# Secure Multiparty Computation and Secret Sharing - An Information Theoretic Appoach

Ronald Cramer
Ivan Damgård
Jesper Buus Nielsen

Book Draft

December 31, 2012

# Contents

# Preface

This is a book on information theoretically secure Multiparty Computation (MPC) and Secret-Sharing, and about the intimate and fascinating relationship between the two notions.

We decided to write the book because we felt that a comprehensive treatment of unconditionally secure techniques for MPC was missing in the literature. In particular, because some of the first general protocols were found before appropriate definitions of security had crystallized, proofs of those basic solutions have been missing so far.

We present the basic feasibility results for unconditionally secure MPC from the late 80ties, generalizations to arbitrary access structures using linear secret sharing, and a selection of more recent techniques for efficiency improvements. We also present our own variant of the UC framework in order to be able to give complete and modular proofs for the protocols we present.

We also present a general treatment of the theory of secret-sharing and in particular, we focus on asymptotic results for multiplicative secret sharing which has various interesting applications to MPC that we present in the final chapter.

Our ambition has been to create a book that will be of interest to both computer scientists and mathematicians, and can be used for teaching at several different levels. Therefore, there are several different ways to read the book, and we give a few suggestions for this below.

The book is intended to be self-contained enough to be read by advanced undergraduate students, and the authors have used large parts of the material in the book for teaching courses at this level. By covering a selection of more advanced material, the book can also be used for a graduate course.

## How to Use this Book

For a course on advanced undergrad level for computer science students, we recommend to cover chapters 1-5. This will include the basic feasibility results for unconditionally secure MPC and the UC model. For some extra perspective it may also be a good idea to cover chapter 7 which is basically a survey of cryptographically secure solutions.

For a graduate level computer science course, we recommend to include also chapters 6 and 8 as they contains several recent techniques an surveys some open problems.

For a course in mathematics on secret-sharing and applications, we recommend to cover first chapters 1, 3 and 6. This will give an intuition for what secret-sharing is and how it is used in MPC. Then chapter 10 should be covered to present the general theory of secret sharing. Finally chapter 11 can be used to present some of the more advanced applications.

## Acknowledgements

We thank Morten Dahl Jørgensen for doing an enormous amount of work in proof reading and commenting. We also thank Sarah Zakarias for writing some of the simulation proofs and helping us avoid many more blunders than are present now.

# Chapter 1

# Introduction

## Contents

## 1.1 Private Information, Uses and Misuses

In a modern information-driven society, the everyday life of individuals and companies is full of cases where various kinds of private information is an important resource. While a cryptographer might think of PIN-codes and keys in this context, this type of secrets is not our main concern here. Rather, we will talk about information that is closer to the "primary business" of an individual or a company. For a private person, this may be data concerning his economic situation, such as income, loans, tax data, or information about health, such as deceases, medicine usage etc. For a company it might be the customer database, or information on how the business is running, such as turnover, profit, salaries, etc.

What is a viable strategy for handling private information? Finding a good answer to this question has become more complicated in recent years. When computers were in their infancy, in the 1950's and 1960's, electronic information security was to a large extent a military business. A military organization is quite special in that confidential information needs to be communicated almost exclusively between its own members, and the primary security goal is to protect this communication from being leaked to external enemies. While it may not be trivial to reach this goal, at least the overall purpose is quite simple to phrase and understand.

In modern society, things get much more complicated: using electronic media, we need to interact and do business with a large number of parties, some of whom we never met, and where many of them may have interests that conflict with ours. So how do you handle your confidential data if you cannot be sure that the parties you interact with are trustworthy?

Of course, one could save the sensitive data in a very secure location and never access it, but this is of course unreasonable. Our private data usually has value only because we want to use them for something. In other words, we have to have ways of controlling leakage of confidential data while this data is being stored, communicated or computed on, *even in cases where the owner of the data does not trust the parties (s)he communicates with.*

A very interesting aspect that makes this problem even more important is that there are many scenarios where a large amount of added value can be obtained by *combining confidential information from several sources*, and from this compute some result that all parties are interested in. To illustrate what is meant by this, we look at a number of different example scenarios in the following subsections.

### 1.1.1 Auctions

Auctions exist in many variants and are used for all kinds of purposes, but we concentrate here on the simple variant where some item is for sale, and where the highest bid wins. We assume the auction is conducted in the usual way, where the price starts at some preset amount and people place increasing bids until no one wants to bid more than the holder of the currently highest bid. When you enter such an auction, you usually have some (more or less precisely defined) idea of the maximal amount you are willing to pay, and therefore when you will stop bidding. On the other hand, every bidder of course wants to pay as small a price as possible for the item.

Indeed, the winner of the auction may hope to pay less than his maximal amount. This will happen if all other bidders stop participating long before the current bid reaches this maximum.

For such an auction to work in a fair way, it is obvious that the maximum amount you are willing to pay should be kept private. For instance, if the auctioneer knows your maximum and is working with another bidder, they can force the price to be always just below your maximum and so force you to pay more than if the auction had been honest. Note that the auctioneer has an incentive to do this to increase his own income, which is often a percentage of the price the item is sold for.

On the other hand, the result of the auction could in principle be computed, if one was given as input the true maximum value each bidder assigns to the item on sale.

### 1.1.2 Procurement

A procurement system is a sort of inverted auction, where some party (typically a public institution) asks companies to bid for a contract, i.e., to make an offer on the price for doing a certain job. In such a case, the lowest bid usually wins. But on the other hand, bidders are typically interested in getting as large a price as possible.

It is obvious that bids are private information: a participating company is clearly not interested in the competitors learning its bid before they have to make theirs. This would allow them to beat your bid by always offering a price that is slightly lower that yours. This is also against the interests of the institution offering the contract because it will tend to make the winning bid larger.

On the other hand, the result of the process, namely who wins the contract, can in principle be computed given all the true values of the bids.

### 1.1.3 Benchmarking

Assume you run a company. You will naturally be interested in how well you are doing compared to other companies in the same line of business as yours. The comparison may be concerned with a number of different parameters, such as profit relative to size, average salaries, productivity, etc. Other companies will most likely have similar interests in such a comparison, which is known as a benchmark analysis. Such an analysis takes input from all participating companies. Based on this it tries to compute information on how well a company in the given line of business should be able to perform, and finally each company is told how its performance compares to this "ideal".

It is clear that each company will insist that its own data is private and must not be leaked to the competitors. On the other hand the desired results can be computed from the private data: there are several known methods from information economics for doing such an analysis efficiently.

### 1.1.4 Data Mining

In most countries, public institutions such as the tax authorities or the healthcare system keep databases containing information on citizens. In many cases there are advantages one can get from coordinated access to several such databases. Researchers may be able to get statistics they could not get otherwise, or institutions might get an administrative advantage from being able to quickly gather the information they need on a certain individual.

On the other hand, there is clearly a privacy concern here: access to many different databases by a single party opens the possibility to compile complete dossiers on particular citizens, which would be a violation of privacy. In fact, accessing data on the same person in several distinct databases is forbidden by law in several countries precisely because of this concern.

## 1.2 Do We Have to Trust Someone?

We are now in a position to extract some common features of all the scenarios we looked at. One way to describe them all is as follows: we have a number of parties that each possess some private data. We want to do some computation that needs all the private data as input. The parties are interested in learning the result, or at least some part of it, but still want to keep their private data as confidential as possible.

Hopefully, it is clear from the previous section that if we can find a satisfactory solution to this problem, there are a very large number of applications that would benefit. Moreover, this leads to an extremely intriguing theoretical question, as we now explain:

One possible – and trivial – solution would be to find some party $T$ that everyone is willing to trust. Now all parties privately give their input to $T$, he does the required computation, announces the result to the parties, and forgets about the private data he has seen. A moment's thought will show that this is hardly a satisfactory solution: first, we have created a single point of attack from where all the private data can potentially be stolen. Second, the parties must all completely trust $T$, both with respect to privacy and correctness of the results. Now, the reason why there are privacy concerns is that the parties do not trust each other in the first place, so why should we believe that they can find a new party they all trust?

In some applications one may pay a party $T$ for doing the computation; if the amount paid is thought to be larger than what $T$ could gain from cheating, parties may be satisfied with the solution. This seems to work in some cases, for instance when a consultancy house is paid a large fee for doing a benchmark analysis – but this is of course a very expensive solution.

Thus, we are left with a fundamental question: *can the problem be solved without relying on a trusted party?*

At first sight, it may seem that this cannot be possible. We want to compute a result that depends on private data from *all* involved parties. How could one possibly do this unless data from several parties become known to someone, and hence we have to trust that party?

Nevertheless, as we shall see, the problem is by no means impossible to solve, and solutions do exist that are satisfactory, both from a theoretical and a practical point of view.

## 1.3 Multiparty Computation

Let us be slightly more precise about the problem we have to solve: the parties, or *players* that participate are called $\mathsf{P}_1, \dots, \mathsf{P}_n$. Each player $\mathsf{P}_i$ holds a secret input $x_i$, and the players agree on some function $f$ that takes $n$ inputs. Their goal is to compute $y = f(x_1, \dots, x_n)$ while making sure that the following two conditions are satisfied:

- Corretness: the correct value of $y$ is computed; and
- Privacy: $y$ is the *only* new information that is released.

Regarding the latter property, note that since the purpose of the whole exercise is that we want to learn $y$, the best we can hope for in terms of privacy is that nothing but $y$ is leaked. Computing $f$ such that privacy and correctness are achieved is referred to as computing $f$ *securely.* Later in the book, we will be precise about what secure computing is; for now, we will be content with the above intuitive idea. Note also that one may consider a more general case where each player gets his own private output. We will do so later, for now we focus on a single, public output for simplicity.

As one example of how this connects to the scenarios from the previous section, one may think of $x_i$ as a number, namely $\mathsf{P}_i$'s bid in a auction, and $f(x_1, ..., x_n) = (z, j)$ where $x_j = z$ and $z \geq x_i, i = 1, \dots, n$, i.e., $f$ outputs the highest bid, and the identity of the corresponding bidder. If we do not want the winner to pay his own bid, but the bid of the second highest bidder, we simply change $z$ to be this value, which is again a well-defined function of the inputs. This would give us a function implementing a so-called second price auction.

In this section, we give a first intuition on how one might compute a function securely without relying on trusted parties. This requires that we specify a *protocol*, i.e., a set of instructions that players are supposed to follow to obtain the desired result. For simplicity, we will assume for now that players always follow the protocol. We will later address the case where some parties may deviate from the protocol, in order to get more information than they are supposed to or cause the result to be incorrect. We will also assume that any pair of players can communicate securely, i.e., it is possible for $\mathsf{P}_i$ to send a message $m$ to $\mathsf{P}_j$, such that no third party sees $m$ and $\mathsf{P}_j$ knows that $m$ came from $\mathsf{P}_i$. We discuss later how this can be realized in practice.

### 1.3.1 Secure Addition and Voting

Let us first look at a simple special case, namely where each $x_i$ is a natural number, and $f(x_1, \dots, x_n) = \sum_{i=1}^n x_i$. Secure computation of even such a simple function can have very meaningful applications. Consider the case where $\mathsf{P}_1, ..., \mathsf{P}_n$ want to vote on some yes/no decision. Then we can let $x_i$ represent the vote of $\mathsf{P}_i$ where $x_i = 0$ means "no" and $x_i = 1$ means "yes". If we can compute the sum of the $x_i$ securely, this exactly means we get a way to vote with the properties we usually expect: the result $\sum_{i=1}^n x_i$ is indeed the result of the vote, namely the number of yes-votes. Moreover, if the computation is secure, no information is leaked other than $\sum_{i=1}^n x_i$, in particular, no information is revealed on how a particular player voted.

We will now design a protocol for the voting application. To be consistent with the next example we give, we set $n = 3$. An exercise below shows how to construct a voting solution for any $n$.

#### Secret Sharing

Before we can solve the problem, we need to look at an important tool known as *secret-sharing.* The term may seem self-contradictory at first sight: how can anything be secret if you share it with others? Nevertheless, the name makes good sense: the point is that secret sharing provides a way for a party, say $\mathsf{P}_1$, to spread information on a secret number $s$ across all the players, such that they together hold full information on $s$, yet no player (except of course $\mathsf{P}_1$) has any information on $s$. First, we choose a prime $p$, and we define $\mathbb{Z}_p$ as $\mathbb{Z}_p = \{0, 1, ..., p-1\}$[1]. In the following, we will think of the secret $s$ as a number in $\mathbb{Z}_p$.

In order to *share the secret* $s$, $\mathsf{P}_1$ chooses numbers $r_1, r_2$ uniformly at random in $\mathbb{Z}_p$, and sets

$$r_3 = s - r_1 - r_2 \bmod p.$$

Put another way: he chooses $r_1, r_2, r_3$ randomly from $\mathbb{Z}_p$, subject to the constraint that $s = r_1 + r_2 + r_3 \bmod p$. Note that this way of choosing $r_1, r_2, r_3$ means that each of the three numbers is uniformly chosen in $\mathbb{Z}_p$: for each of them, all values in $\mathbb{Z}_p$ are possible and equally likely. Now $\mathsf{P}_1$ sends privately $r_1, r_3$ to $\mathsf{P}_2$, $r_1, r_2$ to $\mathsf{P}_3$, and keeps $r_2, r_3$ himself.

The $r_j$'s are called the *shares* of the *secret* $s$.

The process we have described satisfies two basic properties: First, the secret $s$ is kept private in the sense that neither $\mathsf{P}_2$ nor $\mathsf{P}_3$ knows anything about that secret. As a result if some hacker breaks into the machine of $\mathsf{P}_2$ or $\mathsf{P}_3$ (but not both) he will learn nothing about $s$. Second, the secret $s$ can be reconstructed if shares from at least two players are available. Let argue that this is true in a more precise way:

**Privacy.** Even though $\mathsf{P}_1$ has distributed shares of the secret $s$ to the other players, neither $\mathsf{P}_2$ nor $\mathsf{P}_3$ has any idea what $s$ is. For $\mathsf{P}_2$ we can argue as follows: he knows $r_1, r_3$ (but not $r_2$) and that $s = r_1 + r_2 + r_3 \bmod p$. Take any $s_0 \in \mathbb{Z}_p$. From $\mathsf{P}_2$'s point of view, could it be that $s = s_0$? The answer is yes, for if $s = s_0$ it would have to be the case that $r_2 = s_0 - r_1 - r_3 \bmod p$. This is certainly a possibility, recall that $r_2$ is uniformly chosen in $\mathbb{Z}_p$, so all values are possible. However, any other choice, say $s = s_0' \neq s_0$ is also a possibility. If this was the answer we would have $r_2 = s_0' - r_1 - r_3 \bmod p$, which is a value that is different from $s_0 - r_1 - r_3 \bmod p$, but just as likely. We conclude that, from $\mathsf{P}_2$'s point of view, all values of $s$ in $\mathbb{Z}_p$ remain possible and are equally likely. A similar argument shows that $\mathsf{P}_3$ has no idea what $s$ is.

**Correctness.** If two of the three parties pool their information, the secret can be reconstructed, since then all three shares will be known and one can simply add them modulo $p$.

Note that the privacy property is *information theoretic*: as long a party does not know all three summands, no amount of computing power can give him any information on the corresponding secret. In this book, we focus primarily on protocols with this type of security. The secret sharing technique shown above is a special case of so-called replicated secret sharing. There are many ways to realize secret sharing with other desirable properties than the method we show here, and we look at several such techniques later, as well as a more general definition of what secret sharing is.

## A Protocol for Secure Addition

Of course players $\mathsf{P}_2$ and $\mathsf{P}_3$ can distribute shares of their private values $x_2, x_3$ in exactly the same way as $\mathsf{P}_1$. It turns out that one can now compute the sum securely by locally adding shares and announcing the result. The complete protocol is shown below.

To analyze the secure addition protocol, let us first see why the result $v$ is indeed the correct result. This is straightforward:

$$v = \sum_j s_j \bmod p = \sum_j \sum_i r_{i,j} \bmod p = \sum_i \sum_j r_{i,j} \bmod p = \sum_i x_i \bmod p \ .$$

This shows that the protocol computes the sum modulo $p$ of the inputs, no matter how the $x_i$'s are chosen. However, if we let the parties choose $x_i = 1$ for yes and $x_i = 0$ for no, and make

[1]To be more precise, $\mathbb{Z}_p$ is another name for $\mathbb{Z}/p\mathbb{Z}$ where we identify $i \in \mathbb{Z}_p$ with the residue class of numbers that are congruent to $i$ modulo $p$. See more details in the preliminaries section.

Protocol Secure Addition

Participants are $\mathsf{P}_1, \mathsf{P}_2, \mathsf{P}_3$, input for $\mathsf{P}_i$ is $x_i \in Z_p$, where $p$ is a fixed prime agreed upon in advance.

1. Each $\mathsf{P}_i$ computes and distributes shares of his secret $x_i$ as described in the text: he chooses $r_{i,1}, r_{i,2}$ uniformly at random in $\mathbb{Z}_p$, and sets $r_{i,3} = x_i - r_{i,1} - r_{i,2} \bmod p$.
2. Each $\mathsf{P}_i$ sends privately $r_{i,2}, r_{i,3}$ to $P_1$, $r_{i,1}, r_{i,3}$ to $P_2$, and $r_{i,1}, r_{i,2}$ to $P_3$ (note that this involves $\mathsf{P}_i$ sending "to himself"). So $\mathsf{P}_1$, for instance, now holds $r_{1,2}, r_{1,3}$, $r_{2,2}, r_{2,3}$ and $r_{3,2}, r_{3,3}$.
3. Each $\mathsf{P}_j$ adds corresponding shares of the three secrets – more precisely, he computes, for $\ell \neq j$, $s_\ell = r_{1,\ell} + r_{2,\ell} + r_{3,\ell} \bmod p$, and announces $s_\ell$ to all parties (hence two values are computed and announced).
4. All parties compute the result $v = s_1 + s_2 + s_3 \bmod p$.

sure that $p > 3$, then $\sum_i x_i \bmod p = \sum_i x_i$, because all $x_i$ are 0 or 1 and so their sum cannot be larger than $p$. So in this case, $v$ is indeed the number of yes-votes.

Now, why is it the case that no new information other than the result $v$ is leaked to any player? Let us concentrate on $\mathsf{P}_1$ for concreteness. Now, in step 1, $x_1, x_2$ and $x_3$ are secret shared, and we have already argued above that this tells $\mathsf{P}_1$ nothing whatsoever about $x_2, x_3$. In the final step, $s_1, s_2, s_3$ are announced. Note that $\mathsf{P}_1$ already knows $s_2, s_3$, so $s_1$ is the only new piece of information. However, we can argue that seeing $s_1$ will tell $\mathsf{P}_1$ what $v$ is and nothing more. The reason for this is that, if one is given $s_2, s_3$ and $v$, one can compute $s_1 = v - s_2 - s_3 \bmod p$. Put another way: given what $\mathsf{P}_1$ is supposed to know, namely $v$, we can already compute what he sees in the protocol, namely $s_1$, and therefore seeing the information from the protocol tells him nothing beyond $v$.

This type of reasoning is formalized later in the book and is called a *simulation argument*: given what a player is supposed to know, we show how to efficiently compute (simulate) everything he sees in the protocol, and from this, we conclude that the protocol tells him nothing beyond what we wanted to tell him.

Note that given the result, $\mathsf{P}_1$ is in fact able to compute some information about other people's votes. In particular, he can compute $v - x_1 = x_2 + x_3$, i.e., the sum of the other players' votes. It is easy to get confused and think that because of this, something must be wrong with the protocol, but in fact there is no problem: it is true that $\mathsf{P}_1$ can compute the sum of the votes of $\mathsf{P}_2$ and $\mathsf{P}_3$, but this follows from information $\mathsf{P}_1$ is *supposed to know*, namely the result and his own input. There is nothing the protocol can do to deprive him of such information – in other words, the best a protocol can do is to make sure players only learn what they are supposed to learn, and this includes whatever can be derived from the player's own input and the intended result.

### 1.3.2 Secure Multiplication and Match-Making

To do general secure computation, we will of course need to do more than secure addition. It turns out that the secret sharing scheme from the previous subsection already allows us to do more: we can also do secure multiplication.

Suppose two numbers $a, b \in \mathbb{Z}_p$ have been secret shared as described above, so that $a = a_1 + a_2 + a_3 \bmod p$ and $b = b_1 + b_2 + b_3 \bmod p$, and we wish to compute the product $ab \bmod p$ securely. We obviously have

$$ab = a_1b_1 + a_1b_2 + a_1b_3 + a_2b_1 + a_2b_2 + a_2b_3 + a_3b_1 + a_3b_2 + a_3b_3 \bmod p \ .$$

It is now easy to see that if the $a_i$'s and $b_i$'s have been distributed as described above, it is the case that for each product $a_i b_j$, there is at least one player among the three who knows $a_i$ and $b_j$ and therefore can compute $a_i b_j$. For instance, $\mathsf{P}_1$ has been given $a_2, a_3, b_2, b_3$ and can therefore compute $a_2b_2, a_2b_3, a_3b_2$ and $a_3b_3$. The situation is, therefore, that the desired result $ab$ is the sum of some numbers where each summand can be computed by at least one of the players. But now we are essentially done, since from Protocol Secure Addition, we already know how to add securely!

The protocol resulting from these observations is shown below. To argue why it works, one first notes that correctness, namely $ab = u_1 + u_2 + u_3 \bmod p$, follows trivially from the above. To show that nothing except $ab \bmod p$ is revealed, one notes that nothing new about $a, b$ is revealed in the first step, and because Protocol Secure Addition is private, nothing except the sum of the inputs is revealed in the last step, and this sum always equals $ab \bmod p$.

Protocol Secure Multiplication

Participants are $\mathsf{P}_1, \mathsf{P}_2, \mathsf{P}_3$, input for $\mathsf{P}_1$ is $a \in \mathbb{Z}_p$, input for $\mathsf{P}_2$ is $b \in \mathbb{Z}_p$, where $p$ is a fixed prime agreed upon in advance. $\mathsf{P}_3$ has no input.

1. $\mathsf{P}_1$ distributes shares $a_1, a_2, a_3$ of $a$, while $\mathsf{P}_2$ distributes shares $b_1, b_2, b_3$ of $b$.
2. $\mathsf{P}_1$ locally computes $u_1 = a_2b_2 + a_2b_3 + a_3b_2 \bmod p$, $\mathsf{P}_2$ computes $u_2 = a_3b_3 + a_1b_3 + a_3b_1 \bmod p$, and $\mathsf{P}_3$ computes $u_3 = a_1b_1 + a_1b_2 + a_2b_1 \bmod p$.
3. The players use Protocol Secure Addition to compute the sum $u_1 + u_2 + u_3 \bmod p$ securely, where $\mathsf{P}_i$ uses $u_i$ as input.

It is interesting to note that even in a very simple case where both $a$ and $b$ are either 0 or 1, secure multiplication has a meaningful application: consider two parties Alice and Bob. Suppose Alice is wondering whether Bob wants to go out with her, and also Bob is asking himself if Alice is interested in him. They would very much like to find out if there is mutual interest, but without running the risk of the embarrassment that would result if for instance Bob just tells Alice that he is interested, only to find that Alice turns him down. The problem can be solved if we let Alice choose $a \in Z_p$ where $a = 1$ if she is interested in Bob and $a = 0$ otherwise. In the same way, Bob chooses $b$ to be 0 or 1. Then we compute the function $f(a, b) = ab \bmod p$ *securely*. It is clear that the result is 1 if and only if there is mutual interest. But on the other hand if, for instance, Alice is not interested, she will choose $a = 0$ and in this case she learns *nothing new* from the protocol. To see why, notice that security of the protocol implies that the only (possibly) new information Alice will learn is the result $ab \bmod p$. But she already knows that result will be 0! In particular, she does not learn whether Bob was interested or not, so Bob is safe from embarrassment. By a symmetric argument, this is of course also the case for Alice.

