors. 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 $\vec{x}$ has $n$ components, $x_1, \ldots, x_n$. If $\vec{x}$ and $\vec{y}$ are $n$-vectors, $\vec{x}\vec{y}$ denotes the $n$-vector whose $i^{th}$ component is $x_i y_i$.

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 $T \subseteq [1..n]$, we define 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.) If $\vec{x}$ and $\vec{x}'$ are $n$-vectors and $T \subseteq [1..n]$, then $x_{\overline{T}} \cup x'_T$ can be regarded as an $n$-vector $\vec{y}$ where $y_i = x'_i$ if $i \in T$ and $y_i = x_i$ otherwise.

For $f\colon\ (\Sigma^\ell)^n \to (\Sigma^l)^n$ and $T \subseteq [1..n]$, $f_T(\vec{x})$ is the function defined by $f_T(\vec{x}) = (f(\vec{x}))_T$. (That is, $f_T(\vec{x})$ is the tagged $T$-coordinates of the image of $\vec{x}$ under $f$.)

Probability. All probability spaces we will be concerned with have underlying set $\Omega = \Sigma^*$, and $\sigma$-field $2^{\Sigma^*}$. Thus we won't distinguish between a probability space and the probability measure of a space. Our probability measures will all have finite support (i.e., the measure is nonzero only on finitely many points). The probability of an event $E \subseteq \Sigma^*$ with respect to some measure $\mu$ is written $\mathrm{Prob}_\mu[E]$, or simply $\mathrm{Prob}[E]$ when the measure $\mu$ is understood. If $E$ is an event of some probability space, $\overline{E}$ is the event which is the set-theoretic complement of $E$ in the underlying set.

A random variable is any function (not necessarily real-valued) on the underlying set of a probability space. The expectation of a real-valued random variable $X$ is denoted $\mathbf{E}X$.

By $U_k$ we denote the uniform distribution on $\Sigma^k$, that is, $U_k$ is the probability measure which assigns mass $2^{-k}$ to each string of length $k$, and assigns mass 0 to all other strings.

### 2.2.2 Basic Notions

This subsection defines some notions fundamental for stating our results: these are the notions of function families, circuits, negligibility, and ensembles.

Function families. A *finite function* is a map $f_{n\ell l} : (\Sigma^\ell)^n \to (\Sigma^l)^n$, or a map $f_{n\ell l} : (\Sigma^\ell)^n \to \Sigma^l$. The former is a *vector-valued* finite function, while the later is a *string-valued* finite function.

Since we are devising definitions tailored for distributed computation by three or more parties, we consider families of functions, each a function of three or more strings. Suppose $L \subseteq \Sigma^*$, with each $c \in L$ specifying values $n_c \geq 3$, $\ell_c, l_c \geq 0$, in some natural manner. Let $f = \{f_c\}$ be a collection of functions, one for each $c \in L$. If each $f_c$ is a vector-valued finite function $f_c : (\Sigma^{\ell_c})^{n_c} \to (\Sigma^{l_c})^{n_c}$, we say that $f$ is a *vector-valued function family*; if each $f_c$ is a string-valued finite function $f_c : (\Sigma^{\ell_c})^{n_c} \to \Sigma^{l_c}$, we say that $f$ is a *string-valued function family*; in either of the above two cases we say that $f$ is a *function family*.

CIRCUITS. A circuit $C$ is a computing device specialized for computing a function from a fixed number of bits to a fixed number of bits. It is a (finite) labeled directed acyclic graph. Each node is labeled by a symmetric Boolean operator drawn from some fixed set of Boolean operators, such as AND, OR, XOR, and their negations. Input nodes (those with in-degree zero) are labeled $x_1, \ldots, x_i$, and output nodes (those with out-degree zero) are labeled $y_1, \ldots, y_o$.

Circuits provide a convenient encoding for finite functions. A circuit $C$ on $i$ inputs and $o$ outputs computes a function $C$: $\Sigma^i \to \Sigma^o$ in the natural way. For $i = n\ell$, $o = nl$, $C$ can be regarded as computing a vector-valued finite function $C : (\Sigma^\ell)^n \to (\Sigma^l)^n$. For $i = n\ell$, $o = l$, $C$ can be regarded as computing a string-valued finite function $C : (\Sigma^\ell)^n \to \Sigma^l$.

If $C$ is a circuit, then $|C|$ is its size—the number of gates in $C$ plus the number of wires in $C$, say—and $\texttt{depth}(C)$ is its depth—the length of a longest path from an input node to an output node. If $C$ is a circuit, we also write $C$ to denote a string describing it in some standard encoding.

WHAT IS FAST? We adopt the notion that the polynomiality of an algorithm captures its running in a "reasonable" amount of time. In this thesis, "polynomial" will always mean a nonnegative-valued polynomial in a single variable.

It will be convenient for us to speak of the time complexity of algorithms which have infinite strings as inputs. To allow such discourse, we always measure the time complexity of a function in terms of the length of its *first* argument, and we assume that *each* argument to an algorithm can be efficiently scanned from left to right. More precisely, let $x_1, x_2, \ldots, x_m$ be finite or infinite strings, the first of which is a finite string. We say that a string-valued function $\mathcal{M}$ on such tuples of strings $(x_1, \ldots, x_m)$ is *polynomial-time computable* if there exists a polynomial $Q$ and a Turing machine $M$, having $m$ input tapes, such that $M$ computes $\mathcal{M}(x_1, x_2, \ldots, x_m)$ within $Q(|x_1|)$-time steps when $M$ is begun in its initial configuration with its $i^{th}$ input tape initialized to the string $x_i$.

WHAT IS SMALL? We will say that a function $\epsilon : \mathbf{N} \longrightarrow \mathbf{R}$ 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*.

There is, of course, some arbitrariness in this notion of negligibility. An alternative advocated by Levin (e.g., [Le85]) says that a function is negligible if it vanishes faster than the inverse of any function in some fixed resource class $\mathcal{R}$, where $\mathcal{R}$ is required to satisfy certain properties. Fortunately, the notions we develop here are essentially independent of the particular definition selected for negligibility.

### 2.2.3 Indistinguishability of Ensembles

A central notion in defining secure protocols is indistinguishability, as introduced by [GM84] in the context of encryption. (The notion has also proven crucial for the complexity theoretic treatment of pseudorandom generation [Ya82b] and for zero-knowledge proofs [GMR85].) Essentially, it captures the fact that two families of probability spaces can be (asymptotically) so close as to be considered insignificantly different. To say this exactly requires some specialized language—the notion of a distinguisher and of a probability ensemble. The reader who wishes a bit more discussion about this notion may consult [GMR85] (pages 191-193).

DISTINGUISHERS. A distinguisher is our formalization of a "judge" who votes to decide among two competing alternatives. As will be discussed shortly, our notion of a distinguisher is a "nonuniform" one.

A *distinguisher* $D^a$ is an (always halting) probabilistic algorithm, $D$, together with an infinite "advice" string, $a$. The algorithm $D$ takes one or more (possibly infinite) strings, $x_1, x_2, \ldots$, and uses its infinite sequence of random bits, $r_D$, and its infinite string, $a$, to compute a value $D(x_1, x_2, \ldots, a, r_D) \in \{0,1\}$. A distinguisher $D^a$ is *polynomial-time* if $D$ is polynomial-time. (Recall that this means polynomial-time in the length of the first argument.)

ENSEMBLES. If $\mathcal{L} = \{L_k : k \in \mathbf{N}\}$ is a family of languages, then an *ensemble* $E$ (over $\mathcal{L}$) is a collection of probability measures on $\Sigma^*$, one for each $(k, \omega) \in \mathbf{N} \times L_k$; that is, $E = \{E_k(\omega) : k \in \mathbf{N},\ \omega \in L_k\}$. The argument $k$ is called the *index* of the ensemble $E$, the argument $\omega$ is called the *parameter* of $E$, and $\mathcal{L}$ is called the *parameter set* of $E$. As the index $k$ is always drawn from the *index set* $\mathbf{N}$, we never specify it, writing $E = \{E_k(\omega)\colon\ \omega \in L_k\}$ to indicate an ensemble over $\mathcal{L}$. 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.

If $E$ and $E'$ are probability ensembles over a common parameter set $\mathcal{L} = \{L_k\}$, then $E \times E'$ is an ensemble over $\mathcal{L}$, where points in $\Sigma^*$ encode pairs of points in $\Sigma^*$ by some fixed, natural encoding. This ensemble is defined by asserting that $\text{Prob}_{(E\times E')_k(\omega)}[(x,x')] = \text{Prob}_{E_k(\omega)}[x]\cdot\text{Prob}_{E'_k(\omega)}[x']$. The notion generalizes in the obvious way to arbitrary finite products. The notation $E^n$ denotes an ensemble which is the $n$-wise product of $E$ with itself.

If $f:\Sigma^*\to\Sigma^*$ is a function on strings and $E$ is an ensemble, then $f(E)$ is an ensemble in the natural way, with $\text{Prob}_{(f(E))_k(\omega)}[x] = \text{Prob}_{E_k(\omega)}[f^{-1}(x)]$.

If $\mathcal{E}E_k(\omega)$ is an ensemble and $A=\{A_k\subseteq\Sigma^*\}$ is a family of events, we say that $A$ occurs *almost certainly* if $\epsilon(k)=\sup_{\omega\in L_k}\text{Prob}_{E_k(\omega)}[A_k]$ is negligible.

Sometimes a simpler notion of an ensemble will do, ensembles which have no parameter $\omega$. In this case, $\mathcal{E}E_k$ is simply an $\mathbf{N}$-indexed family of probability measures on $\Sigma^*$, and all subsequent definitions are made meaningful by interpreting the parameter set of the ensemble to be a collection of singleton languages.

As an example of an unparameterized ensemble, $\mathcal{E}U_k$ is the ensemble of uniform distributions on $k$-bit strings. As an example of a parameterized ensemble, fix $c$ and define $E_k(x_1\cdots x_{k^c})$ (for $|x_i|=1$) as the distribution on tuples $(n,X_1,\cdots,X_{k^c})$ given by first selecting $n$ to be the product of two random $k$-bit primes, then selecting $X_i$ to be a random residue modulo $n$ if $x_i=0$, and selecting $X_i$ to be a random nonresidue modulo $n$ of Jacobi symbol $+1$ if $x_i=1$. Then $\mathcal{E}E_k(x)$ is an interesting ensemble over $\{L_k=\Sigma^{k^c}\}$.

COMPUTATIONAL INDISTINGUISHABILITY. We now specify what it means for two ensembles to be indistinguishable to an observer with bounded computational resources.

**Definition 2.2.1** *Let $E$ and $E'$ be ensembles. We say that $E$ and $E'$ are* computationally indistinguishable, *written $E\approx E'$, if the ensembles are over the same parameter set $\mathcal{L}=\{L_k\}$, and for every polynomial-time distinguisher $D^a$*

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

*is negligible.*

When we wish to emphasize the sequence of languages $\{L_k\}$ which parameterizes the ensembles $E$ and $E'$ we write $E\underset{L_k}{\approx}E'$. Two ensembles over the same parameter set $\mathcal{L}$ which are *not* computationally indistinguishable are called *computationally distinguishable*. This is written $E\not\approx E'$.

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 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 it is more natural for an infinite string to be an input to an 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 results of this thesis hold equally in the uniform model. However, for economy of notions, we choose to describe only the nonuniform notion of security. Some reasons for favoring the nonuniform notion of security over its uniform counterpart are given in Section 2.3.3.

• We remark that the following two variations of the concept of indistinguishability are possible without affecting which pairs of ensembles are indistinguishable and which are not. First, the distinguisher need not be probabilistic: the advice $a$ can always be used to specify a "good" set of coin flips to use. Second, instead of saying that a *single* infinite advice string $a$ is associated to the algorithm $D$, we could instead associate an advice string for each value $k$. To distinguish $E_k(\omega)$ from $E'_k(\omega)$, the distinguishing algorithm $D$ would be given $1^k$, the sample point, the parameter $\omega$, and $a_k$. To see that this "$k$-dependent advice" does not improve one's ability to distinguish ensembles, note that, by the standard diagonalization method, an infinite sequence of infinite advice strings $\{a_k\}$ can be encoded into a single infinite advice string $a$ in such a way that the overhead to read a bit $a_k[i]$ from the advice string $a$ which encodes it is only quadratic in $k$ and $i$.

STATISTICAL INDISTINGUISHABILITY. We define a stronger notion of indistinguishability, one that does not depend on the resource bounds of the observer.

**Definition 2.2.2** *Let $E$ and $E'$ be ensembles. We say that $E$ and $E'$ are* statistically indistinguishable, *written $E \simeq E'$, if the ensembles are over the same parameter set $\mathcal{L} = \{L_k\}$, and for every distinguisher $D^a$,*

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

*is negligible.*

It is an easy theorem that ensembles $\mathcal{E}E_k(\omega)$ and $\mathcal{E}E'_k(\omega)$ over $\{L_k\}$ are statistically indistinguishable if an only if $\epsilon(k) = \sup_{\omega \in L_k} (\sum_{x \in \Sigma^*} |\mathrm{Prob}_{E_k(\omega)}[x] - \mathrm{Prob}_{E'_k(\omega)}[x])$ is negligible.

## 2.3 Protocols and Their Adversaries

In this section we describe not only the communication mechanism provided to the agents of a collaborative computation, but also the adversary which may attack these agents—for one has not described the behavior of a protocol until it is specified how it runs in the presence of the adversary.

The adversary we consider is extremely strong, which 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 and to talk "aloud," so that all other players will agree on what was said and on who said it.

SUMMARY OF THE MODEL. The *players* (or *processors*) are the agents who wish to carry out the joint computation. The collection of players is the *network,* and the program that the players run is called a *protocol.* To be certain that a protocol can be easily described, each player runs the *same* program; it is made specific to the player who is running it by letting the player be aware of his own *identity.* The players have two resources at their disposal: the ability to compute locally, and the ability to communicate with one another. To communicate, the processors may talk privately to one another in pairs, or they may announce messages to the community as a whole. When they do broadcast a message, everyone knows who sent it.

As a player computes, his *computational state* progresses in time. One might imagine that this computational state should progress as the communication *rounds* progress, but instead we formalize matters with a finer level of granularity, thinking of a processor as carrying out many computational steps within a single round. These steps consist of flipping a coin or applying some deterministic function to his computational state. When the round is over, the messages a player sends out to the other players are functions of his computational state, and the messages a player receives from other players—functions of *their* computational states—augment his computational state in a timely manner.

Initially, each player knows only his own *initial* computational state. The information this contains is his *identity*, $i$, his *private input*, $x_i$, and a string called the *common input*, $c$, which is shared by all of the players. The common input contains such information as the number of players in the network, $n$, and a description of the function they wish to compute, $f_c$.

For simplicity, we insist that players' private inputs are all of the *same* length. While it is not important that all inputs be of the same length—this could be accomplished by *padding* the inputs which are too short—it *is* important that the players know a *bound* on the length of the longest possible private input. That this is a natural and reasonable restriction can be motivated as follows. One thing a player $i$ learns about another player $j$ when interacting with him is the number of bits which were sent out by $j$ and received by $i$. For some functions, player $j$ must send out *at least* as many bits as his private input is long—even if there were no privacy constraint at all. A convenient way to make sure that this degree of exposure of one's input is not considered to be damaging is to say that bounds on input lengths were *already* known in advance by each player.[2]

Our goal in this chapter is to properly define those protocols which are *robust* against the incorrect behavior of some of its participants. We adopt an extremely pessimistic view of how players may diverge from faithfully executing a protocol. Namely, we imagine that

[2]When this restriction is not made, the problem of secure computation changes radically in character. See [CGK90] for some work in this framework.

the players are not the only agents involved in the joint computation: also present is an *adversary*, who runs a program the players know nothing about. The unpleasant trait of an adversary is her ability to *corrupt* players. When a player is corrupted, he is made the adversary's loyal servant, with the adversary subsuming all communication and computation associated with the corrupted player. Additionally, the adversary is handed over the private state of the corrupted player at the time at which the player was corrupted.

The computation proceeds in *rounds*. The adversary runs; then the players run; then the adversary runs; then the players; and so forth, until all the players have *terminated.* At this point, the adversary is given one last run, and then the protocol has terminated. Within each of these adversary and player rounds, a good deal of activity may go on—as players do computation, and the adversary computes and corrupts processors.

We will be interested in how much the adversary can learn as a result of her activities, and what the good players compute, despite the interference of the adversary. To concretize these ideas, we now proceed more formally.

### 2.3.1 Protocols

PROTOCOLS FOR $n$-PARTIES. In this section we describe what a protocol for a fixed number of players is. Later we discuss protocols for arbitrary numbers of players.

Recall that $\Sigma$ is the binary alphabet, and words over this alphabet are indicated by writing that they belong to $\Sigma^*$. However, we will sometimes indicate that words which are apparently not over the binary alphabet are to be considered as belonging to $\Sigma^*$—for example, we might write $0\sharp 0{*}1 \in \Sigma^*$. When we do this, it is understood that the symbols which constitute the "expanded" alphabet over which our strings are drawn are encoded as strings over $\Sigma^*$ by some fixed, natural encoding scheme.

In the definition that follows, the *protocol* $P$ is the main object of interest. It specifies how the computational states of the players are to progress in time. Specifying a protocol means defining the map $P$. The *interaction functions* are the "glue" that allows applying the function $P$ repeatedly to capture the network's state progressing in time.

**Definition 2.3.1** *An $n$-*party protocol *is a Turing-computable function*

$$P : \underbrace{\Sigma^*}_{\substack{\text{common}\\ \text{input}}} \times \underbrace{\Sigma^*}_{\substack{\text{current}\\ \text{state}}} \to \underbrace{\Sigma^*}_{\substack{\text{new}\\ \text{state}}} .$$

*A* network interaction function *is any of the following polynomial-time computable functions:*

1. *a* next-action *function* $\eta : \Sigma^* \to \{\mathsf{compute}, \mathsf{flip\text{-}coin}, \mathsf{round\text{-}done}, \mathsf{protocol\text{-}done}\}$,
2. *a* broadcast messages *function* $M : \Sigma^* \to \Sigma^*$,
3. *a* private messages *function* $m : \Sigma^* \times [1..n] \to \Sigma^*$, *and*
4. *an* output *function* $o : \Sigma^* \to \Sigma^*$.

*A* player configuration *is an element of* $\underbrace{\Sigma^*}_{\substack{\text{player's}\\\text{state}}} \times \underbrace{\Sigma^\omega}_{\substack{\text{coins}\\\text{remaining}}} \times \underbrace{\Sigma^*}_{\text{history}}$.

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

DISCUSSION. The next subsection describes, formally, how a protocol runs. Here we give a sketch of this.

To run a protocol, each player starts off in some initial computational state. Each player applies the next action function, $\eta$, to his current computational state to see what he should do. If it says that he should compute, then the algorithm $P$ is applied to the computational state to come up with a new computational state. If it says that he should flip-coin, then the player is given a random coin toss from his infinite sequence of random coins. The consumed coin then "vanishes" from the infinite sequence of future coin tosses. As long as the player computes or gets coin flips, the process continues. When the player is done with all of his activities for this round, this is indicated by $\eta$ assuming the value round-done. When all the players are done computing in this round, their broadcasts and private messages are "delivered"—that is, these strings are properly appended to the computational states of other players. (This will be described in the next subsection.) The messages a player broadcasts and those he sends out privately to other players are determined by applying the functions $M$ and $m$ to his own computational state. Thus, even though these functions assume values before a player is done with a given round, their values are of no importance then. After all the messages are delivered, each player resumes running: the next action function $\eta$ is again applied to his current computational state—the state that the processor is in after the messages he receives from other players have been appended to his computational state. When a processor is finished with *all* the activities he wishes to engage in for this execution of the protocol, instead of simply choosing a computational state in which $\eta$ assumes a value of round-done, he instead selects a computational state which $\eta$ indicates protocol-done. At this point, the output function $o$ defines the player's private output; previous to this, the output function is not meaningful. The protocol terminates when every player is protocol-done.

As the proceeding discussion suggests, a protocol $P$ can only be run with respect to a fixed set of interaction functions. When combined with $P$, these interaction functions specify how a player's computational state is to progress in time, affecting the computational states of other players in the network. Thus we might have considered a protocol to be a five-tuple $(P, \eta, M, m, o)$ consisting of the protocol algorithm proper and the collection of interaction functions. However, it is more convenient to require that the interaction functions 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.

We have not specified these interaction functions since the particular 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 "$\langle$", "$\rangle$", "(", ")", and "," all being formal symbols, we might say that if $s_i$ contains one and only one occurrence of a substring $\langle 1^j \rangle$, then $\eta(s_i) = \mathsf{compute}$ if $j = 1$, $\eta(s_i) = \mathsf{flip\text{-}coin}$ if $j = 2$, $\eta(s_i) = \mathsf{round\text{-}done}$ if $j = 3$, and $\eta(s_i) = \mathsf{protocol\text{-}done}$ otherwise; if $s_i$ contains one and only one occurrence of a substring $(\mu)$, 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; $m_j(s_i) = \Lambda$; and if $s_i$ contains one and only one occurrence of a substring $[\zeta]$, then $o(s_i) = \zeta$; otherwise, $o(s_i) = \Lambda$. This was just an example of how the interaction functions could be defined; the specific choice of conventions is irrelevant.

We have not described a player's configuration. It captures his current computational state, his future coin tosses, and his *history.* The history is information associated to a player which is *not* relevant to the task at hand. The presence of the history is useful for properly dealing with protocol composition, and for proving certain properties of secure protocols. For example, when a protocol is called as a subroutine, the "saved state" which is irrelevant to the subroutine call is tucked away in the history. Since we will not be concerned with subroutine calls in this thesis, the saved state is of no real importance. However, we keep this information around because of its playing a significant role in the more general theory of secure protocols.

We now define more formally how a protocol runs in the absence of an adversary.

### 2.3.2 Executing Protocols in the Absence of an Adversary

Notice how, in the formalism for a protocol, we have a fine level of granularity in how a protocol runs—all the way down to individual coins being tossed. We could have tried to be more succinct, letting a player's computational state progress from round-to-round, with a player doing all the necessary computation "in one shot." However, the approach selected turns out to be more symmetric with the natural way of phrasing the *adversary's* behavior—she gets information repeatedly from the players within a single round. This approach also serves to emphasize that a player may choose to remember a coin toss or he may choose not to remember a coin toss, but—as we shall see when we discuss the adversary—a coin, once flipped, is not recoverable by the adversary except to the extent that it is stored in the player's computational state.

Thus we index the player configurations by two superscripts. The first index, $r$, is a counter for the round number. The second index, $\rho$, is the "micro-round" number, formalizing this finer granularity as the protocol flips coins and does computation. We allow the micro-round to take on the formal value "$\infty$". If a player's computation for round $r$ is completed at some micro-round $\rho$, then all subsequent computational states for micro-rounds during round $r$, including that at time $(r, \infty)$, are fixed. This is notationally convenient.

Configuration sequences. An $n$-party protocol $P$ generates from $n$ initial player configurations, $(C_1^{00}, \ldots, C_n^{00})$, a sequence of configurations $\{C_i^{r\rho}\colon\ i \in [1..n], r \in \mathbf{N}, \rho \in [0..\infty]\}$, which we now describe.

We remark that some configurations may fail to be defined by the recurrences below; this will be dealt with later. We note that the character $r$ is somewhat overworked: with a subscript $i$, it indicates player $i$'s random tape; as a superscript, it indicates a round number. This should cause no confusion. Recall that, if $r_i$ (for example) is an infinite string, then $r_{ij}$ is its $j^{th}$ bit. Finally, the symbols $\{\#, \natural, *, \bullet, \blacksquare\}$, which appear here and elsewhere, are all just formal punctuation symbols.

Fix the notation

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

for *player configurations*, and the notation

$$M_i^r \quad = \quad \begin{cases} M(s_i^{r\infty}) & \text{if } r > 0 \text{ and } \eta(s_i^{(r-1)\infty}) = \textsf{round-done} \\ \Lambda & \text{otherwise} \end{cases}$$

$$m_{ij}^r \quad = \quad \begin{cases} m_j(s_i^{r\infty}) & \text{if } r > 0 \text{ and } \eta(s_i^{(r-1)\infty}) = \textsf{round-done} \\ \Lambda & \text{otherwise} \end{cases}$$

for *broadcast messages* and *messages sent out along private channels.* (Intuitively, the former is what processor $i$ "tries" to broadcast at the end of round $r$, and the latter is what processor $i$ "tries" to send to $j$ at the end of round $r$.) Let the *common input*, $c$, be the "$\natural$"-terminated prefix of $s_1^{00}$. Then the players' configurations progress as follows:

$$C_i^{r(\rho+1)} \quad = \quad \begin{cases} (P(c, s_i^{r\rho}),\ r_i^{r\rho},\ \pi_i^{r\rho}) & \text{if } \eta(s_i^{r\rho}) = \textsf{compute} \text{ and } r > 0 \\[2ex] (s_i^{r\rho} * r_{i1}^{r\rho},\ r_{i2}^{r\rho} r_{i3}^{r\rho} \cdots,\ \pi_i^{r\rho}) & \text{if } \eta(s_i^{r\rho}) = \textsf{flip-coin} \text{ and } r > 0 \\[2ex] C_i^{r\rho} & \text{otherwise} \end{cases}$$

$$C_i^{r\infty} \quad = \quad C_i^{r\rho} \quad \text{if } \exists\ \rho \in \mathbf{N} \text{ s.t. } r = 0 \text{ or } \eta(s_i^{r\rho}) \in \{\textsf{round-done}, \textsf{protocol-done}\}$$

$$C_i^{(r+1)0} \quad = \quad \begin{cases} (s_i^{r\infty} * M_1^r * \cdots * M_n^r * m_{1i}^r * \cdots * m_{ni}^r, & \text{if } \eta(s_i^{r\infty}) = \textsf{round-done} \text{ and} \\ \quad r_i^{r\infty},\ \pi_i^{r\infty}) & \quad r > 0 \\[2ex] C_i^{r\infty} & \text{otherwise} \end{cases}$$

If any configuration fails to be defined by the recurrences above, then the protocol is said to have *diverged* (on this execution).

Nomenclature. As mentioned, the first superscript indexes the *round* while the second superscript indexes the *micro-round.* The string $C_i^{r0}$ is called the *configuration of party $i$ at* *the beginning of round $r$*, while $C_i^{r\infty}$ is the *configuration of party $i$ at the end of round $r$*. The configuration $C_i^{00} = C_i^{0\infty}$ is the *initial configuration of party $i$*. If the round $r$, micro-round $\rho$ configuration of party $i$ is $C_i^{r\rho} = (s_i^{r\rho}, r_i^{r\rho}, \pi_i^{r\rho})$, then we refer to $s_i^{r\rho}$ as the *computational state* of player $i$ at this point in time. The string $M_i^r$ is the message *broadcasted* by $i$ in round $r$, and the string $m_{ij}^r$ is the message sent *from $i$ to $j$* in round $r$.

EXECUTING PROTOCOLS. In the absence of an adversary, an *execution* of an $n$-party protocol $P$ with *common input* $c = 1^k\#1^n\#1^\ell\#1^l\#1^m\#C$, *private inputs* $x_1, \ldots, x_n \in \Sigma^\ell$, *histories* $\pi_1, \ldots, \pi_n \in \Sigma^m$, and *coin sequences* $r_1, \ldots, r_n \in \Sigma^\omega$ is the sequence of configurations $\{C_i^{r\rho}\}$ generated by $P$ when the initial configuration of party $i$ is taken to be $C_i^{00} = (c\natural x_i \natural i,\ r_i,\ \pi_i)$. The set of executions of $P$ with common input $c$, private inputs $\vec{x}$, histories $\vec{\pi}$, and all possible coin sequences $r_1, \ldots, r_n \in \Sigma^\omega$ 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 $\vec{x}$, means sampling according to this distribution.

The values specified on the common input, $c$, in an execution of an $n$-party protocol are called the *security parameter*, $k$, the *number of players*, $n$, the *input length*, $\ell$, the *output length*, $l$, the *history length*, $m$, and *function description*, $C$. Any of these values may take a subscript $c$ to emphasize their being specified on the common input $c$. Since we will be interested in secure computation of functions, $C_c$ will specify—somehow—a vector-valued or string-valued finite function, $C_c : (\Sigma^{\ell_c})^{n_c} \to (\Sigma^{l_c})^{n_c}$, or $C_c : (\Sigma^{\ell_c})^{n_c} \to \Sigma^{l_c}$.

MESSAGE DELIVERY. The string $M_i^r$ is the message that processor $i$ broadcasts at the end of round $r$, or $\Lambda$ if processor $i$ does not broadcast a message in round $r$. The string $m_{ij}^r$ is the messages that processor $i$ sends to processor $j$ at the end of round $r$, or $\Lambda$ if processor $i$ does not send a message to processor $j$ in round $r$. The empty message is delivered to each processor in round 1, since no activity occurred in round 0 (see below).

DISCUSSION. As mentioned, the history of a processor is thought of as information which a processor may possess which is not relevant to the task at hand, but which is nonetheless part of the processor's configuration; for example, it might be the saved state before a subroutine call, and the currently executing protocol is *really* just a subroutine. To ensure that a processor does not "use" information it should not use, we do not include the history in a processor's computational state. But, as we will see, it is there in the sense that when a processor is corrupted, its history is made available to the adversary. We note that we could, alternatively, have said that some properly-marked portion of each processor's computational state is not allowed to be "used" by the protocol $P$. Saying this formally is more awkward than the approach taken.