This argument assumes, of course, that both players choose their inputs honestly according to their real interests. In the following section we discuss what happens is players do not follow the instructions and what we can do about the problems resulting from this.

From Protocol Secure Multiplication, we see that if Alice and Bob play the roles of $\mathsf{P}_1$ and $\mathsf{P}_2$, respectively, they just need to find a third party to help them, to do the multiplication securely. Note that this third party is not a *completely trusted* third party of the kind we discussed before: he does not learn anything about $a$ or $b$ other than $ab \bmod p$. Alice and Bob do have to trust, however, that the third party does not share his information with Bob or with Alice.

It is an obvious question whether one can do secure multiplication such that *only* Alice and Bob have to be involved? The answer turns out to be yes, but then information theoretic security is not possible, as we shall see. Instead one has to use solutions based on cryptography. Such solutions can always be broken if one party has enough computing power, but on the other hand, this is an issue with virtually all the cryptographic techniques we use in practice.

For completeness, we remark that Alice and Bob's problem is a special case of the so-called match-making problem which has somewhat more serious applications than secure dating. Consider a set of companies where each company has a set of other companies it would prefer to do business with. Now we want that each pair of companies finds out whether there is mutual interest, but without forcing companies to reveal their strategy by announcing their interests in public.

**Exercise 1.1** Consider the third party helping Alice and Bob to do secure multiplication. Show that the Protocol Secure Multiplication is indeed insecure if he reveals what he sees in the protocol to Alice or Bob.

**Exercise 1.2** We have used replicated secret sharing where each player receives two numbers in $\mathbb{Z}_p$, even though only one secret number is shared. This was to be able to do both secure addition and multiplication, but for secure addition only, something simpler can be done. Use the principle of writing the secret as a sum of random numbers to design a secret sharing scheme for any number of parties, where each party gets as his share only a single number in $\mathbb{Z}_p$. Use your scheme to design a protocol for secure addition. How many players can go together and pool their information before the protocol becomes insecure?

**Exercise 1.3** You may have asked yourself why Protocol Secure Multiplication uses Protocol Secure Addition as a subroutine. Why not just announce $u_1, u_2$ and $u_3$, and add them to get the result? The reason is that this would reveal too much information. Show that if $\mathsf{P}_1$ was given $u_2$, he could - in some cases - compute $\mathsf{P}_2$'s input $b$. What is the probability that he will succeed?

### 1.3.3 What if Players Do Not Follow Instructions?

Until now we assumed that players always do what they are supposed to. But this is not always a reasonable assumption, since a party may have an interest in doing something different from what he is instructed to do.

There are two fundamentally different ways in which players could deviate from expected behavior: First, they could choose their inputs in a way different from what was expected when the protocol was designed. Second, while executing the protocol, they could do something different from what the protocol instructs them to do. We will consider the two issues separately:

#### Choice of Inputs

Consider the matchmaking application from above as an example. Let us assume that Alice is not really interested in Bob. We argued above that since her input should then be 0, the output is always 0 and she does not learn whether Bob was interested. The reader may have noticed that there seems to be a way Alice could cheat Bob, if she is willing to choose an input that does not represent her actual interests: she could *pretend* to be interested by choosing $a = 1$, in which case the output will be $ab \bmod p = b$ so she now learns Bob's input and breaks his privacy.

A moment's thought will convince the reader that there is no way we could possibly solve this issue by designing a more secure protocol: whatever the protocol is, a player can of course always choose to execute it with any input he wants to. Since this book is about protocol design, this issue of choice of inputs is out of scope for us.

Therefore, if Bob is worried that Alice might behave as we just described, the only answer we can give is that then he should not play the game at all! If he goes ahead, this has to be based on an assumption that Alice (as well as he himself) has an interest in choosing inputs in a “reasonable way”. A possible justification for such an assumption might be that if Alice really thinks Bob is a looser, then the prospect of being stuck with him the rest of the night should be daunting enough to make her choose her input according to her actual preference!

More seriously (and generally): to do secure computation, we have to assume that players have an incentive to provide inputs that will lead to a “meaningful” result they would like to learn. If one can describe and quantify these incentives, it is sometimes possible to analyze what will happen using a different mathematical discipline called game theory, but that is out of scope for this book.

**Deviation from the Protocol**

Regardless of how the inputs are chosen, it might be that deviating from the protocol could enable a player to learn more information than he was supposed to get, or it could allow him to force the computation to give a wrong result. For instance, this way he could appoint himself the winner of an auction at a low price. This is of course undesirable, but (in contrast to the issue of input choice) this is indeed an issue we can handle.

One solution is to add mechanisms to the protocol which ensure that any deviation from the protocol will be detected. To exemplify this we look at Protocol Secure Addition.

In Protocol Secure Addition we first ask each party to distribute shares of their secret. Looking at $\mathsf{P}_1$, we ask it to pick shares $r_{1,1}, r_{1,2}, r_{1,3}$ such that $x_1 = r_{1,1} + r_{1,2} + r_{1,3} \bmod p$ and then send $r_{1,1}, r_{1,3}$ to $\mathsf{P}_2$ and send $r_{1,1}, r_{1,2}$ to $\mathsf{P}_3$. Here there are two ways to deviate:

First, $\mathsf{P}_1$ could pick $r'_{1,1}, r'_{1,2}, r'_{1,3}$ such that $x_1 \neq r'_{1,1} + r'_{1,2} + r'_{1,3} \bmod p$. This is not a problem, as it just corresponds to having used the input $x'_1 \overset{\text{def}}{=} r'_{1,1} + r'_{1,2} + r'_{1,3} \bmod p$, and as mentioned we cannot (and should not) prevent $\mathsf{P}_1$ from being able to pick any input it desires. The second way to deviate is that $\mathsf{P}_1$ could send $r_{1,1}, r_{1,3}$ to $\mathsf{P}_2$ and send $r'_{1,1}, r_{1,2}$ to $\mathsf{P}_3$ with $r'_{1,1} \neq r_{1,1}$. This is more serious, as now the input $x_1$ of $\mathsf{P}_1$ is not well-defined. This might or might not lead to an attack, but it is at least a clear deviation from the protocol. There is, however, a simple way to catch this deviation: when $\mathsf{P}_2$ and $\mathsf{P}_3$ receive their shares from $\mathsf{P}_1$, then $\mathsf{P}_2$ sends its value of $r_{1,1}$ to $\mathsf{P}_3$ and $\mathsf{P}_3$ sends its own value of $r_{1,1}$ to $\mathsf{P}_2$. Then they check that they hold the same value. In a similar way $\mathsf{P}_1$ and $\mathsf{P}_3$ can check that $\mathsf{P}_2$ sends consistent shares and $\mathsf{P}_1$ and $\mathsf{P}_2$ can check that $\mathsf{P}_3$ sends consistent shares. In general, having players reveal more information to each other could make a protocol insecure. However, in this case no new information leaks because a player only send information which the receiver should already have.

After the sharing phase, we then ask the parties to add their shares and make the sums public. Looking again at $\mathsf{P}_1$, we ask it to compute $s_2$ and $s_3$ and make these public. Here $\mathsf{P}_1$ might deviate by sending, e.g., $s'_2 \neq s_2$. This could lead to a wrong result, $v = s_1 + s'_2 + s_3 \bmod p$. Note, however, that $\mathsf{P}_3$ will compute $s_1$ and $s_2$ and make these public. So, both $\mathsf{P}_1$ and $\mathsf{P}_3$ are supposed to make $s_2$ public. Hence, the players can simply check that $\mathsf{P}_1$ and $\mathsf{P}_3$ make the same value of $s_2$ public. And similarly they can check that the two versions of $s_1$ and the two versions of $s_3$ are identical.

This means Protocol Secure Addition has the following property: if any single party does not do what he is supposed to, the other two players will always be able to detect this. This idea of checking other players to ensure that the protocol is followed is something we will see many times in the following.

### 1.3.4 Towards General Solutions

We have now seen how to do secure multiplication and addition of numbers in $\mathbb{Z}_p$ – although under various rather strong assumptions. We have assumed that players always follow the protocol (although we have seen a partial answer on how to deal with deviations). Furthermore we have only considered the harm that a *single* player can do to the protocol, and not what happens if several players go together and try, for instance, to compute information on the other players' inputs.

Nevertheless, it is well known that multiplication and addition modulo a prime is sufficient to efficiently simulate any desired computation. This implies, on a very fuzzy and intuitive level, that we can hope to be able to do *any* computation securely, if that computation was feasible in the first place – at least under certain assumptions.

It turns out that this is indeed the case, but of course lots of questions remain. To name a few: how do we define security in a precise way? how do we scale solutions from 3 players to any number of players? what if players do not all follow the protocol? if several players pool their information to learn more than they were supposed to, how many such players can we tolerate and still have a secure protocol? On the following pages, we will arrive at answers to these questions.

# Chapter 2

# Preliminaries

## Contents

In this chapter we introduce some basic notions used throughout the book, like random variables, families of random variables, interactive systems, and the statistical and computational indistinguishability of these objects. The primary purpose of the chapter is to fix and make precise our notation. A reader who has followed a course in basic cryptography will probably be familiar with most of the notions covered in this chapter, but is encouraged to do a quick reading of the chapter to get familiar with the book's notation. A possible exception is the material on interactive systems. This part is used intensively in Chapter 4. For later use, a cheat sheet with notation can be found in Chapter 12.3.

## 2.1 Linear Algebra

Let $\mathbb{F}$ be any ring. We use $\mathbf{a} = (a_1, \dots, a_n)$ to denote column vectors. Let $\mathbf{a} = (a_1, \dots, a_n) \in \mathbb{F}^n$, $\mathbf{b} = (b_1, \dots, b_n) \in \mathbb{F}^n$ and $\alpha \in \mathbb{F}$. We use the following standard notation from linear algebra.

$$\mathbf{a} + \mathbf{b} = (a_1 + b_1, \dots, a_n + b_n) \ , \tag{2.1}$$

$$\alpha\mathbf{a} = (\alpha a_1, \dots, \alpha a_n) \ , \tag{2.2}$$

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i \ , \tag{2.3}$$

$$\mathbf{a} * \mathbf{b} = (a_1 b_1, \dots, a_n b_n) \ . \tag{2.4}$$

The operator $\cdot$ is called the inner product and $*$ is called the Schur product and entrywise multiplication.

We use capital letters like $A$ to denote matrices. We use $a_{ij}$ to denote the entry in row $i$, column $j$ in matrix $A$. We use $\mathbf{a}_i$ and $\mathbf{a}_{i*}$ to denote row $i$ of matrix $A$, and $\mathbf{a}_{*j}$ to denote column

$j$ of $A$. We use $A^\mathsf{T}$ for the matrix transpose and $AB$ for matrix multiplication. We write $\ker A$ for the kernel of $A$.

## 2.2 Random Variables

The range of a random variable $X$ is the set of elements $x$ for which the random variable $X$ outputs $x$ with probability greater than 0. When we say random variable in this book we always mean *random variable with finite range.* For a random variable $X$ and an element $x$ we use $\Pr[X = x]$ to denote the probability that $X$ outputs $x$. I.e., for a random variable $X$ there is a finite set $D$ such that $\sum_{x \in D} \Pr[X = x] = 1$. The range of a set of random variables $X_1, \ldots, X_\ell$ is the union of their ranges.

**Definition 2.1 (statistical distance)** *Let $X_0$ an $X_1$ be two random variable with range $D$. We call*

$$\delta(X_0, X_1) \overset{\text{def}}{=} \frac{1}{2} \sum_{d \in D} |\Pr[X_0 = d] - \Pr[X_1 = d]|$$

*the statistical distance between $X_0$ and $X_1$.*

Statistical distance is also known as total variation distance.

A randomized algorithm $A$ is an algorithm $A$ which takes two inputs, an input $x$ and randomness $r$. We use $A(x; r)$ to denote that we run $A$ on $x$ and $r$. We use the semi-colon to signal that the part of the input after the semi-colon is not a real input, but an input used to make the random choices that $A$ need to make during the computation. Typically we run $A$ with $r$ being a uniformly random string. So, if $A$, e.g., needs to flip a fair coin, it uses the next fresh bit from $r$. If $A$ is a randomized algorithm using randomness from $\{0, 1\}^\ell$, then we use $A(x)$ to denote the random variable defined as follows:

1. Sample $r \in_{\mathrm{R}} \{0, 1\}^\ell$.
2. Compute $a \leftarrow A(x; r)$.
3. Output $a$.

We write $a \leftarrow A(x)$ for sampling $A(x)$ in this way.

If $X$ is a random variable and $A$ is a randomized algorithm, then $A(X)$ is the random variable defined as follows:

1. Sample $x \leftarrow X$.
2. Sample $a \leftarrow A(x)$.
3. Output $a$.

We call two random variables $X_0$ and $X_1$ with range $D$ disjoint random variables if it holds for all $x \in D$ that either $\Pr[X_0 = x] = 0$ or $\Pr[X_1 = x] = 0$. I.e., we call them disjoint iff there is no element in the range which they can both output.

**Proposition 2.1** *Let $A$ be any (possibly randomized) algorithm and let $X_0, X_1, X_2$ be any random variables. Then the following holds:*

1. $\delta(X_0, X_1) \in [0, 1]$.
2. $\delta(X_0, X_1) = 0$ *iff* $\Pr[X_0 = x] = \Pr[X_1 = x]$ *for all $x$.*
3. $\delta(X_0, X_1) = 1$ *iff $X_0$ and $X_1$ are disjoint.*
4. $\delta(X_0, X_1) = \delta(X_1, X_0)$.

5. $\delta(X_0, X_2) \le \delta(X_0, X_1) + \delta(X_1, X_2)$.
6. $\delta((X_0, X_2), (X_1, X_2)) = \delta(X_0, X_1)$.
7. $\delta(A(X_0), A(X_1)) \le \delta(X_0, X_1)$.

This, in particular, means that statistical distance is a metric on probability distributions. The last property shows that computation cannot make statistical distance increase.

**Exercise 2.1** Prove Proposition 2.1.

### 2.2.1 Distinguishing Advantage of an Algorithm

For an algorithm $A$ and two random variables $X_0$ and $X_1$ we need a measure of the ability of $A$ to distinguish between $X_0$ and $X_1$. For this purpose we play a game, where we give $A$ either a sample from $X_0$ or a sample from $X_1$ and then ask it to guess which random variable it received a sample from.

1. Sample a uniformly random bit $b \leftarrow \{0, 1\}$.
2. Sample $x \leftarrow X_b$.
3. Sample $c \leftarrow A(x)$.

We think of $c$ as $A$'s guess at $b$. A fixed guess, $c = 0$ say, will allow $A$ to be correct with probability $\frac{1}{2}$ as $b = 0$ with probability $\frac{1}{2}$. When we measure how good $A$ is at distinguishing $X_0$ from $X_1$ we are therefore only interested in how much better than $\frac{1}{2}$ it does. For technical reasons we consider the absolute value, i.e., we use the measure $|\Pr[c = b] - \frac{1}{2}|$. Note that this is a value between 0 and $\frac{1}{2}$. Again for technical reason we prefer the measure to be between 0 and 1, so we scale by a factor of 2, and we get the measure $2|\Pr[c = b] - \frac{1}{2}|$, which we call the advantage of $A$.

Note that if $A$ can always guess $b$ correctly, then $\Pr[c = b] = 1$ and the advantage will be 1. If $A$ makes a random guess, a fixed guess or any other guess independent of $b$, then $\Pr[c = b] = \frac{1}{2}$ and the advantage will be 0. We therefore think of the advantage as measuring how well $A$ can distinguish $X_0$ and $X_1$, with 1 meaning *perfectly* and 0 meaning *not at all*.

Before we make a formal definition it is convenient to rewrite the advantage. For this purpose, observe that if we restrict $A$ to outputting $c \in \{0, 1\}$, then

$$\begin{aligned}
\Pr[c = b] &= \frac{1}{2}\Pr[c = b | b = 0] + \frac{1}{2}\Pr[c = b | b = 1] \\
&= \frac{1}{2}(\Pr[c = 0 | b = 0] + \Pr[c = 1 | b = 1]) \\
&= \frac{1}{2}(\Pr[A(X_0) = 0] + \Pr[A(X_1) = 1]) \\
&= \frac{1}{2}(\Pr[A(X_0) = 0] + (1 - \Pr[A(X_1) = 0])) \\
&= \frac{1}{2} + \frac{1}{2}(\Pr[A(X_0) = 0] - \Pr[A(X_1) = 0]) \ ,
\end{aligned}$$

where the first equation is just an application of the law of total probability, and $\Pr[A(X_1) = 1] = 1 - \Pr[A(X_1) = 0]$ as $A$ outputs either 0 or 1. This means that

$$2\left|\Pr[c = b] - \frac{1}{2}\right| = |\Pr[A(X_0) = 0] - \Pr[A(X_1) = 0]| \ . \tag{2.5}$$

We can thus think of the advantage as the difference between the probability that $A$ guesses 0 when it sees $X_0$ and the probability that $A$ guesses 0 when it sees $X_1$. It is usual to use the rewritten expression for defining the advantage.

**Definition 2.2 (advantage)** *Let $X_0$ an $X_1$ be two random variables. Let $A$ be any algorithm outputting a bit $c \in \{0,1\}$ (we call such an algorithm a* distinguisher*). The* advantage *of $A$ in distinguishing $X_0$ and $X_1$ is*

$$\textsc{Adv}_A(X_0, X_1) \overset{\text{def}}{=} |\Pr[A(X_0) = 0] - \Pr[A(X_1) = 0]| \ .$$

The reason why the rewritten expression is typically used for the formal definition is that it is easily related to the statistical distance between the random variables $A(X_0)$ and $A(X_1)$, which is sometimes convenient when working with the notion of advantage. Namely, note that

$$\begin{aligned}
\delta(A(X_0), A(X_1)) &= \frac{1}{2} \sum_{c=0,1} |\Pr[A(X_0) = c] - \Pr[A(X_1) = c]\,| \\
&= \frac{1}{2}(|\Pr[A(X_0) = 0] - \Pr[A(X_1) = 0]\,| + |\Pr[A(X_0) = 1] - \Pr[A(X_1) = 1]\,|) \\
&= \frac{1}{2}(|\Pr[A(X_0) = 0] - \Pr[A(X_1) = 0]\,| + \\
&\qquad |(1 - \Pr[A(X_0) = 0]) - (1 - \Pr[A(X_1) = 0])|) \\
&= \frac{1}{2}(|\Pr[A(X_0) = 0] - \Pr[A(X_1) = 0]\,| + |\Pr[A(X_1) = 0] - \Pr[A(X_0) = 0]\,|) \\
&= \frac{1}{2}(|\Pr[A(X_0) = 0] - \Pr[A(X_1) = 0]\,| + |\Pr[A(X_0) = 0] - \Pr[A(X_1) = 0]\,|) \\
&= |\Pr[A(X_0) = 0] - \Pr[A(X_1) = 0]\,| \ .
\end{aligned}$$

This shows that

$$\textsc{Adv}_A(X_0, X_1) = \delta(A(X_0), A(X_1)) \ . \tag{2.6}$$

We can therefore think of the advantage of an algorithm $A$ as the statistical distance between its outputs when it gets $X_0$ as input respectively when it gets $X_1$ as input. It is clear that if we allowed $A$ to output an arbitrarily long bit-string, then it could simply output its input. In that case $A(X_0) = X_0$ and $A(X_1) = X_1$ and we would have $\textsc{Adv}_A(X_0, X_1) = \delta(X_0, X_1)$. I.e., the advantage would simply be the statistical distance. We have, however, required that $A$ outputs a single bit, which means that in general the advantage $\textsc{Adv}_A(X_0, X_1)$ could be smaller than the statistical distance $\delta(X_0, X_1)$. In this light, the job of a distinguisher is so to say to boil down the statistical distance between $X_0$ and $X_1$ to a single bit.

The relation between advantage and statistical distance is convenient as it tells us that advantage inherits all the nice properties of statistical distance. As an example Proposition 2.1 directly allows us to conclude the following.

**Corollary 2.2** *Let $A$ be any (possibly randomized) algorithm and let $X_0, X_1, X_2$ be any random variables. Then the following holds:*

1. $\textsc{Adv}_A(X_0, X_1) \in [0, 1]$.
2. $\textsc{Adv}_A(X_0, X_1) = 0$ *if $X_0$ and $X_1$ are identically distributed.*
3. $\textsc{Adv}_A(X_0, X_1) = \textsc{Adv}_A(X_1, X_0)$.
4. $\textsc{Adv}_A(X_0, X_2) \leq \textsc{Adv}_A(X_0, X_1) + \textsc{Adv}_A(X_1, X_2)$.
5. $\textsc{Adv}_A(X_0, X_1) \leq \delta(X_0, X_1)$.

We have just seen that advantage can be phrased in terms of a statistical distance. Conversely, statistical distance can be phrased in terms of advantage: It is possible to show that

$$\delta(X_0, X_1) = \max_A \textsc{Adv}_A(X_0, X_1) \ , \tag{2.7}$$

where the maximum is taken over all possible distinguishers. To show this, one considers the particular maximum likelihood distinguisher $A_{\text{ML}}$ which on input $x$ outputs $c = 0$ if $\Pr[X_0 = x] \geq \Pr[X_1 = x]$ and otherwise outputs $c = 1$. One can argue that $\delta(A_{\text{ML}}(X_0), A_{\text{ML}}(X_1)) = \delta(X_0, X_1)$, which establishes Eq. 2.7 via Eq. 2.6.

**Exercise 2.2** Prove Eq. 2.7 by fleshing out the details of the argument following that equation.

### 2.2.2 Distinguishing Advantage

The notion of an algorithm distinguishing random variables can be extended to classes of algorithms simply by considering the best distinguisher in the class.

**Definition 2.3 (advantage of a class of algorithms)** *Let $X_0$ an $X_1$ be two random variables. Let $\mathcal{A}$ be any class of algorithms outputting a bit $c \in \{0,1\}$. The advantage of $\mathcal{A}$ in distinguishing $X_0$ and $X_1$ is*

$$\textsc{Adv}_{\mathcal{A}}(X_0, X_1) \overset{\text{def}}{=} \max_{A \in \mathcal{A}} \textsc{Adv}_A(X_0, X_1) \ .$$

From Eq. 2.7 we know that $\delta(X_0, X_1) = \textsc{Adv}_{\mathcal{A}}(X_0, X_1)$ when $\mathcal{A}$ is the class of all algorithms. In general $\textsc{Adv}_{\mathcal{A}}$ can be much smaller than the statistical distance.

## 2.3 Families of Random Variables

By a family of random variables we mean a function $X$ from the non-negative integers into random variables. I.e., for each $\kappa \in \mathbb{N}$, $X(\kappa)$ is a random variable. We write a family of random variables as

$$X = \{X(\kappa)\}_{\kappa \in \mathbb{N}} \ .$$

If $X = \{X(\kappa)\}_{\kappa \in \mathbb{N}}$ is a family of random variables and $A$ is a two-input algorithm which takes the security parameter $\kappa$ as its first input, then we let

$$A(X) \overset{\text{def}}{=} \{A(\kappa, X(\kappa))\}_{\kappa \in \mathbb{N}} \ .$$

I.e., $A(X)$ is again a family of random variables and the random variable associated to $\kappa$ is $A(\kappa, X(\kappa))$, which is defined by first sampling $x \leftarrow X(\kappa)$ and then running $A(\kappa, x)$ to define the output of $A(\kappa, X(\kappa))$.

### 2.3.1 Statistical Indistinguishability

We will say that two families of random variables $X_0$ and $X_1$ are statistically indistinguishable if the statistical distance between $X_0(\kappa)$ and $X_1(\kappa)$ goes to 0 very quickly as $\kappa$ grows. By very quickly we will mean that the statistical distance goes to 0 faster than any inverse polynomial – i.e., it is a so-called negligible function, as defined below.

**Definition 2.4 (negligible function)** *We call a function $\delta : \mathbb{N} \to [0,1]$ negligible in $\kappa$ if for all $c \in \mathbb{N}$ there exists $\kappa_c \in \mathbb{N}$ such that $\delta(\kappa) \leq \kappa^{-c}$ for all $\kappa \geq \kappa_c$.*

**Definition 2.5 (statistical indistinguishability)** *We say that $X_0$ and $X_1$ are statistically indistinguishable, written $X_0 \overset{\text{stat}}{\equiv} X_1$, if $\delta(X_0(\kappa), X_1(\kappa))$ is negligible in $\kappa$. If $X_0$ and $X_1$ are* not *statistically indistinguishable, we write $X_0 \overset{\text{stat}}{\not\equiv} X_1$.*

*We say that $X_0$ and $X_1$ are perfectly indistinguishable, written $X_0 \overset{\text{perf}}{\equiv} X_1$, if $\delta(X_0(\kappa), X_1(\kappa)) = 0$ for all $\kappa$. If $X_0$ and $X_1$ are* not *perfectly indistinguishable, we write $X_0 \overset{\text{perf}}{\not\equiv} X_1$.*

Two families of random variables which are perfectly indistinguishable cannot be distinguished by looking at their output, as they make the same output with the same probability. Two families of random variables which are statistically indistinguishable are very close to being perfectly indistinguishable—by moving a negligible amount of probability mass on their output distributions they can be made to be perfectly indistinguishable—and we therefore think of them as being essentially impossible to distinguish by looking at they output. We formalize this later.

When working with indistinguishability of families of random variables, it is convenient to know that the notion is transitive and maintained under computation:

**Proposition 2.3** *Let $A$ be any (possibly randomized) algorithm and let $X_0, X_1, X_2$ be any families of random variables. Then the following holds:*

1. $X_0 \overset{\text{stat}}{\equiv} X_0$.
2. *If* $X_0 \overset{\text{stat}}{\equiv} X_1$ *and* $X_1 \overset{\text{stat}}{\equiv} X_2$*, then* $X_0 \overset{\text{stat}}{\equiv} X_2$.
3. *If* $X_0 \overset{\text{stat}}{\equiv} X_1$*, then* $A(X_0) \overset{\text{stat}}{\equiv} A(X_1)$.

*The same properties hold for* $\overset{\text{perf}}{\equiv}$.

**Exercise 2.3** Prove Proposition 2.3 using Proposition 2.1.

### 2.3.2 Distinguishing Advantage of a Class of Algorithms

We now look at the advantage of an algorithm $A$ in distinguishing two families of random variables $X_0 = \{X_0(\kappa)\}_{\kappa \in \mathbb{N}}$ and $X_1 = \{X_0(\kappa)\}_{\kappa \in \mathbb{N}}$. We are interested in how well $A$ can distinguish $X_0(\kappa)$ and $X_1(\kappa)$ as $\kappa$ grows. For this purpose we define a function

$$\textsc{Adv}_A(X_0, X_1) : \mathbb{N} \to [0, 1]$$

which for each $\kappa \in \mathbb{N}$ measures how well $A$ distinguishes $X_0(\kappa)$ and $X_1(\kappa)$. I.e.,

$$\textsc{Adv}_A(X_0, X_1)(\kappa) \overset{\text{def}}{=} \textsc{Adv}_{A(\kappa,\cdot)}(X_0(\kappa), X_1(\kappa)) ,$$

where $A(\kappa, \cdot)$ is the algorithm that takes one input $x$ and runs $A(\kappa, x)$.

We say that $A$ statistically cannot distinguish $X_0$ and $X_1$ if $\textsc{Adv}_{A(\kappa,\cdot)}(X_0(\kappa), X_1(\kappa))$ goes to 0 very quickly as $\kappa$ grows. We say that $A$ perfectly cannot distinguish $X_0$ and $X_1$ if $\textsc{Adv}_{A(\kappa,\cdot)}(X_0(\kappa), X_1(\kappa)) = 0$ for all $\kappa$.

**Exercise 2.4** Shows that $\textsc{Adv}_A(X_0, X_1)$ is negligible in $\kappa$ iff $A(X_0) \overset{\text{stat}}{\equiv} A(X_1)$ and that $\textsc{Adv}_A(X_0, X_1)(\kappa) = 0$ for all $\kappa$ iff $A(X_0) \overset{\text{perf}}{\equiv} A(X_1)$.

The above exercise shows that we really do not need a notion of distinguishing advantage between families of random variables $X_0$ and $X_1$. We can simply work with statistical distance between families of random variables $A(X_0)$ and $A(X_1)$. It turns out that it is more convenient to work directly with statistical distance, so this is what we will do. For intuition it is, however, convenient to bare in mind that when $A$ is a distinguisher (an algorithm outputting a single bit) then $A(X_0) \overset{\text{perf}}{\equiv} A(X_1)$ essentially means that $A$ perfectly cannot distinguish $X_0$ and $X_1$ and $A(X_0) \overset{\text{stat}}{\equiv} A(X_1)$ means that $A$ only has a negligible advantage in distinguishing $X_0$ and $X_1$.

Typically we are in cryptography not interested in whether one *specific* algorithm can distinguish two families of random variables. The distinguisher is typically modeling the adversary, which can have any behavior. We are therefore more interested in whether *some* algorithm can distinguish two families of random variables. We do, however, often restrict the adversary to performing only realistic computation, like at most $2^{100}$ basic operations. For this purpose, and others, it is convenient to have a notion of indistinguishability for some *class* of distinguishers.

**Definition 2.6 (indistinguishability by class of algorithms)** *Let $X_0 = \{X_0(\kappa)\}_{\kappa \in \mathbb{N}}$ and $X_1 = \{X_1(\kappa)\}_{\kappa \in \mathbb{N}}$ by families of random variables and let $\mathcal{A}$ be any class of distinguishers (algorithms outputting a single bit). We say that $X_0$ and $X_1$ are indistinguishable by $\mathcal{A}$ if $A(X_0) \overset{\text{stat}}{\equiv} A(X_1)$ for all $A \in \mathcal{A}$. We write this as*

$$X_0 \overset{\mathcal{A}}{\equiv} X_1 \ .$$

*If $X_0 \overset{\mathcal{A}}{\equiv} X_1$, where $\mathcal{A}$ is the class of all poly-time algorithms, then we say that $X_0$ and $X_1$ are* computationally indistinguishable *and write*

$$X_0 \overset{\text{comp}}{\equiv} X_1 \ .$$

*If $X_0$ and $X_1$ are* not *computationally indistinguishable, we write $X_0 \overset{\text{comp}}{\not\equiv} X_1$.*

**Example 2.1** The following example shows that two families of random variables might be perfectly distinguishable in the statistical sense yet computationally *in*distinguishable.

Let $(G, E, D)$ be a public-key encryption scheme. On input the security parameter $\kappa$, the key generator $G$ generates a key-pair $(pk, sk) \leftarrow G(\kappa)$, where $pk$ is the public-key and $sk$ is the secret key and $\kappa$ specifies the desired security level of the key. On input $pk$ and a message $m$, the encryption algorithm outputs a ciphertext $C \leftarrow E_{pk}(m)$. Note the $E$ is allowed to use internal randomness and that $C$ depends on this randomness. On input $C$ and $sk$ the decryption algorithm outputs a message $m \leftarrow D_{sk}(C)$. We require that $D_{sk}(C) = m$ when $C \leftarrow E_{pk}(m)$.

For $b \in \{0, 1\}$ let $X_b = \{X_b(\kappa)\}$, where $X_b(\kappa)$ is defined as follows: Sample $(pk, sk) \leftarrow G(\kappa)$, samples $C \leftarrow E_{pk}(b)$ and let $X_b(\kappa) = (pk, C)$. In words, $X_b(\kappa)$ outputs a random public key and a random encryption of $b$, using $\kappa$ as the security level.

We first consider the statistical distance between $X_0$ and $X_1$. Since a given ciphertext $C$ cannot be both an encryption of 0 and an encryption of 1 there is no value which both $X_0(\kappa)$ and $X_1(\kappa)$ could output, i.e., they are disjoint random variables. By Proposition 2.1 this means that $\delta(X_0(\kappa), X_1(\kappa)) = 1$. This is of course trivial: the distinguisher is given $(pk, C)$ and then simply checks whether $C$ is an encryption of 0 or 1, and outputs the corresponding guess $c$. It might do this check by doing, e.g., an exhaustive search for $sk$ and then decrypting.

We then consider the "computational distance" between $X_0$ and $X_1$. For public-key cryptosystem a standard security notion is that of semantic security. Semantic secure cryptosystems are known to exist under a number of assumptions, like the RSA assumption. The notion of semantic security essentially requires that no efficient algorithm can distinguish an encryption of 0 from an encryption of 1. Technically this is defined by requiring that the distinguishing advantage of any poly-time adversary is negligible in the security parameter. This directly implies that $X_0 \overset{\text{comp}}{\equiv} X_1$.

All in all, this shows that $X_0$ and $X_1$ have as large a statistical distance as is possible, yet they are computationally indistinguishable. In particular, $X_0 \overset{\text{perf}}{\not\equiv} X_1$ and $X_0 \overset{\text{stat}}{\not\equiv} X_1$, yet $X_0 \overset{\text{comp}}{\equiv} X_1$. The explanation is that the statistical distance between $X_0$ and $X_1$ cannot be noticed by a poly-time distinguisher. △

Let $\mathcal{A}$ be any class of algorithms. For any two algorithms $A$ and $B$ let $A \circ B$ be the algorithm which runs as follows on input $x$:

1. Sample $b \leftarrow B(x)$.
2. Sample $a \leftarrow A(b)$.
3. Return $a$.

We let $\mathcal{A}^\circ$ be the class of algorithms $B$ for which $\mathcal{A} \circ B \subseteq \mathcal{A}$, by which we simply mean that it holds for all $A \in \mathcal{A}$ that $A \circ B \in \mathcal{A}$. In words, $\mathcal{A}^\circ$ are those algorithms $B$ for which it holds that if it is composed with an algorithm from $\mathcal{A}$ it is again an algorithm from $\mathcal{A}$. As an example, if $\mathcal{A}$ is the set of all poly-time algorithms, then $\mathcal{A}^\circ$ is also the class of all poly-time algorithms.

**Theorem 2.4** *Let $X_0$ and $X_1$ be families of random variables and let $\mathcal{A}$ be any class of distinguishers. Then the following holds.*

1. $X_0 \overset{\mathcal{A}}{\equiv} X_0$.
2. *If* $X_0 \overset{\mathcal{A}}{\equiv} X_1$ *then* $X_1 \overset{\mathcal{A}}{\equiv} X_0$.
3. *If* $X_0 \overset{\mathcal{A}}{\equiv} X_1$ *and* $X_1 \overset{\mathcal{A}}{\equiv} X_2$, *then* $X_0 \overset{\mathcal{A}}{\equiv} X_2$.
4. *If* $X_0 \overset{\mathcal{A}}{\equiv} X_1$ *and* $B \in \mathcal{A}^\circ$, *then* $B(X_0) \overset{\mathcal{A}}{\equiv} B(X_1)$.
5. *If* $X_0 \overset{\mathcal{A}}{\equiv} X_1$ *and* $B \in \mathcal{A}^\circ$, *then* $B(X_0) \overset{\mathcal{A}}{\equiv} B(X_1)$.

*Proof* The first three properties follows directly from Proposition 2.1. To show that $B(X_0) \overset{\mathcal{A}}{\equiv} B(X_1)$, we have to show that $A(B(X_0)) \overset{\text{stat}}{\equiv} A(B(X_1))$ for all $A \in \mathcal{A}$. So, let $A$ be any algorithm from $\mathcal{A}$. Since $A \in \mathcal{A}$ and $B \in \mathcal{A}^\circ$ we have that $C \in \mathcal{A}$ when $C = A \circ B$. From $C \in \mathcal{A}$ and $X_0 \overset{\mathcal{A}}{\equiv} X_1$ it follows that $C(X_0) \overset{\text{stat}}{\equiv} C(X_1)$. Combining this with $C(X_b) = (A \circ B)(X_b) = A(B(X_0))$ it follows that $A(B(X_0)) \overset{\text{stat}}{\equiv} A(B(X_1))$ for all $A \in \mathcal{A}$, as desired. □

**Exercise 2.5** Show that if $X_0 \overset{\text{comp}}{\equiv} X_1$ and $B$ is a poly-time algorithm, then $B(X_0) \overset{\text{comp}}{\equiv} B(X_1)$.

## 2.4 Interactive Systems

An interactive agent $A$ is a computational device which receives and sends messages on named ports and which holds an internal state.

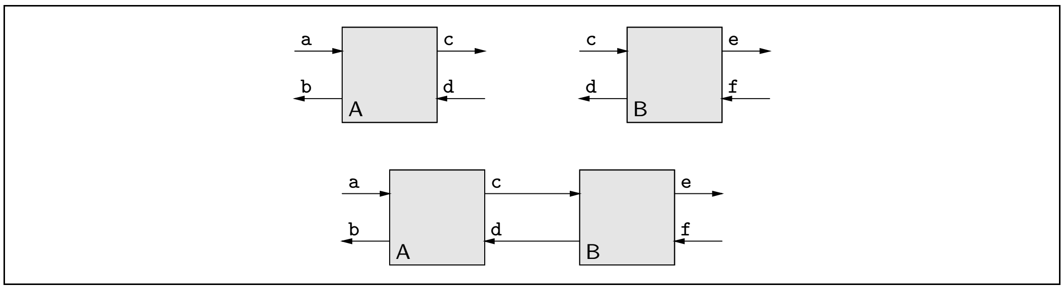


Figure 2.1: An interactive agent $A$ with $\text{In}(A) = \{\mathtt{a}, \mathtt{d}\}$ and $\text{Out}(A) = \{\mathtt{b}, \mathtt{c}\}$; An interactive agent $B$ with $\text{In}(B) = \{\mathtt{c}, \mathtt{f}\}$ and $\text{Out}(B) = \{\mathtt{d}, \mathtt{e}\}$; And the interactive system $\mathcal{IS} = A \diamond B$ with $\text{Out}(\mathcal{IS}) = \{\mathtt{a}, \mathtt{f}\}$ and $\text{In}(\mathcal{IS}) = \{\mathtt{b}, \mathtt{e}\}$.

More formally, an interactive agent is a tuple, $\mathsf{A} = (\text{In}, \text{Out}, \text{State}, \text{Msg}, T, \sigma_0)$, where In is a finite set of names of inports, Out is a finite set of names of outports, State is a set of possible states, Msg is a set of possible messages with at least $0, 1 \in \text{Msg}$ and $T$ is the transition algorithm: it is a randomized algorithm which takes an input $(\kappa, \sigma, I)$, where $\kappa \in \mathbb{N}$ is the security parameter, $\sigma \in \text{State}$ is the current state and $I \in \text{Msg}$ or $I \in \text{In}$ or $I = \mathtt{EOQ}$ or $I = \mathtt{SNT}$. If the agent just tried to read on one of its inports, then it receives $I \in \text{Msg}$ if there were messages ready and otherwise $I = \mathtt{EOQ}$. The input $I = \mathtt{SNT}$ is given to the agent when it just sent a message. The input $I \in \text{In}$ is for the first activation of an agent, to tell it on which port it was activated — this is called the activation port. The output of $T$ is of one of the following forms:

- $(\mathtt{send}, \mathtt{P}, m)$, where $\mathtt{P} \in \text{Out}$ is the port to send on and $m \in \text{Msg}$ is the message to send.
- $(\mathtt{read}, \mathtt{P})$, where $\mathtt{P} \in \text{In}$ is the port to read on.

- $(\texttt{return}, \texttt{RP})$, where $\texttt{RP} \in \text{Out}$ is the so-called return port.

The purpose of the return port is to specify which agent to activate next in a larger system. The agent connected to the return port will be activated next.

In the following we let $\text{In}(\mathsf{A})$ be the component In from $\mathsf{A}$, we let $\text{Out}(\mathsf{A})$ be the component Out from $\mathsf{A}$, and we let

$$\text{Ports}(\mathsf{A}) \overset{\text{def}}{=} \text{In}(\mathsf{A}) \cup \text{Out}(\mathsf{A}) \ .$$

Running an agent is called an activation. In each activation the agent can read on ports several times and send on ports several times and update its current state — the initial state is $\sigma_0$. At the end of the activation it then specifies a return port. See Algorithm Activate Agent for the details.

Algorithm Activate Agent

The activation of an agent $\mathsf{A}$ takes as input the value $\kappa$ of the security parameter, a current state $\sigma$, an activation port $\texttt{AP} \in \text{In}$ and a queue $\text{Q}_{\texttt{P}}$ for each $\texttt{P} \in \text{Ports}(\mathsf{A})$.

```
1. Let $I = \texttt{AP}$.
2. Let $(\sigma', c) \leftarrow T(\kappa, \sigma, I)$.
3. Update the current state: $\sigma \leftarrow \sigma'$.
4. Process the command $c$ as follows:
   send If $c = (\texttt{send}, \texttt{P}, m)$, then enter $m$ to the end of the queue $\text{Q}_{\texttt{P}}$, let $I \leftarrow \texttt{SNT}$
        and go to Step 2.
   read If $c = (\texttt{read}, \texttt{P})$, then let $I \leftarrow \texttt{EOQ}$ if the queue $\text{Q}_{\texttt{P}}$ is empty. Otherwise,
        remove the first element from $\text{Q}_{\texttt{P}}$ and let $I$ be this element. Then go to
        Step 2.
   return If $c = (\texttt{return}, \texttt{RP})$, then the activation returns. The output is the new
        current state $\sigma$ and the return port RP. A side effect of the activation is that
        the queues $\{\text{Q}_{\texttt{P}}\}_{\texttt{P} \in \text{Ports}(\mathsf{A})}$ were updated.
```

We describe agents it is useful with a little short hand. If we say that an agent *sends the activation token on P* we mean that it returns on $\texttt{P}$, i.e., it outputs $(\texttt{return}, \texttt{P})$. If we say that and agent *sends $m$ on P* and we don't say anything else we mean that it sends $m$ on $\texttt{P}$ and then sends the activation token on $\texttt{P}$. If we describe part of an agent by saying *on input $m$ on P do $A(m)$*, we mean that when the agent is activated, it reads on $\texttt{P}$ and if it gets back $m \neq \texttt{EOQ}$, then it executes $A(m)$, where $A(m)$ is some action which depends on $m$.

We are often interested in how efficient agents are to compute, for which the following definition is handy.

**Definition 2.7 (responsive, poly-time)** *We call an agent responsive if it holds for all contexts (i.e., all $\kappa \in \mathbb{N}$, all states $\sigma \in \text{State}$, all activation ports $AP \in \text{In}$ and all queues $\{\text{Q}_P\}_{P \in \text{Ports}(A)}$) that if we activate A in this context, then it will eventually return. We only allow responsive agents. We call an agent poly-responsive if it holds for all contexts that if we activate A in this context, then it will return after having executed at most a number of commands c which is polynomial in $\kappa$. We call an agent step-wise poly-time if $T$ can be computed in expected poly-time in $\kappa$. We call an agent poly-time if it is step-wise poly-time and poly-responsive.*

It is straight forward to see that the activation of a poly-time agent can be computed in expected polynomial time.

We connect a set of interactive agents to become an interactive system simply by connecting outports and inports with the same name. For this procedure to be well-defined we need that no two agents have identically named inports, and similarly for outports — we say that they are compatible.

**Definition 2.8** *We say that interactive agents $\mathsf{A}_1, \ldots, \mathsf{A}_n$ are* port compatible *if* $\forall i, j \in [n], j \neq i : \mathrm{In}(\mathsf{A}_i) \cap \mathrm{In}(\mathsf{A}_j) = \emptyset \wedge \mathrm{Out}(\mathsf{A}_i) \cap \mathrm{Out}(\mathsf{A}_j) = \emptyset$. *If $\mathsf{A}_1, \ldots, \mathsf{A}_n$ are port compatible we call $\mathcal{IS} = \{\mathsf{A}_1, \ldots, \mathsf{A}_n\}$ an* interactive system. *We say that interactive systems are* port compatible *if all their interactive agents are port compatible. If two interactive systems $\mathcal{IS}_1$ and $\mathcal{IS}_2$ are port compatible, then we define their* composition *to be $\mathcal{IS}_1 \diamond \mathcal{IS}_2 \overset{\text{def}}{=} \mathcal{IS}_1 \cup \mathcal{IS}_2$; Otherwise we let $\mathcal{IS}_1 \diamond \mathcal{IS}_2 = \perp$. For any interactive system $\mathcal{IS}$ we let $\mathcal{IS} \diamond \perp \overset{\text{def}}{=} \perp$ and $\perp \diamond \mathcal{IS} \overset{\text{def}}{=} \perp$.*

The following proposition is convenient when reasoning about composed interactive systems.

**Proposition 2.5** *The following holds for all interactive systems $\mathcal{IS}_1, \mathcal{IS}_2, \mathcal{IS}_3$:*

1. $\mathcal{IS}_1 \diamond \mathcal{IS}_2 = \mathcal{IS}_2 \diamond \mathcal{IS}_1$.
2. $(\mathcal{IS}_1 \diamond \mathcal{IS}_2) \diamond \mathcal{IS}_3 = \mathcal{IS}_1 \diamond (\mathcal{IS}_2 \diamond \mathcal{IS}_3)$.

For an interactive system $\mathcal{IS}$ and a port name `P` which is an inport of an agent in $\mathcal{IS}$, we use $\mathsf{A}_{\mathtt{P}}$ to denote the agent from $\mathcal{IS}$ which has an inport named `P`.

For an interactive system $\mathcal{IS}$ we let $\mathrm{In}(\mathcal{IS})$ be the inports which are not connected to outports and we let $\mathrm{Out}(\mathcal{IS})$ be the outports which are not connected to inports. I.e.,

$$\mathrm{In}(\mathcal{IS}) \overset{\text{def}}{=} (\cup_{\mathsf{A} \in \mathcal{IS}} \mathrm{In}(\mathsf{A})) \setminus (\cup_{\mathsf{A} \in \mathcal{IS}} \mathrm{Out}(\mathsf{A})) ,$$

$$\mathrm{Out}(\mathcal{IS}) \overset{\text{def}}{=} (\cup_{\mathsf{A} \in \mathcal{IS}} \mathrm{Out}(\mathsf{A})) \setminus (\cup_{\mathsf{A} \in \mathcal{IS}} \mathrm{In}(\mathsf{A})) .$$

We call $\mathrm{In}(\mathcal{IS})$ and $\mathrm{Out}(\mathcal{IS})$ the open ports of the system. We say that an interactive system is closed if $\mathrm{In}(\mathcal{IS}) = \emptyset$ and $\mathrm{Out}(\mathcal{IS}) = \emptyset$. We say that $\mathcal{IS}$ is executable if it is closed and there is some agent $\mathsf{A}$ in $\mathcal{IS}$ which has an inport named $\epsilon$ and an outport named $\epsilon$ — this is a technical requirement to have some well-defined initial activation port and final return port: the execution will begin by activating on the port $\epsilon$ and will end the first time an agent returns on the port $\epsilon$.

The execution is activation driven. In each step we activate an agent. Initially we activate the agent $\mathsf{A}_\epsilon$ with inport $\epsilon$, and we do it on the port $\epsilon$. After this, the next agent to be activated is the one which has an inport with the name that the previous agent specified as return port, and the agent is activated on that port. We think of this as some activation token `@` being passed around.