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. This will not, however, be a concern for us here.

### 2.3.3 Adversaries (Informal Treatment)

MODELING ERRORS. So far we have described how a fault-less network operates. We now consider the possibility for some players becoming "bad" in an execution of a protocol—that is, of deviating from their prescribed instructions. In fact the goal of this chapter is to properly define those protocols that can "withstand" the action of some bad players.

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—for example—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 we consider is that even bad processors may (perhaps) be computationally bounded, and there may be a limit on the number of bad players possible. We also allow bad players to secretly cooperate with each other. Actually, to guarantee their "perfect cooperation," we envisage a *single* agent, the *adversary,* that during an execution of a protocol, may corrupt and control players.

Let us now address an equally important question: *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 for the current execution, 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. This appears to capture the worst natural model of malicious behavior which one might hope to defeat in our scenario.[3] Such an adversary is provably more powerful than a static one (see [MR91]), and security with respect to a dynamic adversary is both harder to achieve and harder to properly define.

If an adversary were allowed to corrupt all players, then nothing could be said about the behavior of the network. Thus the number of corruptible players will appear in our definition of security.

[3] For the Byzantine agreement problem [PSL80]—the problem perhaps most responsible for clarifying notions of adversarial behavior—even stronger adversaries can be postulated and defeated, including an adversary capable of "seeing" the internal states of all players, but capable of gaining control of the output of only a certain fraction of them.

We now refine these ideas, until we are ready to specify a "mathematical" description of an adversary, and how a protocol runs in the presence of such an agent.

| |
|---|
| START |
| Player's round 0 |
| Adversary's round 0 |
| Player's round 1 |
| Adversary's round 1 |
| ⋮ |
| Player's round $\omega$ |
| Adversary's round $\omega$ |
| END |

Figure 2.1: *An execution of a protocol with the adversary. The player's $0^{th}$ round is a round on which there is not activity—so, effectively, the adversary begins and ends the execution of a protocol.*

ADVERSARIES INTERACTING WITH NETWORKS. 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 2.1.

In the beginning, the adversary and all the players in the network are quiescent. The adversary and players will take turns being active. In the beginning, all of the players are *good*, and so they remain unless the adversary *corrupts* them. Initially, all the information the adversary has on which to base these corruptions is the same common input which the players share. (If the players are entitled to this information without doing any work, so is the adversary!)

At the start of every round, the adversary is quiescent. Once all of the still good players have finished their activities for this round, having well-defined out-going private messages and broadcast messages, the players go to sleep and adversary $A$ is awakened and receives all broadcast messages just computed, together with all of the messages the players composed for already-corrupted processors. The adversary may then choose to corrupt some new processor $i$. When she does so, within the *same* round, she learns all of $i$'s internal state (his computational state, history, and initial private input), all of the messages which were just sent out by processor $i$ (as a consequence), and all messages which were just sent to processor $i$.

After this, still within the same round, $A$ 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 outgoing 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.

This process continues until all of the processors have either been corrupted or have halted. Then the adversary is given one last period of activity. After this, the protocol is said to have terminated.

To formalize what we have just described, we will say that the execution of a protocol in the presence of an adversary follows the sequence of "macro-rounds" shown in Figure 2.1. Within each macro-round, there may be many "micro-rounds," in which the players perform their local computations, or the adversary computes and corrupts various players, in sequence. We choose to think of the players as having a $0^{th}$-round in which no activity occurs; after that void round, it is the adversary's turn to be active.

THE ADVERSARY'S ADVICE. To obtain a robust notion of security, we demand that our protocols remain secure *even if there is some information $a_A$ known to the adversary in advance of the protocol's execution.* Oren has pointed out the importance of providing such *advice* in the context of zero-knowledge proof systems [Or87]. The adversary advice might, for example, consist of information which the adversary has somehow come to acquire about the input vector $\vec{x}$—perhaps from previous executions of other protocols. Or, it might be information that depends on the security parameter $k$, which, perhaps, the adversary worked for years to come up with in a "preprocessing stage" of her attack on the protocol. The advice will be doled out in a manner like the coin flips are: when the adversary asks for the next advice bit, it is given to her and vanishes from the infinite string of advice remaining.

WHY THE NONUNIFORMITY? Adversaries like the one we have described—who may possess some fixed information before the protocol begins, information, possibly hard to compute, tailored to the choice of security parameter or, in our case, the private inputs and processor histories, as well—such adversaries are called called *nonuniform* adversaries (because, traditionally, such adversaries have been modeled by (possibly nonuniform) polynomial-size circuit families). Nonuniform adversaries have several advantages over their uniform counterparts, advantages which we now enumerate. 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, the main theorem of this thesis talks 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.

### 2.3.4 Adversaries (Formal Treatment)

ADVERSARIES FOR $n$-PARTY PROTOCOLS. Formally, an adversary will not be very different from the other participants of the collaborative computation; but she has additional abilities which allow her to make life difficult for the players. Defining how a protocol runs in the presence of an adversary will be a matter of properly modifying the players' and adversary's computational state as she "interacts" with the protocol.

**Definition 2.3.2** *An* adversary for an $n$-party protocol *is a function*

$$A : \underbrace{\Sigma^*}_{\substack{\text{common}\\ \text{input}}} \times \underbrace{\Sigma^*}_{\substack{\text{current}\\ \text{state}}} \to \underbrace{\Sigma^*}_{\substack{\text{new}\\ \text{state}}} .$$

*An* adversary interaction function *is any of the following polynomial-time functions:*

1. *a* next-action *function* $\tilde{\eta} : \Sigma^* \to \{\mathsf{compute}, \mathsf{flip\text{-}coin}, \mathsf{get\text{-}advice}, \mathsf{corrupt}_1, \ldots, \mathsf{corrupt}_n, \mathsf{round\text{-}done}\}$,
2. *a* broadcast messages *function* $\tilde{M} : \Sigma^* \times [1..n] \to \Sigma^*$, *and*
3. *a* private messages *function,* $\tilde{m} : \Sigma^* \times [1..n] \times [1..n] \to \Sigma^*$.

*An* adversary configuration *is an element of*

$$\underbrace{\Sigma^*}_{\substack{\text{adversary's}\\ \text{state}}} \times \underbrace{\Sigma^\omega}_{\substack{\text{coins}\\ \text{remaining}}} \times \underbrace{\Sigma^\omega}_{\substack{\text{advice}\\ \text{remaining}}} \times \underbrace{2^{[1..n]}}_{\substack{\text{corrupted}\\ \text{players}}} \times \underbrace{\Sigma^*}_{\text{traffic}} .$$

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.

Discussion. We do not explicitly specify the interaction functions since the particular conventions selected for them is irrelevant. 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 interaction 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 second component, the adversary's current state—we make a separate argument of it to facilitate specifying the computational complexity of an adversary.

The function $\tilde{\eta}$ describes the action an adversary wishes to perform: does she do some computation, flip a coin, corrupt some processor $i$, or complete her activities for this round? Note that while a player may terminate, we choose to say that an adversary does not; we will select a formalization in which the adversary effectively terminates after all processors have done so.

The string $\tilde{M}_i(s_A)$ denotes the message that the adversary will broadcast for player $i$, if $i$ has been corrupted. The string $\tilde{m}_{ij}(s_A)$ denotes the message that the adversary sends to processor $j$ on behalf of processor $i$, if $i$ has been corrupted.

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. However, possible restrictions on the power of an adversary will be defined in Section 2.3.7.

The *adversary configuration* captures that information about the adversary's computation which needs to be remembered and updated across the application of $A$. The adversary's computational state, the coins which she has not yet used, and the advice which she has not yet read are all components of the adversary's configuration. Also included are the set of processors which are currently corrupted; this set grows with each corruption, and never shrinks. The final component of the adversary configuration is an encoding of the *traffic* of exchanges between the adversary and the players with whom she speaks. This quantity will prove sufficiently important to warrant explicitly specifying how it evolves as the adversary interacts with the players.

### 2.3.5 Executing Protocols in the Presence of an Adversary

We now describe how an $n$-party protocol $P$ executes in the presence of an adversary $A$. After specifying the behavior formally, we will describe what it means in English.

CONFIGURATION SEQUENCES. Let $A$ be an adversary for attacking an $n$-party protocol, and let $P$ be such a protocol. We describe how, from any $n$ initial player configurations, $(C_1^{00}, \ldots, C_n^{00})$, and any initial adversary configuration, $C_A^{00} = (s_A, r_A, a_A, \kappa_A, \tau_A)$, protocol $P$ and adversary $A$ generate a sequences of player configurations, $\{C_i^{r\rho}\colon\ i \in [1..n], r \in \mathbf{N}, \rho \in [0..\infty]\}$, and a sequence of adversary configurations, $\{C_A^{r\rho}\colon r \in \mathbf{N}, \rho \in [0..\infty]\}$.
Fix the notation

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

and let the *common input* $c$ and the *private input* $\vec{x}$ be given by $c\sharp x_i \sharp i = s_i^{00}$. Once again, the symbols $\{\#, \sharp, *, \bullet, \blacksquare\}$ are just formal punctuation symbols. The players' configurations progress as before,

$$C_i^{r(\rho+1)} = \begin{cases} (P(c, s_i^{r\rho}),\ r_i^{r\rho},\ \pi_i^{r\rho}) & \text{if } \eta(s_i^{r\rho}) = \textsf{compute} \text{ and } r > 0 \\ (s_i^{r\rho} * r_{i1}^{r\rho},\ r_{i2}^{r\rho} r_{i3}^{r\rho} \cdots,\ \pi_i^{r\rho}) & \text{if } \eta(s_i^{r\rho}) = \textsf{flip-coin} \text{ and } r > 0 \\ C_i^{r\rho} & \text{otherwise} \end{cases}$$

$$C_i^{r\infty} = C_i^{r\rho} \quad \text{if } \exists\ \rho \in \mathbf{N} \text{ s.t. } r = 0 \text{ or } \eta(s_i^{r\rho}) \in \{\textsf{round-done}, \textsf{protocol-done}\}$$

$$C_i^{(r+1)0} = \begin{cases} (s_i^{r\infty} * M_1^r * \cdots * M_n^r * m_{1i}^r * \cdots * m_{ni}^r, & \text{if } \eta(s_i^{r\infty}) = \textsf{round-done} \text{ and} \\ \quad r_i^{r\infty},\ \pi_i^{r\infty}) & \quad r > 0 \\ C_i^{r\infty} & \text{otherwise} \end{cases}$$

while the adversary's sequence of configurations progresses as follows:

$$C_A^{r(\rho+1)} = \begin{cases} (A(c, s_A^{r\rho}),\ r_A^{r\rho},\ a_A^{r\rho},\ \kappa_A^{r\rho},\ \tau_A^{r\rho}) & \text{if } \tilde{\eta}(s_A^{r\rho}) = \textsf{compute} \\ \\ (s_A^{r\rho} * r_{A1}^{r\rho},\ r_{A2}^{r\rho} r_{A3}^{r\rho} \cdots,\ a_A^{r\rho},\ \kappa_A^{r\rho},\ \tau_A^{r\rho}) & \text{if } \tilde{\eta}(s_A^{r\rho}) = \textsf{flip-coin} \\ \\ (s_A^{r\rho} * a_{A1}^{r\rho},\ r_A^{r\rho},\ a_{A2}^{r\rho} a_{A3}^{r\rho} \cdots,\ \kappa_A^{r\rho},\ \tau_A^{r\rho}) & \text{if } \tilde{\eta}(s_A^{r\rho}) = \textsf{get-advice} \\ \\ (s_A^{r\rho} * s_i^{r\infty} * \pi_i^{r\infty} * x_i * \bar{m}_{1i}^r * \cdots * \bar{m}_{ni}^r, & \text{if } \tilde{\eta}(s_A^{r\rho}) = \textsf{corrupt}_i \\ \quad r_A^{r\rho},\ a_A^{r\rho},\ \kappa_A^{r\rho} \cup \{i\}, & \quad \text{and } r > 0 \\ \quad \tau_A^{r\rho} \bullet i \bullet s_i^{r\infty} * \pi_i^{r\infty} * x_i * \bar{m}_{1i}^r * \cdots * \bar{m}_{ni}^r) & \\ \\ (s_A^{r\rho} * s_i^{r\infty} * \pi_i^{r\infty} * x_i, & \text{if } \tilde{\eta}(s_A^{r\rho}) = \textsf{corrupt}_i \\ \quad r_A^{r\rho},\ \pi_A^{r\rho},\ \kappa_A^{r\rho} \cup \{i\} & \quad \text{and } r = 0 \\ \quad \tau_A^{r\rho} \bullet i \bullet s_i^{r\infty} * \pi_i^{r\infty} * x_i) & \\ \\ C_A^{r\rho} & \text{otherwise} \end{cases}$$

$$C_A^{r\infty} = \begin{cases} (s_A^{r\rho},\ r_A^{r\rho},\ a_A^{r\rho},\ \kappa_A^{r\rho}, & \text{if } \exists\ \rho \in \mathbf{N} \text{ s.t. } r > 0 \\ \quad \tau_A^{r\rho} * \tilde{M}_1^r * \cdots * \tilde{M}_n^r * \tilde{m}_{11}^r * \cdots * \tilde{m}_{1n}^r * \cdots & \quad \text{and } \tilde{\eta}(s_A^{r\rho}) = \textsf{round-done} \\ \qquad \cdots * \tilde{m}_{n1}^r * \cdots * \tilde{m}_{nn}^r \blacksquare) & \\ \\ (s_A^{r\rho},\ r_A^{r\rho},\ a_A^{r\rho},\ \kappa_A^{r\rho},\ \tau_A^{r\rho} \blacksquare) & \text{otherwise} \end{cases}$$

$$C_A^{(r+1)0} = \begin{cases} (s_A^{r\infty} * \bar{M}_1^{r+1} * \cdots * \bar{M}_n^{r+1} * \bar{m}_{11}^{r+1} * \cdots * \bar{m}_{1n}^{r+1} * \cdots * \bar{m}_{n1}^{r+1} * \cdots * \bar{m}_{nn}^{r+1}, & \\ \quad r_A^{r\infty},\ a_A^{r\rho},\ \kappa_A^{r\infty}, & \text{if } r > 0 \\ \quad \tau_A^{r\infty} * \bar{M}_1^{r+1} * \cdots * \bar{M}_n^{r+1} * \bar{m}_{11}^{r+1} * \cdots * \bar{m}_{1n}^{r+1} * \cdots * \bar{m}_{n1}^{r+1} * \cdots * \bar{m}_{nn}^{r+1}) & \\ \\ (s_A^{r\infty} *,\ r_A^{r\infty},\ a_A^{r\infty},\ \kappa_A^{r\infty},\ \tau_A^{r\infty}) & \text{if } r = 0 \end{cases}$$

where

$$M_i^r = \begin{cases} M(s_i^{r\infty}) & \text{if } r > 0 \text{ and } i \notin \kappa_A^{r\infty} \text{ and } \eta(s_i^{(r-1)\infty}) \neq \textsf{protocol-done} \\ \tilde{M}_i(s_A^{r\infty}) & \text{if } r > 0 \text{ and } i \in \kappa_A^{r\infty} \\ \Lambda & \text{otherwise} \end{cases}$$

$$m_{ij}^r = \begin{cases} m_j(s_i^{r\infty}) & \text{if } r > 0 \text{ and } i \notin \kappa_A^{r\infty} \text{ and } \eta(s_i^{(r-1)\infty}) \neq \textsf{protocol-done} \\ \tilde{m}_{ij}(s_A^{r\infty}) & \text{if } r > 0 \text{ and } i \in \kappa_A^{r\infty} \\ \Lambda & \text{otherwise} \end{cases}$$

$$\bar{M}_i^r = \begin{cases} M(s_i^{r\infty}) & \text{if } r > 0 \text{ and } i \notin \kappa_A^{(r-1)\infty} \text{ and } \eta(s_i^{(r-1)\infty}) \neq \textsf{protocol-done} \\ \Lambda & \text{otherwise} \end{cases}$$

$$\bar{m}_{ij}^r = \begin{cases} m_j(s_i^{r\infty}) & \text{if } r > 0 \text{ and } i \notin \kappa_A^{(r-1)\infty} \text{ and } j \in \kappa_A^{(r-1)\infty} \text{ and} \\ & \quad \eta(s_i^{(r-1)\infty}) \neq \textsf{protocol-done} \\ \Lambda & \text{otherwise} \end{cases}$$

$$\tilde{M}_i^r = \begin{cases} \tilde{M}_i(s_A^{r\infty}) & \text{if } r > 0 \text{ and } i \in \kappa_A^{r\infty} \\ \Lambda & \text{otherwise} \end{cases}$$

$$\tilde{m}_{ij}^r = \begin{cases} \tilde{m}_{ij}(s_A^{r\infty}) & \text{if } r > 0 \text{ and } i \in \kappa_A^{r\infty} \text{ and } j \notin \kappa_A^{r\infty} \\ \Lambda & \text{otherwise} \end{cases}$$

If any configuration fails to be defined by the recurrences above, then the execution with the specified initial configuration is said to have *diverged.*

NOMENCLATURE. The tuple $C_i^{r0}$ is called the *configuration of party $i$ at the beginning of round $r$,* while $C_i^{r\infty}$ is the *configuration of party $i$ at the end of round $r$.* The configuration $C_i^{00} = C_i^{0\infty}$ is the *initial configuration of the party $i$.* The tuple $C_A^{r0}$ is called the *configuration of the adversary at the beginning of round $r$,* while $C_A^{r\infty}$ is the *configuration of the adversary at the end of round $r$.* The configuration $C_A^{00} = C_i^{0\infty}$ is the *initial configuration* of the adversary.

If the *round $r$, micro-round $\rho$* configuration of party $i$ is $C_i^{r\rho} = (s_i^{r\rho}, r_i^{r\rho}, \pi_i^{r\rho})$, then we refer to $s_i^{r\rho}$ as the *computational state* of player $i$ at this point in time. The string $M_i^r$ is the message *broadcasted* by $i$ in round $r$, and the string $m_{ij}^r$ is the message sent *from $i$ to $j$* in round $r$.

If $\eta(s_i^{r\infty}) = \textsf{protocol-done}$, then $C_i^{r\infty}$ is called the *final configuration of party $i$.* The value $C_A^{r\infty}$ is called the *adversary configuration at the end of round $r$.*

Player $j$ is *corrupted* in round $r$ if $\tilde{\eta}(s_A^{r\rho}) = \textsf{corrupt}_i$ for some $\rho$. Processor $i$ *terminates* at round $r$ if $r$ is the first round for which $\eta(s_i^{r\infty}) = \textsf{protocol-done}$.

EXECUTING PROTOCOLS. In the presence of an adversary $A$, an *execution* of an $n$-party protocol $P$ with *common input* $c = 1^k\#1^n\#1^\ell\#1^l\#1^m\#C$, *private inputs* $x_1, \ldots, x_n \in \Sigma^\ell$, *histories* $\pi_1, \ldots, \pi_n \in \Sigma^m$, *player coin sequences* $r_1, \ldots, r_n \in \Sigma^\omega$, *adversary coin sequence* $r_A \in \Sigma^\omega$, *adversary advice* $a_A \in \Sigma^\omega$, and *initial corruptions* $\kappa_A$, is the collection of configurations $\{C_i^{r\rho}, C_A^{r\rho}\}$ generated by $P$ and $A$ when the initial configuration of party $i$ is taken to be $C_i^{00} = (c\natural x_i\natural i,\ r_i,\ \pi_i)$, and the initial configuration of $A$ is taken to be $(c{*}\kappa_A,\ r_A,\ a_A,\ \kappa_A,\ c{*}\kappa_A)$. The set of executions of $P$ with common input $c$, private inputs $\vec{x}$, histories $\vec{\pi}$, adversary advice $a_A$, initial corruptions $\kappa_A$, and all possible coin sequences $\vec{r}$

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 $c$, private inputs $\vec{x}$, histories $\vec{\pi}$, and adversary advice $a_A$ means sampling from this probability space.

MESSAGE DELIVERY. We explain the meaning of the various $m$&$M$'s. The string $\bar{M}_i^r$ is the message an uncorrupted processor $i$ "tries" to broadcast to the network in its round $r$. Due the adversary's activities, some other message $M_i^r$ may actually be broadcast to the network on behalf of player $i$. The string $\bar{m}_{ij}^r$ is the message an adversary receives from an uncorrupted processor $i$, intended for (corrupted) processor $j$. The string $m_{ij}^r$ is the message that a (good) player $j$ receives on behalf of player $i$, the source of this message depending on whether $i$ is corrupted or not. The string $\tilde{M}_i^r$ is the message the adversary broadcasts in round $r$ on behalf of the corrupted processor $i$; the string $\tilde{m}_{ij}^r$ is the message the adversary sends in round $r$ to the uncorrupted processor $j$ on behalf of corrupted processor $i$.

THE ADVERSARY'S INTERACTION WITH THE NETWORK. When the adversary corrupts a processor, she learns the current computational state of that processor, and the history associated to that processor. Additionally, she learns that processor's private input. As long as there are messages which were delivered to the corrupted processor in this round (i.e., as long as it is not round 0), the adversary is given those messages which did not originate from the adversary herself. When the adversary terminates her activities for some round, the messages she composes on behalf of the corrupted processors are then delivered. When the processors terminate their activities, the messages which they compose and which the adversary can see (broadcast messages or messages sent to corrupted processors along private channels) are delivered to the adversary. So that the formalism matches the intuition, we demand that an adversary corrupts a processor *at most once*: if $\tilde{\eta}(s_A^{r\rho}) = \mathsf{corrupt}_i$, then $\tilde{\eta}(s_A^{r'\rho'}) \neq \mathsf{corrupt}_i$ for all $(r', \rho') < (r, \rho)$.

TRAFFIC. The "traffic" of exchanges between the adversary and the uncorrupted processors consists of everything the adversary "gets" from the currently 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 the currently 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. As we have formalized it, the traffic also includes (tacked onto the beginning) the common input and a description of the initially corrupted processors.

In words, each time a processor $i$ is corrupted, the information learned from that processor which augments the adversary's state is tacked onto the transcript of traffic, preceded by an indication of which processor was corrupted; each time the adversary completes her round-$r$ activities, the messages she has composed on behalf of corrupted players is appended to the transcript of traffic, terminated by a special marker; and, finally, each time the adversary is awakened as a result of the player's completing a round, those messages sent by good players which are received by the adversary are properly delineated and appended to the transcript of traffic.

If $\tau_A$ specifies traffic, then *corrupted*$(\tau_A)$ denotes the set of processors who are corrupted in this transcript—that is, the set of all $i$ such that $\bullet i \bullet$ appears in $\tau_A$[4]—and $\ell(\tau_A)$ denotes the input length in this traffic—that is, the value $\ell$ such that $c = 1^k\#1^n\#1^\ell\#\cdots$ is a prefix of $\tau_A$.

Output and termination. A protocol is said to *terminate* in round $r$ if $r$ is the first round at the end of which every uncorrupted processor has terminated. We demand the following of any protocol $P$: for any adversary $A$ with which $P$ may interact, when $P$ runs with $A$, $P$ terminates with probability 1.

*Player i's output* in an $r$-round execution of a protocol $P$ is the image $y_i = o(s_i^{r\infty})$ under $o$ of that players final computational state, if the player is uncorrupted at the end of round $r$, and the distinguished value corrupted otherwise. The *players' output* is the tagged vector consisting of each player's output different from corrupted.

Adversary's view. In an execution $e$ of an adversary $A$ with a network, the *view* of $A$ in this execution is "everything the adversary sees" before termination: that is, the triple $(\tau_A, r'_A, a'_A)$ consisting of the traffic, the adversary's coins actually consumed (a prefix $r'_A$ of the infinite string $r_A$), and the adversary's advice which is consumed (a prefix $a'_A$ of the infinite string $a_A$). A random bit is consumed when $\tilde{\eta}$ becomes flip-coin; an advice bit is consumed when $\tilde{\eta}$ becomes get-advice.

### 2.3.6 Dealing with Arbitrary Numbers 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. (It is discussed in [MR91] why imagining $n$ fixed is too restrictive.) Thus we treat a protocol $P$ as a family of $n$-party protocols $P = \{P_n\}$. However, recall that we demanded that a protocol be "describable" by a Turing machine (that is, we demanded a protocol be Turing-computable). In passing to more 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.

**Definition 2.3.3** *A* protocol *$P$ is a polynomial-time computable function that maps a number $1^n$ to a standard encoding of an $n$-party protocol $P_n$.*

[4] If corruption is interrogation followed by murder, surrounding $i$ with bullets is quite mnemonic!

We will usually suppress the subscript $n$ when considering an $n$-party protocol $P_n$, using $P$ to denote either a general protocol or a particular $n$-party protocol that it specifies.

General adversaries. Likewise, to talk about a protocol with arbitrary numbers of players under attack by an adversary, we must suitably relax our notion of an adversary.

**Definition 2.3.4** *An* adversary *$A$ is a function that maps a number $1^n$ to an $n$-party adversary $A_n$.*

Again, we usually suppress the subscript $n$ when considering an adversary for an $n$-party protocol, using $A$ to denote either a general adversary or an adversary for a particular value of $n$.

### 2.3.7 Complexity Measures

Adversaries with bounded charisma. If an adversary corrupts all the players, then nothing can be said about their behavior in her presence. We thus prefer less captivating adversaries.

**Definition 2.3.5** *A* t(n)-adversary *for a protocol $P$ is an adversary $A$ for which $|\kappa_A^{r\infty}| \le t(n)$ for any $r$-round execution of $A_n$ with the $n$-party protocol $P_n$.*

For our purposes, we may imagine that the constraint above is strengthened to demand that the adversary always corrupts *exactly* $t(n)$ players, rather than *at most* $t(n)$ players. As long as $t(n)$ is efficiently computable and the adversary "knows" when the protocol will terminate, the notions are equivalent. This will be the case for us. But in contexts like the Byzantine agreement protocol of Feldman and Micali [FM88a], these conditions are not met. Often, $t(n)$ is a step-function associated to linear function of $n$, such as $t(n) = \lfloor (n-1)/2 \rfloor$.

Local computation complexity. Let $e = \{C_i^{r\rho}, C_A^{r\rho}\}$ be an execution of an $n$-party protocol. The number of *player micro-rounds* for execution $e$ is

$$|\{C_i^{r\rho}:\ i \in [1..n],\ r,\rho \in \mathbf{N},\ \eta(s_i^{r\rho}) \notin \{\textsf{round-done}, \textsf{protocol-done}\}, \text{ and } i \notin \kappa_A^{r\infty}\}|\,,$$

while the number of *adversary micro-rounds* is

$$|\{C_A^{r\rho}:\ r,\rho \in \mathbf{N},\ \eta(s_A^{r\rho}) \ne \textsf{round-done}, \text{ and the protocol has not terminated before round } r\}|\,.$$

A protocol $P = \{P_n\}$ is *polynomial-time* if there is a polynomial $Q$ such that for any $n$, $P_n$ is $Q(|c|)$-time computable (where $c$ is the common input); and for any execution $e$ with any adversary $A$, the number of player micro-rounds is bounded above by $Q(|c|)$.

An adversary $A$ who interacts with a polynomial-time protocol $P$ is *polynomial-time* if there is a polynomial $Q$ such that the encoding of $A_n$ is computed by $A$ in time bounded above by $Q(n)$; and $A_n$ is computable in time at most $Q(|c|)$; and, last of all, the number of adversary micro-rounds is bounded above by $Q(|c|)$.

Round and communication complexity. The *round complexity* of a protocol $P$ is least value $r$ such that when protocol $P$ is run in the presence of an adversary $A$ it necessarily terminates within $r$ rounds. The round complexity is "$\infty$" if no such number exists. The round complexity is regarded as a a function of $|c|$.

The *communication complexity* of a protocol $P$ is the least value $K$ such that when protocol $P$ is run in the presence of an adversary $A$, the total number of bits sent out by uncorrupted processors is at most $K$. The total number of bits sent out by uncorrupted processors is $\sum_{r,i} \delta(i \notin \kappa_A(s_A^{(r-1)\infty}))(|M_i^r| + \sum_j |m_{ij}^r|)$, where $\delta$ is 1 for a true predicate, 0 for a false predicate. The communication complexity is "$\infty$" if no such number exists. The communication complexity is regarded as a a function of $|c|$.

## 2.4 Secure Function Evaluation

With the language of Sections 2.2 and 2.3 in hand, we begin to phrase our definition of security. We begin by providing some intuition as to what correctly and privately computing a function *should* mean in the presence of an adversary as strong as the one we have defined.

### 2.4.1 The Ideal Evaluation

A secure protocol should mimic—as closely as possible—the computation of a function $f$ by an *ideal protocol* for computing it. An ideal protocol for $f$ can be considered as achieving security by adopting a model of computation which provides a *trusted party* for computing the function. The rest of the parties, though, are subject to corruption. We describe the ideal protocol somewhat more colorfully below.

Imagine distributively computing $f : (\Sigma^\ell)^n \to (\Sigma^l)^n$ by playing the following game.

Each player sits at a table, his private input string written underneath the lead plate in front of him. One by one, the adversary "corrupts" some of the players, removing a player's lead plate and learning the information written beneath it. Then the adversary substitutes (fake) input strings $x'_T$ in front of each of these corrupted players, and she replaces the lead plates. Once the plates have all been replaced, the value of $f_i(x'_T \cup x_{\overline{T}})$, magically appears underneath the plate of each player $i$. In this way, each player $i$ has computed $f$ evaluated at a vector of inputs which—though partially determined by the adversary—still, has been correctly evaluated at a point $x'_T \cup x_{\overline{T}}$, where $x'_T$ was chosen independent of the good players private inputs.

After the adversary learns the $f_i(x'_T \cup x_{\overline{T}})$-values for the already-corrupted players $i$, one by one, she may corrupt additional players $j$. Removing their lead plates, the adversary learns their $x_j$- and $f_j(x'_T \cup x_{\overline{T}})$-values. All together, she may corrupt a total of $t$ players.

Turning the ideal protocol into a definition. The formalization of security attempts to ensure that the computation of $f$ by the protocol is "just like" the computation of $f$ by the ideal protocol which computes $f$. Unfortunately, it is not obvious what "just like" really means. We attempt to imitate the ideal computation in all essential ways.

For example, in the ideal protocol, the adversary "knows" what values $x'_T$ she substitutes in under the plates of corrupted players, and she "knows" what her share of the output is as a result of these substitutions. If a protocol does not enjoy this property, might it still be secure in some significant sense? Absolutely. But our viewpoint is that *all* important aspects of the ideal protocol should be mimicked in a protocol we call secure—and the adversary's substitution on a given run of a value $x'_T$ (a value she is aware of) on behalf of the corrupted players would seem to be an important aspect of the ideal protocol. See Section 2.5 for some additional discussion.

### 2.4.2 Ideal Evaluation Oracles

To formalize our notion of security, we develop an abstraction for how an adversary can corrupt players and steal information in the ideal protocol for computing $f$. Though the following notion does not appear *explicitly* in our formalization of security, the language it offers in useful, and the informal description is helpful in understanding the formalism of Section 2.4.3. In Chapter 4, we will will speak in terms of oracles to make more understandable the proof of that chapter.

We imagine a special type of *oracle*, $\mathcal{O}_t(\vec{x}, \vec{\pi}; f)$, whose behavior is dictated by the players' inputs $\vec{x} \in (\Sigma^\ell)^n$, their histories $\vec{\pi} \in (\Sigma^*)^n$, the function $f$ to be computed, and the bound $t \in [0..n]$ on the number of processors the adversary is allowed to corrupt. We now describe how this oracle behaves.

The oracle accepts two types of queries:

- A component query is an integer $i$, $i \in [1..n]$. It is answered by $(x_i, \pi_i)$ if $t$ or fewer component queries have been made so far, and no output query has been made so far; it is answered by $((x_i, \pi_i), y_i)$ if $t$ or fewer component queries have been made so far, and the proper output query $x'_T$ previously made resulted in a response $\vec{y} = f(x'_T \cup x_{\overline{T}})$. Additional component queries are answered by $\Lambda$.
- An output query is a tagged vector $x'_T$. The query is *valid* if $T$ consists precisely of the component queries made so far, and if this is the first output query. Asking a valid output query $x'_T$ results in the oracle computing $\vec{y} = f(x'_T \cup x_{\overline{T}})$, and answering $y_T$. An output query which is not valid is answered by $\Lambda$.

Let us emphasize that the oracle is willing to answer at most $t$ component queries. If the oracle is asked an improper output query (that is, $T$ is not the set of previous component queries), or if the oracle is asked more than one output query, it does not give the requested information. Note that we do allow component queries to follow the output query, so long as the total number of component queries is bounded by $t$. Also, component queries behave differently depending on whether or not the output query has been made: if the output query has been made, then a component query returns (in addition to the requested component) the queried player's "share" of the function value.

Clearly what you (as a $t$-adversary) can learn when speaking to a ideal evaluation oracle *exactly coincides with what you can learn in the ideal computation of $f$.* Privacy is then easy to formalize using ideal evaluation oracles: roughly said, the ensemble of views you get as an adversary interacting with the protocol is indistinguishable from an ensemble of views you can generate without speaking to the network, but speaking to some algorithm equipped with ideal evaluation oracle, instead.

Actually, we will define not only privacy but correctness, too, through the interaction of an algorithm with its ideal evaluation oracle.

If $\mathcal{O}_t(\vec{x}, \vec{\pi}; f)$ is an oracle, we will sometimes omit the subscript $t$ and the parenthesized arguments $\vec{x}$, $\vec{\pi}$ and $f$, and write simply $\mathcal{O}$, instead. The response to a component query of $i$ is then written simply as $\mathcal{O}(i)$, and the response to an output query of $x'_T$ is written simply as $\mathcal{O}(x'_T)$.

The description of an ideal evaluation oracles just given handles vector-valued functions. In fact we will be interested both in secure computations in which every player will get his own private output, and in secure computations in which all players will get a common—and thus public—output.

For the purpose of computing string-valued functions, the definition of component queries can be simplified: a component query $i$ returns $x_i$ if there have been $t$ or fewer component queries so far; there is no need to return the value $y_i$ for queries made after the output query.

### 2.4.3 Simulators

We now introduce a key tool for achieving our definition of security: simulators. This notion was also central to the definition of zero-knowledge proofs, of which protocols for secure function evaluation are an extension. There are, however, several adjustments to be made, reflecting the several differences of context.

#### 2.4.3.1 Informal description

SIMULATORS TALK TO ADVERSARIES. A simulator is an algorithm which interacts with an adversary with "mechanics" similar to the interaction between an adversary and a protocol, as suggested by Figure 2.2. We describe this mechanics below. Later we will concentrate on those simulators interacting with which is indistinguishable—to the adversary—from interacting with a network.

We point out that we are demanding more of our simulators than is customary to demand of simulators for zero-knowledge proofs [GMR85]. Our simulators are required to lay down a conversation with an adversary in a message-by-message manner, something

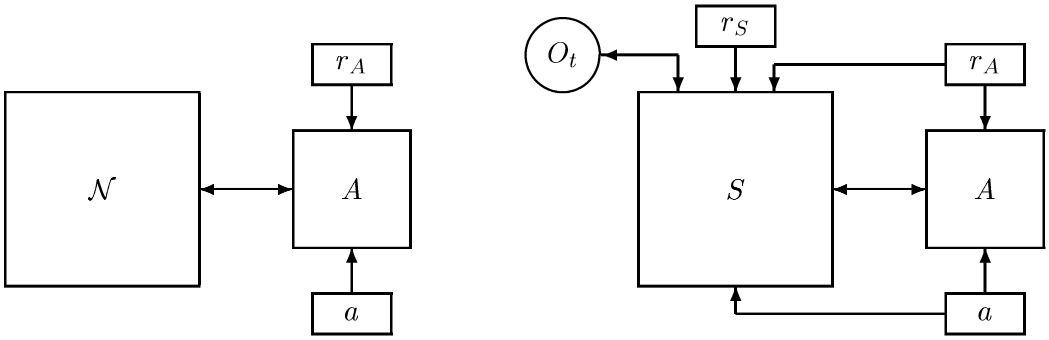


Figure 2.2: *A protocol can be shown private by constructing a simulator $S$ such that the adversary $A$ has no idea if she is talking to the actual network $\mathcal{N}$ or a simulated network maintained by $S$.*

which has never been demanded in simulation for zero-knowledge proofs. Crépeau [Cr90] also explicitly demands this type of simulation, and it is discussed by Kilian as well [Ki89]. This restriction is central to achieving our notion of correctness, since it is through this mechanics of the adversary speaking to the simulator that a meaningful "committal" can be associated with the execution of an adversary with a *network,* not just with a simulator. We caution that this restriction is different from the "black-box" notion sometimes considered for zero-knowledge proofs.

SIMULATORS OWN IDEAL EVALUATION ORACLES. It will be the simulator's responsibility to send to the adversary messages which would "appear" to be from the network itself. To this end, a simulator is provided an ideal evaluation oracle. When and only when the adversary $A$ with which simulator $S$ speaks corrupts a processor $i$, simulator $S$ makes a component query of $i$. Once, and only once, the simulator makes an output query $x'_T$, where $T$ is the set of processors currently corrupted by $A$. We will say that the simulator does this at some point just *following* the the completion of an adversary round, after the adversary's outgoing messages have been composed and sent to the simulator. Presumably, the simulator's output query was required to provide the next round of messages going to the adversary from the simulator on behalf of the uncorrupted players.

A simulator is sometimes productively thought of as the "subconscious" of an adversary who is endowed with an ideal evaluation oracle, but who does not having benefit of a network with which to compute. Everything that a simulator gives to an adversary an adversary could give to herself—if only she had the oracle.

DEPENDENCY OF $S$ ON $A$. There are issues involved in deciding to what extent the simulator's behavior may depend on the adversary $A$ with which it interacts. At one extreme, the simulator might "see" the adversary's current state, the advice string which $A$ has been given, and the coins which $A$ flips; and the simulator algorithm may depend in an arbitrary manner on the adversary's algorithm itself. This is the least restrictive requirements for a simulator, giving rise to the weakest notion of security.

At the other extreme (and there are many in the middle), the simulator knows *nothing* of the adversary's program, advice, coins, or internal state, and it must provide its messages without such benefit, based only on the traffic exchanged between itself and the adversary. This is the most restrictive requirement for a simulator, giving rise to the strongest notion of security. We adopt this notion here. Note that it is completely meaningful to have such a simulator interacting with an adversary who computes an arbitrary function.

Computing the output query. How does the simulator compute its output query? Necessarily, it is a function of the simulator's coins, its oracle responses, and the traffic of exchanges between itself and the adversary. But we wish to be less generous. Why?

Key to achieving strong notions of reducibility and independence (see Section 2.5) is that the adversary should *know* the output query. But if we allow the simulator to compute its output query without any restrictions, then the adversary—not knowing the simulator's coins—cannot possibly know this. One (particularly restrictive) way to demand that the adversary knows the output query (meaning that she *could* compute it, if she wanted to), is to demand that the simulator's output query must be an efficiently computable function $\mathcal{AI}$ (for "adversary input") of *just* the traffic of message exchanges. Though less restrictive notions are possible (and are in fact necessary to properly deal with secure computation in the broadcast model), the issue is always the same: how to ensure that the adversary knows what the output query is, in the sense that it can be calculated by the adversary in roughly the time allotted for the execution.

Adversary output. In the ideal computation of a protocol, the adversary necessarily learns not only what she has sent off to the trusted server, but she also learns what the trusted server returns to her (that is, her share of the output). Through the $\mathcal{AI}$ function we have mimicked the former; through the $\mathcal{AO}$ function (for "adversary output") we imitate the later. Like the adversary input, the adversary output $\mathcal{AO}$ is an efficiently computable function of the traffic. In an $r$-round execution of a protocol, the adversary's share of the common output is $y_{\mathcal{T}} = \mathcal{AO}(c, \tau_A^{r\infty})$, where $\mathcal{T} = \mathit{corrupted}(\tau_A^{r\infty})$ is the set of players who are corrupted when the protocol terminates.

• Strong simulatability. We emphasize that the restriction that $\mathcal{AI}$ and $\mathcal{AO}$ be efficiently computable functions on the traffic is a lot to demand. More general notions of simulatability are less restrictive on this point. The spirit of definitions which capture the *sense* in which the adversary is aware of her input and output is the same, but is not dealt with here. Refer to [MR91].

#### 2.4.3.2 Formal description

DEFINITION OF A SIMULATOR. We define a simulator as a triple consisting of the simulator algorithm proper and two associated (total) functions on the traffic. These specify what the adversary has effectively sent to the trusted server and received from the trusted server.

**Definition 2.4.1** *A* simulator $S = (\mathcal{S}, \mathcal{AI}, \mathcal{AO})$ *is a polynomial-time computable function*

$$\mathcal{S} : \underbrace{\Sigma^*}_{\substack{\text{common}\\\text{input}}} \times \underbrace{\Sigma^*}_{\substack{\text{current}\\\text{state}}} \times \underbrace{\Sigma^\omega}_{\substack{\text{simulator}\\\text{coins}}} \to \underbrace{\Sigma^*}_{\substack{\text{new}\\\text{state}}},$$

*together with a polynomial-time computable* adversary input *function*

$$\mathcal{AI} : \underbrace{\Sigma^*}_{\substack{\text{common}\\\text{input}}} \times \underbrace{\Sigma^*}_{\text{traffic}} \to \underbrace{2^{[1..n]\times\Sigma^*}}_{\substack{\text{substituted}\\\text{values}}} \cup \{\mathsf{not\text{-}now}\},$$

*and a polynomial-time computable* adversary output *function*

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

*The function $\mathcal{AI}$ has the property that for any traffic $\tau_A^{r\infty}$, either $\mathcal{AI}(c, \tau_A^{r\infty}) = \mathsf{not\text{-}now}$ or else $\mathcal{AI}(c, \tau_A^{r\infty}) \in corrupted(\tau_A^{r\infty}) \times \Sigma^{\ell(\tau_A^{r\infty})}$. In the later case, $\mathcal{AI}(c, s_A^{\bar{r}\infty}) = \mathsf{not\text{-}now}$ for all $\bar{r} < r$.*

*A* simulator interaction function *is either of the following polynomial-time computable functions:*

1. *a* next action *function $\Upsilon : \Sigma^* \to \{\mathsf{compute}, \mathsf{protocol\text{-}done}\}$, and*
2. *a* give-to-adversary *function $\Phi : \Sigma^* \to \Sigma^*$.*

*A* simulator configuration *is an element of* $\underbrace{\Sigma^*}_{\substack{\text{simulator's}\\\text{state}}} \times \underbrace{\Sigma^\omega}_{\substack{\text{simulator's}\\\text{coins}}}$.

DISCUSSION. As before, the simulator interaction functions are fixed maps which determine their range points in a natural, easily encoded manner from their domain points.

Recall that we chose to say that the adversary does not terminate; the protocol is said to terminate in the adversary round following the termination of all uncorrupted processors. Since there is no protocol running when an adversary interacts with a simulator, it is a simulator's responsibility to effectively indicate when the protocol should be regarded as being over. It does this with its next action function, $\Phi$.

The simulator's other interaction function is used to read off, from the simulator's computational state, what it provides to the adversary with whom it is interacting.

Running an adversary with a simulator. When having an adversary interact with a protocol, the adversary went through the following sequence of configurations:

$$C_A^{r(\rho+1)} = \begin{cases} (A(c, s_A^{r\rho}),\ r_A^{r\rho},\ a_A^{r\rho},\ \kappa_A^{r\rho},\ \tau_A^{r\rho}) & \text{if } \tilde{\eta}(s_A^{r\rho}) = \textsf{compute} \\ (s_A^{r\rho} {*} r_{A1}^{r\rho},\ r_{A2}^{r\rho} r_{A3}^{r\rho} \cdots,\ a_A^{r\rho},\ \kappa_A^{r\rho},\ \tau_A^{r\rho}) & \text{if } \tilde{\eta}(s_A^{r\rho}) = \textsf{flip-coin} \\ (s_A^{r\rho} {*} a_{A1}^{r\rho},\ r_A^{r\rho},\ a_{A2}^{r\rho} a_{A3}^{r\rho} \cdots,\ \kappa_A^{r\rho},\ \tau_A^{r\rho}) & \text{if } \tilde{\eta}(s_A^{r\rho}) = \textsf{get-advice} \\ (s_A^{r\rho} {*} s_i^{r\infty} {*} \boxed{\pi_i^{r\infty} {*} x_i {*} \bar{m}_{1i}^r {*} \cdots {*} \bar{m}_{ni}^r}, & \text{if } \tilde{\eta}(s_A^{r\rho}) = \textsf{corrupt}_i \\ \quad r_A^{r\rho},\ a_A^{r\rho},\ \kappa_A^{r\rho} \cup \{i\}, & \quad \text{and } r > 0 \\ \quad \tau_A^{r\rho} \bullet i \bullet \boxed{s_i^{r\infty} {*} \pi_i^{r\infty} {*} x_i {*} \bar{m}_{1i}^r {*} \cdots {*} \bar{m}_{ni}^r}) & \\ (s_A^{r\rho} {*} \boxed{s_i^{r\infty} {*} \pi_i^{r\infty} {*} x_i}, & \text{if } \tilde{\eta}(s_A^{r\rho}) = \textsf{corrupt}_i \\ \quad r_A^{r\rho},\ \pi_A^{r\rho},\ \kappa_A^{r\rho} \cup \{i\} & \quad \text{and } r = 0 \\ \quad \tau_A^{r\rho} \bullet i \bullet \boxed{s_i^{r\infty} {*} \pi_i^{r\infty} {*} x_i}) & \\ C_A^{r\rho} & \text{otherwise} \end{cases}$$

$$C_A^{r\infty} = \begin{cases} (s_A^{r\rho},\ r_A^{r\rho},\ a_A^{r\rho},\ \kappa_A^{r\rho}, & \text{if } \exists\ \rho \in \mathbf{N} \text{ s.t. } r > 0 \\ \quad \tau_A^{r\rho} {*} \tilde{M}_1^r {*} \cdots \tilde{M}_n^r {*} \tilde{m}_{11}^r {*} \cdots {*} \tilde{m}_{1n}^r {*} \cdots & \quad \text{and } \tilde{\eta}(s_A^{r\rho}) = \textsf{round-done} \\ \qquad \cdots {*} \tilde{m}_{n1}^r {*} \cdots {*} \tilde{m}_{nn}^r \blacksquare) & \\ (s_A^{r\rho},\ r_A^{r\rho},\ a_A^{r\rho},\ \kappa_A^{r\rho},\ \tau_A^{r\rho} \blacksquare) & \text{otherwise} \end{cases}$$

$$C_A^{(r+1)0} = \begin{cases} (s_A^{r\infty} {*} \boxed{\bar{M}_1^{r+1} {*} \cdots {*} \bar{M}_n^{r+1} {*} \bar{m}_{11}^{r+1} {*} \cdots {*} \bar{m}_{1n}^{r+1} {*} \cdots {*} \bar{m}_{n1}^{r+1} {*} \cdots {*} \bar{m}_{nn}^{r+1}}, & \\ \quad r_A^{r\infty},\ a_A^{r\rho},\ \kappa_A^{r\infty}, & \text{if } r > 0 \\ \quad \tau_A^{r\infty} {*} \boxed{\bar{M}_1^{r+1} {*} \cdots {*} \bar{M}_n^{r+1} {*} \bar{m}_{11}^{r+1} {*} \cdots {*} \bar{m}_{1n}^{r+1} {*} \cdots {*} \bar{m}_{n1}^{r+1} {*} \cdots {*} \bar{m}_{nn}^{r+1}}) & \\ (s_A^{r\infty} {*},\ r_A^{r\infty},\ a_A^{r\infty},\ \kappa_A^{r\infty},\ \tau_A^{r\infty}) & \text{if } r = 0 \end{cases}$$

The boxed quantities indicate the points at which the the adversary has obtained information from the players in the network, and where this information appears in the updated traffic. The idea for defining the behavior of a simulator interacting with an adversary is to have the simulator—rather than the protocol—provide the boxed quantities, above.

To run an adversary $A$ having associated adversary input function $\mathcal{AI}$ with a simulator $S$, common input $c = 1^k\#1^n\#1^\ell\#1^l\#1^m\#C_c$ (where $C_c$ specifies a function $f_c : (\Sigma^\ell)^n \to (\Sigma^l)^n$), adversary coins $r_A$, adversary advice $a_A$, simulator coins $r_S$, and oracle $\mathcal{O}_t(\vec{x}, \vec{\pi}; f)$, where $\vec{x} \in (\Sigma^\ell)^n$ and $\vec{\pi} \in (\Sigma^m)^n$, the boxed quantities above are replaced by that information which the simulator returns—as specified by $\Phi$—and the simulator's state is allowed to progress in time in a manner analogous to a protocol's progress in time.

To describe this, fix the notation that

$$C_S^{r\rho} \quad = \quad (s_S^{r\rho},\, r_S^{r\rho}),$$

and let the adversary configurations progress according to

$$C_A^{r(\rho+1)} \quad = \quad \begin{cases} (A(c, s_A^{r\rho}),\, r_A^{r\rho},\, a_A^{r\rho},\, \kappa_A^{r\rho},\, \tau_A^{r\rho}) & \text{if } \tilde{\eta}(s_A^{r\rho}) = \textsf{compute} \\[2ex] (s_A^{r\rho} * r_{A1}^{r\rho},\, r_{A2}^{r\rho} r_{A3}^{r\rho} \cdots,\, a_A^{r\rho},\, \kappa_A^{r\rho},\, \tau_A^{r\rho}) & \text{if } \tilde{\eta}(s_A^{r\rho}) = \textsf{flip-coin} \\[2ex] (s_A^{r\rho} * a_{A1}^{r\rho},\, r_A^{r\rho},\, a_{A2}^{r\rho} a_{A3}^{r\rho} \cdots,\, \kappa_A^{r\rho},\, \tau_A^{r\rho}) & \text{if } \tilde{\eta}(s_A^{r\rho}) = \textsf{get-advice} \\[2ex] (s_A^{r\rho} * s_i^{r\infty} * \boxed{\Phi(s_S^{r(\rho+1)})}, & \text{if } \tilde{\eta}(s_A^{r\rho}) = \textsf{corrupt}_i \\ \quad r_A^{r\rho},\, a_A^{r\rho},\, \kappa_A^{r\rho} \cup \{i\}, & \quad \text{and } r > 0 \\ \quad \tau_A^{r\rho} \bullet i \bullet \boxed{\Phi(s_S^{r(\rho+1)})}) & \\[2ex] (s_A^{r\rho} * \boxed{\Phi(s_S^{r(\rho+1)})}, & \text{if } \tilde{\eta}(s_A^{r\rho}) = \textsf{corrupt}_i \\ \quad r_A^{r\rho},\, \pi_A^{r\rho},\, \kappa_A^{r\rho} \cup \{i\} & \quad \text{and } r = 0 \\ \quad \tau_A^{r\rho} \bullet i \bullet \boxed{\Phi(s_S^{r(\rho+1)})}) & \\[2ex] C_A^{r\rho} & \text{otherwise} \end{cases}$$

$$C_A^{r\infty} \quad = \quad \begin{cases} (s_A^{r\rho},\, r_A^{r\rho},\, a_A^{r\rho},\, \kappa_A^{r\rho}, & \text{if } \exists\, \rho \in \mathbf{N} \text{ s.t. } r > 0 \\ \quad \tau_A^{r\rho} * \tilde{M}_1^r * \cdots \tilde{M}_n^r * \tilde{m}_{11}^r * \cdots * \tilde{m}_{1n}^r * \cdots & \quad \text{and } \tilde{\eta}(s_A^{r\rho}) = \textsf{round-done} \\ \qquad\qquad \cdots * \tilde{m}_{n1}^r * \cdots * \tilde{m}_{nn}^r \blacksquare) & \\[2ex] C_A^{r\rho} & \text{otherwise} \end{cases}$$

$$C_A^{(r+1)0} \quad = \quad \begin{cases} (s_A^{r\infty} * \boxed{\Phi(s_S^{(r+1)0})}, & \\ \quad r_A^{r\infty},\, a_A^{r\rho},\, \kappa_A^{r\infty}, & \text{if } r > 0 \\ \quad \tau_A^{r\infty} * \boxed{\Phi(s_S^{(r+1)0})}) & \\[2ex] (s_A^{r\infty} *,\, r_A^{r\infty},\, a_A^{r\infty},\, \kappa_A^{r\infty},\, \tau_A^{r\infty}) & \text{if } r = 0 \end{cases}$$

while the simulator's configurations progress according to

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

$$C_S^{r\infty} = C_S^{r\rho} \quad \text{if } \exists \rho \in \mathbf{N} \text{ s.t. } \tilde{\eta}(s_A^{r\rho}) \in \{\textsf{round-done}, \textsf{protocol-done}\}$$

$$C_S^{(r+1)0} = (\mathcal{S}(c,\ s_S^{r\infty} * \mathcal{O}(\mathcal{AI}(c, \tau_S^{r\infty})),\ r_S),\ r_S)$$

where

$$\mathcal{O}(i) = \begin{cases} (x_i, \pi_i) & \text{if } \forall \bar{r} < r,\ \mathcal{AI}(c, \tau_A^{\bar{r}\infty}) = \emptyset \\ \\ ((x_i, \pi_i),\ y_i) & \text{if } \exists\ \bar{r} < r \text{ s.t.} \\ & \quad x'_T = \mathcal{AI}(c, \tau_A^{\bar{r}\infty}) \neq \textsf{not-now} \\ & \quad \text{and } \vec{y} = f_c(x'_T \cup x_{\overline{T}}) \end{cases}$$

$$\mathcal{O}(x'_T) = y_T \text{ where } \vec{y} = f_c(x'_T \cup x_{\overline{T}})$$

and $\mathcal{O}(\textsf{not-now}) = \Lambda$.

In defining simulators for string valued computation, the definition of component queries is simplified to $\mathcal{O}(i) = (x_i, \pi_i)$ but the rest of the definition of a simulator interacting with an adversary is unchanged.

TERMINATION. A simulator is said to *terminate* in round $r$ if $r$ is the first round for which $\Upsilon(s_S^{r0}) = \textsf{protocol-done}$. The adversary with which the simulator interacts is then said to terminate at the end of its round $r$.

ADVERSARY'S VIEW. In an execution $e$ of an adversary $A$ with a simulator, the *view* of $A$ in this execution is "everything the adversary sees" before termination: that is, the triple $(\tau_A, r'_A, a'_A)$ consisting of the traffic, the adversary's coins actually consumed, and the adversary's advice which is consumed.

REQUIREMENTS ON $\mathcal{AI}$ AND $\mathcal{AO}$. For $S = (\mathcal{S}, \mathcal{AI}, \mathcal{AO})$ to be a simulator establishing the security of some protocol $P$, we demand that in any $r$-round execution of an adversary $A$ with $P$, that there is exactly one value $r' \leq r$ such that $\mathcal{AI}(\tau_A^{r'\infty}) = x'_T \neq \textsf{not-now}$. Additionally, $\mathcal{AO}(\tau_A^{r\infty}) \in \mathit{corrupted}(\tau_A^{r\infty}) \times \Sigma^{\ell(\tau_A^{r\infty})}$.

That is, the adversary input assumes a well-defined vector of "substituted inputs" $x'_T$ at *some* point in the execution, and the adversary output, evaluated at the final traffic, specifies a meaningful output on behalf of the corrupted processors.

### 2.4.4 Ensembles for Secure Function Evaluation

THE PARAMETER SET $\mathcal{L}(f)$. A vector-valued function family $f$ is a way to *interpret* each common input $c$ as a map $f_c : (\Sigma^{\ell_c})^{n_c} \to (\Sigma^{l_c})^{n_c}$; a string-valued function family $f$ is a way to interpret each common input $c$ as a map $f_c : (\Sigma^{\ell_c})^{n_c} \to \Sigma^{l_c}$. The description of $f_c$ appears on the common input following the usual quantities. Thus to each function family $f = \{f_c\}$ we associate the $k$-indexed family of languages $\mathcal{L}(f) = \{L_k(f)\}$ given by

$$\begin{aligned} L_k(f) \quad = \quad \bigcup_{n\geq 3,\ k,\ell,l,m\geq 0} & \{(\vec{x},\vec{\pi},a_A,c) :\ \vec{x} \in (\Sigma^\ell)^n,\ \vec{\pi} \in (\Sigma^m)^n,\ a_A \in \Sigma^\omega, \text{ and} \\ & \quad c = 1^k\#1^n\#1^\ell\#1^l\#1^m\#C_c, \text{ where } C_c \text{ specifies a} \\ & \quad \text{function } f_c \text{ from the function family } f\} \end{aligned}$$

of all possible inputs, histories, adversary advice strings, and common inputs, in any "proper" initial configuration of the network.

Restricting our attention to proper initial configurations (with the void set of initially corrupted processors), the initial configuration of the players and the adversary are uniquely identified by a vector $(\vec{x},\vec{\pi},a_A,c,\ \vec{r},r_A)$. This is a point in $L_f(k)$ together with coins $\vec{r}$ for the players and $r_A$ for the adversary.

Just as $(\vec{x},\vec{\pi},a_A,c,\ \vec{r},r_A)$ specifies the initial configuration of the players and the adversary, a point $(\vec{x},\vec{\pi},a_A,c,\ r_S,r_A)$ uniquely specifies the initial configurations of the adversary and a simulator $S$. This is a point in $L_f(k)$ together with simulator coins, $r_S$, and adversary coins, $r_A$.

THE ADVERSARY'S VIEW. Consider an adversary, $A$, attacking a protocol, $P$. Let $f = \{f_c\}$ be a function family. With a proper initial configuration specified by $(\vec{x},\vec{\pi},a_A,c,\ \vec{r},r_A)$, there is an associated view which will be held by the adversary at termination (if the protocol terminates). Omitting mention of $\vec{r}$ and $r_A$, there is an associated distribution on adversary views induced by the taking each bit in $\vec{r}$ and $r_A$ to be selected uniformly at random. We let

$$A\text{-VIEW}_k^P(\vec{x},\vec{\pi},a_A,c)$$

denote this distribution of adversary views. Read this as the view which $A$ gets when speaking to the network running protocol $P$. When $\vec{x}$, $\vec{\pi}$, $a_A$ and $c$ are not regarded as fixed, but are allowed to vary in $L_k(f)$, then $A\text{-VIEW}_k^P(\vec{x},\vec{\pi},a_A,c)$ becomes an ensemble $\mathcal{E}A\text{-VIEW}_k^P(\vec{x},\vec{\pi},a_A,c)$ indexed by $k$ and parameterized by $\mathcal{L}(f)$. Sometimes we simplify the notation by writing $\mathcal{E}P_k(\vec{x},\vec{\pi},a_A,c)$ or $\mathcal{E}P_k$ in place of $\mathcal{E}A\text{-VIEW}_k^P(\vec{x},\vec{\pi},a_A,c)$.