**Definition 2.9 (execution of interactive system)** *An interactive system is called* executable *if it is closed and contains an agent $\mathsf{A}$ with $\epsilon \in \mathrm{In}(\mathsf{A})$ and $\epsilon \in \mathrm{Out}(\mathsf{A})$. For an executable interactive system $\mathcal{IS}$ and a given value $\kappa \in \mathbb{N}$ of the security parameter, we use $\mathcal{IS}(\kappa)$ to denote the message $m$ output in Step 8 in Algorithm* Execute System *when executing $\mathcal{IS}$ as specified in Algorithm* Execute System. *If the execution does not stop, such that $m$ is not defined, we let $\mathcal{IS}(\kappa) = \infty$ for some reserved symbol $\infty$. Note that this makes $\mathcal{IS}(\kappa)$ a random variable with range $\mathrm{Msg} \cup \{\infty\}$, where $\mathrm{Msg}$ is the message space of $\mathsf{A}_\epsilon$. We sometimes use $\mathcal{IS}$ to denote the family of random variables $\{\mathcal{IS}(\kappa)\}_{\kappa \in \mathbb{N}}$.*

**Example 2.2** Let $\mathsf{A}$ be an interactive agent with $\mathrm{In}(\mathsf{A}) = \{\epsilon, \mathtt{1}\}$ and $\mathrm{Out}(\mathsf{A}) = \{\epsilon, \mathtt{1}\}$. On any input on $\epsilon$ or `1` it flips $r \in \{0,1\}^\ell$ for $\ell = \lceil \log_2(\kappa) \rceil$ and interprets it as an integer $R \in \mathbb{Z}_{2^\ell}$. If $R < \kappa$, then it sends $R$ on $\epsilon$. Otherwise, it sends `retry` on `1`. Let $\mathcal{IS} = \{\mathsf{A}\}$. Then $\mathcal{IS}$ is executable and $\Pr[\mathcal{IS}(\kappa) = \infty] = 0$, so $\mathcal{IS}(\kappa)$ is a random variable with range $\mathbb{Z}_\kappa$, and it is uniformly random on that range. △

Algorithm Execute System

1. Initialize the current state of all agents to be the initial state: For all $\mathsf{A} \in \mathcal{IS}$, do $\sigma_{\mathsf{A}} \leftarrow \sigma_0(\mathsf{A})$
2. Initialize an empty queue for all ports: For all $\mathtt{P} \in \cup_{\mathsf{A} \in \mathcal{IS}} \mathrm{Ports}(\mathsf{A})$, do $\mathrm{Q}_{\mathtt{P}} \leftarrow \epsilon$ .
3. Let the initially activation port be $\mathtt{AP}$.
4. Let $\mathsf{A}_{\mathtt{AP}}$ denote the agent with an inport named $\mathtt{AP}$.
5. Activate $\mathsf{A}_{\mathtt{AP}}$ on $(\kappa, \sigma_{\mathsf{A}}, \mathtt{AP}, \{\mathrm{Q}_{\mathtt{P}}\}_{\mathtt{P} \in \mathrm{Ports}(\mathsf{A})})$ — see Algorithm Activate Agent. This will update the queues $\{\mathrm{Q}_{\mathtt{P}}\}_{\mathtt{P} \in \mathrm{Ports}(\mathsf{A})}$ and make $\mathsf{A}_{\mathtt{AP}}$ output a return port $\mathtt{RP}$ and a new current state $\sigma'$.
6. Update the current state of $\mathsf{A}_{\mathtt{AP}}$: $\sigma_{\mathsf{A}} \leftarrow \sigma'$.
7. Update the activation port: $\mathtt{AP} \leftarrow \mathtt{RP}$.
8. If $\mathtt{RP} \neq \epsilon$, go to Step 4. If $\mathtt{RP} = \epsilon$, then the execution stops with output $m$, where $m = 0$ if $\mathrm{Q}_\epsilon$ is empty and were $m$ is the front element in $\mathrm{Q}_\epsilon$ otherwise.

**Example 2.3** Let $\mathsf{A}_1$ be an interactive agent with $\mathrm{In}(\mathsf{A}_1) = \{\epsilon, \mathtt{1}\}$ and $\mathrm{Out}(\mathsf{A}_1) = \{\epsilon, \mathtt{2}\}$. On any input on $\epsilon$ it samples a uniformly random bit $c \in \{0,1\}$ and sends $c$ on $\mathtt{2}$. On input $e \in \{0,1\}$ on $\mathtt{1}$ it sends $e$ on $\epsilon$. Let $\mathsf{A}_2$ be an interactive agent with $\mathrm{In}(\mathsf{A}_2) = \{\mathtt{2}\}$ and $\mathrm{Out}(\mathsf{A}_2) = \{\mathtt{1}\}$. On input $c \in \mathbb{N}$ on $\mathtt{2}$, it samples $d \leftarrow \{0,1\}$ and sends $c + d$ on $\mathtt{1}$. Let $\mathcal{IS} = \{\mathsf{A}_1, \mathsf{A}_2\}$. Then $\mathcal{IS}$ is executable and $\Pr[\mathcal{IS}(\kappa) = 0] = \frac{1}{4}$, $\Pr[\mathcal{IS}(\kappa) = 1] = \frac{1}{2}$, and $\Pr[\mathcal{IS}(\kappa) = \infty] = \frac{1}{4}$. $\triangle$

We will not only be interested in the execution of closed system. To allow to talk about the execution of an open system we introduce the notion of a closure.

**Definition 2.10 (closure)** *Let $\mathcal{IS}$ be an interactive system. A closure of $\mathcal{IS}$ is an interactive system $\mathcal{Z}$ where $\mathcal{IS} \diamond \mathcal{Z} \neq \perp$ and for which $\mathcal{IS} \diamond \mathcal{Z}$ is executable. This, in particular, means that* $\mathrm{In}(\mathcal{Z}) = \mathrm{Out}(\mathcal{IS})$ *and* $\mathrm{Out}(\mathcal{Z}) = \mathrm{In}(\mathcal{IS})$.

We reserve the port name $\epsilon$ for closures, i.e., we assume that normal interactive systems like $\mathcal{IS}$ do not use this port name for internal communication. The reason is that we want the closure $\mathcal{Z}$ to be the one that defines the output of the execution.

Above we implicitly defined the output of an execution to be $\infty$ if the execution runs forever. In some of our later definitions, in particular the definition of indistinguishability of interactive systems, handling systems which run forever is definitional cumbersome. Since we do not need to study such systems we will restrict our attention to systems which do not run forever. We call such systems responsive. Sometimes we also need to restrict systems to be poly-time.

**Definition 2.11 (responsive, poly-time)** *We call an executable interactive system $\mathcal{IS}$ responsive if* $\Pr[\mathcal{IS}(\kappa) = \infty] = 0$ *for all $\kappa$.*

*Let $\mathcal{IS}$ be an open interactive system not using the port name $\epsilon$. We call $\mathcal{IS}$ responsive if it holds for all closures $\mathcal{Z}$ of $\mathcal{IS}$ and all values $\kappa \in \mathbb{N}$ of the security parameter that whenever the activation token is turned over from an agent in $\mathcal{Z}$ to an agent in $\mathcal{IS}$, in the execution $(\mathcal{IS} \diamond \mathcal{Z})(\kappa)$, then with probability 1 the activation token will later be turned over from an agent in $\mathcal{IS}$ to an agent in $\mathcal{Z}$. In words, the probability that $\mathcal{IS}$ at some point keeps the activation token forever is 0.*

*We say that $\mathcal{IS}$ is poly-responsive if there exists a polynomial $\tau : \mathbb{N} \to \mathbb{N}$ such that it holds for all closures $\mathcal{Z}$ of $\mathcal{IS}$ and all values $\kappa \in \mathbb{N}$ of the security parameter that whenever the activation* *token is turned over from an agent in $\mathcal{Z}$ to an agent in $\mathcal{IS}$, in the execution $(\mathcal{IS} \diamond \mathcal{Z})(\kappa)$, then the expected number of activations of agents in $\mathcal{IS}$ until the activation token is again turned over from an agent in $\mathcal{IS}$ to an agent in $\mathcal{Z}$ is bounded by $\tau(\kappa)$. In other words, when $\mathcal{IS}$ is activated on an open inport, then the expected number of internal agent activations before it returns the activation token on an open outport is polynomial.*

*We say that $\mathcal{IS}$ is a* poly-time *interactive system if all agents are poly-time and $\mathcal{IS}$ is poly-responsive.*

The main important fact to remember about this definition is the following:

**Proposition 2.6** *A poly-time interactive system $\mathcal{IS}$ can be executed in expected polynomial time. More precisely, for any activation of $\mathcal{IS}(\kappa)$ the expected running time spent until the activation token is returned on an open port grows at most polynomially in $\kappa$.*

**Exercise 2.6** Prove Proposition 2.6. The crucial step is the following:

Let $B(\kappa, \cdot)$ be a stateful algorithm taking the security parameter as input plus an additional input $y$. Stateful means, as usual, that $B$ keeps a state that is stored between calls to $B$ and the output depends on the state as well as the input. Assume that $B$ runs in expected polynomial time. I.e., there exists a polynomial $q(\kappa)$ such that the expected running time of $B(\kappa, y)$ is bounded by $q(\kappa)$ for all states and inputs $y$.

Let $A(\kappa, \cdot)$ be an algorithm taking the security parameter as input plus an additional input $x$, and assume that $A(\kappa, \cdot)$ uses $B(\kappa, \cdot)$ as sub-routine, i.e., $A(\kappa, x)$ might call $B(\kappa, y)$, and it might do so several times and on different inputs $y$. Assume that $A(\kappa, \cdot)$ calls $B(\kappa, \cdot)$ an expected polynomial number of times. In other words, there exists a polynomial $p(\kappa)$ such that the expected number of times that $A(\kappa, x)$ calls $B(\kappa, \cdot)$ is bounded by $p(\kappa)$ for all inputs $x$. Let $t(\kappa, x)$ be the expected value of the sum of the running times of all the calls that $A(\kappa, x)$ makes to $B(\kappa, \cdot)$.

As a first step in your proof, you should show that there exists a polynomial $t(\kappa)$ such that $t(\kappa, x) \leq t(\kappa)$ for all inputs $x$.

### 2.4.1 Indistinguishable Interacting Systems

We say that two interactive systems are (behaviorly) indistinguishable if one cannot tell the difference between them by sending and receiving messages over the open ports of the systems. For this purpose we work with the notion of an environment, which is just an interactive system which closes the system and makes it executable. We think of the execution as the environment playing with the system over the open ports. Its job is then to guess which system it is connected to. The closure therefore acts as a distinguisher. For this reason we require that it outputs a bit $c$, which is its guess at which system it is playing with.

**Definition 2.12** *Let $\mathcal{IS}_0$ and $\mathcal{IS}_1$ be responsive interactive systems with $\mathrm{In}(\mathcal{IS}_0) = \mathrm{In}(\mathcal{IS}_1)$ and $\mathrm{Out}(\mathcal{IS}_0) = \mathrm{Out}(\mathcal{IS}_1)$. We call an interactive system $\mathcal{Z}$ an* environment *for $\mathcal{IS}_0$ and $\mathcal{IS}_1$ if $\mathcal{Z}$ is a closure of both $\mathcal{IS}_0$ and $\mathcal{IS}_1$ and $\mathcal{IS}_0 \diamond \mathcal{Z}$ and $\mathcal{IS}_1 \diamond \mathcal{Z}$ are responsive.*[1] *We also require that when $\mathcal{Z}$ produces an output on the port $\epsilon$, then it is a bit $c$. The responsiveness, and the fact that $\mathcal{Z}$ outputs a bit on $\epsilon$ means that $(\mathcal{IS}_0 \diamond \mathcal{Z})(\kappa)$ and $(\mathcal{IS}_1 \diamond \mathcal{Z})(\kappa)$ are random variables with range $\{0, 1\}$.*

- *We say that the systems are perfectly indistinguishable, written $\mathcal{IS}_0 \overset{\text{perf}}{\equiv} \mathcal{IS}_1$, if it holds that $\{(\mathcal{Z} \diamond \mathcal{IS}_0)(\kappa)\}_{\kappa \in \mathbb{N}} \overset{\text{perf}}{\equiv} \{(\mathcal{Z} \diamond \mathcal{IS}_1)(\kappa)\}_{\kappa \in \mathbb{N}}$ for all environments $\mathcal{Z}$ for $\mathcal{IS}_0$ and $\mathcal{IS}_1$.*

[1] Since we require that $\mathcal{IS}_0$ and $\mathcal{IS}_1$ are responsive, the requirement that $\mathcal{IS}_0 \diamond \mathcal{Z}$ and $\mathcal{IS}_1 \diamond \mathcal{Z}$ are responsive just requires that $\mathcal{Z}$ does not make an infinite number of internal activations.

- *We say that the systems are statistically indistinguishable, written* $\mathcal{IS}_0 \overset{\text{stat}}{\equiv} \mathcal{IS}_1$*, if it holds that* $\{(\mathcal{Z} \diamond \mathcal{IS}_0)(\kappa)\}_{\kappa\in\mathbb{N}} \overset{\text{stat}}{\equiv} \{(\mathcal{Z} \diamond \mathcal{IS}_1)(\kappa)\}_{\kappa\in\mathbb{N}}$ *for all environments* $\mathcal{Z}$ *for* $\mathcal{IS}_0$ *and* $\mathcal{IS}_1$*, such that* $\mathcal{Z}$ *makes a polynomial (in* $\kappa$*) number of activations of* $\mathcal{IS}_0$ *(*$\mathcal{IS}_1$*).*
- *For a class of interactive systems* Env *we say that* $\mathcal{IS}_0$ *is indistinguishable from* $\mathcal{IS}_1$ *for* Env *(written* $\mathcal{IS}_0 \overset{\text{Env}}{\equiv} \mathcal{IS}_1$*) if* $\{(\mathcal{Z} \diamond \mathcal{IS}_0)(\kappa)\}_{\kappa\in\mathbb{N}} \overset{\text{stat}}{\equiv} \{(\mathcal{Z} \diamond \mathcal{IS}_1)(\kappa)\}_{\kappa\in\mathbb{N}}$ *for all* $\mathcal{Z} \in \text{Env}$ *for which* $\mathcal{Z}$ *is an environment for* $\mathcal{IS}_0$ *and* $\mathcal{IS}_1$*.*
- *We say that* $\mathcal{IS}_0 \overset{\text{comp}}{\equiv} \mathcal{IS}_1$ *if* $\mathcal{IS}_0 \overset{\text{Env}}{\equiv} \mathcal{IS}_1$ *for the class* Env *of poly-time interactive systems.*

The case of statistical indistinguishability is meant to capture systems $\mathcal{IS}_0, \mathcal{IS}_1$ that behave "almost" in the same way, for instance where the behavior is exactly the same, except if some event $E$ occurs, where $\Pr[E]$ is negligible. To capture such cases, we need the restriction to a polynomial number of activations. Otherwise an unbounded environment could distinguish easily: it would just keep activating the system until the "error event" occurs.

For a class of interactive systems Env we let $\text{Env}^\diamond$ be the class of interactive systems $\mathcal{IS}$ for which $\mathcal{Z} \diamond \mathcal{IS} \in \text{Env}$ whenever $\mathcal{Z} \in \text{Env}$ and $\mathcal{Z}$ and $\mathcal{IS}$ are port compatible.

**Proposition 2.7** *The following holds for all interactive systems* $\mathcal{IS}_0, \mathcal{IS}_1, \mathcal{IS}_2$ *and all classes of environments* Env*.*

1. $\mathcal{IS}_0 \overset{\text{Env}}{\equiv} \mathcal{IS}_0$*.*
2. *If* $\mathcal{IS}_0 \overset{\text{Env}}{\equiv} \mathcal{IS}_1$*, then* $\mathcal{IS}_1 \overset{\text{Env}}{\equiv} \mathcal{IS}_0$*.*
3. *If* $\mathcal{IS}_0 \overset{\text{Env}}{\equiv} \mathcal{IS}_1$ *and* $\mathcal{IS}_1 \overset{\text{Env}}{\equiv} \mathcal{IS}_2$*, then* $\mathcal{IS}_0 \overset{\text{Env}}{\equiv} \mathcal{IS}_2$*.*
4. *If* $\mathcal{IS}_0 \overset{\text{Env}}{\equiv} \mathcal{IS}_1$ *and* $\mathcal{IS}_2 \in \text{Env}^\diamond$*, then* $\mathcal{IS}_2 \diamond \mathcal{IS}_0 \overset{\text{Env}}{\equiv} \mathcal{IS}_2 \diamond \mathcal{IS}_1$*.*
5. *If* $\mathcal{IS}_0 \overset{\text{Env}_1}{\equiv} \mathcal{IS}_1$ *and* $\mathcal{IS}_2 \diamond \text{Env}_2 \subseteq \text{Env}_1$*, then* $\mathcal{IS}_2 \diamond \mathcal{IS}_0 \overset{\text{Env}_2}{\equiv} \mathcal{IS}_2 \diamond \mathcal{IS}_1$*.*

*The same properties hold for* $\overset{\text{perf}}{\equiv}$*,* $\overset{\text{stat}}{\equiv}$ *and* $\overset{\text{comp}}{\equiv}$*.*

*Proof* The first three properties are straight forward. To show that $\mathcal{IS}_2 \diamond \mathcal{IS}_0 \overset{\text{Env}}{\equiv} \mathcal{IS}_2 \diamond \mathcal{IS}_1$, we have to show that $\mathcal{Z} \diamond (\mathcal{IS}_2 \diamond \mathcal{IS}_0) \overset{\text{stat}}{\equiv} \mathcal{Z} \diamond (\mathcal{IS}_2 \diamond \mathcal{IS}_1)$ for all $\mathcal{Z} \in \text{Env}$ which are environments for $\mathcal{IS}_2 \diamond \mathcal{IS}_0$ and $\mathcal{IS}_2 \diamond \mathcal{IS}_1$. Since $\mathcal{Z} \in \text{Env}$ and $\mathcal{IS}_2 \in \text{Env}^\diamond$, we have that $\mathcal{Z} \diamond \mathcal{IS}_2 \in \text{Env}$. From $\mathcal{Z} \diamond \mathcal{IS}_2 \in \text{Env}$ and $\mathcal{IS}_0 \overset{\text{Env}}{\equiv} \mathcal{IS}_1$ it follows that $(\mathcal{Z} \diamond \mathcal{IS}_2) \diamond \mathcal{IS}_0 \overset{\text{stat}}{\equiv} (\mathcal{Z} \diamond \mathcal{IS}_2) \diamond \mathcal{IS}_1$. The claim then follows from $(\mathcal{Z} \diamond \mathcal{IS}_2) \diamond \mathcal{IS}_b = \mathcal{Z} \diamond (\mathcal{IS}_2 \diamond \mathcal{IS}_b)$ for $b = 0, 1$. The proof of the last property follows in the same manner. $\square$

We are particularly interested in Property 4 in the above proposition. It basically says that if two interactive systems are hard to distinguish, then this also holds if we use them in some context $\mathcal{IS}_2$. I.e., as long as the context $\mathcal{IS}_2$ does not perform any computation that the class of distinguishers that we consider cannot perform on its own. This is very useful in comparing systems like $\mathcal{IS}_2 \diamond \mathcal{IS}_0$ and $\mathcal{IS}_2 \diamond \mathcal{IS}_1$, as we only have to compare their non-common parties, $\mathcal{IS}_0$ and $\mathcal{IS}_1$. This is sometimes called modularity of indistinguishability and sometimes called composability of indistinguishability.

In Property 4 we showed composability only for $\mathcal{IS}_2 \in \text{Env}^\diamond$. This is, in fact, the best general result we can hope for. If, e.g., $\mathcal{Z}$ is the class of poly-time environments and $\mathcal{IS}_2$ performs exponential time computations, then it might happen that $\mathcal{IS}_0 \overset{\text{Env}}{\equiv} \mathcal{IS}_1$ but $\mathcal{IS}_2 \diamond \mathcal{IS}_0 \overset{\text{Env}}{\not\equiv} \mathcal{IS}_2 \diamond \mathcal{IS}_1$: it might be that $\mathcal{IS}_0$ and $\mathcal{IS}_1$ are indistinguishable to poly-time environments due to the use of cryptography but that $\mathcal{IS}_2$ breaks this cryptography using its exponential time computing power. The reason why this does not contradict the proposition is that when Env is the class of poly-time environments and $\mathcal{IS}_2$ is an exponential time system, then $\mathcal{IS}_2 \notin \text{Env}^\diamond$.[2]

[2]Assume namely that $\mathcal{IS}_2 \in \text{Env}^\diamond$. By definition, this means that $\mathcal{IS}_2 \diamond \mathcal{IS} \in \text{Env}$ for all $\mathcal{IS} \in \text{Env}$. Since the empty interactive system $\mathcal{IS} = \emptyset$ is clear poly-time we have that $\emptyset \in \text{Env}$, which implies that $\mathcal{IS}_2 \diamond \emptyset = \mathcal{IS}_2 \in \text{Env}$. This contradicts the premises that Env is the poly-time systems and that $\mathcal{IS}_2$ is an exponential time system.

**Example 2.4** Let $A$ be an interactive agent with $\text{In}(A) = \{\texttt{a}, \texttt{d}\}$ and $\text{Out}(A) = \{\texttt{b}, \texttt{c}\}$ and the following behavior: On input $x \in \mathbb{Z}$ on `a` output $x + 1$ on `c`, and on input $x \in \mathbb{Z}$ on `d` output $x+1$ on `b`. Let $B$ be an interactive agent with $\text{In}(B) = \{\texttt{c}\}$ and $\text{Out}(B) = \{\texttt{d}\}$ and the following behavior: On input $x \in \mathbb{Z}$ on `c` output $2x$ on `d`. Let $C$ be an interactive agent with $\text{In}(C) = \{\texttt{a}\}$ and $\text{Out}(C) = \{\texttt{b}\}$ and the following behavior: On input $x \in \mathbb{Z}$ on `a` output $2x + 3$ on `b`. Let $\mathcal{IS}_0 = \{A, B\}$ and let $\mathcal{IS}_1 = \{C\}$. Then $\mathcal{IS}_0 \overset{\text{perf}}{\equiv} \mathcal{IS}_1$. △

**Exercise 2.7** Prove Property 5 in Proposition 2.7.

## 2.5 Public-Key Cryptosystems

The following section precisely defines the security of public key cryptosystems, and we will use such systems later, but at the same time gives an example of how interactive systems and the notion of indistinguishability of interactive systems can be used to make precise definitions.

A public-key cryptosystem consists of three algorithms $(G, E, D)$ called the key generator, the encryption algorithm and the decryption algorithm, respectively.

We define security of a public-key cryptosystem by comparing two interactive agents. For $b = 0, 1$, let $\mathsf{A}_b$ be the following interactive agent:

- $\text{In}(\mathsf{A}_b) = \{\texttt{keygen}, \texttt{msgs}, \texttt{dec}\}$.
- $\text{Out}(\mathsf{A}_b) = \{\texttt{pk}, \texttt{target}, \texttt{plaintext}\}$.
- It behaves as follows:
  - The first time a message is input on `keygen` it samples $(pk, sk) \leftarrow G(\kappa)$ and outputs $pk$ on `pk`. It ignores all subsequent messages on `keygen`.
  - On the first message of the form $(m_0, m_1)$ with $|m_0| = |m_1|$ on `msgs` after $(pk, sk)$ has been sampled, it samples a target ciphertext $c^* \leftarrow E_{pk}(m_b)$ and returns $c^*$ on `target`. It ignores all subsequent messages on `msgs`.
  - On each message $c$ on `dec`, after $(pk, sk)$ has been sampled, it returns $m \leftarrow D_{sk}(c)$ on `plaintext`. If $c^*$ has been defined, then it ignores inputs with $c = c^*$.

The following definition captures the standard notions of indistinguishability under chosen ciphertext attack and indistinguishability under chosen plaintext attack.

**Definition 2.13** *We say that $(G, E, D)$ is IND-CCA secure if $\mathsf{A}_0 \overset{\text{comp}}{\equiv} \mathsf{A}_1$. We say that $(G, E, D)$ is IND-CPA secure if $\mathsf{A}'_0 \overset{\text{comp}}{\equiv} \mathsf{A}'_1$, where $\mathsf{A}'_b$ is $\mathsf{A}_b$ with the ports `dec` and `plaintext` removed.*

The notion of IND-CPA (which is sometimes called semantic security) says that it is impossible to distinguish encryptions of different messages in poly-time, even if you get to pick the messages yourself, and even if you get to pick the messages after seeing the public key. The notion of IND-CCA says that the task remains hard even if you get access to a decryption oracle, unless of course you decrypt the ciphertext you were given as challenge (which would make the task trivial).

It is possible to build cryptosystems which are IND-CPA secure and IND-CCA secure under many assumptions, e.g., the RSA assumption and the discrete logarithm assumption.

# Chapter 3

# MPC Protocols with Passive Security

## Contents

## 3.1 Introduction

In Chapter 1, we gave an introduction to the multiparty computation problem and explained what it intuitively means for a protocol to compute a given function securely, namely, we want a protocol that is correct and private. However, we only considered 3 players, we only argued that a single player does not learn more than he is supposed to, and finally we assumed that all players would follow the protocol.

In this chapter, we will consider a more general solution to multiparty computation, where we remove the first two restrictions: we will consider any number $n \geq 3$ of players, and we will be able to show that as long as at most $t < n/2$ of the players go together after the protocol is executed and pool all their information, they will learn nothing more than their own inputs and the outputs they were supposed to receive, even if their computing power is unbounded. We will still assume, however, that all players follow the protocol. This is known as *semi-honest* or *passive* security.

To argue security in this chapter, we will use a somewhat weak but very simple definition that only makes sense for semi-honest security. We then extend this in the next chapter to a fully general model of what protocols are, and what security means.

Throughout this chapter, we will assume that each pair of players can communicate using a perfectly secure channel, so if two players exchange data, a third player has no information at all about what is sent. Such channels might be available because of physical circumstances, or we can implement them relative to a computational assumption using cryptography. For more details on this, see Chapter 7.

We end the chapter by showing that the bound $t < n/2$ is optimal: if $t \geq n/2$, then some functions cannot be computed securely.

## 3.2 Secret Sharing

Our main tool to build the protocol will be so called secret sharing schemes. The theory of secret sharing schemes is a large and interesting field in its own right with many applications to MPC, and we look at this in more detail in Chapter 10, where a formal definition of the notion as well as a treatment of the theory of secret-sharing can be found.

Here we concentrate on a particular example scheme that will be sufficient for our purposes in this chapter, namely Shamir's secret sharing scheme. This scheme is based on polynomials over a finite field $\mathbb{F}$. The only necessary restriction on $\mathbb{F}$ is that $|\mathbb{F}| > n$, but we will assume for concreteness and simplicity that $\mathbb{F} = \mathbb{Z}_p$ for some prime $p > n$.

A value $s \in \mathbb{F}$ is *shared* by choosing a random polynomial $f_s(\mathtt{X}) \in \mathbb{F}[\mathtt{X}]$ of degree at most $t$ such that $f_s(0) = s$. And then sending privately to player $P_j$ the share $s_j = f_s(j)$. The basic facts about this method are that any set of $t$ or fewer shares contain no information on $s$, whereas it can easily be reconstructed from any $t+1$ or more shares. Both of these facts are proved using Lagrange interpolation.

#### Lagrange Interpolation

If $h(\mathtt{X})$ is a polynomial over $\mathbb{F}$ of degree at most $l$ and if $C$ is a subset of $\mathbb{F}$ with $|C| = l+1$, then

$$h(\mathtt{X}) = \sum_{i \in C} h(i)\delta_i(\mathtt{X}) ,$$

where $\delta_i(\mathtt{X})$ is the degree $l$ polynomial such that, for all $i, j \in C$, $\delta_i(j) = 0$ if $i \neq j$ and $\delta_i(j) = 1$ if $i = j$. In other words,

$$\delta_i(\mathtt{X}) = \prod_{j \in C, j \neq i} \frac{\mathtt{X} - j}{i - j} .$$

We briefly recall why this holds. Since each $\delta_i(\mathtt{X})$ is a product of $l$ monomials, it is a polynomial of degree at most $l$. Therefore the right hand side $\sum_{i \in C} h(i)\delta_i(\mathtt{X})$ is a polynomial of degree at most $l$ that on input $i$ evaluates to $h(i)$ for $i \in C$. Therefore, $h(\mathtt{X}) - \sum_{i \in C} h(i)\delta_i(\mathtt{X})$ is 0 on all points in $C$. Since $|C| > l$ and only the zero-polynomial has more zeroes than its degree (in a field), it follows that $h(\mathtt{X}) - \sum_{i \in C} h(i)\delta_i(\mathtt{X})$ is the zero-polynomial, from which it follows that $h(\mathtt{X}) = \sum_{i \in C} h(i)\delta_i(\mathtt{X})$.

Using the same argument, one easily sees that the Lagrange interpolation works, even if no polynomial is predefined. Given any set of values $\{y_i \in \mathbb{F} | \ i \in C\}$, $|C| = l+1$, we can construct a polynomial $h$ with $h(i) = y_i$ of degree at most $l$ as

$$h(\mathtt{X}) = \sum_{i \in C} y_i \delta_i(\mathtt{X}) .$$

A consequence of Lagrange interpolation is that there exist easily computable values $\mathbf{r} = (r_1, ..., r_n)$, such that

$$h(0) = \sum_{i=1}^{n} r_i h(i) \tag{3.1}$$

for all polynomials $h(X)$ of degree at most $n-1$. Namely, $r_i = \delta_i(0)$. We call $(r_1, ..., r_n)$ the recombination vector. Note that $\delta_i(\mathtt{X})$ does not depend on $h(\mathtt{X})$, so neither does $\delta_i(0)$. Hence, the *same* recombination vector $\mathbf{r}$ works for all $h(\mathtt{X})$. It is a public piece of information that all players can compute.

A final consequence is that for all secrets $s \in \mathbb{F}$ and all $C \subset \mathbb{F}$ with $|C| = t$ and $0 \notin C$, if we sample a uniformly random $f$ of degree $\leq t$ and with $f(0) = s$ then the distribution of the $t$ shares

$$(f(i))_{i \in C}$$

is the uniform distribution on $\mathbb{F}^t$. Since the uniform distribution on $\mathbb{F}^t$ clearly is independent of $s$, it in particular follows that given only $t$ shares one gets no information on the secret.

One way to see that any $t$ shares are uniformly distributed is as follows: One way to sample a polynomial for sharing of a secret $s$ is to sample a uniformly random $a = (a_1, \ldots, a_t) \in \mathbb{F}^t$ and let $f_a(\mathtt{X}) = s + \sum_{j=1}^{t} a_j \mathtt{X}^t$ (as clearly $f_a(0) = s$.) For a fixed $s$ and fixed $C$ as above this defines an evaluation map from $\mathbb{F}^t$ to $\mathbb{F}^t$ by mapping $a = (a_1, \ldots, a_t)$ to $(f_a(i))_{i \in C}$. This map is invertible. Namely, given any $(y_i)_{i \in C} \in \mathbb{F}^t$, we know that we seek $f_a(\mathtt{X})$ with $f_a(i) = y_i$ for $i \in C$. We furthermore know that $f_a(0) = s$. So, we know $f_a(\mathtt{X})$ on $t+1$ points, which allows to compute $f_a(\mathtt{X})$ and $a \in \mathbb{F}^t$, using Lagrange interpolation. So, the evaluation map is invertible. Any invertible map from $\mathbb{F}^t$ to $\mathbb{F}^t$ maps the uniform distribution on $\mathbb{F}^t$ to the uniform distribution on $\mathbb{F}^t$.

**Example Computations**

We look at an example. Assume that we have five parties $\mathsf{P}_1, \ldots, \mathsf{P}_5$ and that we want to tolerate $t = 2$ corrupted parties. Assume that we work in $\mathbb{F} = \mathbb{Z}_{11}$ and want to share $s = 7$. We pick $a_1, a_2 \in_{\mathrm{R}} \mathbb{F}$ uniformly at random, say they become $a_1 = 4$ and $a_2 = 1$, and then we define

$$h(\mathtt{X}) = s + a_1\mathtt{X} + a_2\mathtt{X}^2 = 7 + 4\mathtt{X} + \mathtt{X}^2 \ . \tag{3.2}$$

Then we compute $s_1 = h(1) = 7 + 4 + 1 \bmod 11 = 1$, $s_2 = h(2) = 19 \bmod 11 = 8$, $s_3 = h(3) = 6$, $s_4 = h(4) = 6$, $s_5 = h(5) = 8$. So, the sharing is

$$[s] = (1, 8, 6, 6, 8) \ .$$

We send $s_i$ securely to $\mathsf{P}_i$.

Assume now that someone is given just the shares $s_3, s_4, s_5$. Since $3 > 2$, she can use Lagrange interpolation to compute the secret.

We first compute

$$\delta_3(\mathtt{X}) = \prod_{j=4,5} \frac{\mathtt{X} - j}{3 - j} = \frac{(\mathtt{X} - 4)(\mathtt{X} - 5)}{(3 - 4)(3 - 5)} = (\mathtt{X}^2 - 9\mathtt{X} + 20)((3 - 4)(3 - 5))^{-1} \pmod{11} \ .$$

We have that $(3 - 4)(3 - 5) = 2$ and $2 \cdot 6 \bmod 11 = 1$, so $((3 - 4)(3 - 5))^{-1} \bmod 11 = 6$. So,

$$\delta_3(\mathtt{X}) = (\mathtt{X}^2 - 9\mathtt{X} + 20)6 = (\mathtt{X}^2 + 2\mathtt{X} + 9)6 = 6\mathtt{X}^2 + 12\mathtt{X} + 54 = 6\mathtt{X}^2 + \mathtt{X} + 10 \pmod{11} \ .$$

We check that

$$\begin{aligned}
\delta_3(3) &= 6 \cdot 3^2 + 3 + 10 = 67 = 1 \pmod{11} \ , \\
\delta_3(4) &= 6 \cdot 4^2 + 4 + 10 = 110 = 0 \pmod{11} \ , \\
\delta_3(5) &= 6 \cdot 5^2 + 5 + 10 = 165 = 0 \pmod{11} \ ,
\end{aligned}$$

as it should be.

We then compute

$$\begin{aligned}
\delta_4(\mathtt{X}) = \prod_{j=3,5} \frac{\mathtt{X} - j}{4 - j} &= \frac{(\mathtt{X} - 3)(\mathtt{X} - 5)}{(4 - 3)(4 - 5)} \\
&= (\mathtt{X}^2 - 8\mathtt{X} + 15)(-1)^{-1} \\
&= (\mathtt{X}^2 + 3\mathtt{X} + 4)10 \\
&= 10\mathtt{X}^2 + 8\mathtt{X} + 7 \ .
\end{aligned}$$

We can check that $\delta_4(3) = 121 = 0 \pmod{11}$, $\delta_4(4) = 199 = 1 \pmod{11}$, and $\delta_4(5) = 297 = 0 \pmod{11}$.

We then compute

$$\begin{aligned}\delta_5(\mathtt{X}) = \prod_{j=3,4} \frac{\mathtt{X}-j}{5-j} &= \frac{(\mathtt{X}-3)(\mathtt{X}-4)}{(5-3)(5-4)} \\ &= (\mathtt{X}^2 - 7\mathtt{X} + 12)(2)^{-1} \\ &= (\mathtt{X}^2 + 4\mathtt{X} + 1)6 \\ &= 6\mathtt{X}^2 + 2\mathtt{X} + 6 \ .\end{aligned}$$

We can check that $\delta_5(3) = 66 = 0 \pmod{11}$, $\delta_5(4) = 110 = 0 \pmod{11}$, and $\delta_5(5) = 166 = 1 \pmod{11}$.

It is now clear that if for any $s_3, s_4, s_5$ we let

$$h(\mathtt{X}) = s_3 \cdot \delta_3(\mathtt{X}) + s_4 \cdot \delta_4(\mathtt{X}) + s_5 \cdot \delta_5(\mathtt{X}) \ ,$$

then $h(3) = s_3 \cdot 1 + s_4 \cdot 0 + s_5 \cdot 0 = s_3$, $h(4) = s_3 \cdot 0 + s_4 \cdot 1 + s_5 \cdot 0 = s_4$ and $h(5) = s_3 \cdot 0 + s_4 \cdot 0 + s_5 \cdot 1 = s_5$, which implies that if $s_3 = f(3)$, $s_4 = f(4)$ and $s_5 = f(5)$ for some quadratic polynomial, then $h(\mathtt{X}) = f(\mathtt{X})$. This allows to compute $h(\mathtt{X})$ from the three shares.
More concretely, notice that

$$\begin{aligned}h(\mathtt{X}) &= s_3\delta_3(\mathtt{X}) + s_4\delta_4(\mathtt{X}) + s_5\delta_5(\mathtt{X}) \\ &= (6s_3 + 10s_4 + 6s_5)\mathtt{X}^2 + (s_3 + 8s_4 + 2s_5)\mathtt{X} + (10s_3 + 7s_4 + 6s_5) \ .\end{aligned}$$

Since we consider $h(\mathtt{X})$ of the form $h(\mathtt{X}) = s + a_1\mathtt{X} + a_2\mathtt{X}^2$, we have that

$$\begin{aligned}s &= 10s_3 + 7s_4 + 6s_5 \bmod 11 \\ a_1 &= s_3 + 8s_4 + 2s_5 \bmod 11 \\ a_2 &= 6s_3 + 10s_4 + 6s_5 \bmod 11 \ ,\end{aligned}$$

which is then the general formula for computing $h(\mathtt{X})$ from the three shares $s_3 = h(3), s_4 = h(4), s_5 = h(5)$.

In our concrete example we had the shares $s_3 = 6$, $s_4 = 6$ and $s_5 = 8$. If we plug this in we get

$$\begin{aligned}s &= 10 \cdot 6 + 7 \cdot 6 + 6 \cdot 8 \bmod 11 = 150 \bmod 11 = 7 \\ a_1 &= 6 + 8 \cdot 6 + 2 \cdot 8 \bmod 11 = 70 \bmod 11 = 4 \\ a_2 &= 6 \cdot 6 + 10 \cdot 6 + 6 \cdot 8 \bmod 11 = 144 \bmod 11 = 1 \ ,\end{aligned}$$

which gives exactly the polynomial in (3.2).

If we had only been interested in finding the secret $s$ and not the entire polynomial we would only need the equation $s = 10s_3 + 7s_4 + 6s_5 \bmod 11$. We see that $r = (10, 7, 6)$ is the recombination vector for finding $h(0)$ from $h(3), h(4), h(5)$ when $h(\mathtt{X})$ is a polynomial of degree at most 2.

## 3.3 A Passive Secure Protocol

### 3.3.1 Arithmetic Circuits

We will present a protocol which can securely evaluate a function with inputs and outputs in a finite field $\mathbb{F}$. For notational convenience we construct the protocol for the case where each party has exactly one input and one output from $\mathbb{F}$. I.e., $f : \mathbb{F}^n \to \mathbb{F}^n, (x_1, \ldots, x_n) \to (y_1, \ldots, y_n)$. We assume that the mapping $(y_1, \ldots, y_n) = f(x_1, \ldots, x_n)$ is described using an arithmetic circuit.

More formally, such a circuit is an acyclic directed graph, where each node is called a gate, the edges are called wires. Each gate has at most 2 in-coming wires. There are $n$ input gates with no in-coming and 1 out-coming wire, each such gate is labeled by $i$ for the player $\mathsf{P}_i$ who is going to supply the secret input value $x_i$ for that gate. Then there are a number of internal addition and multiplication gates, with 2 input wires and any number of output wires, these add or multiply their two inputs. We also have multiply-by-constant gates, they have 1 input wire and any number of output wires, each such gate is labeled by a constant $\alpha \in \mathbb{F}$ and does multiplication by $\alpha$. Finally there is for each $\mathsf{P}_i$ exactly one output gate labeled by $i$, with 1 input wire and no output wires. The value eventually assigned to the input wire of this gate is going to be $y_i$.

Evaluating a circuit can be done as follows: we assume that the input values have been specified initially, and we think of this as assigning the value $x_i$ to the wire(s) coming out of the input gate labeled $i$. We assume that the gates have been numbered in some (arbitrary) way. We then take the first gate for which values have been assigned to all its input wires, compute the output value and assign it to the output wire(s) of the gate. Repeat this until all wires have had values assigned. This then also defines the output values.

The order in which we visit the gates in this procedure will be called the computational ordering in the following.

Considering arithmetic circuits is without loss of generality: It is well known that any function that is feasible to compute at all can be specified as a polynomial-size Boolean circuit using *and* and *negation*. But any such circuit can be simulated by operations in $\mathbb{F}$: Boolean values `true` or `false` can be encoded as 1 resp. 0. Then the *negation* of bit $b$ is $1 - b$, and the *and* of bits $b$ and $b'$ is $b \cdot b'$.

**Exercise 3.1** Assume that you are given a protocol that securely computes any function $f : \mathbb{F}^n \to \mathbb{F}^n$ given by an arithmetic circuit. Show how it can be used to securely compute any function $g : \{0,1\}^n \to \{0,1\}^n$, where each party has a bit as input and $g$ can be any poly-time computable function. Assume that you have a poly-sized Boolean circuit for $g$. We argued how to do that above, but the solution only works if players select inputs correctly. If parties may not do this, there is the problem when the Boolean circuit is coded as an arithmetic circuit: it is important that all parties input 0 or 1. If players may input any $x_i \in \mathbb{F}$ we may be in trouble: Consider a case with three parties, each with an input $x_i \in \{0,1\}$. Assume that $y_1 = (1 - x_1)x_2 + x_1 x_3$. If $x_1 = 0$, then $y_1 = x_2$, and if $x_1 = 1$, then $y_1 = x_3$. I.e., $\mathsf{P}_1$ can choose to learn either $x_2$ or $x_3$, but not both.

1. Argue that a cheating $P_i$ which inputs $x_1 \notin \{0,1\}$ can learn both $x_2$ and $x_3$.

2. Give a general construction which prevents this type of attack. [Hint: Assume that $\mathbb{F}$ is small and try to map all possible inputs $x_i \in \mathbb{F}$ to an input $x'_i \in \{0,1\}$ and then do the actual computation on $(x'_1, \ldots, x'_n)$.]

### 3.3.2 The Protocol

We recall that we will give a protocol for the scenario where there are secure channels between all parties. We assume that some subset $C$ of the players, of size at most $t$, will go together after the protocol and try to learn as much as they can from the data they have seen. We will say in the following that the players in $C$ are corrupt, while players not in $C$ are honest. Since the function we are to compute is specified as an arithmetic circuit over $\mathbb{F}$, our task is, loosely speaking, to compute a number of additions and multiplications in $\mathbb{F}$ of the input values (or intermediate results), while revealing nothing except for the final result(s).

We begin by defining some convenient notation

**Definition 3.1** *We define $[a; f]_t$, where $a \in \mathbb{F}$ and $f$ is a polynomial over $f$ with $f(0) = a$ and degree at most $t$:*

$$[a; f]_t = (f(1), ...., f(n)) \ ,$$

*i.e., the set of shares in secret $a$, computed using polynomial $f$. Depending on the context, we sometimes drop the degree or the polynomial from the notation, so we write $[a; f]$ or just $[a]$.*

*Notice that on one hand the notation $[a; f]$* describes *an object, namely $(f(1), \ldots, f(n))$. On the other hand, it is also a* statement. *It states that $f(0) = a$. The notation $[a; f]_t$ describes the* same *object, $(f(1), \ldots, f(n))$, but further more states that* $\deg(f) \leq t$.

Using the standard notation for entrywise addition, multiplication by a scalar and the Schur product we have for $\alpha \in \mathbb{F}$:

$$[a; f]_t + [b; g]_t = (f(1) + g(1), ...., f(n) + g(n)) \ , \tag{3.3}$$

$$\alpha[a; f]_t = (\alpha f(1), ..., \alpha f(n)) \ , \tag{3.4}$$

$$[a; f]_t * [b; g]_t = (f(1)g(1), ..., f(n)g(n)) \ . \tag{3.5}$$

By the trivial observations that $f(i) + g(i) = (f + g)(i)$, $f(i)g(i) = (fg)(i)$, the following very important facts are easy to see:

**Lemma 3.1** *The following holds for any $a, b, \alpha \in \mathbb{F}$ and any polynomials $f, g$ over $\mathbb{F}$ of degree at most $t$ with $f(0) = a$ and $g(0) = b$:*

$$[a; f]_t + [b; g]_t = [a + b; f + g]_t \ , \tag{3.6}$$

$$\alpha[a; f]_t = [\alpha a; \alpha f]_t \ , \tag{3.7}$$

$$[a; f]_t * [b, g]_t = [ab; fg]_{2t} \ . \tag{3.8}$$

Since our notation for secret sharing both describe objects and make statements, the equations in the lemma actually say more than might be obvious on a first reading. As an example, the equation $[a; f]_t * [b; g]_t = [ab; fg]_{2t}$ states that the object $(f(1), \ldots, f(n)) * (g(1), \ldots, g(n))$ is identical to the object $((fg)(1), \ldots, (fg)(n))$. It also states that $(fg)(0) = ab$ and $\deg(fg) \leq 2t$.

The importance of the lemma is that it shows that by computing only on shares, we can – implicitly – compute on the corresponding secrets. For instance, the first of the above three relations shows that, if secrets $a, b$ have been shared, then if each player adds his shares of $a$ and $b$, then we obtain shares in the secret $a + b$. It will be useful in the following to have language for this sort of thing, so we define:

**Definition 3.2** *When we say that a player $\mathsf{P}_i$* distributes *$[a; f_a]_t$, this means that he chooses a random polynomial $f_a(\mathsf{X})$ of degree $\leq t$ with $f_a(0) = a$, and then sends the share $f_a(j)$ to $\mathsf{P}_j$, for $j = 1, \ldots, n$. Whenever players have obtained shares of a value $a$, based on polynomial $f_a$, we say that* the players hold *$[a; f_a]_t$.*

*Suppose the players hold $[a; f_a]_t$, $[b; f_b]_t$. Then, when we say that* the players compute *$[a; f_a]_t + [b; f_b]_t$, this means that each player $\mathsf{P}_i$ computes $f_a(i) + f_b(i)$, and by Lemma 3.1, this means that the players now hold $[a + b; f_a + f_b]_t$. In a similar way we define what it means for the players to compute $[a; f]_t * [b, g]_t$ and $\alpha[a; f]_t$.*

Using the tools and terms defined so far, we can describe a protocol for securely evaluating an arithmetic circuit, see Protocol CEPS (Circuit Evaluation with Passive Security).

### 3.3.3 Analysis

We will now analyze the security of the CEPS protocol. We will require two conditions:

**Perfect Correctness** With probability 1, all players receive outputs that are correct based on the inputs supplied.

Protocol CEPS (Circuit Evaluation with Passive Security)

The protocol proceeds in three phases: the input sharing, computation and output reconstruction phases.

**Input Sharing:** Each player $\mathsf{P}_i$ holding input $x_i \in \mathbb{F}$ distributes $[x_i; f_{x_i}]_t$.

We then go through the circuit and process the gates one by one in the computational order defined above. Just after the input sharing phase, we consider all input gates as being processed. We will maintain the following:

**Invariant:** Recall that computing with the circuit on inputs $x_1, ..., x_n$ assigns a unique value to every wire. Consider an input or an output wire for any gate, and let $a \in \mathbb{F}$ be the value assigned to this wire. Then, if the gate has been processed, the players hold $[a; f_a]_t$ for a polynomial $f_a$.