THE SIMULATED VIEW. Consider an adversary, $A$, interacting with a simulator, $S$. Let $f = \{f_c\}$ be a function family. With a proper initial configuration specified by $(\vec{x},\vec{\pi},a_A,c,\ r_S,r_A)$, there is an associated view which will be held by the adversary at termination (if the protocol terminates) when $A$ talks to simulator $S$, where $S$ has an ideal evaluation oracle $\mathcal{O}(\vec{x},\vec{\pi};f_c)$.

Omitting mention of $r_S$ and $r_A$, there is an associated distribution on adversary views induced by the taking each bit in $r_S$ and $r_A$ to be selected uniformly at random. We let

$$A\text{-VIEW}_k^S(\vec{x}, \vec{\pi}, a_A, c)$$

denote this distribution of adversary views. Read this as the view which $A$ gets when speaking to the simulator $S$. When $\vec{x}$, $\vec{\pi}$, $a_A$ and $c$ are not regarded as fixed, but are allowed to vary in $L_k(f)$, then $A\text{-VIEW}_k^S(\vec{x}, \vec{\pi}, a_A, c)$ becomes an ensemble $\mathcal{E}A\text{-VIEW}_k^S(\vec{x}, \vec{\pi}, a_A, c)$ indexed by $k$ and parameterized by $\mathcal{L}(f)$. Sometimes we simplify the notation for this ensemble by writing $\mathcal{E}S_k^{\mathcal{O}}(\vec{x}, \vec{\pi}, a_A, c)$ or $\mathcal{E}S_k^{\mathcal{O}}$ in place of $\mathcal{E}A\text{-VIEW}_k^S(\vec{x}, \vec{\pi}, a_A, c)$.

Network input. Consider an adversary, $A$, attacking a protocol, $P$. Let $f = \{f_c\}$ be a function family, and let $S$ be a simulator with adversary input function $\mathcal{AI}$. With a proper initial configuration specified by $(\vec{x}, \vec{\pi}, a_A, c,\ \vec{r}, r_A)$, there is (as long as the protocol terminates) a well defined tagged vector $x'_T$ which is the $x'_T$-value obtained by evaluating the adversary input function $\mathcal{AI}$ on traffic values $\tau_A^{r\infty}$ for $r = 0, 1, \ldots$, until a value different from not-now is obtained. Define

$$\mathcal{NI}(\vec{x}, \vec{\pi}, a_A, c,\ \vec{r}, r_A) = x'_T \cup x_{\overline{T}}.$$

This is the $n$-vector of good players' private inputs "shuffled in" with the values entered into the computation by the adversary, as indicated by $\mathcal{AI}$. It is called the *network input function*, and, intuitively, it specifies, on a particular run, what the *network* has committed to on this run.

Network output. Consider an adversary, $A$, attacking a protocol, $P$. Let $f = \{f_c\}$ be a function family, and let $S$ be a simulator with adversary output function $\mathcal{AO}$. With a proper initial configuration specified by $(\vec{x}, \vec{\pi}, a_A, c,\ \vec{r}, r_A)$, there is (as long as the protocol terminates) a well defined tagged vector $y'_T$ which is the result of applying the $\mathcal{AO}$-function to the traffic $\tau_A^{r\infty}$ which has occurred when the adversary terminates. Define

$$\mathcal{NO}(\vec{x}, \vec{\pi}, a_A, c,\ \vec{r}, r_A) = y'_T \cup y_{\overline{T}}$$

where $y_{\overline{T}}$ is the vector of outputs of good players. The vector above consists of good players' outputs "shuffled in" with the output values for the adversary specified by the function $\mathcal{AO}$. The function $\mathcal{NO}$ is called the *network output*, and specifies, intuitively, what the good players *did* compute as output values, and what the adversary *could* compute as output values on behalf of the corrupted players.

### 2.4.5 The Definition of Secure Function Evaluation

We are now ready to define what it means for a protocol $P$ to securely evaluate a function family $f = \{f_c\}$.

COMPUTATIONAL SECURITY. We first define what it means for a protocol for function evaluation to be secure with respect to a computationally bounded adversary.

**Definition 2.4.2** *Protocol $P$ $t$-*securely computes *$f = \{f_c\}$ if there is a simulator $S = (\mathcal{S}, \mathcal{AI}, \mathcal{AO})$ such that for any polynomial-time $t$-adversary $A$,*

- `(Privacy)` $\mathcal{E}A\text{-VIEW}_k^P(\vec{x}, \vec{\pi}, a_A, c) \underset{L_k(f)}{\approx} \mathcal{E}A\text{-VIEW}_k^S(\vec{x}, \vec{\pi}, a_A, c)$.
- `(Correctness)` *For some negligible function $\epsilon(k)$, for all $(\vec{x}, \vec{\pi}, a_A, c) \in L_k(f)$,*

$$\mathrm{Prob}_{\vec{r}, r_A}\left[\mathcal{NO}(\vec{x}, \vec{\pi}, a_A, c,\ \vec{r}, r_A) \neq f_c(\mathcal{NI}(\vec{x}, \vec{\pi}, a_A, c,\ \vec{r}, r_A))\right] \leq \epsilon(k).$$

Definition 2.4.2 can be explained as follows. When the adversary talks to the network, what she is learning is that which she can computationally closely approximate given an ideal evaluation oracle. How the adversary would compute the output query to this oracle defines what she commits to on a run of the protocol with the simulator. According to this function, when the adversary talks to the network, what she does is to effectively commit to a value $x'_T$. By the time the adversary terminates, she is in possession of a value $y_{\mathcal{T}}$, $\mathcal{T} \supseteq T$. During this run, almost certainly the good players computed $y_{\overline{\mathcal{T}}} = f_{\overline{\mathcal{T}}}(x'_T \cup x_{\overline{T}})$ and the adversary has learned $y_{\mathcal{T}} = f_{\mathcal{T}}(x'_T \cup x_{\overline{T}})$.

STATISTICAL AND PERFECT SECURITY. We now define what it means for a protocol for function evaluation to be secure with respect to an adversary with unlimited computational power. The only change that is made is to require statistical closeness for privacy. The following notion is also called *information-theoretic* security.[5]

**Definition 2.4.3** *Protocol $P$* statistically $t$-securely computes *$f = \{f_c\}$ if there is a simulator $S = (\mathcal{S}, \mathcal{AI}, \mathcal{AO})$ such that for any $t$-adversary $A$,*

- `(Privacy)` $\mathcal{E}A\text{-VIEW}_k^P(\vec{x}, \vec{\pi}, a_A, c) \underset{L_k(f)}{\simeq} \mathcal{E}A\text{-VIEW}_k^S(\vec{x}, \vec{\pi}, a_A, c)$.
- `(Correctness)` *For some negligible function $\epsilon(k)$, for all $(\vec{x}, \vec{\pi}, a_A, c) \in L_k(f)$,*

$$\mathrm{Prob}_{\vec{r}, r_A}\left[\mathcal{NO}(\vec{x}, \vec{\pi}, a_A, c,\ \vec{r}, r_A) \neq f_c(\mathcal{NI}(\vec{x}, \vec{\pi}, a_A, c,\ \vec{r}, r_A))\right] \leq \epsilon(k).$$

There is a corresponding notion of *perfect security,* in which there is equality on the privacy constraint, and no chance of error on the correctness constraint.

[5]For information-theoretic security, one might select a *nonasymptotic* notion of security: the distance between $\mathcal{E}P_k$ and $\mathcal{E}S_k$ is at most $2^{-k}$, and the chance that the $\mathcal{NO}$ differs from $f(\mathcal{NI})$ is at most $2^{-k}$.

## 2.5 Discussion

A great many issues have been considered in constructing our definition of secure function evaluation. In this section, we draw the readers attention to just a *few* of them. Greater discussion and justification for these notions will appear in the paper of Micali and Rogaway [MR91].

THE MARRIAGE OF PRIVACY AND CORRECTNESS. Some earlier definitional work treated privacy and correctness as separate concerns that could be met independently. One must be cautious of approaches like this: in many possible formalizations, protocols exist which would be private with respect to one simulator, correct with respect to another simulator, but not simultaneously private and correct with respect to any simulator.

The idea that correctness should be defined by using the *same* simulator existentially guaranteed for privacy was one of the early ideas underlying this work. In fact, at this point it seems crucial that privacy and correctness be interwoven: though privacy is a meaningful notion in its own right, it is doubtful that a strong notion of correctness is *possible* without its definition being entwine with that of privacy.

STATISTICAL CLOSENESS FOR CORRECTNESS. A protocol should compute what it is supposed to compute—and not just something that "looks like" what the protocol is supposed to compute. For example, a protocol for collaboratively flipping coins is *not* an acceptable protocol for collaboratively computing range points of a pseudorandom generator. (Pseudorandom generators are described in the next chapter.)

REDUCIBILITY. Cryptography is a slippery business. One demonstration of this lies in the fact that many plausible definitions for secure function evaluation fail to make the "obvious" theorems true (or, if they are true, what their proofs are is unclear). Illustrating this, many possible definitions of a secure protocol appear not to provably achieve the following *reducibility* property, informally stated as follows: that a secure protocol for $f$ in the "special" model of computation which is like ours but which provides for the computation of $g$ as a "primitive" should remain a secure protocol for $f$ when a secure computation for $g$ is inserted in place of using the primitive provided by the enriched model.

That is, 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 (an oblivious transfer "box," say, for implementing a voting protocol). You proved that your protocol $P^g$ for computing $f$ was secure in a model of computation in which an "ideal" computation of the primitive $g$ was provided "for free." Now you have continued your work and designed a protocol $P_g$ which securely computes $g$. One would hope that you obtain a secure protocol for $f$ by inserting the code $P_g$ wherever it is necessary in $P^g$ that $g$ be computed.

Our definition of security admits such reducibility. However, stating this theorem precisely and proving that it holds would take us well afield of our goal.

ADVERSARIAL AWARENESS. In the ideal computation of a function, the adversary necessarily "knows" what she has sent off to the trusted party on a particular execution of the protocol. She knows that the function will be computed on this value, shuffled in with the good players' inputs. She knows what values, subsequently, are returned to her by the trusted party.

Through the adversary input function, the adversary output function, and our notion of correctness, we have required that all of this is directly paralleled in a protocol we call secure. In particular, the adversary is "knows" the input she has effectively "sent off" to the network to compute on (in the sense that she could easily calculate this $x'_T$ value, using $\mathcal{AI}$); she knows that, almost certainly, the function will be computed on this value, shuffled in with the good players' inputs; and, finally, the adversary is aware of what values, subsequently, have been effectively returned to her (in the sense that she could easily calculate them, using $\mathcal{AO}$). A failure to mimic any of these properties of the ideal protocol would be a significant breach of the abstraction.

INDEPENDENCE. We wish to ensure the highest possible standard for *independence*—the adversary's inability to significantly correlate her behavior with the private input values held by uncorrupted processors. Our definitions do this, though we shall not in this thesis formalize statements which capture the extent to which independence has been obtained.

CHAPTER 3

# The Constant-Round Protocol

The protocol described in this chapter was invented with a goal of *minimizing* what would seem to be the key resource for secure distributed computation—*interaction*. Of course as interaction is minimized, other complexity measures must simultaneously be kept in check.

This chapter does not prove—or even rigorously state—anything about the protocol we exhibit. This is left to Chapter 4. We begin with an overview.

## 3.1 Overview

To develop some intuition, we first look at why interaction rounds were formerly used in such excess.

THE GMW-PARADIGM. In the multiparty protocols of [GMW87, CDG87, GHY87, GV87, BGW88, CCD88, BG89, RB89, GL90] and others, the underlying notions of security are often quite different, and so are the assumed communication models. Nonetheless, all of them follow the same basic paradigm of Goldreich, Micali and Wigderson [GMW87], which we now describe.

There are three stages to collaboratively compute the value of some function $y = f(x_1, \ldots, x_n)$. (For simplicity, take $f$ to be a Boolean function of $n$ binary strings.) In the first stage, each player "breaks up" his private input into pieces, or *shares*, and distributes these pieces. When sharing a value $b$, for some parameter $t$, $t < n/2$, we require that no $t$ players get information about $b$ from the shares received; and yet, the value $b$ is recoverable given the cooperation of the "good" players—even if the "bad" players try to obstruct this recovery to the best of their abilities.

After the sharing stage, a *computation* stage follows, in which each player, given his own shares of the input $(x_1, \ldots, x_n)$, computes his own share of $f(x_1, \ldots, x_n)$. To accomplish this, the function $f$ to be evaluated is represented by a Boolean circuit, $C$. Thus, in Stage 1, each player got shares of the values along the input wires of $C$. In Stage 2, for each gate of the circuit, from shares for the input wires for this gate the parties compute *in a privacy-preserving manner*, the shares for the output wire of this gate. In general, this privacy-preserving computation employs interaction. In this way, the parties work their way up the circuit, from leaves to root, and eventually hold shares for the value corresponding to the output wire of circuit $C$.

In the third and final stage, the result of the function evaluation is recovered by having the players properly combine the shares held for the output wire of the circuit $C$.

THE PROBLEM. In view of even this brief description, it can be seen that all of these protocols for secure function evaluation run in unbounded "distributed time"—that is, using an unbounded number of rounds of communication. Even though the interaction for each gate can be implemented in a way that requires only a constant number of rounds, the total number of rounds will still be linear in the depth of the underlying circuit.

Bar-Ilan and Beaver [BB89] were the first to investigate reducing the round complexity for secure function evaluation. They exhibited a method that (for information-theoretic security) always saves a logarithmic factor of rounds (logarithmic in the total length of the players' inputs), while the total amount of communication grows only by a polynomial factor. While their result shows that the depth of a circuit is not a lower bound for the number of rounds necessary for securely evaluating it, the savings is far from being substantial in the general setting. Thus, the key question for making secure function evaluation practical or for understanding its complexity is:

> *How many rounds are necessary to securely evaluate a circuit, while keeping the communication and local computation polynomial in the size of the circuit?*

CONSTANT ROUND SECURE COMPUTATION. Many of us believed that more complicated functions (i.e., those with greater circuit complexity) required more rounds for secure evaluation. We now know this to be false, for the case of complexity-theoretic secure computation:

**Main Theorem — Informal version** *There exists a protocol which, using a constant number of rounds and a polynomial amount of communication, allows its participants to evaluate any circuit securely. The protocol works in the model of computation described in Chapter 2, in which parties can communicate privately in pairs and can broadcast messages to everyone. It assumes the existence of a one-way function. The protocol tolerates any polynomial-time adversary who can corrupt fewer than half of the total number of players.*

Informally, the theorem says that interaction is like an atom. Without interaction secure function evaluation is impossible; but with a tiny bit of interaction, it is fully possible. The formal statement of the theorem above is given by Theorem 4.3.1.

A new approach. Achieving low round complexity necessitates a break from the gate-by-gate approach described above. The protocol described here is the first multiparty secure function evaluation protocol which does this. Subsequently, Beaver, Feigenbaum, Kilian and Rogaway [BFKR90] also developed a secure function evaluation protocol which does not follow the gate-by-gate approach.

A bird's-eye view of the constant-round protocol. The method for achieving fast secure function evaluation can be described as finding the right way to generalize the older two-party protocol of Yao [Ya86]. His approach—what I call computing a "garbled circuit"—had been used within the GMW-paradigm for computing the desired "out-going shares" of each gate from its "incoming shares" by engaging in many, suitably chosen, two-party computations. This use, however, leads to an unbounded number of rounds. We, instead, modify the construction "from the inside," generalizing it to many parties but preserving the constant round complexity.

The idea is to have the community use the limited interaction available to it to construct a publicly-known *garbled circuit*, along with a set of *garbled inputs* to feed to this circuit. Taken together, the garbled circuit and the garbled input are called the *garbled program.* The garbled program is sufficiently "scrambled" that its revelation does not divulge more information than what is permissible to divulge. The garbled program is computed using the older gate-by-gate approach to secure function evaluation. Once issued, each individual player evaluates the garbled program on his own, without interacting with other players.

The garbled program is defined is a very special way, so as to allow the players to compute this object in a way that permits them to perform the brunt of the scrambling locally, rather than use intensive interaction to collaboratively scramble the program step-by-step.

To efficiently come up with the garbled program, the players join various pieces together, each piece contributed by an individual participant. Of course, no player is trusted to contribute a correct piece, so each player uses interaction to prove to the community that he has done his work correctly. As usual, verification is simpler than computation, and correctness of very deep circuits (evaluated locally by individual players) can be verified by small, shallow circuits. These can be evaluated securely in a constant number of rounds using the gate-by-gate approach of previous protocols. In the end, the community can be certain that it is issuing a correct garbled program, which has been found with very little interaction and is evaluated without any interaction at all.

Why bootstrapping helps. In the remainder of this chapter, even some of what is described above is abstracted out, as we show how to implement a secure function evaluation protocol on top of *any* other secure function evaluation protocol. In the next chapter, we show that if the underlying secure function evaluation is correct and private, then the protocol implemented on top of it will also be correct and private.

How, intuitively, can we possibly gain something by implementing a secure function evaluation protocol on top of another secure function evaluation protocol? Seems like a strange strategy.

In general, if one wants to securely evaluate some function $f$ in a particularly efficient manner, one would expect to have to exploit the specific structure of the particular function. But in devising a general method of producing secure protocols, the function being evaluated is arbitrary, so there would seem to be no structure there to exploit. The bootstrapping strategy works because the evaluation of $f$ is implemented on top of the evaluation of a function $\hat{f}$ which is concocted to have a remarkably simple structure—so simple that $\hat{f}$ can be evaluated securely in constant rounds. Thus the arbitrary, "structureless" function $f$ necessarily *does* have enough structure to allow it to be distributively evaluated efficiently.

IDEAS FROM WHICH THE PROTOCOL SPRINGS. As mentioned, the idea of performing two-party computation by using a garbled circuit is due to Yao [Ya86].

We require the use of a secure function evaluation protocol, as developed by [GMW87, BGW88, CCD88, RB89]. To achieve constant round complexity, we demand that constant-depth circuits can be evaluated in constant rounds; the exact statement of what is required is given by Theorem 4.1.1. Any of the information-theoretic secure protocols mentioned above, [BGW88, CCD88, RB89], can be implemented to have this property. We note that the security of the constant round protocol, though, does not depend on situating it on top of an information-theoretic secure protocol; a complexity-theoretic secure protocol would do as well.

The protocol of [Ya86] required the intractability of factoring. This assumption was reduced to a trapdoor permutation in [GMW87]. The protocol we develop assumes the mildest of common cryptographic assumptions—the existence of a one-way function. However, this relaxation in assumptions is not new. Galil, Haber and Yung [GHY87] had already showed that a pseudorandom generator suffices to produce the underlying "encryptions" for the garbled circuit that were formerly achieved using public-key cryptography in the [GMW87] protocol. In [BG89], Beaver and Goldwasser explicitly recognize that a one-way function is adequate for the job.

A DETOUR. We have managed to state the general strategy for achieving fast secure function evaluation without even *hinting* at what a garbled program looks like or how it is evaluated! Before we rectify this, we take a short detour and describe a central tool needed to answer these question: the tool is a *pseudorandom generator*.

## 3.2 Pseudorandom Generators

A pseudorandom generator deterministically stretches a short, truly random "seed" into a longer "pseudorandom" string. The distribution induced on the pseudorandom strings when the seeds are sampled from uniformly at random is such that the pseudorandom output "looks random" with respect to any polynomial-time computation.

This notion of a pseudorandom generator was first defined and investigated by Blum and Micali [BM82], and by Yao [Ya82b]. These authors showed that pseudorandom generators exist under appropriate complexity assumptions.

**Definition 3.2.1** *A* pseudorandom generator *is a polynomial-time computable function $G : \Sigma^* \to \Sigma^*$ taking $k$-bit strings to $Q(k)$-bit strings, $Q(k) > k$, such that the $k$-indexed ensembles $\mathcal{E}G(U_k)$ and $\mathcal{E}U_{Q(k)}$ are computationally indistinguishable.*

Call the function $Q(k)$ in Definition 3.2.1 the *stretch* of the generator $G$. The following theorem appears in Boppana and Hirschfeld [BH88]. It says that the ability to stretch the truly random seed just a little implies the ability to stretch it a lot.

**Theorem 3.2.2** *If there exists a pseudorandom generator with stretch $k+1$, then, for any polynomial $Q(k) \geq k+1$, there exists a pseudorandom generator with stretch $Q(k)$.* ■

A major effort has gone into weakening the conditions known sufficient for the existence of a pseudorandom generator, including the work of Levin [Le85], and of Goldreich, Krawczyk, and Luby [GKL88]. This effort has culminated in the work of Impagliazzo, Levin and Luby [ILL89], and Håstad [Ha90]. They characterize nonuniform and uniform pseudorandom generation by the existence of nonuniform and uniform one-way functions, respectively. (But recall that we are limiting the scope of our definitions to nonuniform notions.) We now define a (nonuniform) one-way function. Informally, this is an efficiently computable function whose inverse is computable only a negligible fraction of the time.

**Definition 3.2.3** *A* one-way function *is a polynomial-time computable function $f : \Sigma^* \to \Sigma^*$ such that for any polynomial-time "inverting" algorithm $I$ and any infinite string $a_I$,*

$$\epsilon(k) = \mathrm{Prob}_{x \in U_k}\left[I(1^k, f(x), a_I) \in f^{-1}(f(x))\right]$$

*is negligible.*

The existence of a function $f'$ satisfying the definition above turns out to be equivalent to the existence of a function $f$ satisfying the apparently weaker condition that any inverting algorithm should *fail* to invert $f$ a significant fraction of the time—more precisely, that there exists a constant $c$ such that for any inverting algorithm $I$ and any infinite string $a_I$,

$$\Delta(k) = \mathrm{Prob}_{x \in U_k}\left[I(1^k, f(x), a_I) \notin f^{-1}(f(x))\right]$$

is at least $n^{-c}$ for all sufficiently large $n$ [Ya82b]. Similarly, it is also equivalent to allow the inverting algorithm to be a *probabilistic* polynomial-time algorithm; the probability that $I$ successfully inverts is now taken over the inverting algorithm's coins as well as over $x \in U_k$.

A theorem mentioned previously is the following:

**Theorem 3.2.4 ([ILL89, Ha90])** *A pseudorandom generator exists if and only if there exists a one-way function.* ■

As a simple example of the results cited, consider the *multiplication* function

$$f(x_1x_2) = x_1 \cdot x_2,$$

where $|x_1| = |x_2| + \delta$, $\delta \in \{0,1\}$, and $x_1 \cdot x_2$ is the product of $x_1$ and $x_2$, treated as binary numbers. Suppose that every polynomial-time algorithm fails a fraction $n^{-10}$ of the time (for big enough $n$) to split a number $x$ into numbers $x_1$ and $x_2$ with $x_1x_2 = x$ and $|x_1| = |x_2| + \delta$, $\delta \in \{0,1\}$, when $x$ is the product of random numbers of lengths varying by at most one. Then there exists a pseudorandom generator stretching length-$k$ strings to length $2\bar{n}k + 2$ strings, where $\bar{n}$ is any fixed polynomial in $k$.

## 3.3 High-Level Description

We have said that constant-round secure function evaluation is achieved by efficiently issuing to each player a garbled circuit and a set of garbled inputs at which to evaluate this circuit. This section informally describes what a garbled program looks like, how it is evaluated, why it is plausible that it can be computed quickly, and why it might be issued to the players without compromising their private inputs. The next section more formally describes how to collaboratively compute a function $f$ given a protocol for computing some related function $\hat{f}$, instead.

THE SET-UP. The players want to collaboratively evaluate some function. This function must be represented somehow; we represent it by a Boolean circuit $C$. We assume that $C$ is made up of only two-input gates. Though the gates have bounded fan-in, they may have unbounded fan-out. Each player knows the input bits along some of the input wires to $C$—namely, player $i$, who possesses private input $x_i$, knows the $|x_i|$ bits along the input wires which take $x_i$. The community wants to learn the bits induced along the output wires.

Figure 3.1 is an example of a circuit three players may want to collaboratively evaluate.

The circuit has two gates and five wires. Players 1, 2, and 3 provide the bits $x_1$, $x_2$, and $x_3$ along wires 1, 2, and 3, respectively. Thus the players are trying to evaluate the function $f(x_1, x_2, x_3) = x_1x_2 \vee x_3$. We will suppose that $x_1 = 0$, $x_2 = 1$, and $x_3 = 1$, so the players should compute the bit 1.

(In this example, each player just provides a single bit, and there is only a single bit of output. But neither of these facts will be relevant to the method we describe.)

EVALUATING AN (UNGARBLED) CIRCUIT. How would a circuit $C$ normally be evaluated? Here is one way to describe it.

See Figure 3.2. In the circuit $C$, each wire carries one of two possible *signals*—the formal

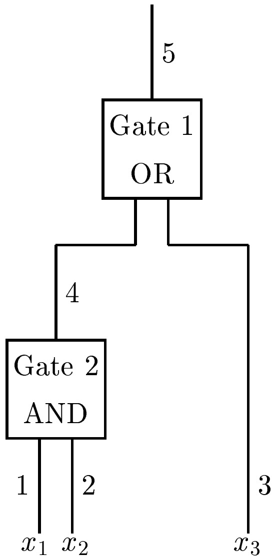


Figure 3.1: *A circuit for three players to distributively compute* $x_1x_2 \vee x_3$.

string **0** or the formal string **1**. The two possible signals are the same for each wire, and everyone knows what the two signals associated to a wire are.

Each signal has a corresponding, "publicly-known" semantics. The **0**-signal means that the wire has semantics 0 (or "false"), while the **1**-signal means that the wire has a semantics of 1 (or "true").

If you know the signals along the incoming wires of a gate, you can mechanically propagate a signal to the out-going wire of the gate. For example, an AND gate with incoming signals of **0** and **1** gets an out-going signal of **0**. In this way, signals propagate up the circuit, from input wires to the output wires, eventually defining signals for all output wires. Since the signals have a fixed, known, semantics, the circuit has then been evaluated.

Evaluating a garbled circuit. Evaluating a garbled circuit is not so very different. Once again, there will be wires, gates, and signals, and these will mirror the structure of the corresponding "ungarbled" circuit. See Figure 3.3, which depicts a garbled circuit, and information related to it.

Wires will still carry one of two signals—but this time, the signals will not be the strings **0** and **1**. Instead, each wire $\omega$ will have longer strings associated to it, signals $s_0^\omega$ and $s_1^\omega$. These will be *random* strings of length $nk+1$—random except that signal $s_0^\omega$ will always end in a **0**, and signal $s_1^\omega$ will always end with a **1**.

Before, the signals associated with a wire were *publicly known*, and the same two signals were associated with each wire. Now, the signals associated with a wire will be *secret*, and they will vary from wire to wire.

Before, the two signals had a fixed semantics, **0** meaning 0 (false) and **1** meaning 1

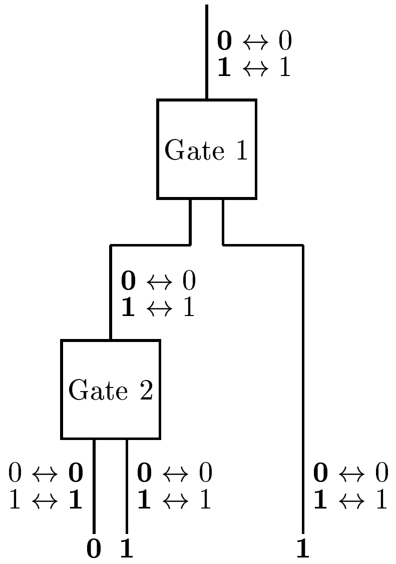


Figure 3.2: *The circuit and its input. Each wire caries one of two signals,* **0** *or* **1**, *with associated semantics* $0 \leftrightarrow \mathbf{0}$ *and* $1 \leftrightarrow \mathbf{1}$. *To compute, signals are propagated mechanically up the circuit.*

(true). Now, the signals will have a secret, "hidden" semantics: $s_0^\omega$ having associated semantics $\lambda^\omega$ and $s_1^\omega$ having associated semantics $\overline{\lambda^\omega}$. For wires other than the output wires, this semantics is *random*, and is not known by anybody. In Figure 3.3, the signals $s_0^1$ and $s_1^1$ have been given the semantics 0 and 1, respectively, while $s_0^2$ and $s_1^2$ have semantics 1 and 0, respectively.

Just as evaluating an ungarbled circuit consists of learning the correct signal for each wire by mechanically propagating signals up the circuit, evaluating a garbled circuit consists of learning one of the two signals associated with each wire, and propagating these signals up the circuit. Initially, you will *hold* (that is, you will "know") one of the two possible signals for each input wire—you will hold the signal with the correct semantics for this input wire. Holding a signal $s_b^\omega$ for a wire $\omega$ will correspond to that wire having semantics of $\lambda^{\omega \oplus b}$. Given the two incoming signals for a gate, a method will be provided allowing you to learn the correct out-going signal for that gate.

For example, in Figure 3.3, knowing $s_0^1$ and $s_0^2$ "means" that the left and right incoming wires to Gate 1 carry 0 and 1, respectively. Consequently, in evaluating this "garbled gate," the players should learn the signal $s_1^4$, since this is the signal for the out-going wire which carries the semantics of $0 \wedge 1 = 0$.