We then continue with the last two phases of the protocol:

**Computation Phase:** Repeat the following until all gates have been processed (then go to the next phase): Consider the first gate in the computational order that has not been processed yet. According to the type of gate, do one of the following

**Addition gate:** The players hold $[a; f_a]_t, [b; f_b]_t$ for the two inputs $a, b$ to the gate. The players compute $[a; f_a]_t + [b; f_b]_t = [a + b; f_a + f_b]_t$.

**Multiply-by-constant gate:** The players hold $[a; f_a]_t$ for the inputs $a$ to the gate. The players compute $\alpha[a; f_a]_t = [\alpha a; \alpha f_a]_t$.

**Multiplication gate:** The players hold $[a; f_a]_t, [b; f_b]_t$ for the two inputs $a, b$ to the gate.

1. The players compute $[a; f_a]_t * [b; f_b]_t = [ab; f_a f_b]_{2t}$.
2. Define $h \overset{\text{def}}{=} f_a f_b$. Then $h(0) = f_a(0) f_b(0) = ab$ and the parties hold $[ab; h]_{2t}$, i.e., $\mathsf{P}_i$ holds $h(i)$. Each $\mathsf{P}_i$ distributes $[h(i); f_i]_t$.
3. Note that $\deg(h) = 2t \leq n - 1$. Let $\mathbf{r}$ be the recombination vector defined in section 3.2, that is, the vector $\mathbf{r} = (r_1, \ldots, r_n)$ for which it holds that $h(0) = \sum_{i=1}^{n} r_i h(i)$ for any polynomial $h$ of degree $\leq n - 1$. The players compute

$$\sum_i r_i [h(i); f_i]_t = [\sum_i r_i h(i); \sum_i r_i f_i]_t = [h(0); \sum_i r_i f_i]_t = [ab; \sum_i r_i f_i]_t \ .$$

**Output Reconstruction:** At this point all gates, including the output gates have been processed. So do the following for each output gate (labeled $i$): The players hold $[y; f_y]_t$ where $y$ is the value assigned to the output gate. Each $\mathsf{P}_j$ securely sends $f_y(j)$ to $\mathsf{P}_i$, who uses Lagrange interpolation to compute $y = f_y(0)$ from $f_y(1), \ldots, f_y(t + 1)$, or any other $t + 1$ points.

**Perfect Privacy** Any subset $C$ of corrupt players, of size at most $t < n/2$, learns no information beyond $\{x_j, y_j\}_{\mathsf{P}_j \in C}$ from executing the protocol, regardless of their computing power.

To be more precise about privacy, we will use the simulation paradigm that we also alluded to in the introduction: to show that someone learns only information $X$, we show that *everything* he sees can be *efficiently* recreated, or simulated, given only $X$.

To express what this exactly means in our case, we define $\text{view}_j$ to be all the values that $\mathsf{P}_j$ sees during the execution of the protocol. More precisely, $\text{view}_j$ is a vector, where we first add $x_j$, and then each time $\mathsf{P}_j$ makes a random choice we add it to $\text{view}_j$, and each time $\mathsf{P}_j$ receives a message, we add the message to $\text{view}_j$. Notice that $\text{view}_j$ is a random variable, as it depends on the random choices made by $\mathsf{P}_j$ and the other parties (via their messages to $\mathsf{P}_j$).

Privacy now means that there exists an efficient probabilistic algorithm – a simulator $\mathsf{S}$ – which, given $\{x_j, y_j\}_{\mathsf{P}_j \in C}$, can produce an output whose distribution is exactly the same as the entire set of data seen by $C$ during the protocol, i.e., we want that

$$\mathsf{S}(\{x_j, y_j\}_{\mathsf{P}_j \in C}) \overset{\text{perf}}{\equiv} \{\text{view}_j\}_{\mathsf{P}_j \in C} \ .$$

The requirement that the distributions be exactly equal is the reason we talk about *perfect* privacy here.

We warn the reader already now that the general security definition we give in the next chapter looks quite different, although it too is based on the simulation paradigm. This is necessary, as correctness and privacy are simply not sufficient to define security if corrupt players may deviate from the protocol.

## Correctness

If the invariant defined in the CEPS protocol is maintained, it is clear that once all gates are processed, then in particular for every wire going into an output gate, players hold $[y; f_y]_t$ where $y$ is the correct value for that wire. So the player who is to receive that value will indeed get $y$.

To check that the invariant holds, note that this is trivially true for the input gates after the first phase. In the computation phase we easily see that the protocol for each type of gate maintains the invariant, by Lemma 3.1, and (for multiplication) by definition of the recombination vector.

## Privacy

We now argue that the values seen by $t$ corrupted parties gives them no information extra to their inputs and outputs.

Note that the corrupted parties only receive two types of messages from the honest parties.

**Type 1:** In Input Sharing and Multiplication gate, they receive shares in the random sharings $[x_i; f_{x_i}]_t$ respectively $[h(i); f_i]_t$ made by honest parties.

**Type 2:** In Output Reconstruction they receive all shares in $[y, f_y]_t$ for each output $y$ which is for a corrupted party.

The privacy of the protocol now follows from the following two observations:

**Observation 1:** All values sent by honest parties to corrupted parties in Input Sharing and Multiplication are shared using random polynomials, $f_{x_i}$ respectively $f_i$, of degree at most $t$. Since there are at most $t$ corrupted parties, the set of corrupted parties receive at most $t$ shares of these polynomials, and $t$ shares of random polynomials of degree at most $t$ are just uniformly random values.

**Observation 2:** The values sent by honest parties to corrupted parties in Output Reconstruction to reconstruct $[y; f_y]_t$ could have been computed by the corrupted parties themselves given the values they know prior to Output Reconstruction plus the output $y$. The reason is that prior to Output Reconstruction the corrupted parties already known $f_y(i)$ for each of the $t$ corrupted parties $\mathsf{P}_i$. This gives them $t$ point on $f_y(\mathtt{X})$. If we give them the result $y$, they additionally know that $f_y(0) = y$. So, with their view prior to Output Reconstruction and the result $y$, they know $t+1$ points on $f_y(\mathtt{X})$. Since $f_y(\mathtt{X})$ has degree at most $t$, this allows the corrupted parties to efficiently compute $f_y(\mathtt{X})$ using Lagrange interpolation. From $f_y(\mathtt{X})$ they can compute $f_y(i)$ for all honest parties $\mathsf{P}_i$, and $f_y(i)$ is exactly the share in $y$ sent by $\mathsf{P}_i$ in Output Reconstruction.

Let us first see why these two observations intuitively should make us consider the protocol secure. Recall that the goal is that the corrupted parties learn nothing extra to their own inputs and outputs. But clearly they don't! In Input Sharing and Multiplication they just receive uniformly random values. This gives them nothing, as they could have sampled such values themselves. In Output Reconstruction they then receive values which they could have computed themselves from their outputs $y$.

The observant reader will notice that this intuition is in fact also close to being a formal proof. Recall that our formal requirement to call a protocol perfectly private is that there exists a simulator $\mathsf{S}$ which given the inputs and outputs of the corrupted parties outputs a value with the same distribution as the view of the corrupted parties in the protocol. From the above intuition it is clear how $\mathsf{S}$ would proceed:

1. It runs the corrupted parties on their own inputs according to the protocol. It can do so, as it is given the inputs of the corrupted parties.
2. During Input Sharing and Multiplication it simulates the shares sent to the corrupted parties from the honest parties by simply sampling uniformly random values, which produces the right distribution by Observation 1.
3. In Output Reconstruction it will for each output $y$ to a corrupted party compute the honest parties' shares in $[y; f_y]$ to be those consistent with the $t$ simulated shares of the corrupted parties and $f_y(0) = y$. It can do so, as $y$ is the output of a corrupted party and hence was given as input to $\mathsf{S}$. This gives the right distribution by Observation 2.

We will now give the proof in more detail. It follows the above proof very closely. In the rest of the book, when we prove protocols secure, we will give proofs at the level of detail above. The purpose of the detailed proof below is to enable the reader interested in rigorous proofs to flesh out such proofs along the lines used below.

**Formalizing Observation 1**

We start by formalizing Observation 1. For this purpose, we define a function $\text{Strip}(\text{view}_j)$ which takes as input the view of a party $\mathsf{P}_j \in C$ and removes from $\text{view}_j$ the $n - t$ shares $f_{y_j}(i)$ sent to $\mathsf{P}_j$ in Output Reconstruction from parties $\mathsf{P}_i \notin C$. The function outputs this stripped view. Notice that $\text{Strip}(\text{view}_j)$ still contains $x_j$, the random choices made by $\mathsf{P}_j$, the shares sent to $\mathsf{P}_j$ in Input Sharing and Multiplication, and the shares sent to $\mathsf{P}_j$ from *corrupted* parties in Output Reconstruction.

We can now formalize Observation 1 as follows.

**Lemma 3.2** *Let $C \subset \{P_1, \ldots, P_n\}$ be a set of corrupted parties, and assume for simplicity that $|C| = t$. Consider two global input vectors $\mathbf{x}^{(0)} = (x_1^{(0)}, \ldots, x_n^{(0)})$ and $\mathbf{x}^{(1)} = (x_1^{(1)}, \ldots, x_n^{(1)})$, where $x_j^{(0)} = x_j^{(1)}$ for $\mathsf{P}_j \in C$, i.e., the corrupted parties have the same inputs in both vectors. For $b = 0, 1$, let $\text{view}_j^b$ be the view of $\mathsf{P}_j$ after running Protocol* CEPS (Circuit Evaluation with Passive Security) *on $\mathbf{x}^b$. Then it holds that*

$$\{\text{Strip}(\text{view}_j^{(0)})\}_{P_j \in C} \overset{\text{perf}}{\equiv} \{\text{Strip}(\text{view}_j^{(1)})\}_{P_j \in C} \ . \tag{3.9}$$

*Proof* For each part of the protocol we check that $C$ sees the same distribution in case (0) and in case (1):

**Input Sharing:** Each $P_i \in C$ distributes $[x_i^{(b)}]_t$. Since $x_i^{(0)} = x_i^{(1)}$, this clearly leads to the same distribution on the shares in the two cases.

Each $P_j \notin C$ distributes $[x_i^{(b)}]_t$. It might be the case that $x_j^{(0)} \neq x_j^{(1)}$, but parties in $C$ see at most $t$ shares, and these are uniformly random and independent of $x_j^{(b)}$; In particular, the distribution is the same when $b = 0$ and $b = 1$.

**Addition:** Here no party sends or receives anything, so there is nothing to show.

**Multiplication:** Follows as for Input Sharing: We can assume that all values held by parties in $C$ before the current multiplication have the same distribution in case (0) and in case (1). This is trivially the case for the first multiplication and can be assumed by induction for the following ones. In particular, the values they are supposed to locally multiply have the same distribution. Therefore all values generated and sent by parties in $C$ have the same distribution. The honest parties only send shares of random sharings, and $C$ only sees $t$ shares of each sharing. These are just uniformly random values.

**Output Reconstruction:** Here all shares in $[y_j^{(b)}; f_{y_j^{(b)}}]$ are securely sent to $P_j$, for $j = 1, \dots, n$. If $P_j$ is honest, parties in $C$ see nothing. If $P_j \in C$ is corrupt, then parties in $C$ see *all* shares in $[y_j^{(b)}; f_{y_j^{(b)}}]$. But Strip exactly deletes the shares sent from honest parties to corrupted parties. The shares sent from corrupted parties to corrupted parties are computed from their view prior to Output Reconstruction, which we, by induction, can assume are identically distributed when $b = 0$ and $b = 1$.

□

**Formalizing Observation 2**

We then formalize Observation 2. For this purpose we need another function, Dress.

For any value $\mathbf{y}_C = \{y_j\}_{\mathsf{P}_j \in C} \in \mathbb{F}^t$, we define a function $\text{Dress}_{\mathbf{y}_C}$ which takes an input $\{\text{view}'_j\}_{\mathsf{P}_j \in C}$, where each $\text{view}'_j$ is a stripped view of $\mathsf{P}_j$, i.e., $\text{view}'_j = \text{Strip}(\text{view}_j)$, where $\text{view}_j$ is a possible view of $\mathsf{P}_j$. The output is of the form $\{\text{view}''_j\}_{\mathsf{P}_j \in C}$, where each $\text{view}''_j$ is of the form of a full view of $\mathsf{P}_j$. The purpose of $\text{Dress}_{\mathbf{y}_C}$ is to try to fill in those values that Strip removed. It attempts this as follows: For each $\mathsf{P}_j \in C$, it takes the $t$ shares that the corrupted parties hold in the output of $\mathsf{P}_j$, then it assumes that the output of $\mathsf{P}_j$ is the value $y_j$ given in $\mathbf{y}_C$, computes which shares the honest parties would hold in $y_j$ in that case, and extends $\text{view}'_j$ with these shares, as if they were sent to $\mathsf{P}_j$. In more detail, first $\text{Dress}_{\mathbf{y}_C}$ inspects each $\text{view}'_j$, $\mathsf{P}_j \in C$, and reads off the shares of the output of $\mathsf{P}_j$ that was sent to $\mathsf{P}_j$ from corrupted parties $\mathsf{P}_i \in C$ — recall that we did not delete these shares, only the shares sent by honest parties. So, for $\mathsf{P}_j \in C$, it now knows the shares of the $t$ corrupted parties in the output of $\mathsf{P}_j$ — call these $f(i)_{y_j}$ for $\mathsf{P}_i \in C$. Then it takes the value $y_j$ from $\mathbf{y}_C$ and defines $f_{y_j}(0) \overset{\text{def}}{=} y_j$. After this, $\text{Dress}_{\mathbf{y}_C}$ has defined $f_{y_j}(\mathtt{X})$ in $t+1$ points and can hence compute the unique degree-$t$ polynomial $f(\mathtt{X})$ consistent with these points. Then for $\mathsf{P}_i \notin C$ it adds the share $f_{y_j}(i)$ to $\text{view}_j$ as if $\mathsf{P}_i$ sent $f_{y_j}(i)$ to $\mathsf{P}_j$. Call the result $\text{view}''_j$, and let $\text{Dress}_{\mathbf{y}_C}(\{\text{view}'_j\}_{\mathsf{P}_j \in C}) = \{\text{view}''_j\}_{\mathsf{P}_j \in C}$.

We can now formalize Observation 2 as follows.

**Lemma 3.3** *Let $C \subset \{P_1, \dots, P_n\}$ be a set of corrupted parties, and assume for simplicity that $|C| = t$. Let $\mathbf{x}$ be any input vector, let $(y_1, \dots, y_n) = f(\mathbf{x})$, let $\mathbf{y}_C = \{y_j\}_{P_j \in C}$, and let* $\text{view}_j$ *be the view of $P_j$ after running Protocol* CEPS (Circuit Evaluation with Passive Security) *on $\mathbf{x}$. Then*

$$\text{Dress}_{\mathbf{y}_C}(\{\text{Strip}(\text{view}_j)\}_{P_j \in C}) = \{\text{view}_j\}_{P_j \in C} \ .$$

*Proof* The view $\text{view}_j$ contains $n$ shares in the sharing $[y'_j, f_{y'_j}]_t$ of the output $y'_j$ that $\mathsf{P}_j$ computes in the protocol. By perfect correctness of Protocol CEPS (Circuit Evaluation

WITH PASSIVE SECURITY), we know that $y'_j = y_j$ for the $y_j$ given by $(y_1, \ldots, y_n) = f(\mathbf{x})$. So, view$_j$ contains $n$ shares in a sharing $[y_j; f_{y_j}]_t$ of $y_j$. Then Strip removes the shares of the honest parties. The function Dress$_{\mathbf{y}_C}$, however, recomputes the same shares, as it computes them from the same correct $y_j$ (the $y_j$ in $\mathbf{y}_C$ was taken from $(y_1, \ldots, y_n)$) and the $t$ shares of the corrupted parties, which were not deleted from view$_j$. By Lagrange interpolation this gives exactly the shares back that Strip deleted. See Exercise 3.2 for the case where less than $t$ parties are corrupted. □

### Formalizing the Simulator

We can also formalize the simulator S via the two functions defined above. It proceeds as follows:

1. The input to S is $\{x_j, y_j\}_{j \in C}$.
2. It defines a global input vector $\mathbf{x}^{(0)} = (x_1^{(0)}, \ldots, x_n^{(0)})$, where $x_j^{(0)} = x_j$ for $\mathsf{P}_j \in C$ and $x_j^{(0)} = 0$ for $\mathsf{P}_j \notin C$. Here $x_j$ for $\mathsf{P}_j \in C$ are the values it received as input.
3. It runs Protocol CEPS (CIRCUIT EVALUATION WITH PASSIVE SECURITY) on $\mathbf{x}^{(0)}$ and lets view$_j^{(0)}$ be the view of $\mathsf{P}_j$ in this execution of the protocol.
4. It lets $\mathbf{y}_C = \{y_j\}_{\mathsf{P}_j \in C}$, where $y_j$ is the value it received as input.
5. It outputs $\text{Dress}_{\mathbf{y}_C}(\{\text{Strip}(\text{view}_j^{(0)})\}_{\mathsf{P}_j \in C})$.

Notice that running the protocol on wrong inputs for the honest parties results in sending $t$ shares of wrong values during Input Sharing and Computation Phase. Since the corrupted parties only receive $t$ shares, this is the same as just sending them uniformly random values. Then Strip and Dress$_{\mathbf{y}_C}$ removes the shares sent by the honest parties in Output Reconstruction and replaces them with shares consistent with the shares of the corrupted parties and the correct output. So, the above way to specify the simulator is equivalent to the one given earlier.

### Formally Analyzing the Simulator

We now prove that the simulator produces the right distribution.

**Theorem 3.4** *Let* $\mathbf{x} = (x_1, \ldots, x_n)$ *be a global input vector and let* $(y_1, \ldots, y_n) = f(\mathbf{x})$ *and let* $C \subset \{P_1, \ldots, P_n\}$ *with* $|C| = t$. *For* $P_j \in C$, *let* view$_j$ *be the view of* $P_j$ *when running Protocol* CEPS (CIRCUIT EVALUATION WITH PASSIVE SECURITY) *on the input* $\mathbf{x}$. *Then*

$$S(\{x_j, y_j\}_{P_j \in C}) \overset{\text{perf}}{\equiv} \{\text{view}_j\}_{P_j \in C} \ .$$

*Proof* Let $\mathbf{y}_C = \{y_j\}_{\mathsf{P}_j \in C}$ for the values $y_j$ defined in the premise of the theorem. Define $\mathbf{x}^{(0)} = (x_1^{(0)}, \ldots, x_n^{(0)})$ where $x_j^{(0)} = x_j$ for the $x_j$ given in the premise of the theorem for $\mathsf{P}_j \in C$ and $x_j^{(0)} = 0$ for $\mathsf{P}_j \notin C$. Let view$_j^{(0)}$ be the view of $\mathsf{P}_j$ generated by S by running Protocol CEPS (CIRCUIT EVALUATION WITH PASSIVE SECURITY) on the input $\mathbf{x}^{(0)}$.

By construction of the simulator we have that

$$\mathsf{S}(\{x_j, y_j\}_{\mathsf{P}_j \in C}) \overset{\text{perf}}{\equiv} \text{Dress}_{\mathbf{y}_C}(\{\text{Strip}(\text{view}_j^{(0)})\}_{\mathsf{P}_j \in C}) \ . \tag{3.10}$$

Since S generates view$_j^{(0)}$ by running Protocol CEPS (CIRCUIT EVALUATION WITH PASSIVE SECURITY) on the input $\mathbf{x}^{(0)}$, and because $x_j = x_j^{(0)}$ for $\mathsf{P}_j \in C$ and $|C| = t$, it follows from Lemma 3.2 that

$$\{\text{Strip}(\text{view}_j^{(0)})\}_{\mathsf{P}_j \in C} \overset{\text{perf}}{\equiv} \{\text{Strip}(\text{view}_j)\}_{\mathsf{P}_j \in C} \ . \tag{3.11}$$

We then apply Proposition 2.3, Property 3 to Eq. 3.11. This gives us that

$$A(\{\text{Strip}(\text{view}_j^{(0)})\}_{\mathsf{P}_j \in C}) \overset{\text{perf}}{\equiv} A(\{\text{Strip}(\text{view}_j)\}_{\mathsf{P}_j \in C}) \tag{3.12}$$

for all functions $A$. Let $A = \text{Dress}_{\mathbf{y}_C}$. Plugging this $A$ into Eq. 3.12 we get that

$$\text{Dress}_{\mathbf{y}_C}(\{\text{Strip}(\text{view}_j^{(0)})\}_{\mathsf{P}_j \in C}) \overset{\text{perf}}{\equiv} \text{Dress}_{\mathbf{y}_C}(\{\text{Strip}(\text{view}_j)\}_{\mathsf{P}_j \in C}) \ . \tag{3.13}$$

By Lemma 3.3 we have that

$$\text{Dress}_{\mathbf{y}_C}(\{\text{Strip}(\text{view}_j)\}_{\mathsf{P}_j \in C}) \overset{\text{perf}}{\equiv} \{\text{view}_j\}_{\mathsf{P}_j \in C} \ . \tag{3.14}$$

Using Eq. 3.10, Eq. 3.13 and Eq. 3.14 and transitivity of $\overset{\text{perf}}{\equiv}$ (Proposition 2.3, Property 2) we get that

$$\mathsf{S}(\{x_j, y_j\}_{\mathsf{P}_j \in C}) \overset{\text{perf}}{\equiv} \{\text{view}_j\}_{\mathsf{P}_j \in C} \ ,$$

as desired. □

**Exercise 3.2** To formally prove that Protocol CEPS (Circuit Evaluation with Passive Security) is perfectly private we have to prove Theorem 3.4 for all $C$ with $|C| \le t$, not just $|C| = t$. Give a rigorous proof for the case $|C| < t$. [Hint: Dress is going to be a probabilistic algorithm.]

### 3.3.4 Example Computations and Proofs by Example

The above argument for the output reconstruction shows that it does not harm to give all shares of an output to the corrupted parties. This, in particular, shows that the shares do not carry information about how the result was computed: If $c = a + b$ is reconstructed and the result is 6, then the $n$ shares of $c$ will be consistent with both $a = 2, b = 4$ and $a = 1, b = 5$ — otherwise the protocol could not be secure. We will, however, look at two exercises to exemplify this phenomenon.

Consider a setting where variables $a$ and $b$ have been computed, and where then a variable $c = a + b$ is computed and output to $\mathsf{P}_1$. Assume that $n = 3$ and $t = 1$. I.e., we have three parties $\mathsf{P}_1, \mathsf{P}_2, \mathsf{P}_3$ and one can be corrupted. For sake of example, say it is $\mathsf{P}_1$. Since $t = 1$ we are using polynomials of the form $f(\mathtt{X}) = \alpha_0 + \alpha_1\mathtt{X}$, a.k.a. lines.

Assume that $a = 2$ and that $a$ is shared using the polynomial $a(\mathtt{X}) = 2 + 2X$, and assume that $b = 4$ and that $b$ is shared using the polynomial $a(\mathtt{X}) = 4 + X$. This gives the following computation:

| Variable | Value | $\mathsf{P}_1$ | $\mathsf{P}_2$ | $\mathsf{P}_3$ |
|---|---|---|---|---|
| $a$ | 2 | **4** | 6 | 8 |
| $b$ | 4 | **5** | 6 | 7 |
| $c = a + b$ | 6 | **9** | **12** | **15** |

We show the shares $a(1) = 4$, $a(2) = 6$, $a(3) = 8$ and the shares $b(1) = 5$, $b(2) = 6$, $b(3) = 7$ in the rows to the right of the variables and their values. When the parties compute the variable $c = a + b$, they simply add locally and compute the shares 9, 12 respectively 15. In the table all shares that $\mathsf{P}_1$ would see in this case are put in bold.

We want that $\mathsf{P}_1$ only learns that $c = 6$, and nothing about $a$ and $b$ except that $a + b = 6$. We demonstrate that this is the case by an example. We let the party $\mathsf{P}_1$ make the hypothesis that $a = 1$ and $b = 5$. Hopefully it cannot exclude this hypothesis. For starters, $\mathsf{P}_1$ has the following view (not knowing the shares of $\mathsf{P}_1$ and $\mathsf{P}_2$):

| Variable | Value | $\mathsf{P}_1$ | $\mathsf{P}_2$ | $\mathsf{P}_3$ |
|---|---|---|---|---|
| $a$ | 1 | **4** | ? | ? |
| $b$ | 5 | **5** | ? | ? |
| $c = a + b$ | 6 | **9** | **12** | **15** |

If $a(0) = 1$ and $a(1) = 4$, then it must be the case that $a(\mathtt{X}) = 1 + 3\mathtt{X}$, which would imply that $a(2) = 7$ and $a(3) = 10$. Furthermore, if $b(0) = 5$ and $b(1) = 5$, then it must be the case that $b(\mathtt{X}) = 5 + 0\mathtt{X}$, which would imply that $b(2) = 5$ and $b(3) = 5$. If $\mathsf{P}_1$ fills these values into the table it concludes that the network must be configured as follows for its hypothesis to hold:

| Variable | Value | $\mathsf{P}_1$ | $\mathsf{P}_2$ | $\mathsf{P}_3$ |
|---|---|---|---|---|
| $a$ | 1 | **4** | 7 | 10 |
| $b$ | 5 | **5** | 5 | 5 |
| $c = a + b$ | 6 | **9** | **12** | **15** |

Note that this hypothesis is consistent with the protocol and what $\mathsf{P}_1$ have seen, as $7 + 5 = 12$ and $10 + 5 = 15$. Therefore $a = 1$ and $b = 5$ is as possible as $a = 2$ and $b = 4$.

**Exercise 3.3** In the above example, $\mathsf{P}_1$ could also have made the hypothesis that $a = 0$ and $b = 6$. Show that $\mathsf{P}_1$ cannot exclude this example, by filling in the below table and noting that it is consistent.

| Variable | Value | $\mathsf{P}_1$ | $\mathsf{P}_2$ | $\mathsf{P}_3$ |
|---|---|---|---|---|
| $a$ | 0 | **4** | ? | ? |
| $b$ | 6 | **5** | ? | ? |
| $c = a + b$ | 6 | **9** | **12** | **15** |

We now consider an example of a multiplication of variables $a = 2$ and $b = 3$. The polynomials used to share them are $a(\mathtt{X}) = 2 + \mathtt{X}$ and $b(\mathtt{X}) = 3 - \mathtt{X}$:

| Variable | Value | $\mathsf{P}_1$ | $\mathsf{P}_2$ | $\mathsf{P}_3$ |
|---|---|---|---|---|
| $a$ | 2 | **3** | 4 | 5 |
| $b$ | 3 | **2** | 1 | 0 |
| $d = ab$ | 6 | **6** | 4 | 0 |
| $d_1$ | **6** | **4** | **2** | **0** |
| $d_2$ | 4 | **6** | 8 | 10 |
| $d_3$ | 0 | **0** | 0 | 0 |
| $c = 3d_1 - 3d_2 + d_3$ | 6 | **−6** | **−18** | **−30** |

We explain the lower part of the table soon, but first note that the shares of $d = ab = 6$ are not on a line, as all the other shares are. The reason is that $d(\mathtt{X}) = a(\mathtt{X})b(\mathtt{X})$ is not a line, but a quadratic polynomial. In fact, $d(\mathtt{X}) = (2 + \mathtt{X})(3 - \mathtt{X}) = 6 + \mathtt{X} - \mathtt{X}^2$, which is consistent with $d(1) = 6$, $d(2) = 4$ and $d(3) = 0$.

After having computed the local products $d_i$, the next step in the multiplication algorithm uses the Lagrange formula for computing $d$ from $d_1, d_2, d_3$, so we derive that one. Since $2t = 2$ we are looking at quadratic polynomials $y(\mathtt{X}) = \alpha_0 + \alpha_1\mathtt{X} + \alpha_2\mathtt{X}^2$, where $\alpha_0$ is the secret. Therefore the shares are $y_1 = y(1) = \alpha_0 + \alpha_1 + \alpha_2$, $y_2 = y(2) = \alpha_0 + 2\alpha_1 + 4\alpha_2$ and $y_3 = y(3) = \alpha_0 + 3\alpha_1 + 9\alpha_2$. It follows that $\alpha_0$ can always be computed from the shares as $\alpha_0 = 3y_1 - 3y_2 + y_3$. This formula was found using simple Gaussian elimination, but is also given by the Lagrange interpolation formula. I.e., in our case the recombination vector is $r = (3, -3, 1)$.

In our example we have $d_1 = 6$, $d_2 = 4$ and $d_3 = 0$, and indeed $3d_1 - 3d_2 + d_3 = 18 - 12 = 6 = ab$, as it should be. Each party now shares its value $d_i$. In the table $\mathsf{P}_1$ used the polynomial $d_1(\mathtt{X}) = 6 - 2\mathtt{X}$, $\mathsf{P}_2$ used the polynomial $d_2(\mathtt{X}) = 4 + 2\mathtt{X}$ and $\mathsf{P}_3$ used the polynomial $d_3(\mathtt{X}) = 0 + 0\mathtt{X}$. The parties then locally combine their shares by an inner product with the recombination vector $(3, -3, 1)$, leading to the shares in the table: as an example, $\mathsf{P}_1$ computed $-6 = 3 \cdot 4 + (-3) \cdot 6 + 1 \cdot 0$.

Again, an example will reveal that any other hypothesis, like $a = 1$ and $b = 6$ would have given the exact same view to $\mathsf{P}_1$. The reader is encourage to do that, by solving the following exercise.

**Exercise 3.4** Show that the values seen by $\mathsf{P}_1$ are consistent with the hypothesis $a = 1$ and $b = 6$ by filling in the following table and noting that it is consistent.

| Variable | Value | $\mathsf{P}_1$ | $\mathsf{P}_2$ | $\mathsf{P}_3$ |
|---|---|---|---|---|
| $a$ | 1 | **3** | ? | ? |
| $b$ | 6 | **2** | ? | ? |
| $d = ab$ | 6 | **6** | ? | ? |
| $d_1$ | **6** | **4** | **2** | **0** |
| $d_2$ | ? | **6** | ? | ? |
| $d_3$ | ? | **0** | ? | ? |
| $c = 3d_1 - 3d_2 + d_3$ | 6 | $\mathbf{-6}$ | $\mathbf{-18}$ | $\mathbf{-30}$ |

[To check solution: It must say $-42$ and 84 somewhere.]

## 3.4 Optimality of the Corruption Bound

What happens if the number $t$ of corrupt players is larger than what we assumed so far, i.e., what if $t \geq n/2$? Then the CEPS protocol no longer works, the multiplication subprotocol breaks down. You may want to take a moment to see why this is the case. We will now show that this is no coincidence. In fact there are functions that cannot be computed securely if $t \geq n/2$.

Towards a contradiction, suppose there is a protocol $\pi$, with *perfect privacy* and *perfect correctness* (as defined earlier) for two players $\mathsf{P}_1, \mathsf{P}_2$ to securely evaluate the logical AND of their respective private input bits $b_1, b_2$, i.e., $b_1 \wedge b_2$. The set $C$ of corrupt players from the definition may in this case be $C = \{\mathsf{P}_1\}$ or $C = \{\mathsf{P}_2\}$, i.e., $n = 2, t = 1$.

We assume as usual that the players communicate using a perfect *error-free communication channel*.

Without loss of generality, we may assume the protocol is of the following form.

1. Each player $\mathsf{P}_i$ has a private input bit $b_i$. Before the protocol starts, they select private random strings $r_i \in \{0,1\}^*$ of appropriate length.

   Their actions in the forthcoming protocol are now uniquely determined by these initial choices.

2. $\mathsf{P}_1$ sends the first message $m_{11}$, followed by $\mathsf{P}_2$'s message $m_{21}$.

   This continues until $\mathsf{P}_2$ has sent sufficient information for $\mathsf{P}_1$ to compute $y = b_1 \wedge b_2$. Finally, $\mathsf{P}_1$ sends $y$ (and some halting symbol) to $\mathsf{P}_2$.

   The $\mathsf{transcript}$ of an $s$-round conversation is

   $$\mathcal{T} = (m_{11}, m_{21}, \ldots, m_{1t}, m_{2s}, y).$$

   For $i = 1, 2$, the $\mathsf{view}$ of $P_i$ is

   $$\mathrm{view}_i = (b_i, r_i, \mathcal{T}) .$$

To be concrete about the assumptions, perfect correctness means here that the protocols always halts (in a number of rounds $t$ that may perhaps depend on the inputs and the random coins) and that the correct result is always computed.

Perfect privacy means that the distribution of $\mathsf{P}_i$'s view of the protocol depends only on his input bit $b_i$ and the result $r = b_1 \wedge b_2$.

Note that these conditions imply that if one of the players has input bit equal to 1, then he learns the other player's input bit with certainty, whereas if his input bit equals 0, he should have no information at all about the other player's input bit.

Let $\mathcal{T}(0,0)$ denote the set of transcripts $\mathcal{T}$, which can arise when $b_1 = 0$ and $b_2 = 0$ and let $\mathcal{T}(0,1)$ denote the set of transcripts $\mathcal{T}$, which can arise when $b_1 = 0$ and $b_2 = 1$.

Given a transcript $\mathcal{T} = (m_{11}, m_{21}, \ldots, m_{1t}, m_{2s}, y)$ we say that it is *consistent with input* $b_1 = 1$ if there exists $r_1$ such that running $\mathsf{P}_1$ with input $b_1 = 1$ and randomness $r_1$ might result in $\mathcal{T}$ being generated. Namely, if we assume that $\mathsf{P}_1$ receives the message $m_{2i}$ in round $i$ for $i = 1, \ldots, s$, then with $b_1 = 1$ and randomness $r_1$, $\mathsf{P}_1$ would send exactly the message $m_{1i+1}$ in round $i+1$. Let $\mathcal{C}_{b_1=1}$ denote the transcripts consistent with $b_1 = 1$.

It follows from perfect privacy that

$$\mathcal{T}(0,0) \subset \mathcal{C}_{b_1=1} \ .$$

Assume namely that $\mathsf{P}_1$ has input $b_1 = 0$ and $\mathsf{P}_2$ has input $b_2 = 0$, and that $\mathsf{P}_2$ sees a transcript $\mathcal{T}$ which is not from $\mathcal{C}_{b_1=1}$. Then $\mathsf{P}_2$ can conclude that $b_1 = 0$, contradicting privacy.

From the perfect correctness we can conclude that

$$\mathcal{C}_{b_1=1} \cap \mathcal{T}(0,1) = \emptyset \ .$$

Consider namely $\mathcal{T} \in \mathcal{T}(0,1)$. By perfect correctness it follows that $y = 0$ in $\mathcal{T}$. Therefore $\mathcal{T}$ is clearly not consistent with input $b_1 = 1$, as that would mean that $\mathcal{T}$ could be produced with $b_1 = 1$ and $b_2 = 1$, which would give result $y = 1$ (again by perfect correctness).

From these two observations we see that $\mathcal{T}(0,0) \cap \mathcal{T}(0,1) = \emptyset$.

Now consider a case where $\mathsf{P}_1$ executes the protocol with $b_1 = 0$. Then he sees a transcript $\mathcal{T}$ from either $\mathcal{T}(0,0)$ or $\mathcal{T}(0,1)$. Since they are disjoint, $\mathsf{P}_1$ can determine the input of $\mathsf{P}_2$ simply by checking whether $\mathcal{T} \in \mathcal{T}(0,0)$ or $\mathcal{T} \in \mathcal{T}(0,1)$. This contradicts perfect privacy.

One concrete algorithm $\mathsf{P}_1$ could use to check whether $\mathcal{T} \in \mathcal{T}(0,0)$ or $\mathcal{T} \in \mathcal{T}(0,1)$ is to do the equivalent test of whether $\mathcal{T} \in \mathcal{C}_{b_1=1}$. This can be done by a brute force search of all possibilities for his own randomness. Such an algorithm is of course not efficient, but this is a not a problem as we consider corrupt players that may have unlimited computing power.

The argument can be generalized to show impossibility of protocols where privacy or correctness may fail with small but negligible probability, or even with small constant probability.

**Exercise 3.5** Show that there is no 2-party perfectly secure and perfectly correct protocol for the OR function $(b_1, b_2) \mapsto b_1 \vee b_2$.

**Exercise 3.6** Show that the following protocol is a perfectly secure (in the sense of poly-time simulation) and perfectly correct protocol for the XOR function $(b_1, b_2) \mapsto b_1 \oplus b_2$. Party $\mathsf{P}_1$ sends $b_1$ to $\mathsf{P}_2$ and $\mathsf{P}_2$ sends $b_2$ to $\mathsf{P}_1$. Then they both output $b_1 \oplus b_2$.

**Exercise 3.7** Any binary Boolean function $B : \{0,1\} \times \{0,1\} \to \{0,1\}$ can be given by a vector $(o_{00}, o_{01}, o_{10}, o_{11}) \in \{0,1\}^4$, by letting $B(b_1, b_2) = o_{b_1 b_2}$. The AND function is given by $(0,0,0,1)$, the OR function is given by $(0,1,1,1)$, the XOR function is given by $(0,1,1,0)$, and the NAND function is given by $(1,1,1,0)$. Show that all functions specified by a vector with an even number of 1's can be securely computed as defined above and that none of the other functions can.

### Computational Security

The assumptions about the players' unbounded computational resources and the communication channel are essential for the impossibility results.

It can be shown that any of the following conditions is sufficient for the existence of a secure two-party protocol for the AND function (as well as OR).

1. Existence of trapdoor one-way permutations.
2. Both players are memory bounded.
3. The communication channel is noisy.

We sketch a secure AND protocol based on the assumption that there exists a public-key cryptosystem where the public keys can be sampled in two different ways: There is the usual key generation which gives an encryption key $ek$ and the corresponding decryption key $dk$. The other method only generates $ek$ and even the party having generated $ek$ cannot decrypt ciphertexts under $ek$. We assume that these two key generators give encryption keys with the same distribution

If $b_1 = 0$, then $\mathsf{P}_1$ samples $ek$ uniformly at random without learning the decryption key. If $b_1 = 1$, then $\mathsf{P}_1$ samples $ek$ in such a way that it learns the decryption key $dk$. It sends $ek$ to $\mathsf{P}_2$. Then $\mathsf{P}_2$ sends $C = E_{pk}(b_2)$ to $\mathsf{P}_1$. If $b_1 = 0$, then $\mathsf{P}_1$ outputs 0 and sends $y = 0$ to $\mathsf{P}_2$ which outputs $y$. If $b_1 = 1$, then $\mathsf{P}_1$ decrypts $C$ to learn $b_2$, outputs $b_2$ and sends $y = b_2$ to $\mathsf{P}_2$ which outputs $y$.

Since the two ways to sample the public key gives the same distribution, the protocol is perfectly secure for $\mathsf{P}_1$. The security of $\mathsf{P}_2$ depends on the encryption hiding $b_2$ when $b_1 = 0$ and is therefore computational. In particular, a computationally unbounded $\mathsf{P}_1$ could just use brute force to decrypt $C$ and learn $b_2$ even when $b_1 = 0$.

This protocol can be in principle be made secure against deviations by letting the parties use generic zero-knowledge proofs to show that they followed the protocol. This in principle leads to secure two-party protocols for any function.

**Exercise 3.8**

1. Use the special cryptosystem from above to give a secure protocol for the OR function.
2. Try to generalize to any two-party function, where one of the parties has a constant number of input bits and the other party might have an arbitrary number of inputs. [Hint: The party with a constant number of inputs will have perfect security and will not send just a single encryption key.]

# Chapter 4

# Models

**Contents**

## 4.1 Introduction

The protocol we described in Chapter 3 is perfectly correct and perfectly private, but cannot tolerate that any of the parties deviate from the protocol. In later chapters we will present protocols which are correct and private even if some parties do not follow the protocol. Such protocols are called robust. Before we can prove that these protocols are robust, we need a good definition of what we mean by that.

In this section we will describe a security model for cryptographic protocols known as the UC model. Here *UC* stands for *universally composable.* The name was adopted because a protocol proven secure in this model remains secure regardless of the context in which it is used. In other words, it can be "universally" composed with any set of other protocols. Our formulation of the UC model differs in several respects from the way it was originally formalized in the literature. We will hint at these differences as we go, some additional comments can be found in Chapter 12.

Before we look at the formal details, let us start by discussing the basic ideas of how to define privacy and robustness of a protocol and last, but not least, why and how these two concepts should be combined.

### 4.1.1 Defining Privacy

It is convenient to first look back on how we defined privacy in Chapter 3. Informally we defined a protocol to be private (against corruptions of size $t$) as follows. First pick an input $(x_1, \ldots, x_n)$ for the protocol and make a run of the protocol on this input. Then pick some $C \subset \{\mathsf{P}_1, \ldots, \mathsf{P}_n\}$ with $|C| \leq t$ and consider the values $\{\text{view}_j\}_{\mathsf{P}_j \in C}$, where $\text{view}_j$ is the view of party $\mathsf{P}_j$ in the execution. The values $\{\text{view}_j\}_{\mathsf{P}_j \in C}$ constitute exactly the information leaked to the corrupted parties, $C$, during an execution. In the following we therefore call the values $\{\text{view}_j\}_{\mathsf{P}_j \in C}$ the leaked values.

We then want to say that the leaked values do not allow the corrupted parties to learn anything that they should not learn. It is clear that the corrupted parties necessarily must learn their own inputs and outputs, in fact this is the entire purpose of the protocol. We therefore call the values $\{x_j, y_j\}_{\mathsf{P}_j \in C}$ the allowed values.

In these new terms we can give the following informal definition of what it means for a protocol to be private.

*A protocol is private if it always holds that the leaked values contain no more information than the allowed values.*

This leaves us with the problem of defining what it means for one value $V_1$ to contain no more information than some other value $V_2$. One very convenient way to define this is to note that if $V_1$ can be computed from (only) $V_2$, then clearly $V_1$ cannot contain more information than $V_2$. However, this definition would be a bit too liberal for our use, in particular if we consider security which is guaranteed using cryptography. Suppose, for instance, that we make $V_2 = n$ public where $n = pq$ is an integer that is the product of two large primes $p, q$. And let us define $V_1 = (p, q)$. Now, since the prime factors of integers are unique, it is certainly *possible* to compute $V_1 = (p, q)$ from $V_2 = n$. Nevertheless, if factoring large integers is a hard problem (as we hope if we use the RSA public-key cryptosystem), we could make $n$ public and expect that, in practice, $p, q$ will remain secret.

From this point of view, one would claim that $V_2 = (p, q)$ contains more than $V_1 = n$, because given $V_2$ we can efficiently compute much more than if we were given only $V_1$. We therefore say that a value $V_1$ contains no more knowledge than $V_2$ if $V_1$ can be computed *efficiently* from $V_2$.

Following this line of thought, we get the following more developed definition of privacy:

*A protocol is private if it always holds that the leaked values can be computed efficiently from the allowed values.*

To prove that a protocol is private one therefore has to show that there exists an efficient algorithm which takes the allowed values as input and outputs the leaked values. This program, which demonstrates that it is possible to efficiently compute the leaked values from the allowed values, is what we called the simulator. By our latest attempt at a definition, the simulator would be required to efficiently compute the leaked values from the allowed values. Note, however, that the traces view$_i$ are generated by the parties in the protocol, and they are allowed to make random choices. Therefore $\{\text{view}_j\}_{\mathsf{P}_j \in C}$ is not a fixed value, but actually a random variable. Therefore the simulator will also have to output a random variable, i.e., use internal random choices. This is why the actual definition of privacy says that

*There exists an efficient simulator* $\mathsf{S}$ *such that the* simulated values $\mathsf{S}(\{x_j, y_j\}_{\mathsf{P}_j \in C})$ *and the leaked values* $\{\text{view}_j\}_{\mathsf{P}_j \in C}$ *have the same* distribution.

### 4.1.2 Defining Robustness

We now sketch how we define robustness. Robustness has to do with how an attack influences the protocol and achieves some effect, in that the outputs may be different. In other words, an attack on robustness tries to achieve influence rather than knowledge. For the definition, we will therefore play exactly the same game as we did with privacy, just using influence as the basic currency instead of knowledge.

When we consider robustness, we assume that the corrupted parties are what we call Byzantine. This just means that we make absolutely no assumptions on how they behave. Concretely, we assume that an adversary has taken control over the corrupted parties, and every time the protocol instructs them to send some message, which is supposed to be computed in a particular way according to the protocol, the adversary may instruct them send any other message. Our job, as protocol designer, is then to design the protocol to work correctly regardless of how the adversary behaves.

Before discussing influence in more general term it is instructive to see two examples of what constitutes influence.

**Example 4.1** Consider Protocol CEPS (Circuit Evaluation with Passive Security), and let us run it with three parties and $t = 1$. Let us compute the function $f(x_1, x_2, x_3) = (x_1x_2x_3, x_2, x_3)$. That is, $\mathsf{P}_1$ is to learn $x_1x_2x_3$ and the other parties are only to learn their own input. Let us now assume that $\mathsf{P}_1$ is corrupted and show an example of how $\mathsf{P}_1$ can influence the protocol. Suppose $\mathsf{P}_1$ replaces its input $x_1$ by $x_1' = 1$, i.e., in the Input sharing phase it does a secret sharing of 1. Then it follows the rest of the protocol honestly. As a result $\mathsf{P}_1$ will learn $y_1 = x_1'x_2x_3 = 1x_2x_3 = x_2x_3$. $\triangle$

The above example shows that Protocol CEPS (Circuit Evaluation with Passive Security) allows $\mathsf{P}_1$ to influence the protocol in such a way that he always learns $x_2x_3$. The reader should take a moment to contemplate whether we should consider this a security problem? and if so, why?

**Example 4.2** Consider again Protocol CEPS (Circuit Evaluation with Passive Security) with three parties, $t = 1$ and $f(x_1, x_2, x_3) = (x_1x_2x_3, x_2, x_3)$. Say we order that gates in the corresponding circuit such that we first compute $[u]_t = [x_1x_2]_t$ and then $[y_1]_t = [ux_3]_y$. Finally, in the output phase, players send all their shares in $[y_1]_t$ to $\mathsf{P}_1$.[1] Let us again assume that $\mathsf{P}_1$ is corrupt. Now, however, we assume that $\mathsf{P}_1$ follows the protocol during the Input sharing phase using the input $x_1' = 0$. Then during the Multiplication protocol where $[u]_t = [x_1'x_2]_t$ is computed, players locally compute products that define a polynomial $h$ with $h(0) = x_1'x_2$. However, $\mathsf{P}_1$ does not distribute $[h(1)]_t$ as he is supposed to, but instead he distributes $[r_1^{-1} + h(1)]$, where we recall that $r_1$ is the first entry in the reconstruction vector. This means that the result

[1]Note that the other parties can just output their input, though the generic protocol would secret share their inputs and the reconstruct them towards the parties.

becomes

$$\begin{aligned}
[u]_t &= r_1[r_i^{-1} + h(1)]_t + \sum_{i=2}^{n} r_i[h(i)]_t \\
&= [r_1 r_1^{-1} + r_1 h(1) + \sum_{i=2}^{n} r_i h(i)]_t \\
&= [1 + \sum_{i=1}^{n} r_i h(i)]_t \\
&= [1 + x_1' x_2]_t \\
&= [1]_t \ ,
\end{aligned}$$

where the last equality follows because $x_1' = 0$. Then $\mathsf{P}_1$ follows the protocol honestly in the multiplication protocol for $[y_1]_t = [ux_3]_t$ and in the Reconstruction phase. As a result $\mathsf{P}_1$ will learn $y_1 = ux_3 = 1x_3 = x_3$. $\triangle$

The above example shows that Protocol CEPS (Circuit Evaluation with Passive Security) allows $\mathsf{P}_1$ to influence the protocol in such a way that $\mathsf{P}_1$ always learns $x_3$. Should we consider this a security problem? and if so, why?