As mentioned, the $s_0^\omega$ and $s_1^\omega$ signals are concocted to be differentiable: we asserted that the last bit of $s_0^\omega$ is a **0** and the last bit of $s_1^\omega$ is a **1**. Thus these signals are referred to as the *even* and *odd* signals, respectively. If you know a signal for a wire, you know whether you possess the even or the odd signal for the wire, but this tells you nothing about the

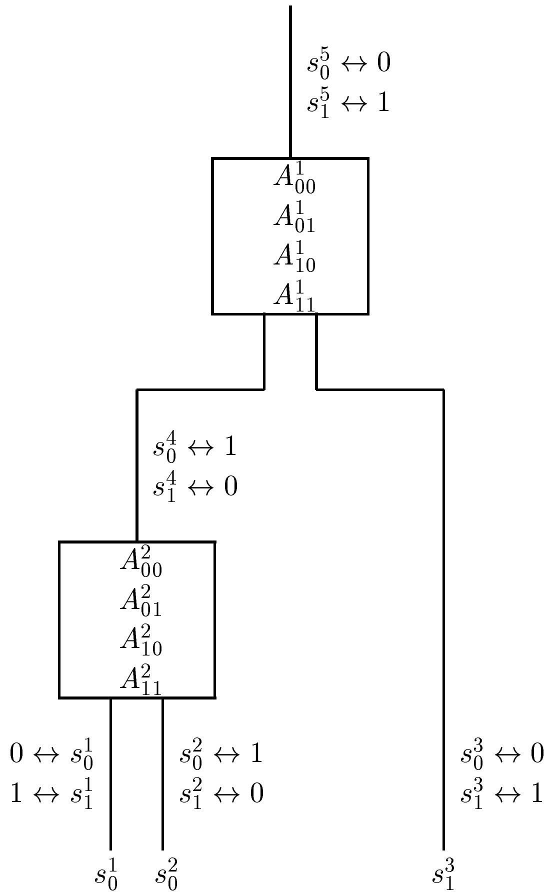


Figure 3.3: *A sample garbled circuit (the eight gate labels) and garbled input (the three signals that are fed in along the input wires). Also shown are the two signals (secretly) associated with each wire, and their corresponding (secret) semantics.*

underlying semantics of the wire, because this was chosen at random independent of the signal being even or odd.

An exception to this is made for output wires, for which we *want* the semantics to be public. We simply assert that the even signals for these wires have semantics of 0, and the odd signals have semantics of 1. Now when a player has computed the signals for the output wires of a circuit, he has also computed the "semantics" of the circuit's output.

In evaluating the circuit of Figure 3.3, each player initially knows $s_0^1$, $s_0^2$, and $s_1^3$. The first two of these are combined and—somehow—each player learns $s_1^4$. Then $s_1^4$ and $s_1^3$ are combined and—somehow—each player learns $s_1^5$. Since this signal belongs to an output wire and is odd, the circuit evaluates to 1.

How signals propagate across a gate. What, exactly, is this mechanism that allows a player, given knowledge of two incoming signals to a gate, to compute the correct out-going

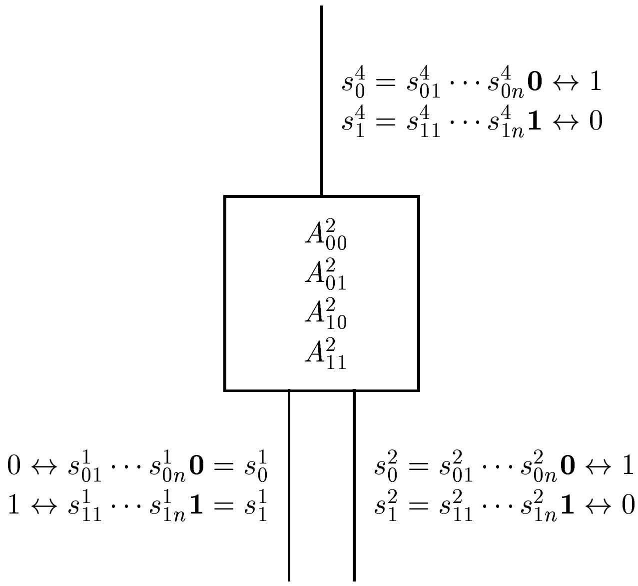


Figure 3.4: *A closer look at the signals associated with the wires of Gate 2, and their corresponding semantics. Each signal consists of $n$ length-$k$ strings, and one additional bit.*

signal for the gate?

Each gate $g$ of the garbled circuit provides "help" for accomplishing this task *for gate $g$*. The help is in the form of a table of four strings of length $nk+1$. Each of these strings is called a *gate label*. The *garbled circuit* is the collection of all of the gate labels for all of the gates.

The four gate labels associated with gate $g$ are written $A_{00}^g$, $A_{01}^g$, $A_{10}^g$, and $A_{11}^g$. If you possess two *even* incoming signals for gate $g$, then $A_{00}^g$ allows you to compute the correct out-going signal; if you possess an even signal for the left incoming wire of gate $g$ and an odd signal for its right incoming wire, then $A_{01}^g$ lets you recover the out-going signal; and so forth.

We now describe how the out-going signal is computed from the incoming signals and the gate labels.

The signals for a wire are not treated atomically. Rather, each signal is considered as consisting of $n$ strings of length $k$, together with one more bit that specifies whether this signal is even or odd. Figure 3.4 shows the makeup of the signals for Gate 2 of Figure 3.3, together with their semantics.

Each of the $n$ "pieces" of each incoming signal serves as the seed of a pseudorandom generator $G$ stretching $k$ bits to $2(\bar{n}k+1)$ bits, where $\bar{n}$ is a fixed polynomial in $k$ which bounds the number of players, $n$, in terms of the security parameter used by the algorithm, $k$. (For concreteness, we will later select $\bar{n} = k^{10}$, quite arbitrarily.) Define $G_0'$ and $G_1'$ by $G(s) = G_0'(s)G_1'(s)$, for $|G_0'(s)| = |G_1'(s)| = \bar{n}k + 1$, and set $G_0(s) = G_0'(s)[1 : nk+1]$ and

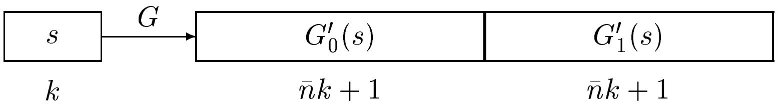


Figure 3.5: *The pseudorandom generator $G$, stretching $k$ bits to $2\bar{n}k+2$ bits, for $\bar{n}=\bar{n}(k)$ a fixed polynomial in $k$. The map $G$ defines $G_0(s)=G(s)[1:nk+1]$ and $G_1(s)=G(s)[\bar{n}k+2:\bar{n}k+nk+2]$.*

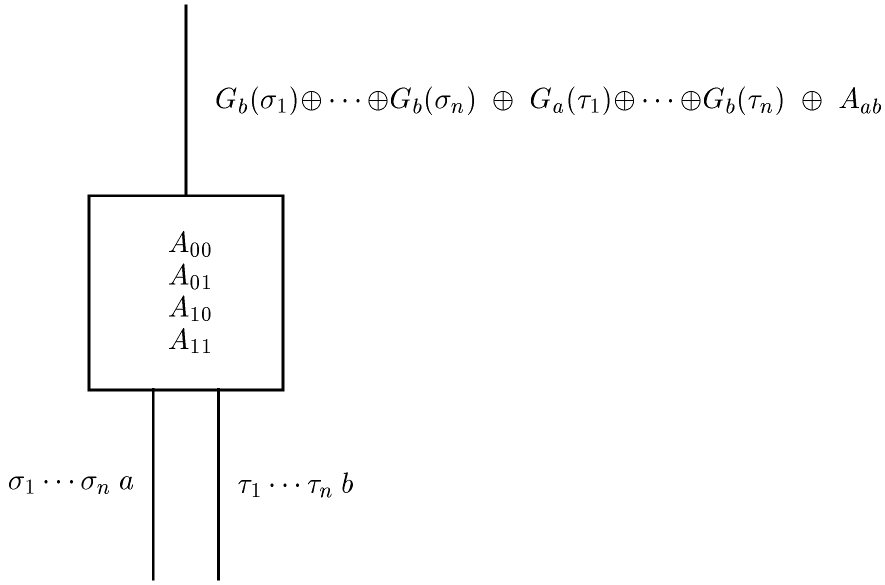


Figure 3.6: *How the out-going signal for a gate is determined from the two incoming signals.*

$G_1(s)=G'_1(s)[1:nk+1]$. See Figure 3.5.

In evaluating the garbled circuit, for each wire incoming to some gate $g$, *either* $G_0$ *or* $G_1$ is computed at each of the $n$ pieces of the incoming signal carried along that wire. To illustrate, suppose that you possess two *even* signals coming into a gate $g$. That is, both of the incoming signals are strings of length $nk+1$ which end in $\mathbf{0}$. Then each of the two signals, minus the last bit, is disected into $n$ $k$-bit long strings. Apply $G_0$ to each of these $2n$ strings to get $2n$ strings of length $nk+1$. The bitwise exclusive-or of all of these images, *also exclusive-or'ed with* $A^g_{00}$, is the desired out-going signal for gate $g$.

More generally (had the signals not both been even), the out-going signal is computed as follows. If the left incoming signal to gate $g$ has parity $a$ and prefix $\sigma_1\cdots\sigma_n$ (each $\sigma_i$ of length $k$); and the right incoming signal to gate $g$ has parity $b$ and prefix $\tau_1\cdots\tau_n$ (each $\tau_i$ of length $k$); and the gate labels for this gate are $A_{00},A_{01},A_{10},A_{11}$; then the out-going signal is defined to be

$$G_b(\sigma_1)\oplus\cdots\oplus G_b(\sigma_n)\ \oplus\ G_a(\tau_1)\oplus\cdots\oplus G_a(\tau_n)\ \oplus\ A_{ab}.$$

This is illustrated in Figure 3.6.

The string $A^g_{00}$ can be thought of as the *encryption* of the out-going signal with the correct semantics given that both incoming signals were even. To decrypt this encryption, you must hold both of the two even incoming signals.

More generally, the convention on the signals being even and odd lets you know which of the four gate labels is meaningfully decrypted given the "secret key" $(\sigma_1 \cdots \sigma_n\, a,\ \tau_1 \cdots \tau_n\, b)$.

We have specified what the garbled circuit looks like: it is the collection of $A^g_{ab}$-values. The garbled input is the collection of those signals which carry the correct semantics for each input wire of the circuit. In Figure 3.3, the garbled input consists of the strings $s^1_0 s^2_0 s^3_1$, while the garbled circuit is $A^1_{00}A^1_{01}A^1_{10}A^1_{11}A^2_{00}A^2_{01}A^2_{10}A^2_{11}$. The garbled circuit together with its garbled input is called the *garbled program.*

We have now described how to evaluate a garbled program, and how to understand the result.

Why can a garbled program be found quickly? For any choice for the even and odd signals of length $nk+1$ for the wires of the circuit $C$, and for any semantic assignment $\omega \mapsto \lambda^\omega$ for these signals, the gate labels for each gate are well defined so that the gate computes correctly with respect to the assigned semantics. For example, in Figure 3.4, we must arrange that

$$A^2_{00} = G_0(s^1_{01}) \oplus \cdots \oplus G_0(s^1_{0n}) \ \oplus\ G_0(s^2_{11}) \oplus \cdots \oplus G_0(s^2_{1n}) \ \oplus\ s^4_1$$

for the garbled Gate 2 to compute correctly on two even incoming signals.

Though in this section we will not write out the general expression for what $A^g_{ab}$ should be, it is clearly a simple function of the signals and semantics for the three wires that touch gate $g$. In particular, *the gate labels depend only on local information.* This means that in calculating the garbled circuit, all the gate labels can be calculated *in parallel.* Since signals and their semantics are selected randomly, the calculation of the garbled circuit completely parallelizes. This will mean that while the communication complexity for computing the garbled circuit depends on the size of the circuit $C$, the number of rounds will not.

Thus to achieve constant rounds the key is to ensure that the calculation of a *single* set of gate labels can be accomplished in constant rounds. It is in order to accomplish this that we have treated signals as consisting of $n$ separate seeds to the generator $G$. In constructing the garbled circuit, each player will be responsible for computing $G$ on "his own" portion of each signal. That is, each player $i$ will be required to locally compute the image under $G$ of the $i^{th}$ piece of each signal; and he will enter this information into the collaborative computation. For example, in Figure 3.3, player $i$—if honest—selects ten random length $k$ strings $s^\omega_{\beta i}$, where $\omega \in [1..5]$, $\beta \in \{0,1\}$, applies $G$ to each of these, and enters all of this data into the collaborative computation. Applying the generator $G$ to the strings may be time-consuming—but this computation is done locally, not distributively. It is in this way that the brunt of the scrambling operation is performed locally.

Given the images of the seeds under the generator and the $\lambda^\omega$-values specifying their semantics, the calculation necessary to compute the gate labels is fixed and straightforward; it is not surprising that this can be performed efficiently.

Why is the protocol private? The technical answer that "the simulation goes through" is not particularly satisfying! We try to give some intuition why the strategy outlined might be private.

The intent is that knowledge of two incoming signals for a gate gives rise to knowledge of the correct out-going signal, *and nothing else that is meaningful.* So if you are magically handed a garbled circuit and a garbled input to feed to it, you should learn *only* a randomly selected signal for each wire not an output wire. For each output wire, you should learn a random signal of the "correct" semantics. Apart from the output wires, knowledge of a single signal for a wire should have no discernible meaning.

In some sense, the secure collaborative computation of the garbled program *does* amount to magically being handed the garbled circuit and the garbled input—except that the signals are not entirely secret, since each player knows one piece of each signal (the piece that player "was responsible for"). The main concern, then, is that the garbled program does divulge extractable information, even if taken together with those pieces of signals a bad player might already know about. That is, as a dishonest player, you know one complete set of signals for the input wires (from knowing the garbled input), and you know those pieces of various other signals that your dishonest friends have told you about. But, since there is some honest player who has not told you the seeds *he* was responsible for, you are *denied* knowledge of some piece of each incoming signal that is not a garbled input. Intuitively, in order for one of the gate labels to be meaningful, you must not be denied knowledge of *any* of the $2n$ seeds whose images under the generator are exclusive-or'ed to decrypt the out-going signal.

In fact, replacing a gate label for which you lack a seed required to "decrypt" that gate label with a truly random gate label does not give rise to a noticeable change in distributions. (Of course, we will be proving this.) Three of the four gate labels for each gate will be like this, and so all of these can be exchanged with truly random strings without a noticeable change in distributions. But this new distribution—a distribution on "fake" garbled circuits—is easily generable. So, releasing a garbled circuit reveals no significant information, insofar as an indistinguishable "fake" garbled circuit can be released if one knows only what the garbled circuit ought compute.

In the next section, we repeat the protocol more formally, without attempting to provide additional intuition.

## 3.4 The Protocol

Notation. We fix notation to be used throughout the remainder of the thesis. Let $G$ denote a pseudorandom generator that stretches a $k$-bit string $s$ to a $2(\bar{n}k+1)$-bit string $G(s)$, where $\bar{n}=k^{10}$ bounds the actual number of players, $n$, in terms of the security parameter, $k$, which will be used by the protocol. (The constant 10 is an arbitrary constant.) Let $G_0$ and $G_1$ be given by $G_0'(s)=G(s)[1\!:\!\bar{n}k+1]$, and $G_1'(s)=G(s)[\bar{n}k+2\!:\!\bar{n}k+2+2\bar{n}k+2]$, with $G_0=G_0'[1\!:\!nk+1]$ and $G_1=G_1'[1\!:\!nk+1]$.

We consider the collaborative computation of a string-valued function $f_c:(\Sigma^\ell)^n\to\Sigma^l$. This function is represented by a circuit $C_c:(\Sigma^\ell)^n\to\Sigma^l$, with $f_c(x_1\cdots x_n)=C_c(x_1\cdots x_n)$. A description of this circuit appears on the common input $c=1^{k'}\#1^n\#1^\ell\#1^l\#1^m\#C_c$. The circuit $C_c$ has $\Gamma$ gates, which are labeled $1,2,\ldots,\Gamma$, and it has $W$ wires, which are labeled $1,2,\ldots,W$. The gates are taken to be two-input gates of arbitrary functionality and arbitrary fan-out. Each gate has distinguished *left* and *right* incoming wire. The numbering of wires is chosen so that the *input wires* are wires $1,\ldots,n\ell$, and the *output wires* are wires $W-l+1,\ldots,W$. The $j^{th}$-bit of private input $x_i$ appears along input wire $\ell(i-1)+j$, and the $j^{th}$-bit of the output $y$ appears along output wire $W-l+j$.

In the description of the protocol that follows, the index $i$ ranges over the players, $i\in[1..n]$. The index $\omega$ ranges over the wires, $\omega\in[1..W]$, or—when indicated—some subset of these wires (like the input wires). The index $g$ ranges over the gates, $g\in[1..\Gamma]$.

Comments on the protocol. The following remarks and terminology will be used in the proof presented in Chapter 4, or are otherwise useful in understanding the protocol.

- When run on common input $c=1^{k'}\#1^n\#1^\ell\#1^l\#1^m\#C$, with $k'$ as the security parameter specified by $c$, the "true" security parameter $k$ used by the players in the protocol is not necessarily $k'$—because we insist that $k$ be at least a polynomial fraction of $|c|$. This is necessary because the adversary is given time polynomial in $|c|$, rather than time polynomial in $k$. We enforce this requirement by saying that $k=\max\{k',|c|^{1/10}\}$, where 10 is an arbitrary absolute constant. Because of this convention, $n$ is bounded above by a fixed polynomial in $k$, $n\le\bar{n}(k)=k^{10}$.
- The *garbled circuit* is the collection of *gate labels* issued to the players, $A^g_{ab}\in\Sigma^{nk+1}$. The *garbled input* is the collection of signals $\sigma^\omega\in\Sigma^{nk+1}$ issued to the players, one for each input wire $\omega$. The *garbled program* is the garbled circuit together with the garbled input. Thus a garbled program looks like

$$\hat{y}=A^1_{00}A^1_{01}A^1_{10}A^1_{11}\cdots A^\Gamma_{00}A^\Gamma_{01}A^\Gamma_{10}A^\Gamma_{11}\,\sigma^1\cdots\sigma^{n\ell}\in\Sigma^{\hat{l}},$$

  where $\hat{l}=(4\Gamma+n\ell)(nk+1)$.
- The mapping specified by Step 1 *from* player inputs $\vec{x}$ and player random coins $\vec{r}$ *to* the garbled program $\hat{y}$, is referred to as the function $\hat{f}$. The function $\hat{f}$ can be

considered as a map from $(\Sigma^{\hat{\ell}})^n$ to $\Sigma^{\hat{l}}$, for $\hat{\ell} = \ell + \rho$, with $\rho = 2kW + W - l$. Of course $\hat{f}$ is actually a function family; it depends on the common input.

- The mapping specified by Step 2 of the protocol, from the garbled program $\hat{y} \in \Sigma^{\hat{l}}$ to the string $y \in \Sigma^{\ell}$ to which it evaluates, is called the *evaluation* function, $\mathtt{Eval}$. We write $y = \mathtt{Eval}(c, \hat{y})$.
- In the evaluation of a garbled program $\hat{y}$, the collection of $W$ signals which come to be *held* are called the *on-path signals.* The other $W$ signals are the *off-path signals.* The collection of $\Gamma$ gate-labels which are used for computing the on-path signals—one $A^g_{ab}$ for each gate $g$—these are called the *on-path gate labels.* The other $3\Gamma$ gate labels are called the *off-path gate labels.*

  Evaluating a garbled program entails learning the on-path signals and outputting the parity of each on-path signal for each output wire, in the proper order.

Step 1: Collaboratively compute the garbled program $\hat{y}$.

`Common Input:` The shared string $c = 1^{k'}\#1^n\#1^\ell\#1^l\#1^m\#C$.

`Private Input:` Each player $i$ has private input $x_i \in \Sigma^\ell$.

`Coin Tosses:` Each player $i$ uses random coins $r_i \in \Sigma^{2kW+W-l}$.

`Compute:` Players compute *gate labels* $A^g_{00}, A^g_{01}, A^g_{10}, A^g_{11} \in \Sigma^{nk+1}$ and *input signals* $\sigma^\omega \in \Sigma^{nk+1}$, for $g \in [1..\Gamma]$ and $\omega \in [1..n\ell]$.

Let $k = \max\{k', |c|^{1/10}\}$. The parties information-theoretically $\lfloor (n-1)/2 \rfloor$-securely compute (with security parameter $k$), gate labels and input signals, defined as follows:

**(a)** **(i)** Each string $r_i$ defines length $k$ strings $s^1_{0i}, s^1_{1i}, \ldots, s^W_{0i}, s^W_{1i}$ and bits $\lambda^1_i, \ldots, \lambda^{W-l}_i$ by asserting that $r_i = s^1_{0i}s^1_{1i}\cdots s^W_{0i}s^W_{1i}\ \lambda^1_i \cdots \lambda^{W-l}_i$.

**(ii)** The private inputs $x_1, \ldots, x_n$ define the bits $b^1, \ldots, b^{n\ell}$ associated with each input wire, according to $b^1 \cdots b^{n\ell} = x_1 \cdots x_n$.

**(iii)** For $b \in \{0,1\}$, let $s^\omega_b = s^\omega_{b1} \cdots s^\omega_{bn}\ b$.

**Comment:** $s^\omega_0$ and $s^\omega_1$ are the even and odd *signals* associated to wire $\omega$.

**(iv)** Define

$$\lambda^\omega = \begin{cases} \lambda^\omega_1 \oplus \cdots \oplus \lambda^\omega_n & \text{for } 1 \le \omega \le W-l, \\ 0 & \text{for } W-l+1 \le \omega \le W \end{cases} \tag{3.1}$$

**Comment:** $\lambda^\omega$ is the *semantics* of signal $s^\omega_0$, and $\overline{\lambda^\omega}$ is the semantics of signal $s^\omega_1$.

**(b)** For each input wire $\omega$ of the circuit, $\omega \in [1..n\ell]$, the string $\sigma^\omega$ is given by

$$\sigma^\omega = s^\omega_{b^\omega \oplus \lambda^\omega}. \tag{3.2}$$

**Comment:** That is, $\sigma^\omega = s^\omega_0$ if $b^\omega = \lambda^\omega$, and $\sigma^\omega = s^\omega_1$ if $b^\omega \neq \lambda^\omega$. In general, $s^\omega_{b \oplus \lambda^\omega}$ is the signal for wire $\omega$ which carries semantics $b$.

**(c)** For each gate $g$ of the circuit, $g \in [1..\Gamma]$, the strings $A^g_{00}$, $A^g_{01}$, $A^g_{10}$ and $A^g_{11}$ are defined as follows: let $\otimes$ denote the function computed by gate $g$, suppose the left wire is wire $\alpha$, the right wire is wire $\beta$, and the output wire is wire $\gamma$, for $\alpha, \beta, \gamma \in [1..W]$. Then, for $a, b \in \{0,1\}$, define the gate label $A^g_{ab}$ by

$$\begin{aligned} A^g_{ab} \;=\; & G_b(s^\alpha_{a1}) \oplus \cdots \oplus G_b(s^\alpha_{an}) \;\oplus\; G_a(s^\beta_{b1}) \oplus \cdots \oplus G_a(s^\beta_{bn}) \;\oplus \\ & s^\gamma_{[(\lambda^\alpha \oplus a) \otimes (\lambda^\beta \oplus b)] \oplus \lambda^\gamma} \end{aligned} \tag{3.3}$$

**Comment:** $\lambda^\alpha \oplus a$ is the semantics of the left wire if you possess a parity-$a$ signal for the left wire, and $\lambda^\beta \oplus b$ is the semantics of the right wire if you possess a parity-$b$ signal for the right wire. In this case, the gate should output the signal of semantics $(\lambda^\alpha \oplus a) \otimes (\lambda^\beta \oplus b)$, which is signal $s^\gamma_{[(\lambda^\alpha \oplus a) \otimes (\lambda^\beta \oplus b)] \oplus \lambda^\gamma}$.

Figure 3.7: *Step 1: the parties collaboratively compute the garbled program $\hat{y}$.*

Step 2: Locally evaluate garbled program $\hat{y}$, computing $y = \texttt{Eval}(\hat{y})$.

| | |
|---|---|
| `Common Input:` | The shared string $c = 1^{k'}\#1^n\#1^\ell\#1^l\#1^m\#C$. |
| `Input:` | A garbled program $\hat{y}$, i.e., gate labels $A^g_{00}, A^g_{01}, A^g_{10}, A^g_{11} \in \Sigma^{nk+1}$, and input signals $\sigma^\omega \in \Sigma^{nk+1}$, for $g \in [1..\Gamma]$, $\omega \in [1..n\ell]$. |
| `Output:` | The string $y \in \Sigma^l$ that this garbled program "evaluates to." |

Let $k = \max\{k', |c|^{1/10}\}$. Each player $i$ behaves as follows:

Initially, player $i$ is said to *hold* the signal

$$\sigma^\omega = \sigma^\omega_1 \cdots \sigma^\omega_n \ \operatorname{lsb}(\sigma^\omega),$$

for each input wire $\omega$, where $|\sigma^\omega_1| = \cdots = |\sigma^\omega_n| = k$.

Consider a gate $g$, having left input wire $\alpha$, right input wire $\beta$, and output wire $\gamma$, with $\alpha, \beta, \gamma \in [1..W]$. Suppose inductively that player $i$ holds the signal $\sigma^\alpha = \sigma^\alpha_1 \cdots \sigma^\alpha_n\ a$ for wire $\alpha$, and the signal $\sigma^\beta_1 \cdots \sigma^\beta_n\ b$ for wire $\beta$, where $|\sigma^\alpha_1| = \cdots = |\sigma^\alpha_n| = |\sigma^\beta_1| = \cdots = |\sigma^\beta_n| = k$ and $a, b \in \{0,1\}$. Then player $i$ computes and is said to *hold* the signal

$$\sigma^\gamma = G_b(\sigma^\alpha_1) \oplus \cdots \oplus G_b(\sigma^\alpha_n) \ \oplus \ G_a(\sigma^\beta_1) \oplus \cdots \oplus G_a(\sigma^\beta_n) \ \oplus \ A^g_{ab} \tag{3.4}$$

for wire $\gamma$. In this way, $\sigma^\omega$ values are computed for each wire $\omega$ of the circuit.

When a value $\sigma^\omega$ is held for each wire $\omega$ of the circuit, each player $i$ outputs

$$y = \operatorname{lsb}(\sigma^{W-l+1}) \ \cdots \ \operatorname{lsb}(\sigma^W) \tag{3.5}$$

as his private output.

Figure 3.8: *Step 2, in which the players, on their own, evaluate the garbled program $\hat{y}$.*

CHAPTER 4

# Proving the Protocol Secure

## 4.1 Introduction

The protocol described in Chapter 3 is not specified fully because we have not said how to implement the secure function evaluation on which it rests. All we have specified is *what* we want to collaboratively compute, and how a computation of this is used to define each player's private output.

We will not remedy this; in fact, we will exploit it. Our strategy has been to abstract out the specifics of the already very complicated information-theoretically secure collaborative computation, and argue that—*whatever* its implementation—if it securely computes the function $\hat{f}$ that it is designed to compute, then the protocol as a whole securely computes the function $f$ that it was designed to compute.

Here, precisely, is what we will assume: that existing protocols—those of [BGW88, CCD88, RB89]—can be used as the basis for proving Theorem 4.1.1, below. We do not say what *particular* protocol is used—though, for concreteness, we state the theorem with the bound on fault-tolerance of the [RB89] protocol, $t = \lfloor (n-1)/2 \rfloor$. If a protocol with a weaker bound on fault tolerance is used, this bound is simply mirrored as the fault-tolerance in Theorem 4.3.1. If an error-free protocol is employed as the basis of Theorem 4.1.1, there will, correspondingly, be no error in the correctness constraint for the complexity-theoretic constant-round protocol.

**Theorem 4.1.1** *Let $f = \{f_c\}$ be a string-valued function family, in which each $f_c : (\Sigma^{\ell_c})^{n_c} \to \Sigma^{l_c}$ is described by a circuit $C_c$, with $f_c(x_1 \cdots x_{n_c}) = C_c(g_1(x_1) \cdots g_{n_c}(x_{n_c}))$, where $g : \Sigma^* \times \mathbf{N} \to \Sigma^*$ is polynomial-time computable and $\{C_c\}$ is a family of constant-depth circuits, each containing only two-input* NAND*-gates and unbounded fan-in* XOR*-gates. Then, for some absolute constant $R$, there is a polynomial-time, $R$-round protocol $P$ which information-theoretically $t$-securely computes $f$, where $t = \lfloor (n-1)/2 \rfloor$.* ■

More general statements than the one above are possible; this one is tailored to our specific needs.

Theorem 4.1.1 says something more general than that a function can be information-theoretically securely computed in constant rounds if it has constant-depth circuits. Rather, it says that this is possible if each player need only apply a polynomial-time function to his private input, and then the result of *this* computation is the input to a collaborative computation of a function having constant-depth circuits.

The intuition why Theorem 4.1.1 holds is the following. Secure function evaluation using the gate-by-gate approach was sketched in Chapter 1, and, for constant depth circuits, protocols employing this approach require a constant number of rounds. Furthermore, because of the *specifics* of the mechanisms for secure function evaluation developed in [BGW88, CCD88, RB89], not only can the shared out-going bit for a bounded fan-in gate be computed from the shared incoming bits in a constant number of rounds, but this is possible for unbounded fan-in XOR gates, as well.[1] For the function $g$ that each player is "asked" to contribute an image under, each player shares the preimage as well as the image. He then "proves" to the community that the shared image was properly computed from the shared preimage. Though this proof will use an amount of communication which grows with the depth of a circuit for computing $g$, it requires only a fixed number of rounds.

How does Theorem 4.1.1 apply to us? Basically, the function we collaboratively evaluate in Step 1 of the protocol of Chapter 3 does indeed factor into a "hard" part, $g$—which players evaluate locally—and an "easy part," $\hat{C}_c$—which the players collaboratively evaluate. Details specifying this decomposition are given in the proof of Theorem 4.3.1.

Since the round complexity for computing the output $y$ in our protocol is precisely the round complexity for computing the garbled program $\hat{y}$, and since the local computational complexity of our protocol as a whole is within a polynomial factor of the local computational complexity for just computing $\hat{y}$, Theorem 4.1.1 assures us (after setting up the appropriate language) of having specified in Chapter 3 a constant-round, polynomial-time protocol, $P$. However, showing that $P$ $t$-securely computes $f_c$ is not a small task. In Section 4.3 we do this. First, in Section 4.2, we establish some preliminaries needed for the argument.

## 4.2 Preliminaries

In this section we collect up some facts which will be useful in proving our main result.

Perhaps the most basic fact we require is that indistinguishability of ensembles defines an equivalence relation. In particular, indistinguishability is transitive (reflexivity and sym-

[1] Alternatively, the result of Bar-Ilan and Beaver [BB89] allows any function with log-depth circuits to be securely evaluated in a constant number of rounds. The unbounded fan-in XOR gates could thus be replaced by a complete binary tree of bounded fan-in XOR gates, and then this result applied.

metry are obvious).

**Proposition 4.2.1** *Suppose $R$ and $S$ are computationally indistinguishable ensembles, and suppose $S$ and $T$ are computationally indistinguishable ensembles. Then $R$ and $T$ are computationally indistinguishable ensembles.*

*Proof*: The proof is a gentle introduction to a standard cryptographic argument. Suppose for contradiction that $R = \mathcal{E}R_k(\omega)$ is computationally distinguishable from $T = \mathcal{E}T_k(\omega)$, as ensembles over a common parameter set $\mathcal{L} = \{L_k\}$. Then there is a polynomial-time distinguisher $D^a$, a polynomial $Q$, and a collection of strings $\{\omega_k \in L_k\}$, such that

$$\left|\mathbf{E}D(1^k, R_k(\omega_k), \omega_k, a) - \mathbf{E}D(1^k, T_k(\omega_k), \omega_k, a)\right| \geq 1 \,/\, Q(k)$$

for infinitely many $k \in \mathbf{N}$. By the triangle inequality, for each such $k$, either

$$\left|\mathbf{E}D(1^k, R_k(\omega_k), \omega_k, a) - \mathbf{E}D(1^k, S_k(\omega_k), \omega_k, a)\right| \geq 1 \,/\, 2Q(k)$$

or

$$\left|\mathbf{E}D(1^k, S_k(\omega_k), \omega_k, a) - \mathbf{E}D(1^k, T_k(\omega_k), \omega_k, a)\right| \geq 1 \,/\, 2Q(k)$$

(or both). Thus either there is an infinite collection of $k \in \mathbf{N}$ such that

$$\left|\mathbf{E}D(1^k, R_k(\omega_k), \omega_k, a) - \mathbf{E}D(1^k, S_k(\omega_k), \omega_k, a)\right| \geq 1 \,/\, 2Q(k),$$

or there is an infinite collection of $k \in \mathbf{N}$ such that

$$\left|\mathbf{E}D(1^k, S_k(\omega_k), \omega_k, a) - \mathbf{E}D(1^k, R_k(\omega_k), \omega_k, a)\right| \geq 1 \,/\, 2Q(k).$$

In the former case, $R$ is distinguishable from $S$, and in the later case, $S$ is distinguishable from $T$. This contradicts the theorem's assumption. ∎

• NOTATION. The notation in the proof above, with all four argument to $D$, gets a bit tiresome. So we will sometimes simplify expressions such as $D(1^k, R_k(\omega_k), \omega_k, a)$ to $D(R_k(\omega))$.

If you can not distinguish ensembles $R$ and $S$ based on a single sample, then you can not distinguish $R$ and $S$ based on many sample points, either. (Recall that we are only dealing with nonuniform indistinguishability.) The proof is implicit in Yao [Ya82b], and follows the now standard "probability walk" argument, which we shall use again in the proof of Theorem 4.3.1.

**Proposition 4.2.2** *If ensembles $R$ and $S$ are computationally indistinguishable over $\mathcal{L} = \{L_k\}$, then, for any polynomial $Q(k)$, $R^{Q(k)}$ and $S^{Q(k)}$ are computationally indistinguishable over $\mathcal{L}^{Q(k)} = \{L_k^{Q(k)}\}$.*

*Proof*: Assume for contradiction that $R^{Q(k)}$ and $S^{Q(k)}$ are computationally distinguishable. Then there is a distinguisher $D^a$, a polynomial $q$, an infinite set $K$, and a collection of strings $\{\omega_k\}$ such that

$$|\mathbf{E}D(R_k(\omega_k)) - \mathbf{E}D(S_k(\omega_k))| \geq 1 \,/\, q(k)$$

for all $k \in K$. We use $D$ and $a$ to construct a distinguisher $\bar{D}^{\bar{a}}$ for distinguishing $R$ from $S$.

For $i \in [0..Q(k)]$ and $\omega \in L_k$, define $T_k^i(\omega) = R_k(\omega)^i S_k(\omega)^{Q(k)-i}$. Note that $T_k^0(\omega) = S_k(\omega)$, and $T_k^{Q(k)}(\omega) = R_k(\omega)$. By the triangle inequality, for each $k \in K$ there must exist a $g_k \in [1..Q(k)]$ such that

$$\nu(k) \;=\; \left|\mathbf{E}D(T_k^{g_k-1}(\omega_k)) - \mathbf{E}D(T_k^{g_k}(\omega_k))\right| \geq 1 \,/\, Q(k)q(k).$$

(Informally, we have taken a "probability walk" between $R_k(\omega_k)$ and $S_k(\omega_k)$, using $T_k^i(\omega_k)$ to specify a set of steps that take you from the one space to the other. Since the two endpoints have significantly different expectations under $D$, some step in between must have associated to it a nonnegligible "jump" in the induced expectations.)

Consider a "distinguisher" $\bar{D}_{\{\vec{r}_k\vec{s}_k\}}$, where each $\vec{r}_k\vec{s}_k \in (\Sigma^*)^{Q(k)-1}$. This distinguisher behaves as follows: it takes a sample $x$ (which can be thought of as being either $R_k(\omega_k)$-distributed or $S_k(\omega_k)$-distributed), and it computes $D(1^k, (r_k^1 \cdots r_k^{g_k-1} x s_k^{g_k+1} \cdots s_k^{Q(k)}), \omega_k, a)$. Observe that if each $r_k^i$ is drawn according to $R_k(\omega_k)$, and each $s_k^j$ is drawn according to $S_k(\omega_k)$, then

$$\begin{aligned}
\nu(k) \;&=\; \left|\mathbf{E}_{\vec{r}_k\vec{s}_k}\mathbf{E}\bar{D}_{\{\vec{r}_k\vec{s}_k\}}(R_k(\omega_k)) - \mathbf{E}_{\vec{r}_k\vec{s}_k}\mathbf{E}\bar{D}_{\{\vec{r}_k\vec{s}_k\}}(S_k(\omega_k))\right| \\
&\leq\; \mathbf{E}_{\vec{r}_k\vec{s}_k}\left|\mathbf{E}\bar{D}_{\{\vec{r}_k\vec{s}_k\}}(R_k(\omega_k)) - \mathbf{E}\bar{D}_{\{\vec{r}_k\vec{s}_k\}}(S_k(\omega_k))\right|.
\end{aligned}$$

Since the average of a bunch of elements cannot exceed the maximum of those elements, the equation above implies that there exists, for each $k$, *particular* values $r_k^1, \cdots, r_k^{g_k-1}$ and $s_k^{g_k+1}, \cdots, s_k^{Q(k)}$ such that

$$\nu(k) \;\leq\; \left|\mathbf{E}\bar{D}_{\{\vec{r}_k\vec{s}_k\}}(R_k(\omega_k)) - \mathbf{E}\bar{D}_{\{\vec{r}_k\vec{s}_k\}}(S_k(\omega_k))\right|.$$

Letting $\bar{a}$ reasonably encode the $\{g_k\}$-values, these $\{\vec{r}_k\vec{s}_k\}$-values, and the infinite string $a$, we have constructed a distinguisher $\bar{D}^{\bar{a}}$ which, along the infinite set $K$, distinguishes $R$ and $S$ by at least the inverse polynomial $1 \,/\, q(k)Q(k)$. This is a contradiction. ∎

We draw the following immediate conclusion to Proposition 4.2.2, where $\bar{n} = k^{10}$ (or any other polynomial in $k$):

**Lemma 4.2.3** *Let $G : \Sigma^k \to \Sigma^{2\bar{n}k+2}$ be a pseudorandom generator, and let $Q(k)$ be a polynomial. Then the $k$-indexed ensembles $\mathcal{E}U_{(2\bar{n}k+2)Q(k)}$ and $\mathcal{E}[G(U_k)]^{Q(k)}$ are computationally indistinguishable.* ∎

## 4.3 Proof of the Main Theorem

Our goal is to show the following:

**Theorem 4.3.1 (Main theorem—string-valued computation)** *Assume a one-way function exists, and let* $f = \{f_c\}$ *be a string-valued function family in which each* $f_c : (\Sigma^{\ell_c})^{n_c} \to \Sigma^{l_c}$ *is described by a circuit* $C_c$ *for computing it. Then, for some absolute constant* $R$*, there is a polynomial-time,* $R$*-round protocol* $P$ *that* $t$*-securely computes* $f$*, where* $t = \lfloor (n-1)/2 \rfloor$.

We emphasize that the protocol $P$ which Theorem 4.3.1 asserts the existence of is a *fixed* protocol: in our formulation the common input contains a description of the function the protocol distributively evaluates. Though the polynomial that bounds $P$'s running time in terms of the length of the common input has degree which depends on the underlying one-way function, the constant $R$ does not depend on this.

*Proof*: We have described the protocol $P$ in Chapter 3, given a protocol for information-theoretic secure function evaluation. We must establish that this protocol securely computes $f$. This entails exhibiting a simulator $S = (\mathcal{S}, \mathcal{AI}, \mathcal{AO})$, and showing that the simulator "works" to establish $P$'s privacy and correctness.

### Part 1: Strategy, and exhibition of the simulator $S$.

A MENAGERIE OF PROTOCOLS, SIMULATORS, AND ORACLES. To describe the simulator $S$ and prove that it works, we will introduce some extra terminology. There are sufficiently many terms that the table of Figure 4.1 may help the reader keep them straight. We begin by defining the first row of terms in this table, taking a closer look at the secure computation of Step 1.

THE PROTOCOL $\hat{P}$. In Step 1 of the protocol, a player $i$ gets private input $x_i \in \Sigma^\ell$ and common input $c = 1^{k'}\#1^n\#1^\ell\#1^l\#1^m\#C$, where $C$ specifies $\Gamma$, $W$, and the topology of a circuit on $W$ wires and $\Gamma$ gates, obeying the conventions indicated on Page 71. He computes $k = \max\{k', |c|^{1/10}\}$, $\rho = 2kW + W - l$, $\hat{\ell} = \ell + \rho$, $\hat{l} = (4\Gamma + n\ell)(nk+1)$, and flips coins $r_i \in \Sigma^\rho$. He then runs a protocol $\hat{P}$ for secure function evaluation on private input $x_i r_i \in \Sigma^{\hat{\ell}}$ and common input $\hat{c} = 1^k\#1^n\#1^{\hat{\ell}}\#1^{\hat{l}}\#1^m\#\hat{C}$, where $\hat{C}$ is a certain constant-depth circuit of two-input NAND gates and $n$-input XOR gates, a description of $\hat{C}$ being efficiently computable from $c$. Circuit $\hat{C}$ specifies the function $\hat{f}$ which protocol $\hat{P}$ computes by asserting that

$$\hat{f}(x_1r_1, \cdots, x_nr_n) \quad = \quad \hat{C}(g_1(cx_1r_1), \cdots, g_n(cx_nr_n)),$$

where

$$g_i(cx_ir_i) \quad = \quad x_ir_iG(s_{0i}^1)G(s_{1i}^1)\cdots G(s_{0i}^W)G(s_{1i}^W),$$

| | | | | | |
|---|---|---|---|---|---|
| $\hat{P}$ | The protocol which $t$-securely computes $\hat{f}(\vec{x}\vec{r})$. | $\hat{S}$ | Simulator for this protocol. | $\hat{\mathcal{O}}$ | $= \hat{\mathcal{O}}(\vec{x}\vec{r}, \vec{\pi}; \hat{f})$, the oracle for the simulator $\hat{S}$. |
| $\bar{P}$ | Player $i$ selects $r_i \leftarrow \$$, computes $\hat{c}$ from $c$, then runs $\hat{P}_i$ with private input $x_i r_i$ and common input $\hat{c}$. | $\bar{S}$ | Same as $\hat{S}$ apart from using $\hat{c}$ instead of $c$ and a syntactic modification of strings handed over to the adversary. | $\bar{\mathcal{O}}$ | Selects $\vec{r} \leftarrow \$$, then answers according to $\hat{\mathcal{O}}(\vec{x}\vec{r}; \hat{f})$. |
| | | | | $\bar{\bar{\mathcal{O}}}$ | Same as $\bar{\mathcal{O}}$, but "turns off" to query of $j$, for some random $j \in [1..n]$. |
| | | | | $\ddot{\mathcal{O}}$ | Same as $\dot{\mathcal{O}}$, but "turns off" to query of $j$, for some random $j \in [1..n]$. |
| | | $\dot{S}$ | Same as $\hat{S}$, except garbled program is evaluated. | $\dot{\mathcal{O}}$ | Same as $\bar{\mathcal{O}}$, but returns a "fake" garbled program, having random strings for 3/4 of the gate labels. |
| $P$ | Same as $\bar{P}$, except outputs the evaluation of the garbled program. | $S$ | Same as $\dot{S}$, except simulates the behavior of $\dot{\mathcal{O}}$ using $\mathcal{O}$. | $\mathcal{O}$ | $= \mathcal{O}(\vec{x}, \vec{\pi}; f)$, the oracle for the simulator $S$. |

Figure 4.1: *A myriad of closely related protocols, simulators, and oracles. Omitted from the table are the oracles $\mathcal{O}_i$, for $0 \le i \le \Gamma$, and the oracle $\mathcal{O}_*$. Only $\hat{\mathcal{O}}$ and $\mathcal{O}$ are "true" oracles; the rest are probabilistic algorithms of $c$, $\vec{x}$, $\vec{\pi}$, $f$, and the oracle's queries.*

with $s^{\omega}_{bi}$ defined from $r_i$ according to the equation of Step 1a(i) of the protocol $P$. The definition of $\hat{C}$ can then be read off the description of Step 1 of the protocol.

We assume that $c$ is encoded within the circuit $\hat{C}$, so that not only is the map $c \mapsto \hat{c}$ easily computed, but so to is the inverse map $\hat{c} \mapsto c$.

Applying Theorem 4.1.1, Step 1 of $P$ can be performed in polynomial-time and constant rounds. Since Step 2 requires polynomial-time and no communication at all, protocol $P$ is a polynomial-time, constant-round protocol.

Definition of $\hat{S}$. Protocol $\hat{P}$ is an information-theoretically secure protocol for $\hat{f}$. Let $\hat{S} = (\hat{\mathcal{S}}, \widehat{\mathcal{AI}}, \widehat{\mathcal{AO}})$ be the simulator which establishes the security of $\hat{P}$. Simulator $\hat{S}$ is given access to an oracle $\mathcal{O} = \mathcal{O}(\vec{x}\vec{r}, \vec{\pi}; \hat{f})$.

It is through $\hat{\mathcal{S}}$, $\widehat{\mathcal{AI}}$, and $\widehat{\mathcal{AO}}$ that $\mathcal{S}$, $\mathcal{AI}$, and $\mathcal{AO}$ will be defined. Defining the functions $\mathcal{AI}$ and $\mathcal{AO}$ is the easy part, so we dispense with it right off.

Definition of $\mathcal{AI}$. The adversary input function for $S$ is defined as follows: $\mathcal{AI}(c, \tau) = x'_T$ when $\widehat{\mathcal{AI}}(\hat{c}, \tau) = x'_T r'_T$. That is—after adjusting the common input in the same way that $P$

adjusts it for $\hat{P}$—the adversary input function associated to $S$ just "reads off" the adversary input as the appropriate substring of $\widehat{\mathcal{AI}}$, applied to the same conversation.

Definition of $\mathcal{AO}$. As the adversary output function for $\hat{S}$ specifies a garbled program, the natural thing is to define the adversary output function for $S$ to simply evaluate that garbled program: thus $\mathcal{AO}(c, \tau)$ is defined as $\mathtt{Eval}(\hat{c}, \widehat{\mathcal{AO}}(\hat{c}, \tau))$.

Construction of $S$, at a glance. In what way does $\hat{S}$ fail to be a good simulator for $P$? At a glance, $\hat{S}$ would seem to be a completely different beast from the simulator $S$ we wish to construct, because $\hat{S}$ has access to an $\mathcal{O}(\vec{x}\vec{r}, \vec{\pi}; \hat{f})$-oracle, while we only have access to an $\mathcal{O}(\vec{x}, \vec{\pi}; f)$-oracle.

All the same, the simulator $S$ we define will behave very much like $\hat{S}$. Except that whenever $\hat{S}$ would interact with its oracle $\hat{\mathcal{O}}(\vec{x}\vec{r}, \vec{\pi}; \hat{f})$, simulator $S$ will interact with its oracle $\mathcal{O}(\vec{x}, \vec{\pi}; f)$, instead. From this, it will come up with a response of its own creation. For example, to a component query of $i$, the oracle won't simply return the pair $(x_i, \pi_i)$, but some $(x_i r_i, \pi_i)$, instead. In effect, we construct a "fake" oracle, $\dot{\mathcal{O}}$, which behaves like the "real" oracle, $\hat{\mathcal{O}}$. Oracle $\dot{\mathcal{O}}$ is not really a "true" oracle at all, but a rather smart little probabilistic algorithm.

This *oracle substitution* method is at the center of the proof described here. Because of the complexity of the arguments involved, oracle substitutions are described as being made in several stages.

The $\hat{\mathcal{O}}(\vec{x}\vec{r}, \vec{\pi}; \hat{f})$-oracle, though not exactly (or even statistically) simulatable with only the aid of an $\mathcal{O}(\vec{x}, \vec{\pi}; f)$-oracle, will be shown to be very closely approximable nonetheless. In fact, our approximation $\dot{\mathcal{O}}$ to the oracle $\hat{\mathcal{O}}$ will be such a good approximation that no polynomial-time test could tell with which of the two oracles it was conversing (as long as the polynomial-time test made fewer than $n$ component queries to the oracle). In particular, the view provided to some polynomial-time $t$-adversary $A$ when talking to the simulator $\hat{S}$ running under $\hat{\mathcal{O}}(\vec{x}\vec{r}, \vec{\pi}; \hat{f})$ will be computationally indistinguishable from the view provided to $A$ when it speaks to the simulator $\dot{S}$ we construct running under the oracle $\dot{\mathcal{O}}$ we devise.

The meaning of $\hat{P}$'s privacy. The protocol $\hat{P}$ $t$-securely computes $\hat{f}$ in the information-theoretic sense. The privacy part of this assertion means that for any polynomial-time $t$-adversary $A$ interacting with the network running $P$, the view $A$ receives from the network is very closely approximated by interacting with the simulator $\hat{S}$, instead. In symbols, and using the "shorthand" notation mentioned following the definitions of $A\text{-VIEW}_k^P$ and $A\text{-VIEW}_k^S$, we have, *by assumption,* that

$$\mathcal{E}\hat{P}_k(\vec{x}\vec{r}, \vec{\pi}, a_A, \hat{c}) \quad \simeq \quad \mathcal{E}\hat{S}_k^{\hat{\mathcal{O}}(\vec{x}\vec{r}, \vec{\pi}; \hat{f})}(\vec{x}\vec{r}, \vec{\pi}, a_A, \hat{c}). \tag{4.1}$$

Equation 4.1 asserts statistical indistinguishability of $\mathcal{L}(\hat{f})$-parameterized ensembles. We sometimes abbreviate statements like the one above to the less cumbersome $\mathcal{E}\hat{P}_k \simeq \mathcal{E}\hat{S}_k^{\hat{\mathcal{O}}}$.

Definition of $\bar{P}$. We define a protocol $\bar{P}$ "in between" protocols $\hat{P}$ and $P$. This helps us reason about $P$.

Protocol $\bar{P}$ is identical to $P$ apart from one matter: in protocol $P$, the garbled program $\hat{y}$ which $\hat{P}$ computes is evaluated to a string $y = \mathtt{Eval}(\hat{c}, \hat{y})$, and player $i$ outputs $y$ instead of $\hat{y}$. Protocol $\bar{P}$ does not do this; it simply outputs the garbled program $\hat{y}$.

In other words, $\bar{P}$ instructs each player $i$ to take his private input $x_i$ and his common input $c$, and *replace* his private input by $x_i r_i$ and replace his common input by $\hat{c}$, where $r_i$ is $i$'s length-$\rho$ prefix of coins. After this initial replacement, $\bar{P}$ instruct player $i$ to behave as $\hat{P}$ dictates.

Definitions of $\bar{S}$ and $\bar{\mathcal{O}}$. Paralleling the modification of $\hat{P}$ to $\bar{P}$, we modify simulator $\hat{S}$ to create a different simulator $\bar{S}$, and modify oracle $\hat{\mathcal{O}}$ to a make a different oracle $\bar{\mathcal{O}}$.

We consider a probabilistic analog to the oracle $\hat{\mathcal{O}}$, which we call $\bar{\mathcal{O}}$. Oracle $\bar{\mathcal{O}}$ behaves as follows: it selects a random $\vec{r} \leftarrow (\Sigma^\rho)^n$, and then behaves like $\hat{\mathcal{O}}(\vec{x}\vec{r}, \vec{\pi}; \hat{f})$, returning $(x_i r_i, \pi_i)$ to a component query of $i$, and returning $\bar{y}_T = \hat{f}(x'_T r'_T \cup x_{\overline{T}} r_{\overline{T}})$ to an output query of $x'_T r'_T$.

The simulator $\bar{S}$ is very similar to the simulator $\hat{S}$. It begins by mapping its common input $c$ to the string $\hat{c}$, and pretending, subsequently, that $\hat{c}$ was the common input it was issued. Additionally, when a processor $i$ is corrupted, simulator $\hat{S}$ would normally provide information to the adversary specifying the private input $x_i r_i$ of the corrupted processor, its history $\pi_i$, and its computational state $s_i^{r\infty}$. Simulator $\bar{S}$ answers identically, except that the private input of $i$ is replaced by $x_i$, and the computational state of $i$ is syntactically modified so that $r_i$ specifies the coins that were initially flipped when player $i$ executes $\bar{P}$.[2]

Protocol $\bar{P}$ is well-approximated by $\bar{S}^{\bar{\mathcal{O}}}$. We argue later that Equation 4.1 and our definitions of $\bar{P}$, $\bar{S}$, and $\bar{\mathcal{O}}$ imply that

$$\mathcal{E}\bar{P}_k(\vec{x}, \vec{\pi}, a_A, c) \quad \simeq \quad \mathcal{E}\bar{S}_k^{\bar{\mathcal{O}}}(\vec{x}, \vec{\pi}, a_A, c). \tag{4.2}$$

This is the content of Claim 4. Equation 4.2 asserts statistical indistinguishability of $\mathcal{L}(f)$-parameterized ensembles.

The simulator $\dot{S}$. When an adversary attacking the network running $\bar{P}$ corrupts a processor $i$ in her final round, she discovers, encoded in $i$'s computational state, $i$'s private output—the garbled program $\bar{y}$. Simulator $\dot{S}$ is obtained from $\bar{S}$ by replacing the string returned by $\bar{S}$ in response to any final-round corruption by the corresponding string in which the specified output value, rather than being the garbled program $\hat{y}$, is the value $y$ that this garbled program evaluates to. (For technical reasons—see page 90—we also demand

[2]Since $\bar{P}$ is a protocol *we* specify the description of, there is no problem in determining how these coins are to be inserted in a description of the computational state so as to "look" like a computational state obtained from a processor running under $\bar{P}$.

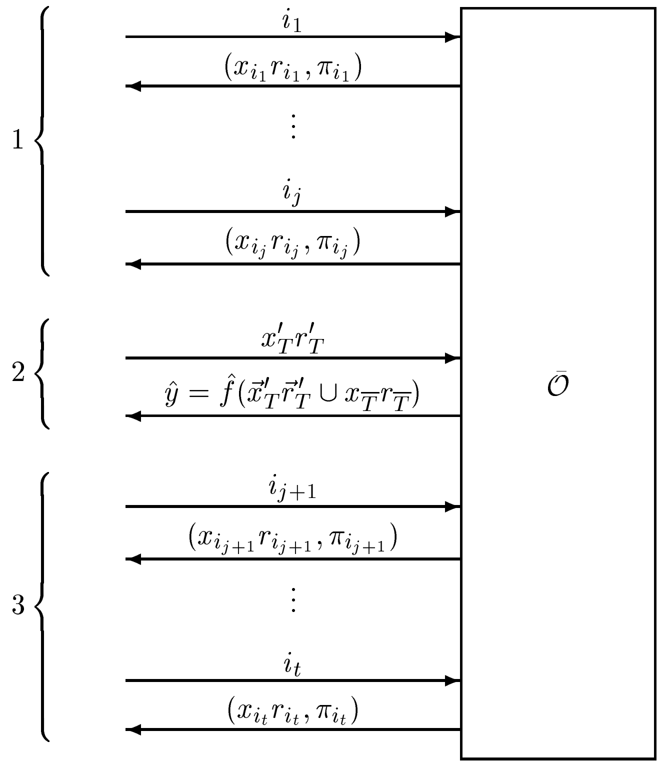


Figure 4.2: *What the simulator $\dot{S}$ asks of its oracle $\bar{\mathcal{O}}$*

that $\dot{S}$ provides a distinguished *forbidden*-value to the adversary with which it speaks if $\dot{S}$'s oracle returns a distinguished *forbidden*-value. Oracles $\bar{\mathcal{O}}$ and $\dot{\mathcal{O}}$ do not return *forbidden*.) We will later show (and it is pretty obvious, given Equation 4.2), that

$$\mathcal{E}P_k(\vec{x}, \vec{\pi}, a_A, c) \quad \simeq \quad \mathcal{E}\dot{S}_k^{\bar{\mathcal{O}}}(\vec{x}, \vec{\pi}, a_A, c). \tag{4.3}$$

This is the content of Claim 5. Equation 4.3 asserts the statistical indistinguishability of $\mathcal{L}(f)$-parameterized ensembles.

THE ORACLE $\dot{\mathcal{O}}$. So far, the modifications to $\hat{P}$, $\hat{S}$, and $\hat{\mathcal{O}}$ have been simple, essentially "syntactic" modifications. In essence, we have simply set ourselves up to do something more interesting: to replace the oracle which knows about $\hat{f}$ with an oracle that only knows about $f$. That is, we will now replace the oracle $\bar{\mathcal{O}}$ by a "fake" oracle $\dot{\mathcal{O}}$, realizable given an oracle $\mathcal{O} = \mathcal{O}(\vec{x}, \vec{\pi}; f)$. We now describe the behavior of $\dot{\mathcal{O}}$.

Let us consider the sequence of queries that $\dot{S}$ makes to its oracle $\bar{\mathcal{O}}$, and describe how we will answer them with $\dot{\mathcal{O}}$. See Figures 4.2 and 4.3.

As with any simulator interacting with its ideal evaluation oracle, there are three phases of oracle queries: (1) the component queries; (2) the output query; and (3) additional component queries.

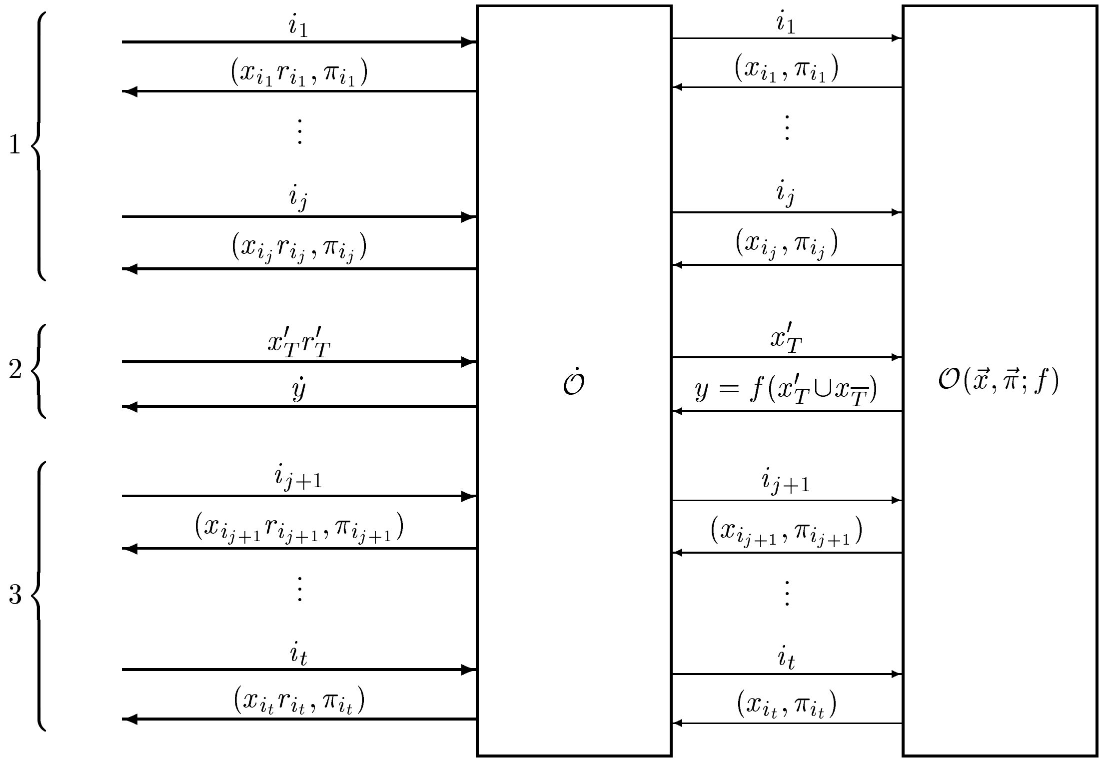


Figure 4.3: *Alternatively, $\dot{S}$'s queries can be answered by the "fake" oracle, $\dot{\mathcal{O}}$, which uses an $\mathcal{O}(\vec{x}, \vec{\pi}; f)$ oracle to compose its answers.*

The oracle $\dot{\mathcal{O}}$ begins by selecting a random $\vec{r} \in (\Sigma^\rho)^n$. To a component query of $i$ (whether before or after the output query), $\dot{\mathcal{O}}$ will make an oracle call to its $\mathcal{O}(\vec{x}, \vec{\pi}; f)$-oracle to learn $x_i$ and $\pi_i$; it will then return $(x_i r_i, \pi_i)$.

Note that the distribution on what $\dot{\mathcal{O}}$ returns in Phase 1 and Phase 3 of the oracle queries is identical to the distribution that $\bar{\mathcal{O}}$ returns for these queries.

What remains is to describe how $\dot{\mathcal{O}}$ answers an output query of $x'_T r'_T$.

The oracle $\dot{\mathcal{O}}$, on receiving the output query $x'_T r'_T$, makes a query of $x'_T$ to $\mathcal{O}(\vec{x}\vec{\pi}; f)$, receiving a string $y \in \Sigma^\ell$. The strings $y$, $r'_T$, $r_{\overline{T}}$, and additional random bits are used in constructing $\dot{\mathcal{O}}$'s response

$$\dot{y} = A^1_{00} A^1_{01} A^1_{10} A^1_{11} \cdots A^\Gamma_{00} A^\Gamma_{01} A^\Gamma_{10} A^\Gamma_{11}\, \sigma^1 \cdots \sigma^{n\ell} \in \Sigma^{\hat{l}}$$

to the output query. The string $\dot{y}$ is called the *fake garbled program*, as opposed to the *real garbled program*, $\bar{y}$, that oracle $\bar{\mathcal{O}}$ returns.

To construct the fake garbled program, define $\vec{r}'' = r_T \cup r'_{\overline{T}}$ and set $s^1_{0i} s^1_{1i} \cdots s^W_{0i} s^W_{1i} = r''_i\,[1\!:\!2kW]$, where $|s^\omega_{bi}| = k$, for $i \in [1..n]$. Define $s^\omega_b = s^\omega_{b1} \cdots s^\omega_{bn}\, b$ for $\omega \in [1..W]$, $b \in \{0, 1\}$. What we have done is produce signals which are a proper "intermixing" of random strings with those strings which the output query $x'_T r'_T$ specifies.

A "random path" through the garbled circuit is selected by flipping a coin $\mu^\omega$ for each wire $\omega$ that is not an output wire, $\omega[1..W-l]$. The *fake garbled program*, $\dot{y}$, will be devised so that $s^\omega_{\mu^\omega}$ will be the signal held for wire $\omega$ when $\dot{y}$ is evaluated, for $\omega \in [1..W-l]$. (Claim 3 establishes that this is really so.) For an output wire $\omega$, $\dot{y}$ will be constructed so that the semantically correct signal will be held, $s^\omega_{y[\omega-(W-l)]}$. For uniformity in notation, then, set $\mu^\omega = y[\omega - W + l]$ for each output wire $\omega$, $\omega \in [W-l+1..W]$.

The conditions specified by the last paragraph are simple to satisfy by the appropriate choice of *fake input signals*, $\sigma^1 \cdots \sigma^{n\ell}$, and a correct definition of just *one* of the four gate labels $A^g_{\alpha\beta}$, for each gate $g$. Namely, select fake garbled inputs of

$$\sigma^\omega = s^\omega_{\mu^\omega} \tag{4.4}$$

for each input wire $\omega$. Then, to a gate $g$ with left input wire $\alpha$, right input wire $\beta$, and output wire $\gamma$, define one of the four gate labels $A^g_{ab}$ for each gate $g$ according to

$$A^g_{\mu^\alpha\mu^\beta} \quad = \quad G_{\mu^\beta}(s^\alpha_{\mu^\alpha 1}) \oplus \cdots \oplus G_{\mu^\beta}(s^\alpha_{\mu^\alpha n}) \ \oplus \ G_{\mu^\alpha}(s^\beta_{\mu^\beta 1}) \oplus \cdots \oplus G_{\mu^\alpha}(s^\beta_{\mu^\beta n}) \ \oplus \ s^\gamma_{\mu^\gamma}. \tag{4.5}$$

We have thus specified fake garbled input $\sigma^1 \cdots \sigma^{n\ell}$ of the fake garbled program $\dot{y}$, and we have specified, for each gate $g$, one of the four gate labels for $g$. All that remains unspecified in $\dot{y}$ are the remaining three gate labels of each gate $g$. *Set all* $3\Gamma$ *of these* $A^g_{\alpha\beta}$*-values to be* <u>*random*</u> *strings of length* $nk+1$.

$\diamondsuit \quad \diamondsuit \quad \diamondsuit$

Having defined oracle $\dot{\mathcal{O}}$ and the simulator $\dot{S}$, we have defined the simulator $S$: $S = S^{\mathcal{O}(\vec{x},\vec{\pi};f)}$ is $\dot{S}^{\dot{\mathcal{O}}}$, except that the oracle responses we have imagined being provided from $\dot{\mathcal{O}}$ are computed by $S$ itself, with the aid of its $\mathcal{O}(\vec{x},\vec{\pi};f)$-oracle. Notice that the responses of $\dot{\mathcal{O}}$ are easily computable from the responses of $\mathcal{O}$ and some coin tosses.

Oracle $\dot{\mathcal{O}}$ well-approximates oracle $\bar{\mathcal{O}}$. The main goal is to establish that

$$\mathcal{E}\dot{S}^{\bar{\mathcal{O}}}_k(\vec{x},\vec{\pi},a_A,c) \quad \approx \quad \mathcal{E}\dot{S}^{\dot{\mathcal{O}}}_k(\vec{x},\vec{\pi},a_A,c). \tag{4.6}$$

This equation asserts computation indistinguishability of $\mathcal{L}(f)$-parameterized ensembles. The right hand side is, by definition, $\mathcal{E}S^{\mathcal{O}}_k$.

Equation 4.6 is still rather awkward to argue directly, so we argue instead (Claim 6) that Equation 4.6 follows from

$$\mathcal{E}\dot{S}^{\bar{\bar{\mathcal{O}}}}_k(\vec{x},\vec{\pi},a_A,c) \quad \approx \quad \mathcal{E}\dot{S}^{\ddot{\mathcal{O}}}_k(\vec{x},\vec{\pi},a_A,c). \tag{4.7}$$

for oracles $\bar{\bar{\mathcal{O}}}$ and $\ddot{\mathcal{O}}$, which are very much like $\bar{\mathcal{O}}$ and $\dot{\mathcal{O}}$, respectively. The difference is that each chooses a random $j \in [1..n]$ and, if there is ever a component query of $j$, it "shuts up," refusing to answer any more questions. Equation 4.7 will then be argued by contradiction (Claim 7), where assuming its falsity allows the breaking the pseudorandom generator $G$.

Summary of strategy. Before proceeding, we observe that, taken together, Equations 4.3 and 4.6 imply that $P$ is $t$-private. The global strategy for arguing privacy, can be described as:

$$\begin{array}{llll} \mathcal{E}\hat{P}_k(\vec{x}\vec{r},\vec{\pi},a_A,\hat{c}) & \underset{\text{Def}}{\simeq} \mathcal{E}\hat{S}_k^{\hat{\mathcal{O}}}(\vec{x}\vec{r},\vec{\pi},a_A,\hat{c}) & \Longrightarrow \\ \mathcal{E}\bar{P}_k(\vec{x},\vec{\pi},a_A,c) & \underset{\text{Claim 4}}{\simeq} \mathcal{E}\bar{S}_k^{\bar{\mathcal{O}}}(\vec{x},\vec{\pi},a_A,c) & \Longrightarrow \\ \mathcal{E}P_k(\vec{x},\vec{\pi},a_A,c) & \underset{\text{Claim 5}}{\simeq} \mathcal{E}\dot{S}_k^{\bar{\mathcal{O}}}(\vec{x},\vec{\pi},a_A,c) & \underset{\text{Claims 6,7}}{\approx} \mathcal{E}\dot{S}_k^{\dot{\mathcal{O}}}(\vec{x},\vec{\pi},a_A,c) \underset{\text{Def}}{=} \mathcal{E}S_k^{\mathcal{O}}(\vec{x},\vec{\pi},a_A,c). \end{array}$$

Under each assertion is written the claim used to establish it. The "heart" of the proof—and the only arguments which really depends on the clever definition of the protocol $P$—is Claim 7, which begins on page 91. (On a first reading, it may be desirable to begin there.)

### Part 2: The simulator $S$ establishes security.

**Claim 1:** Evaluating $\hat{y}$ gives $y$

We show that evaluating a (real) garbled program $\hat{y}$ gives the value $y$ which it "should" evaluate to: for any $x'_T$, $r'_T$, $x_{\overline{T}}$ and $r_{\overline{T}}$, $y = \texttt{Eval}(\hat{c}, \hat{f}(x'_T r'_T \cup x_{\overline{T}} r_{\overline{T}}))$ is equal to $f(x'_T \cup x_{\overline{T}})$. The argument is by induction on the depth of the circuit $C$ defining $f$. We maintain the following invariant: *if, in the evaluation of $\hat{y} = \hat{f}(x'_T r'_T \cup x_{\overline{T}} r_{\overline{T}})$, signal $\sigma^\omega = s_b^\omega$ is held for some wire $\omega$, then wire $\omega$ carries the bit $b \oplus \lambda^\omega$ when $C$ is evaluated at $x'_T \cup x_{\overline{T}}$.* Here, $\lambda^\omega$ is defined from $r = r'_T \cup r_{\overline{T}}$ according to

$$\lambda^\omega = \begin{cases} \oplus_1^n \, r_i[2nW + \omega] & \text{for } \omega \in [1..W-l], \quad \text{and} \\ 0 & \text{for } \omega \in [W-l+1..W], \end{cases}$$

as with the equations of Steps 1a(i) and 1a(iv).

The invariant is seen to hold for input wires, since the signal issued for an input wire $\omega$ is $\sigma^\omega = s^\omega_{b^\omega \oplus \lambda^\omega}$, by Equation 3.2, and this wire carries a truth value of $b^\omega = (b^\omega \oplus \lambda^\omega) \oplus \lambda^\omega$ in $C$.

Inductively, consider a gate $g$ of functionality $\otimes$ having left incoming wire $\alpha$ (carrying a signal $\sigma^\alpha = s^\alpha_{\mu^\alpha}$), right incoming wire $\beta$ (carrying a signal $\sigma^\beta = s^\beta_{\mu^\beta}$), and out-going wire $\gamma$. By Equation 3.4, the signal computed for wire $\gamma$ will be

$$\sigma^\gamma \;=\; G_{\mu^\beta}(s^\alpha_{\mu^\alpha 1}) \oplus \cdots \oplus G_{\mu^\beta}(s^\alpha_{\mu^\alpha n}) \;\oplus\; G_{\mu^\alpha}(s^\beta_{\mu^\beta 1}) \oplus \cdots \oplus G_{\mu^\alpha}(s^\beta_{\mu^\beta n}) \;\oplus\; A^g_{\mu^\alpha \mu^\beta},$$

which, from the definition of $A^g_{\mu^\alpha \mu^\beta}$, Equation 3.3, is

$$\sigma^\gamma \;=\; s^\gamma_{[(\mu^\alpha \oplus \lambda^\alpha) \otimes (\mu^\beta \oplus \lambda^\beta)] \oplus \lambda^\gamma}$$

The inductive hypothesis says that the left incoming wire carries the bit $\mu^\alpha \oplus \lambda^\alpha$ in $C$, and the right incoming wire carries the bit $\mu^\beta \oplus \lambda^\beta$. So we know that out-going wire for gate $g$ of $C$ carries the bit $(\mu^\alpha \oplus \lambda^\alpha) \otimes (\mu^\beta \oplus \lambda^\beta)$. Consequently, the signal $\sigma^\gamma$ we have computed for wire $\gamma$ is $s^\gamma_{b \oplus \lambda^\gamma}$, where wire $\gamma$ carries the bit $b$ in circuit $C$. This establishes the induction.

By Equations 3.1 and 3.2, we conclude that $\mathtt{Eval}(\hat{y}) = (0 \oplus \lambda^{W-l+1}) \cdots (0 \oplus \lambda^W) = y$ $\square$

**Claim 2:** PROTOCOL $P$ IS CORRECT.

We next argue that protocol $P$ is correct. This follows from Claim 1 and definitions of $\mathcal{AI}$ and $\mathcal{AO}$.

The correctness of protocol $\hat{P}$ implies that, almost certainly, each player outputs $\hat{y} = \hat{f}(x'_T r'_T \cup x_{\overline{T}} r_{\overline{T}})$ as a result of the execution of Step 1, where $x'_T r'_T$ is the adversary's committal in $\hat{P}$, as specified by $\widehat{\mathcal{AI}}$.[3] In the protocol $P$, each player then computes his output $y$ by evaluating this garbled program $\hat{y}$. We need that the $y$-value so computed almost certainly equals $f(x'_T \cup x'_{\overline{T}})$, where $x'_T$ is the adversary's committal in protocol $P$. In fact, Claim 1 establishes that whenever each good player computes $\hat{y} = \hat{f}(x'_T r'_T \cup x_{\overline{T}} r_{\overline{T}})$, for *any* $r'_T$ and $r_{\overline{T}}$, each player computes $y = f(x'_T \cup x'_{\overline{T}})$. Thus, almost certainly, each player, when running $P$, outputs $y = f(x'_T \cup x_{\overline{T}})$, where $x'_T$ is given by $\mathcal{AI}$ and the output attributable to the adversary is given by $\mathcal{AO}$. This establishes correctness. $\square$

**Claim 3:** EVALUATING $\dot{y}$ GIVES $y$.

Very similar to Claim 1, we show that, in evaluating the *fake* garbled program $\dot{y}$, one comes to *hold* the signal $s^\omega_{\mu^\omega}$ for each wire $\omega$. (By the same reasoning, "hybrid" garbled programs—where some of the off-path gate labels are meaningful and others are not—will share this property.)

This holds by induction on the depth of the circuit $C$. For an input wire $\omega$, one holds $s^\omega_{\mu^\omega}$ by the definition of $\sigma^\omega$, Equation (4.4). Inductively, from holding $s^\alpha_{\mu^\alpha}$ for the left incoming wire of some gate $g$, and from holding $s^\beta_{\mu^\beta}$ for the right incoming wire of gate $g$, one computes and holds, according to Equation 3.3,

$$\sigma^\gamma \;=\; G_{\mu^\beta}(s^\alpha_{\mu^\alpha 1}) \oplus \cdots \oplus G_{\mu^\beta}(s^\alpha_{\mu^\alpha n}) \;\oplus\; G_{\mu^\alpha}(s^\beta_{\mu^\beta 1}) \oplus \cdots \oplus G_{\mu^\alpha}(s^\beta_{\mu^\beta n}) \;\oplus\; A^g_{\mu^\alpha \mu^\beta}$$

for the out-going wire $\gamma$ of gate $g$. From the definition of $A^g_{\mu^\alpha \mu^\beta}$, Equation (4.5), this is precisely $s^\gamma_{\mu^\gamma}$.

We conclude that $\mathtt{Eval}(\dot{y}) = \mu^{W-l+1} \cdots \mu^W$. By our choice of $\mu^\omega$ values on output wires, Equation (4.3), this is precisely the string $y$ returned from the oracle query $\mathcal{O}(x'_T)$, $y = f(x'_T \cup x_{\overline{T}})$. $\square$

**Claim 4:** EQUATION 4.2 IS CORRECT.

Let $A$ be a $t$-adversary. By the privacy of $\hat{P}$, $\mathcal{E}\hat{P}_k \simeq \mathcal{E}\hat{S}^{\hat{\mathcal{O}}}_k$, which is Equation 4.1. We wish to show that Equation 4.2 holds, that $\mathcal{E}\bar{P}_k \simeq \mathcal{E}\bar{S}^{\bar{\mathcal{O}}}_k$.

[3] That is, the good players *do* output the string $\hat{y}$ in Step 1, while the adversary *could* compute $\hat{y}$ by evaluating $\widehat{\mathcal{AO}}$.

First note that, for *any* distinguisher $\hat{D}^a$, for a negligible function $\epsilon(k')$,

$$\begin{aligned}
\epsilon(k') &= \max_{\substack{(\vec{x},\vec{\pi},a_A,c)\in L_{k'}(f)\\ \vec{r}\in(\Sigma^{\rho_c})^{n_c}}} \left|\mathbf{E}\hat{D}(\hat{P}_k(\vec{x}\vec{r},\vec{\pi},a_A,\hat{c})) - \mathbf{E}\hat{D}(\hat{S}_k^{\hat{\mathcal{O}}}(\vec{x}\vec{r},\vec{\pi},a_A,\hat{c}))\right| \\
&\geq \max_{(\vec{x},\vec{\pi},a_A,c)\in L_{k'}(f)} \left\{2^{-\rho_c n_c}\sum_{\vec{r}\in(\Sigma^{\rho_c})^{n_c}}\left|\mathbf{E}\hat{D}(\hat{P}_k(\vec{x}\vec{r},\vec{\pi},a_A,\hat{c})) - \mathbf{E}\hat{D}(\hat{S}_k^{\hat{\mathcal{O}}}(\vec{x}\vec{r},\vec{\pi},a_A,\hat{c}))\right|\right\} \\
&\geq \max_{(\vec{x},\vec{\pi},a_A,c)\in L_{k'}(f)} \Bigg| 2^{-\rho_c n_c}\sum_{\vec{r}\in(\Sigma^{\rho_c})^{n_c}}\mathbf{E}\hat{D}(\hat{P}_k(\vec{x}\vec{r},\vec{\pi},a_A,\hat{c})) \; - \\
&\qquad\qquad 2^{-\rho_c n_c}\sum_{\vec{r}\in(\Sigma^{\rho_c})^{n_c}}\mathbf{E}\hat{D}(\hat{S}_k^{\hat{\mathcal{O}}}(\vec{x}\vec{r},\vec{\pi},a_A,\hat{c}))\Bigg| . \qquad (4.8)
\end{aligned}$$

(Recall that $\hat{D}(E_k(\omega))$ is short for $\hat{D}(1^k, E_k(\omega), \omega, a)$, and $k = \min\{k', |c|^{1/10}\}$.) That the first quantity is negligible follows directly from $\hat{P}$'s privacy; the next inequality just says that the average of a bunch of elements can not exceed the maximum of these element; and the last line is the triangle inequality: $\sum|A_i - B_i| \geq |\sum(A_i - B_i)| = |\sum A_i - \sum B_i|$.

Now, given a distinguisher $\bar{D}^a$—think of this as having been designed for distinguishing $\mathcal{E}\bar{P}_k$ from $\mathcal{E}\bar{S}_k^{\bar{\mathcal{O}}}$—given any such distinguisher, there is a distinguisher $\hat{D}^a$ such that

$$\begin{aligned}
&\mathbf{E}\bar{D}(1^{k'}, \bar{P}_{k'}(\vec{x},\vec{\pi},a_A,c),(\vec{x},\vec{\pi},a_A,c),a) = \\
&\quad 2^{-\rho_c n_c}\sum_{\vec{r}\in(\Sigma^{\rho_c})^{n_c}}\mathbf{E}\hat{D}(1^k,\hat{P}_k(\vec{x}\vec{r},\vec{\pi},a_A,\hat{c}),(\vec{x}\vec{r},\vec{\pi},a_A,\hat{c}),a) \qquad (4.9)
\end{aligned}$$

and

$$\begin{aligned}
&\mathbf{E}\bar{D}(1^{k'}, \bar{S}_{k'}^{\bar{\mathcal{O}}}(\vec{x},\vec{\pi},a_A,c),(\vec{x},\vec{\pi},a_A,c),a) = \\
&\quad 2^{-\rho_c n_c}\sum_{\vec{r}\in(\Sigma^{\rho_c})^{n_c}}\mathbf{E}\hat{D}(1^k,\hat{S}_k^{\hat{\mathcal{O}}}(\vec{x}\vec{r},\vec{\pi},a_A,\hat{c}),(\vec{x}\vec{r},\vec{\pi},a_A,\hat{c}),a). \qquad (4.10)
\end{aligned}$$

Namely, the distinguisher $\bar{D}$ is defined as follows. It maps its first argument, $k$, to $k'$; it maps its third argument, $(\vec{x}\vec{r},\vec{\pi},a_A,\hat{c})$, to $(\vec{x},\vec{\pi},a_A,c)$; it maps its fourth argument to itself; and it maps it second argument, which is the view $\hat{\nu}$ of the adversary $A$, to an alternative view $\bar{\nu}$, as follows: when $\hat{\nu}$ indicates that a processor $i$ is corrupted, and processor $i$ had private input $x_i r_i$, the view $\bar{\nu}$ is constructed so that $x_i$ is the private input and $r_i$ is the prefix of $i$'s random string. That is, $\bar{\nu}$ is a syntactic modification of $\hat{\nu}$ so that it is a view that would appear to come from $\bar{P}$, not $\hat{P}$. After modifying the arguments in this way, the distinguisher $\hat{D}$ is applied to them. Equations 4.9 and 4.10 follow directly from the definitions of $\hat{P}, \bar{P}, \hat{S}, \bar{S}, \hat{D}$, and $\bar{D}$.

Combining Equations 4.8, 4.9, and 4.10, we have that

$$\epsilon(k') \;\geq\; \left|\mathbf{E}D(\bar{P}_{k'}(\vec{x},\vec{\pi},a_A,c)) - \mathbf{E}D(\bar{S}_{k'}^{\bar{\mathcal{O}}}(\vec{x},\vec{\pi},a_A,c))\right|,$$

which is precisely Equation 4.2. □

**Claim 5:** EQUATION 4.3 IS CORRECT.

The last claim easily gives Equation 4.3. To see this, note that if $\mathcal{E}E_k(\omega) \simeq \mathcal{E}E'_k(\omega)$ and $h$ is any function, then $\mathcal{E}h(E_k(\omega)) \simeq \mathcal{E}h(E'_k(\omega))$. (Likewise, if $\mathcal{E}E_k(\omega) \approx \mathcal{E}E'_k(\omega)$ and $h$ is efficiently computable, then $\mathcal{E}h(E_k(\omega)) \simeq \mathcal{E}h(E'_k(\omega))$.)

More explicitly, the only difference between $\bar{P}$ and $P$ is that $P$ evaluates the garbled circuit which has been collaboratively computed, outputting this evaluation, while $\bar{P}$ outputs the garbled program itself. Let $A$ be a $t$-adversary. By our definition of $P$ and $\bar{P}$,

$$P_k(\vec{x}, \vec{\pi}, a_A, c) \quad = \quad \mathtt{eval}(\bar{P}_k(\vec{x}, \vec{\pi}, a_A, c)), \tag{4.11}$$

where `eval` is the function on views that simply replaces the string encoding $A$'s view with the identical string, except that the garbled program $\bar{y}$ which the view $\bar{\nu}$ specifies to be output is, instead, evaluated, with the function `Eval`; it is this string which $\nu = \mathtt{eval}(\bar{\nu})$ specifies should be output. Likewise,

$$\dot{S}_k^{\bar{\mathcal{O}}}(\vec{x}, \vec{\pi}, a_A, c) \quad = \quad \mathtt{eval}(\bar{S}_k^{\bar{\mathcal{O}}}(\vec{x}, \vec{\pi}, a_A, c)), \tag{4.12}$$

as follows directly by the definition of $\dot{S}$ and $\bar{S}$. Applying `eval` to both sides of Equation 4.2 gives

$$\mathcal{E}\ \mathtt{eval}(\bar{P}_k(\vec{x}, \vec{\pi}, a_A, c)) \quad = \quad \mathcal{E}\ \mathtt{eval}(\bar{S}_k^{\bar{\mathcal{O}}}(\vec{x}, \vec{\pi}, a_A, c)).$$

Combining this with Equations 4.11 and 4.12 gives

$$\mathcal{E}P_k(\vec{x}, \vec{\pi}, a_A, c) \quad \simeq \quad \mathcal{E}\dot{S}_k^{\bar{\mathcal{O}}}(\vec{x}, \vec{\pi}, a_A, c),$$

as desired. □

◊ ◊ ◊

DEFINITIONS OF $\bar{\bar{\mathcal{O}}}$ AND $\ddot{\mathcal{O}}$. We are now ready to address the main concern, that the "fake" oracle $\dot{\mathcal{O}}$ provides a very close approximation to $\bar{\mathcal{O}}$—not in the information theoretic sense, as with all modifications made so far, but with respect to polynomial-time computation.

Actually, it will be convenient to argue this indirectly, showing that $\bar{\mathcal{O}}$ remains closely approximated by $\dot{\mathcal{O}}$ even if each of these oracles "shut up" if you ask some particular randomly-chosen component query. Specifically, define $\bar{\bar{\mathcal{O}}}$ and $\ddot{\mathcal{O}}$ as identical to $\bar{\mathcal{O}}$ and $\dot{\mathcal{O}}$, respectively, except that each oracle initially chooses a random $j \in [1..n]$ and then, to a component query of $j$, $\bar{\bar{\mathcal{O}}}$ and $\ddot{\mathcal{O}}$ answer *forbidden*—rather than $x_j r_j$, as they otherwise would. Once $\bar{\bar{\mathcal{O}}}$ or $\ddot{\mathcal{O}}$ answers *forbidden*, they answer $\Lambda$ on any subsequent queries. We now show that it suffices to show that $\bar{\bar{\mathcal{O}}}$ is well approximated by $\ddot{\mathcal{O}}$.

**Claim 6:** IF $\dot{S}^{\bar{\bar{\mathcal{O}}}} \approx \dot{S}^{\ddot{\mathcal{O}}}$, THEN $\dot{S}^{\bar{\mathcal{O}}} \approx \dot{S}^{\dot{\mathcal{O}}}$.

We may assume that $\hat{S}$ always makes exactly $t$ component queries, since a simulator which makes fewer queries than that can always be modified to make exactly $t$ component queries, ignoring the unneeded responses. Notice that $\dot{S}$ inherits this property from $\hat{S}$.

To establish Claim 6, suppose for contradiction that

$$\dot{S}_k^{\bar{\bar{\mathcal{O}}}} \quad \approx \quad \dot{S}_k^{\ddot{\mathcal{O}}}, \tag{4.13}$$

yet

$$\dot{S}_k^{\bar{\mathcal{O}}} \quad \not\approx \quad \dot{S}_k^{\dot{\mathcal{O}}}. \tag{4.14}$$

Equation 4.14 means that there exists a polynomial-time $t$-adversary $A$, a polynomial-time distinguisher $D^a$ and a sequence $\{(\vec{x}_k, \vec{\pi}_k, a_k, c_k) \in L_k(f)\}$ such that

$$\nu(k) = \left|\mathbf{E}D(\dot{S}_k^{\bar{\mathcal{O}}}(\vec{x}_k, \vec{\pi}_k, a_k, c_k)) - \mathbf{E}D(\dot{S}_k^{\dot{\mathcal{O}}}(\vec{x}_k, \vec{\pi}_k, a_k, c_k))\right|$$

is nonnegligible. Henceforth we omit the arguments to the simulators, writing an equation like the one above as simply

$$\nu(k) = \left|\mathbf{E}D(\dot{S}_k^{\bar{\mathcal{O}}}) - \mathbf{E}D(\dot{S}_k^{\dot{\mathcal{O}}}).\right|$$

We will contradict Equation 4.13. To do this, define a distinguisher $D'$, which is identical to $D$ except that $D$ outputs the bit 0 on views containing a distinguished *forbidden*-value. As mentioned on page 82, $\dot{S}$ provides $A$ with a distinguished *forbidden*-value when $\dot{S}$'s oracle responds *forbidden*. Then

$$\begin{aligned}
\mathbf{E}D'(\dot{S}^{\bar{\bar{\mathcal{O}}}}) &= \text{Prob}\left[\dot{S}_k^{\bar{\bar{\mathcal{O}}}} \text{ does } \textit{not} \text{ get a } \textit{forbidden}\right] \cdot \\
&\qquad \mathbf{E}\left[D(\dot{S}_k^{\bar{\bar{\mathcal{O}}}}) |\ \dot{S}_k^{\bar{\bar{\mathcal{O}}}} \text{does not get a } \textit{forbidden}\right] &(4.15)\\
&= \text{Prob}\left[j \leftarrow [1..n] : \dot{S}_k^{\bar{\mathcal{O}}} \text{ does not ask component query } j\right] \cdot \\
&\qquad \mathbf{E}D(\dot{S}_k^{\bar{\mathcal{O}}}) &(4.16)\\
&= (1 - t/n)\ \mathbf{E}D(\dot{S}_k^{\bar{\mathcal{O}}}), &(4.17)
\end{aligned}$$

where Equation 4.15 holds by our extension of $D'_k$ to 0 on *forbidden*, and

$$\mathbf{E}\left[D(\dot{S}_k^{\bar{\bar{\mathcal{O}}}}) |\ \dot{S}_k^{\bar{\bar{\mathcal{O}}}} \text{does not get a } \textit{forbidden}\right] = \mathbf{E}D(\dot{S}_k^{\bar{\mathcal{O}}})$$

follows because of $\dot{S}_k$ making exactly $t$ component queries. Similarly, we have

$$\begin{aligned}
\mathbf{E}D'(\dot{S}_k^{\ddot{\mathcal{O}}}) &= \text{Prob}\left[\dot{S}_k^{\ddot{\mathcal{O}}} \text{ does } \textit{not} \text{ get a } \textit{forbidden}\right] \cdot \\
&\qquad \mathbf{E}\left[D(\dot{S}_k^{\ddot{\mathcal{O}}}) | \dot{S}_k^{\ddot{\mathcal{O}}} \text{does not get a } \textit{forbidden}\right] \\
&= \text{Prob}\left[j \leftarrow [1..n] : \dot{S}_k^{\dot{\mathcal{O}}} \text{ does not ask component query } j\right] \cdot \\
&\qquad \mathbf{E}D(\dot{S}_k^{\dot{\mathcal{O}}}) \\
&= (1 - t/n)\ \mathbf{E}D(\dot{S}_k^{\bar{\mathcal{O}}}).
\end{aligned}$$

Thus

$$\begin{aligned}\left|\mathbf{E}D'(\dot{S}_k^{\bar{\bar{\mathcal{O}}}}) - \mathbf{E}D'(\dot{S}_k^{\ddot{\mathcal{O}}})\right| &= (1 - t/n)\ \nu(k) \\ &\geq \nu(k)/2\end{aligned}$$

is nonnegligible, contradicting Equation 4.13 and establishing Claim 6. □

**Claim 7:** $\dot{S}^{\bar{\bar{\mathcal{O}}}} \approx \dot{S}^{\ddot{\mathcal{O}}}$.

Finally, we address the heart of the matter. Proceeding by contradiction, assume that

$$\dot{S}^{\bar{\bar{\mathcal{O}}}} \quad \not\approx \quad \dot{S}^{\ddot{\mathcal{O}}}. \tag{4.18}$$

This means that there is a polynomial-time $t$-adversary $A$, a polynomial-time distinguisher $D^a$, a sequence $\{(\vec{x}_k, \vec{\pi}_k, (a_A)_k, c_k) \in L_k(f)\}$, a polynomial $Q(k)$, and an infinite set $K \subseteq \mathbf{N}$ such that

$$\left|\mathbf{E}D(S_k^{\bar{\bar{\mathcal{O}}}}) - \mathbf{E}D(S_k^{\ddot{\mathcal{O}}})\right| > \frac{1}{Q(k)} \tag{4.19}$$

for $k \in K$.

We will use this to construct a probabilistic polynomial-time distinguisher $(D')^{a'}$ for distinguishing the ensembles

$$\mathcal{ER}_k \;=\; \mathcal{E}(U_{2\bar{n}k+2})^2 \qquad \text{and} \qquad \mathcal{EPR}_k \;=\; \mathcal{E}(G(U_k))^2,$$

where $G : \Sigma^k \to \Sigma^{2(\bar{n}k+1)}$ is the pseudorandom generator used by protocol $P$. Recall that $\bar{n} = k^{10}$ is a bound on the size of $n$ in terms of the associated security parameter $k$. The existence of such a distinguisher contradicts Lemma 4.2.3.

To describe $D'$ and $a'$, fix $k \in K$ and let $\vec{x} = \vec{x}_k$, $\vec{\pi} = \vec{\pi}_k$, $a_A = (a_A)_k$, and $c = c_k$. Let $n, \ell, l, m, C, \Gamma$, and $\rho$ all be as indicated by $c$.

The distinguisher $D'$ takes as inputs the two strings, $A', B' \in \Sigma^{2\bar{n}k+2}$. It uses a substring of each of these strings to compute a "good guess" as to whether $A$ and $B$ are random or pseudorandom. Let $A_0 = A'[1 : nk+1]$, $A_1 = A'[\bar{n}k+2 : \bar{n}k+nk+2]$, $B_0 = B'[1 : nk+1]$, and $B_1 = B'[\bar{n}k+2 : \bar{n}k+nk+2]$.

Definition of the oracle $\mathcal{O}_i$. We define a sequence of oracles, $\mathcal{O}_0, \mathcal{O}_1, \ldots, \mathcal{O}_\Gamma$, the first oracle will be identical to $\ddot{\mathcal{O}}$, and each oracle will be successively more and more like $\bar{\bar{\mathcal{O}}}$, until $\mathcal{O}_\Gamma = \bar{\bar{\mathcal{O}}}$. This might be depicted as

$$\ddot{\mathcal{O}} = \mathcal{O}_0 \Rightarrow \mathcal{O}_1 \Rightarrow \cdots \Rightarrow \mathcal{O}_{\Gamma-1} \Rightarrow \mathcal{O}_\Gamma = \bar{\bar{\mathcal{O}}}.$$

Passing from oracle $\mathcal{O}_{g-1}$ to oracle $\mathcal{O}_g$ can be thought of as "fixing up" the gate labels on gate $g$. That is, recall that $\ddot{\mathcal{O}}$ provides "meaningless" (random) gate labels on three of the four gate labels per gate, while $\bar{\bar{\mathcal{O}}}$ provides "correct" gate labels throughout. The intermediate oracles have meaningful gate labels on more and more gates.

The behavior of $\mathcal{O}_i$ is as follows. The oracle begins by selecting a random $\vec{r} \in (\Sigma^\rho)^n$ and a random $j \in [1..n]$. To a component query of $j$, $\mathcal{O}_i$ responds with *forbidden*, and it responds with $\Lambda$ to all queries following a *forbidden* response. Otherwise, to a component query of $\iota$ $\mathcal{O}_i$ responds with $x_\iota r_\iota$.

We must describe how $\mathcal{O}_i$ answers an output query of $x'_T r'_T$, returning some string

$$\check{y} = A^1_{00}A^1_{01}A^1_{10}A^1_{11}\cdots A^\Gamma_{00}A^\Gamma_{01}A^\Gamma_{10}A^\Gamma_{11}\;\sigma^1\cdots\sigma^{n\ell},$$

which we now define.

Exactly paralleling the definitions in the protocol and in the description of $\dot{\mathcal{O}}$, define $\vec{r}'' = r_T \cup r'_{\overline{T}}$, and set

$$s^1_{0i}s^1_{1i}\cdots s^W_{0i}\,s^W_{1i}\;\lambda^1_i\cdots\lambda^{W-l}_i = r''_i,$$

for $i \in [1..n]$, $|s^\omega_{bi}| = k$. For $\omega \in [1..W]$, declare

$$\lambda^\omega = \begin{cases} \lambda^\omega_1 \oplus \cdots \oplus \lambda^\omega_n & \text{if } \omega \in [1..W-l], \quad \text{and} \\ 0 & \text{if } \omega \in [W-l+1..W], \end{cases}$$

and set $s^\omega_b = s^\omega_{b1}\cdots s^\omega_{bn}\,b$. Let $b^\omega$ be the value carried by wire $\omega$ of $C$ when $C$ is evaluated at $x'_T \cup x_{\overline{T}}$, and define $\mu^\omega = \lambda^\omega \oplus b^\omega$.

Now to answer the output query, set $\sigma^\omega = s^\omega_{\mu^\omega}$, for $\omega \in [1..n\ell]$. This defines the garbled input. To define the gate labels, for a gate $g$ of functionality $\otimes$, left incoming wire $\alpha$, right incoming wire $\beta$, and out-going wire $\gamma$, define $A^g_{ab}$ according to

$$A^g_{ab} = \begin{cases} G_b(s^\alpha_{a1}) \oplus \cdots \oplus G_b(s^\alpha_{an}) \oplus \\ G_a(s^\beta_{b1}) \oplus \cdots \oplus G_a(s^\beta_{bn}) \oplus \\ s^\gamma_{[(a\oplus\lambda^\alpha)\otimes(b\oplus\lambda^\beta)]\oplus\lambda^\gamma} & \text{if } g \in [1..i] \text{ or } ab = \mu^\alpha\mu^\beta \\ \\ R^g_{ab} & \text{otherwise} \end{cases} \tag{4.20}$$

where $R^g_{ab}$ is a randomly selected string from $\Sigma^{nk+1}$.

Informally, we have put in correct gate-labels for *all* gates numbered $1, \ldots, i$, and we have also put in correct *on-path* gate labels for the remaining gates. The other $3(\Gamma - i)$ gate-labels are garbage—just random strings.

**Claim 8:** $\bar{\bar{\mathcal{O}}} = \mathcal{O}_\Gamma$.

This claim means that $\bar{\bar{\mathcal{O}}}$ and $\mathcal{O}_\Gamma$ exhibit exactly the same (probabilistic) behavior when used as ideal evaluation oracles which are asked at most $t$ component queries.

This statement is immediate from the definitions of of $\bar{\bar{\mathcal{O}}}$ and $\mathcal{O}_\Gamma$. Both oracles choose a random $j \in [1..n]$, and answer *forbidden* to a component query of $j$, answering $\Lambda$ subsequently. Apart from that, both oracles return $x_i r_i$ (for a random $r_i$) to a component query of $i$. To an output query of $x'_T r'_T$, both oracles return $\hat{f}(x'_T r'_T \cup x_{\overline{T}} r_{\overline{T}})$. $\square$

**Claim 9:** $\ddot{\mathcal{O}} \;=\; \mathcal{O}_0$.

This claim means that $\bar{\bar{\mathcal{O}}}$ and $\mathcal{O}_0$ exhibit exactly the same (probabilistic) behavior when used as ideal evaluation oracles which are asked at most $t$ component queries.

As before, both oracles choose a random $j \in [1..n]$, and answer *forbidden* to a component query of $j$, answering $\Lambda$ subsequently. Both oracles then choose a random $\vec{r}$, and, to a component query of $i$, each answers $x_i r_i$.

To an output query of $x'_T r'_T$, the oracle $\ddot{\mathcal{O}}$ chooses a random path $\mu^1, \ldots, \mu^{W-l}$ and returns a random fake garbled program $\dot{y}$ that computes $f(x'_T \cup x_{\overline{T}})$ along this path, having random off-path gate labels. On the other hand, $\mathcal{O}_i$ explicitly chooses no such random path; the path is implicitly specified by the $\vec{r}''$-values.

Still, the path chosen, being a function of the randomly selected $\oplus_1^n r_i$-values, is independent of the values returned by the $t < n$ component queries. Furthermore, the path does not depend on the output query itself. Thus both oracle $\ddot{\mathcal{O}}$ and $\mathcal{O}_0$ answer an output query $x'_T r'_T$ by a random path computing $f(x'_T \cup x_{\overline{T}})$, with $U_{nk+1}$-distributed off-path gate labels. This establishes Claim 9 □

**Claim 10:** THERE IS A $g_k \in [1..\Gamma]$ SUCH THAT $\left|\mathbf{E}D(\dot{S}_k^{\mathcal{O}_{g_k-1}}) - \mathbf{E}D(\dot{S}_k^{\mathcal{O}_{g_k}})\right| \geq 1 \,/\, \Gamma Q(k)$.

The claim follows immediately from Claims 8, 9, Equation 4.19, and the triangle inequality. In common parlance, we have taken a "probability walk" through the oracles $\mathcal{O}_0 \Rightarrow \mathcal{O}_1 \Rightarrow \cdots \Rightarrow \mathcal{O}_{\Gamma-1} \Rightarrow \mathcal{O}_\Gamma$. The distinguishing probabilities differ at the two endpoints; somewhere, there must be a nonnegligible "jump" in the distinguishing probabilities. We have let $g_k$ denote a place in which there is a jump between the behavior induced by oracles $\mathcal{O}_{g_k-1}$ and $\mathcal{O}_{g_k}$. The same proof technique was used in the proof of Proposition 4.2.2. □

◊ ◊ ◊

Before proceeding, we make the following definition: $\alpha_k$, $\beta_k$, and $\gamma_k$ denote the left incoming wire, right incoming wire, and out-going wire to gate $g_k$, respectively.

THE ORACLE $\mathcal{O}_*$.[4] The distinguisher $D'$, recall, takes a string $A_0 A_1 B_0 B_1 \in \Sigma^{4(nk+1)}$ (extracted from $A'B'$). Consider the oracle $\mathcal{O}_*$, whose behavior depends on such a string $A_0 A_1 B_0 B_1$, defined to behave as follows: it flips coins to determine the values $j$, $\vec{r}$, and $R^g_{ab}$, exactly as $\mathcal{O}_{g_k}$ did before. Component queries are answered as $\mathcal{O}_{g_k}$ would answer them. Then, to an output query of $x'_T r'_T$, the values $\vec{r}''$, $s^\omega_{bi}$, $\lambda^\omega_i$, $\lambda^\omega$, $s^\omega_b$, $b^\omega$, $\mu^\omega$ and $\sigma^\omega$ are all computed as $\mathcal{O}_{g_k}$ would compute these values. However, in forming the garbled program $y_*$, the $A^g_{ab}$-values are computed differently than they were for $\mathcal{O}_{g_k}$—but just for the case of

[4] Introduced for the benefit of readers who believe there have not been enough oracles described in this proof.

gate $g_k$. Namely, set

$$A^g_{ab} = \begin{cases} \mathtt{G}_b(s^\alpha_{a1})\oplus\cdots\oplus\mathtt{G}_b(s^\alpha_{an})\oplus \\ \mathtt{G}_a(s^\beta_{b1})\oplus\cdots\oplus\mathtt{G}_a(s^\beta_{bn})\oplus \\ s^\gamma_{[(a\oplus\lambda^\alpha)\otimes(b\oplus\lambda^\beta)]\oplus\lambda^\gamma} & \text{if } g\in[1..g_k] \text{ or } ab=\mu^\alpha\mu^\beta \\ \\ R^g_{ab} & \text{otherwise} \end{cases} \tag{4.21}$$

where

$$\mathtt{G}_\beta(s^\omega_{\delta i}) \quad = \quad \begin{cases} A_\beta & \text{if } \omega=\alpha_k \text{ and } i=j \text{ and } \delta=\overline{\mu^{\alpha_k}} \\ B_\beta & \text{if } \omega=\beta_k \text{ and } i=j \text{ and } \delta=\overline{\mu^{\beta_k}} \\ G_\beta(s^\omega_{\delta i}) & \text{otherwise} \end{cases}$$

for $\beta,\delta\in\{0,1\}$. We can describe the preceding as follows. Player $j$'s contribution to the left wire feeding gate $g_k$ is $A_0A_1$ for the (supposed) generator output corresponding to the off-path signal. For the right wire, his contribution is $B_0B_1$ for the (supposed) generator output corresponding to the off-path signal. Now if $A_0A_1$ and $B_0B_1$ really *are* pseudorandom, this is what player $j$ *should* contribute for these wires, and we will compute a meaningful set of off-path gate labels for this gate. But if $A_0A_1$ and $B_0B_1$ are truly random, then this contribution will cause *all* of the off-path gate labels to be random. We formalize this as follows:

**Claim 11:** If $A'$ and $B'$ are $G(U_k)$-distributed, then $\mathcal{O}^{A_0A_1B_0B_1}_* = \mathcal{O}_{g_k}$.
**Claim 12:** If $A'$ and $B'$ are $U_{(2\bar{n}k+2)}$-distributed, then $\mathcal{O}^{A_0A_1B_0B_1}_* = \mathcal{O}_{g_k-1}$.
The first of the these two claims is immediate. The only concern is that a component query would necessitate revealing a preimage under the generator of an $A$ or $B$-value. We don't have any such values to reveal. Fortunately, we have arranged that a component query of $j$ need only return a value of the response *forbidden*.

The second of the these claims is just a little more complicated. The behavior of $\mathcal{O}_*$ and $\mathcal{O}_{g_k-1}$ potentially differ only by the gate labels associated to gate $g_k$. In $\mathcal{O}_{g_k-1}$, random strings are used for the three off-path gate labels. In $\mathcal{O}_*$, on the other hand, the three off-path gate labels are produced according to Equation 4.21. The calculation of these gate labels can be viewed as XOR-ing three $\Sigma^{nk+1}$-valued random variables, $X_1$, $X_2$, and $X_3$, with random variables $A_0$, $B_0$, and $A_1$, respectively, where $X_0$, $X_1$, and $X_2$ are independent of $A_0$, $B_0$, and $A_1$, respectively. Thus, when $A'$ and $B'$ are $U_{2\bar{n}k+2}$-distributed, the three off-path gate-labels are $U_{nk+1}$-distributed. □

Wrapping up. We now complete the proof of Claim 7. Observe that, given $\vec{x},\vec{\pi},a_A,c$ and $g_k$, and given $A$ and $B$, it is easy to compute a sample from $\dot{S}^{\mathcal{O}_*}$ in time polynomial in $|c|$. The distinguisher $D'$, using advice which encodes both the (infinite collection) of strings $a'=\{(\vec{x}_k,\vec{\pi}_k,(a_A)_k,c_k,g_k)\}$ and the infinite string $a$ in a reasonable manner, and using $A$

and $B$, works as follows: it computes a sample $\nu$ from $\dot{S}^{\mathcal{O}_*^{AB}}$ and then computes the bit $D(1^k, \nu, (\vec{x}_k, \vec{\pi}_k, (a_A)_k, c_k), a)$. It output this bit. We know that

$$\left|\mathbf{E}D'(\mathcal{R}_k) - \mathbf{E}D'(\mathcal{PR}_k)\right| \geq \frac{1}{\Gamma_k Q(k)} \geq \frac{1}{k^{10}Q(k)}$$

for all $k \in K$. This contradicts the assumption given by Equation 4.18, and so have established Claim 7, as well as the Theorem 4.3.1. □

■

## 4.4 Postmortem

We begin with a few comments about the proof of Theorem 4.3.1.

Requirement on honest majority. The assumption that $t < n/2$ was only required because Theorem 4.1.1 requires this; a garbled program reveals no useful information as long as $t < n$. Indeed, the proof we have given remains unmodified to establish this claim, apart from relaxing the bound in the conclusion of Claim 6 to "$\geq \nu(k)/n \geq \nu(k)/k^{10}$."

Generality of techniques. Some of the techniques used in the proof of Theorem 4.3.1 are applicable to other simulator arguments. One of these methods is considering the probabilistic analog of deterministic protocols and simulators, as we did in passing from $\hat{P}$ to $\bar{P}$ and from $\hat{\mathcal{O}}$ to $\bar{\mathcal{O}}$. Another is considering oracles which "shut up" when asked oracle queries they find inconvenient to answer. The high-level method of replacing one oracle by another—the "oracle substitution argument" we have employed—is likely to be generally applicable.

Vector-valued secure computation. For simplicity, we have described the protocol for string-valued computation. There is no difficulty extending the protocol to vector valued computation, and there is no difficulty carrying the proof over to this case, as well.

To collaboratively compute a vector-valued function $f : (\Sigma^\ell)^n \to (\Sigma^l)^n$, the players compute, instead, a string-valued function $f_* : (\Sigma^{\ell+l})^n \to \Sigma^{\ell l}$, defined by $y_* = f_*(\vec{x}\vec{r}) = (r_1 \oplus f_1(\vec{x})) \cdots (r_n \oplus f_n(\vec{x}))$. To compute $f$, each player is instructed to choose a random $r_i \in \Sigma^l$ and then run the protocol for securely computing $f_*$. Then, each player $i$ outputs $y_i = r_i \oplus y_*[(i-1)l + 1 : il]$. Using this method, the following extension of Theorem 4.3.1 can be obtained:

**Theorem 4.4.1 (Main theorem—vector-valued computation)** *Assume a one-way function exists, and let $f = \{f_c\}$ be a vector-valued function family in which each $f_c : (\Sigma^{\ell_c})^{n_c} \to (\Sigma^{l_c})^{n_c}$ is described by a circuit $C_c$ for computing it. Then, for some absolute constant $R$, there is a polynomial-time, $R$-round protocol $P$ that $t$-securely computes $f$, where $t = \lfloor (n-1)/2 \rfloor$.* ■

# Bibliography


[BB89] J. Bar-Ilan and D. Beaver, "Non-Cryptographic Fault Tolerant Computing in a Constant Number of Rounds of Interaction," *Proc. of the 8th PODC* (1989), 201-209.

[Ba86] D. Barrington, "Bounded-Width Polynomial-Size Branching Programs Recognize Exactly those Languages in $NC^1$," *Proc. of the 18th STOC* (1986), 1–5.

[Be88a] D. Beaver, "Secure Multiparty Protocols Tolerating Half Faulty Processors," CRYPTO-89 Proceedings, Springer-Verlag. To appear in *J. of Cryptology.*

[Be88b] D. Beaver, "Distributed Non-Cryptographic Oblivious Transfer with Constant Rounds Secret Function Evaluation," Harvard University Technical Report TR-13-88.

[Be91] 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.)

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

[BFKR90] D. Beaver, J. Feigenbaum, J. Kilian and P. Rogaway, "Security with Low Communication Overhead," CRYPTO-90 Proceedings, Springer-Verlag.

[BMR90] D. Beaver, S. Micali and P. Rogaway, "The Round Complexity of Secure Protocols," *Proc. of the 22nd STOC* (1990), 503–513,

[Be87a] J. Benaloh, "Secret Sharing Homomorphisms: Keeping Shares of a Secret Sharing," CRYPTO-86 Proceedings, Springer-Verlag.

[Be87b] J. Benaloh, "Verifiable Secret-Ballot Elections," Yale University Ph.D. Thesis (1987), TR-561.

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

[BL85] M. Ben-Or and N. Linial, "Collective Coin Flipping, Robust Voting Schemes, and Minima of Banzhaf Values," *Proc of the 26th FOCS*.

[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.

[BS81] G. Blakley and R. Swanson, "Security Proofs for Information Protection Systems," *Proc. of IEEE Symposium on Security and Privacy* (1981), 75–81.

[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*, 13(4):850–864, 1984. Earlier version in *Proc. of the 23rd FOCS* (1982).

[BD84] A. Broder and D. Dolev, "Flipping Coins in Many Pockets," *Proc of the 25th FOCS* (1984).

[BH88] R. Boppana and R. Hirschfeld, "Pseudorandom Generators and Complexity Classes," *Advances in Computing Research*, Vol. 5, *Randomness in Computation*, Silvio Micali, editor. JAI Press, 1989, 1–26.

[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, Springer-Verlag, 87–119.

[CK91] B. Chor and E. Kushilevitz, "A Zero-One Law for Boolean Privacy," *SIAM Journal on Discrete Math*, February 1991. Earlier version in *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, S. Goldwasser, S. Micali and B. Awerbuch, "Verifiable Secret Sharing and Achieving Simultaneity in the Presence of Faults," *Proc. of the 26th FOCS* (1985), 383–395.

[CR87] B. Chor and M. Rabin, "Achieving Independence in Logarithmic Number of Rounds," *Proc. of the 6th PODC* (1987).

[Cl85] R. Cleve, "Limits on the Security of Coin Flips When Half the Processors are Faulty," *Proc. of the 18th STOC* (1986) 364–369.

[Co85] S. Cook, "A Taxonomy of Problems with Fast Parallel Algorithms," *Information and Control*, **64** (1985) 2–22.

[Cr90] C. Crépeau, "Correct and Private Reductions Among Oblivious Transfers," MIT Ph.D. Thesis, 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):644–654, (November 1976.

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

[FM88a] P. Feldman and S. Micali, "Optimal Algorithms for Byzantine Agreement," *Proc. of the 20th STOC* (1988), 148–161.

[FM88b] P. Feldman and S. Micali, "One Can Always Assume Private Channels," unpublished manuscript (1988).

[GHY87] Z. Galil, S. Haber and M. Yung, "Cryptographic Computation: Secure Fault-Tolerant Protocols and the Public-Key Model," CRYPTO-87 Proceedings, Springer-Verlag, 135–155.

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

[GGM86] O. Goldreich, S. Goldwasser and S. Micali, "How to Construct Random Functions," *J. of the ACM*, 33(4):792–807, 1986.

[GKL88] O. Goldreich, H. Krawczyk, and M. Luby, "On the Existence of Pseudorandom Generators," *Proc. of the 29th FOCS* (1988), 12–24.

[GL89] O. Goldreich and L. Levin, "A Hard-Core Predicate for all One-Way Functions," *Proc. of the 21st STOC* (1989), 25–32.

[GMW86] O. Goldreich, S. Micali and A. Wigderson, "Proofs that Yield Nothing but their Validity and a Methodology of Cryptographic Protocol Design," *Proc. of the 27th FOCS* (1986), 174–187.

[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, Springer-Verlag, 76–86.

[GL90] S. Goldwasser and L. Levin, "Fair Computation of General Functions in Presence of Immoral Majority," CRYPTO-90 Proceedings, Springer-Verlag, 75–84.

[GM84] S. Goldwasser and S. Micali, "Probabilistic Encryption," *Journal of Computer and System Sciences,* 28(2):270–299, 1984.

[GMR85] S. Goldwasser, S. Micali, and C. Rackoff, "The Knowledge Complexity of Interactive Proof Systems," *SIAM Journal on Computing,* 18(2):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.

[GMT82] S. Goldwasser, S. Micali, and P. Tong, "Why and How to Establish a Private Code on a Public Network," *Proc. of the 23rd FOCS*, (1982), 134–144.

[GMY83] S. Goldwasser, S. Micali, and A. Yao, "Strong Signature Schemes," *Proc. of the 15th STOC* (1983), 431–439.

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

[Ha90] J. Håstad, "Pseudo-Random Generators under Uniform Assumptions," *Proc. of the 22nd STOC* (1990), 395–404.

[ILL89] R. Impagliazzo, L. Levin and M. Luby, "Pseudo-random generation from one-way functions," *Proc. of the 21st STOC* (1989), 12–23.

[Ki88] J. Kilian, "Founding Cryptography on Oblivious Transfer," *Proc. of the 20th STOC* (1988), 20–29.

[Ki89] J. Kilian, "Uses of Randomness in Algorithms and Protocols," MIT Ph.D. Thesis (1989).

[Ki91] J. Kilian, "General Completeness Theorems for 2-Party Games," *Proc. of the 23rd FOCS* (1991).

[KMR90] J. Kilian, S. Micali and P. Rogaway, "The Notion of Secure Computation," unpublished manuscript, 1990.

[Le85] L. Levin, "One-Way Functions and Pseudorandom Generators," *Combinatorica,* 17:357–363, 1988. 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.

[MR91] S. Micali and P. Rogaway, "Secure Computation," in preparation, 1991.

[MRS88] S. Micali, C. Rackoff and B. Sloan, "The Notion of Security for Probabilistic Cryptosystems," *SIAM J. of Computing*, 17(2):412–26, 1988.

[NIV] Unattributed, *Ecclesiastes* (New International Version, date of original unknown), International Bible Society (1977).

[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* 27(2), 1980.

[Ra81] M. Rabin, "How to Exchange Secrets by Oblivious Transfer," Technical Memo TR-81, Aiken Computation Laboratory, Harvard University, 1981.

[Ra88] T. Rabin, "Robust Sharing of Secrets when the Dealer is Honest or Cheating," Hebrew University Master's Thesis. Also described in [RB89].

[RB89] T. Rabin and M. Ben-Or, "Verifiable Secret Sharing and Multiparty Protocols with Honest Majority," *Proc. of the 21st STOC* (1989), 73–85.

[RSA78] R. Rivest, A. Shamir and L. Adleman, "A Method for Obtaining Digital Signatures and Public Key Cryptosystems," *Communication of the ACM,* 21(2):120–126, February 1978.

[Sh79] A. Shamir, "How to Share a Secret," *Communication of the ACM,* 22(11):612–613, November 1979.

[SRA81] A. Shamir, R. Rivest, and L. Adleman, "Mental Poker," in *Mathematical Gardener,* D. D. Klarner, editor, Wadsworth International (1981) 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.

[Ya79] A. Yao, "Some Complexity Questions Related to Distributed Computing," *Proc. of the 11th STOC* (1979), 209–213.

[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).