It seems natural to say that a protocol is robust if the adversary cannot gain anything from influencing the protocol. Note, however, that if the adversary is able to take over a party $\mathsf{P}_i$, then there is some influence which is inevitable, namely input substitution. If, e.g., the adversary is controlling the party $\mathsf{P}_1$, then it can of course force $\mathsf{P}_1$ to use some value $x_1'$ as the input to the protocol instead of $x_1$. As a result, the protocol would compute $f(x_1', x_2, \ldots, x_n)$ instead of $f(x_1, x_2, \ldots, x_n)$. There is no way to combat this, as it is identical to the situation where an honest $\mathsf{P}_1$ is running with input $x_1'$, which is of course a perfectly allowable situation. We therefore have a certain allowed influence which we must accept. In the secure function evaluation protocol, the only allowed influence is input substitution.

In general we decide on what the allowed influence is by specifying an ideally secure way to solve the problem at hand. Then we say that any influence which is possible in this ideal world is allowed influence. To specify an ideal world for secure function evaluation, we could imagine a perfectly trusted hardware box which allows each $\mathsf{P}_i$ to securely input $x_i$ to the box without the other parties learning anything about $x_i$. Then the box computes $(y_1, \ldots, y_n) = f(x_1, \ldots, x_n)$ and outputs $y_i$ to $\mathsf{P}_i$ in such a way that the other parties learn nothing about $y_i$. After this the box blows up such that its internal state is lost forever. Except for the explosion, it is hard to imagine a more secure way to do function evaluation. And, it is easy to see that the only power a set of corrupted parties have in this ideal world is to pool their inputs and then compute some alternative inputs to the box, i.e., they can only do input substitution.

The influence that the attacker has on the actual protocol execution is called the actual influence. For a secure function evaluation protocol that runs on a network with secure channels, the actual influence of the adversary is that he can send arbitrary values to the honest parties on behalf of one or more of the corrupted parties.

We then say that a protocol is robust if no matter how the adversary uses his actual influence, the effect will always correspond to an effect of an allowed influence. This would, namely, show that whatever effect the actual influence has, the attacker could have obtained the same using an allowed influence, so the effect is by definition allowed. This leads to the following first attempt at defining robustness.

*A protocol is robust if it always holds that the effect of an actual influence can also be obtained using an allowed influence.*

To make this more precise, recall that when we considered privacy we said that a protocol was private if the actual values leaked by the protocol could be efficiently simulated given the allowed values. We will reuse this approach. So, we will require that for every adversary attacking the protocol one can efficiently compute an allowed way to influence the protocol which has the same effect. The efficient algorithm computing this allowed influence is called a simulator.

*A protocol is robust if there exists an efficient simulator $\mathsf{S}$, such that for every adversary attacking the protocol, $\mathsf{S}$ can efficiently compute an allowed influence with the same effect.*

For the secure function evaluation protocol, this definition just means that for each attack on the protocol resulting in outputs $(y'_1, \ldots, y'_n)$, one should be able to efficiently compute an input substitution for the corrupted parties which makes the ideal box for function evaluation give the same outputs $(y'_1, \ldots, y'_n)$.

Let us then return to the above two examples of possible attacks. In the first example $\mathsf{P}_1$ influences the protocol as to always learn $x_2 x_3$. This is not a problem, as $\mathsf{P}_1$ can obtain this by simply using $x_1 = 1$ as input, which is an allowed influence. So, this is not an attack on the robustness. In the second example $\mathsf{P}_1$ influences the protocol as to always learn $x_3$. This *is* an attack on the robustness, as there is no input substitution which allows $\mathsf{P}_1$ to always learn $x_3$. To see this, suppose we are in a case where $\mathsf{P}_2$ has input 0. This means, of course, that no matter how the other inputs are chosen, the result will be 0. In particular, no matter how $\mathsf{P}_1$ chooses his input, he will learn nothing about $x_3$. However, in the actual attack, he learns $x_3$ with certainty.

### 4.1.3 Combining Privacy and Robustness

The last attack we just saw looks at first like an attack on the correctness only: $\mathsf{P}_1$ makes the result be different from what it should be. However, it actually turns out to be an attack on the privacy as well: the incorrectness makes the protocol leak the input of $\mathsf{P}_3$ in cases where it should be perfectly hidden from $\mathsf{P}_1$. The fact that correctness and privacy can be entangled in this way makes it impossible to consider them in isolation. They are really two sides of the same thing.

What this concretely means for our full formal definition is that we must require existence of *one single simulator* that simultaneously demonstrates both privacy and robustness.

What this means is that the simulator receives information on how the adversary tries to influence the real protocol, and it must translate this efficiently into an allowed influence. In the "opposite" direction, it receives the allowed values and must efficiently simulate the leaked values. If this can be done, it essentially shows that anything that an adversary could obtain in an attack could also be obtained using an attack that is by definition harmless.

### 4.1.4 A Sketch of UC Security

In this subsection, we will sketch how our intuitive ideas on privacy and correctness materialize when we move towards a formal definition by considering players as computational entities. In Section 4.2 we then give the full formal model and definition of security.

The UC model was originally phrased in terms of interactive Turing machines. We will take a slightly more abstract approach and model protocols using so-called interactive systems, which in turn consists of a number of so-called interactive agents, which can communicate by sending messages on so-called ports. Interactive systems are defined in Chapter 2. Here we recall briefly the main terminology.

The ports of an agent are named and are divided into inports and outports. This is a simple mechanism for describing how the agents are connected. If some agent $\mathsf{A}_1$ outputs a message $m$ on an outport named P and some agent $A_2$ has an inport also named P, then $m$ will be input to $\mathsf{A}_2$ on the port P.

An interactive agent $\mathsf{A}$ can be thought of as a computer which has some open connections, or ports, on which it can receive and send messages. The main technical difference is that an agent only changes state when it is explicitly activated, and that the activation happens on an inport. Let AP be the name of an inport of A. An activation starts by inputting AP to A to tell it which port it was activated on — the port AP is called the activation port. Now A might read some of the messages waiting on its inports, possibly change state and possibly sends messages on some of its outports. This is called an activation of the agent. At the end of an activation, the agent also outputs the name RP of one of its outports — the port RP is called the return port. In a larger system, the return port specifies the next activation port. We think of this as some activation token @ being passed around between the agents.

An interactive system $\mathcal{IS}$ is just a set of agents, where no two agents have inports with the same name and no two agents have outports with the same name — this is to ensure that it is uniquely given how the ports should be connected.

If some agent A has an outport named P and some agent B has an inport named P, then we say that A and B are connected by P. Technically, when the interactive system is executed, a message queue will be associated to P. When A sends a message on P it is entered to the end of the queue. When B reads from P it pops the front element of the queue or receives EOQ if the queue is empty.

If some agent A in an interactive system $\mathcal{IS}$ has a inport named P and no agent in $\mathcal{IS}$ has an outport named P, then we call P an open inport of the system $\mathcal{IS}$. In the same way, if some agent A in $\mathcal{IS}$ has an outport named P and no agent in $\mathcal{IS}$ has an inport named P, then we call P an open outport of the system $\mathcal{IS}$.

An interactive system $\mathcal{IS}$ can receive messages from its environment on its open inports — these are just entered into the message queues for the open inports. An interactive system with open inports can also be activated. This happens by specifying an open inport AP of the system. The system is activated as follows: first the agent who has AP as inport is activated with activation port AP. As a result of this it reads messages on some of its inports, changes state and sends messages on some of its outports. It also specifies a return port RP. The agent with an inport named RP is then activated next, with the activation port begin RP, and so on. When at some point an agent specifies a return port RP which is an open outport of the system (such that there is no agent in the system with an inport named RP), the activation of the system stops. We say that $\mathcal{IS}$ returned with return port RP. As a result of this the system might have read some messages on its open inports, might have changed state, might have output messages on some of its open outports, and will have specified a return port RP. Hence an interactive system is much like an interactive agent — it just has more internal structure.

## Behavioral Equivalence

Seen from outside the system (not having access to the internal structure of $\mathcal{IS}$) the only observable events in an activation of $\mathcal{IS}$ is that some values were input on some open inports and that later some values appeared on some open outports. We are often not interested in the internal structure of a system (as it will correspond to how a protocol is implemented) but only this externally observable input/output behavior (as it will correspond to the inputs and outputs of the protocol, the allowed influence on the protocol and the leakage from the protocol). We therefore have a notion of two systems being indistinguishable if they give the same outputs on the same open outports whenever they get the same inputs on the same open inports. Because two indistinguishable systems need not have the same internal structure, as long as they behave in the same way, we sometimes call them behaviorally equivalent.

## Security by Behavioral Equivalence.

In the UC model, the security of a protocol is based on the notion of behaviorally equivalent systems.

The first step in the formalization is to model the protocol using an interactive system $\pi$. The system $\pi$ will contain an agent $\mathsf{P}_i$ for each party in the protocol and some agent $\mathcal{R}$ modeling the communication resource that the parties have access to, like an agent securely moving messages between the parties. The party $\mathsf{P}_i$ will have an inport $\mathtt{in}_i$ on which it receives inputs for the protocol and an outport $\mathtt{out}_i$ on which it delivers outputs from the protocol. It also has ports connecting it to the resource $\mathcal{R}$. The resource $\mathcal{R}$ takes care of moving messages between parties. In addition to moving messages, $\mathcal{R}$ also models the leakage of the communication network and the possible influence that an attacker might have on the communication network.

**Example 4.3** If we want $\mathcal{R}$ to model authenticated communication without order preservation of the messages, it could have an outport R.leak on which it would output all messages sent via $\mathcal{R}$, to model that authenticated channels not necessarily hide the messages sent, and it could have an inport R.infl on which it takes instruction about which order to deliver the messages in, to model that it does not preserve the ordering of messages. This is exactly the allowed leakage and the allowed influence on an authenticated channel without order preservation. △

To specify how a protocol is *supposed* to work, a potentially much simpler system F is formulated. The system F is formulated such that it always has the intended input/output behavior, such that it only leaks the allowed values, and such that it only allows the allowed influence. We sometimes call F the intended functionality and sometimes we call F the ideal functionality. The ideal functionality F is often without internal structure (just a single agent) as its only job is to have the correct input/output behavior, such that we can compare the protocol $\pi$ to F.

**Example 4.4** For the secure function evaluation problem, F could simply take one input $x_i$ from each party, on some inport $\mathtt{in}_i$ designated for that party, then it internally computes $(y_1, \ldots, y_n) = f(x_1, \ldots, x_n)$ and outputs each $y_i$ on an outport $\mathtt{out}_i$ designated for the party who is to learn $y_i$. Its only leakage would be that it reveals *when* a party provides input. The only influence would be that it allows the attacker to specify when the outputs are to be delivered.[2] To model this leakage and the influence, we give F an outport F.leak and an inport F.infl. When it receives $x_i$ on $\mathtt{in}_i$, it outputs "party $i$ gave input" on F.leak. After computing $(y_1, \ldots, y_n) = f(x_1, \ldots, x_n)$, if it receives an input "deliver to party $i$" on F.infl, it outputs $y_i$ on $\mathtt{out}_i$, unless it already gave output to party number $i$. △

Notice that if there existed such a thing as F in the real world and party number $i$ was magically and locally connected to $\mathtt{in}_i$ and $\mathtt{out}_i$ on F, then secure function evaluation would be easy: each party inputs $x_i$ to F and then at some point gets back $y_i$. The adversary would learn when and if F was used and determine when it delivers messages, but the parties would always be guaranteed that the results are correct and that their inputs are kept secret by F. It does not get better than this, which is why we call F the *ideal* functionality.

A protocol $\pi$ is then said to be secure if the system $\pi$ behaves "in the same way" as the ideal functionality F. This means that the security of a protocol $\pi$ is always relative to another system F. We can therefore not speak about a protocol $\pi$ being secure. We can only say that $\pi$ is at least as secure as F. This is natural, as a statement of the form $\pi$ *is secure* is actually absurd; It claims that the protocol $\pi$ is secure without even stating what problem the protocol $\pi$ is supposed to solve. The statement $\pi$ *is at least as secure as F* can however be made precise. The job of F in this statement is then to specify the task that $\pi$ is supposed to solve, by having the required input/output behavior, by allowing only the allowed influence and leaking only the allowed leakage. When $\pi$ is as secure as the intended functionality F, then we also say that $\pi$ securely implements F.

[2]Creating protocols where it is not visible when a party gives input and where an attacker with some control over the communication network cannot influence when the parties are ready to give outputs is hard. Hence we can turn this into allowed leakage and influence by adding it to F, to make F easier to realize.

### The Simulator

When comparing a protocol $\pi$ and an ideal functionality $\mathsf{F}$, we are left with the following problem. The ideal functionality $\mathsf{F}$ has a leakage port `F.leak` and an influence port `F.infl`, and it typically allows very limited leakage and influence. The protocol $\pi$ will consist of $n$ parties $\mathsf{P}_1, \dots, \mathsf{P}_n$, and these will be connected using an agent $\mathcal{R}$ modeling the communication network used by the protocol. All these agents will have their own leakage ports and influence ports, say `R.leak`, `R.infl` for $\mathcal{R}$.

Hence we cannot expect $\pi$ and $\mathsf{F}$ to be behaviorally equivalent, the sets of available ports are completely different. To make matters worse, the ports of the players and of $\mathcal{R}$ will typically allow much more leakage and influence than does $\mathsf{F}$.

We therefore introduce the simulator whose job it is to fix this problem. The simulator is another interactive agent $\mathcal{S}$ meant to be connected to $\mathsf{F}$. It connects to the leakage port `F.leak` of $\mathsf{F}$ and the influence port `F.infl` of $\mathsf{F}$, i.e., it sees the leakage of $\mathsf{F}$ and gets to influence $\mathsf{F}$. At the same time it has all the leakage and influence ports of the players in $\pi$ and of $\mathcal{R}$. So, if we connect $\mathcal{S}$ and $\mathsf{F}$, then $\mathsf{F} \diamond \mathcal{S}$ has the same set of open ports $\pi$. Now it makes sense to ask for $\pi$ and $\mathsf{F} \diamond \mathcal{S}$ to be behaviorally equivalent.

Here we used the notion $\mathcal{IS}_0 \diamond \mathcal{IS}_1$ for composing interactive systems. This notation is defined in detail i Chapter 2, here we just remind the reader that the two composed systems must have matching port names in order for the composition to be well defined.

Notice that whether $\pi$ and $\mathsf{F} \diamond \mathcal{S}$ are behaviorally equivalent depends heavily on the simulator $\mathcal{S}$. The job of $\mathcal{S}$ is to make the systems look the same to any distinguisher. I.e., it translates commands on the influence ports corresponding to the protocol into commands on the influence port of the ideal functionality, and it sees the values output on the leakage port of the ideal functionality and must then output values on the leakage ports corresponding to the protocol. It must do so in such a way that $\mathsf{F} \diamond \mathcal{S}$ behaves like $\pi$. In other words, the simulator simulates the leakage of the protocol using the the leakage from the ideal functionality and it simulates influence on the protocol using its influence on the ideal functionality. This is exactly according to our informal definitions of privacy and robustness from above. Note that we have not yet talked about how we model corruption of players, we will return to this shortly when we give the complete definition.

### Universal Composition

A primary reason for the popularity of the UC model is its so-called universal composition Theorem (UC Theorem). The UC Theorem says that if $\pi$ is a secure protocol for some task (specified by an intended functionality $\mathsf{F}$), then it is safe to use the protocol $\pi$ as a sub-protocol in any context where one needs to solve the task at hand. More precisely, if some protocol is secure when it uses $\mathsf{F}$ as an auxiliary resource, then that protocol is also secure if $\mathsf{F}$ is replaced by the protocol $\pi$. This allows to analyze a complex protocol $\pi_{\mathtt{CMPLX}}$ in an easier way by abstracting away some sub-protocol $\pi$: we replace calls to $\pi$ by calls to an ideal functionality $\mathsf{F}$ with the indented input/output behavior of $\pi$. We then prove two claims which are reduced in complexity, namely that 1) $\pi_{\mathtt{CMPLX}}$ is secure when it uses the ideal sub system $\mathsf{F}$ as resource and we prove that 2) the protocol $\pi$ securely implements $\mathsf{F}$. From this we get for free, using the UC Theorem, that $\pi_{\mathtt{CMPLX}}$ is also secure when it uses $\pi$ as sub protocol instead of calling $\mathsf{F}$.

## 4.2 The UC Model

We now proceed to give the formal details of our version of the UC framework.

### 4.2.1 Clock-Driven Execution

One minor deviation we do from the way the UC framework was originally formulated is that we use a clock-driven execution, where we introduce a very abstract notion of local clocks in most entities. The original formulation of the UC framework was more message driven. The reason for choosing clock-driven activation is that we want our framework to handle both synchronous protocols and asynchronous protocols. We can do this by either letting the local clocks be synchronized or by allowing them to drift apart. The original formulation of the UC framework was targeted against asynchronous protocols only, and hence does not capture synchronous protocols in a convenient way.

By a clocked entity in our framework we mean an interactive agent with some extra properties as specified below. Recall that in Chapter 2, we defined interactive systems that are composed of interactive agents. When executing such a system, an agent passes control to another by handing it an activation token. An interactive system of clocked entities will simply be a special case, where the token is passed according to certain rules that we will specify in a moment.

All clocked entities in our framework are going to have one or more inports with a name ending in `infl` or $\texttt{infl}_i$. This could, e.g., be an inport named `N.infl` or $\texttt{N.infl}_i$. If a clocked entity has an inport named `N.infl`, then it must have a matching outport named `N.leak`, called a leakage port. And vice versa, if a clocked entity has an outport named `N.infl`, then it must have a matching inport named `N.leak`. Among many other things these ports will be used to drive the execution of the interactive systems we consider.

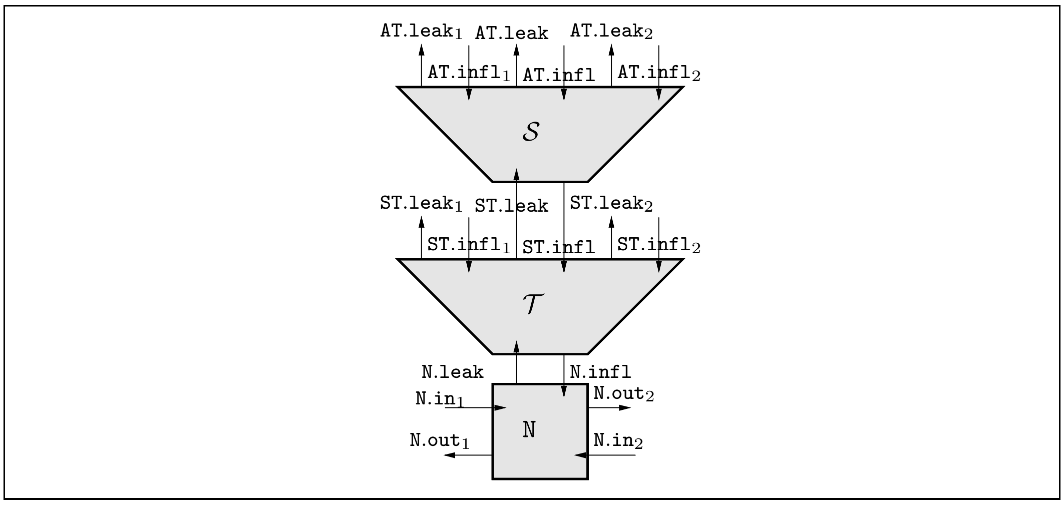


Figure 4.1: Diagram used for explaining the rules for activation.

A clocked entity receives a clocking signal (we say it is clocked) if it receives the activation token on an inport with a name of form `N.infl` (or $\texttt{N.infl}_i$). When it is clocked it must eventually return the activation on the matching outport `N.leak` (respectively $\texttt{N.leak}_i$). We introduce this rule, and other rules for passing the activation around, for technical reasons, which we discuss later. Before returning the activation token on `N.leak` or $\texttt{N.leak}_i$, a clocked entity C is allowed to do recursive calls to other clocked entities. Concretely, if C has an outport with a name of form `R.infl` or $\texttt{R.infl}_i$, it is allowed to send the activation token on that port. When it later receives the activation token back on `R.leak` respectively $\texttt{R.leak}_i$, it must either return the activation token on `N.leak` (respectively $\texttt{N.leak}_i$) or do another recursive call to a clocked entity. Summarizing, a clocked entity must obey the following list of rules for activation.

**initialization** A clocked entity holds as part of its state a bit active $\in \{0, 1\}$, which is initially set to 0. When active $= 0$ the clocked entity is said to be inactive. When active $= 1$ the clocked entity is said to be active. It also holds a bit calling $\in \{0,1\}$, which is initially set to 0. When calling $= 1$ the clocked entity is said to be calling.

**activation bounce during inactivity** If an inactive clocked entity receives the activation token on any port with a name not of the form `N.infl` or $\texttt{N.infl}_i$, then it returns the activation token on the matching outport without doing anything else, i.e., it does not read messages, it does not change state and it does not send messages.

**clocking** If a inactive clocked entity receives the activation token on an open inport with a name of the form `N.infl` or $\texttt{N.infl}_i$, then it sets active $\leftarrow 1$ and stores the name `CP` of the inport on which it was activated. We say that it was called on `CP`.

**return or clock** An active clocked entity is only allowed to send the activation token on an outport matching the inport `CP` on which it was called or an open outport with a name of the form `R.infl` or $\texttt{R.infl}_i$.

**return the call** If an active clocked entity sends the activation token on the outport matching the inport `CP` on which it was called, then it sets active $\leftarrow 0$. In that case we say that it returned the call.

**recursive call** If an active clocked entity sends the activation token on an open outport named `R.infl` or $\texttt{R.infl}_i$, it first sets calling $\leftarrow 1$ and stores the name `RP` of the port on which it did the recursive call. We say that it did a recursive call on `RP`.

**activation bounce during calls** If a *calling* clocked entity receives the activation token any inport `P` which does not match the outport `RP` on which it did the recursive call, then it returns the activation token on the outport matching `P` without doing anything else.

**result** If a *calling* clocked entity receives the activation token on the inport matching the outport `RP` on which it did the recursive call, then it sets calling $\leftarrow 0$.

Together, these rules ensure that all clocked entities pass the activation token in a simple recursive manner over matching influence and leakage ports, and that they only do work when they are called on their influence ports or after having being returned a recursive call.

As an example, see Fig. 4.1. If we call $\mathcal{S}$ on $\texttt{AT.leak}_1$, then it must return the activation on $\texttt{AT.leak}_1$. But before it does so, it is allowed to make a number of calls to $\mathcal{T}$ on `ST.infl`.[3] Whenever $\mathcal{S}$ does a recursive call to $\mathcal{T}$, the clocked entity $\mathcal{T}$ must return the activation on `ST.leak`, but before it does so, it is allowed to make a number of calls to the agent named `N` on `N.infl`.[4] Whenever $\mathcal{T}$ does a recursive call to `N`, the clocked entity `N` must return the activation on `N.leak`. And, since `N` has no outport ending in `infl` or $\texttt{infl}_i$, it is not allowed to do recursive calls before returning the activation. The net result of this is that when $\mathcal{S} \diamond \mathcal{T} \diamond \texttt{N}$ is called on $\texttt{AT.infl}_1$, then it may send messages on some of its outports, and it eventually returns the activation on $\texttt{AT.leak}_1$.

As another example, assume that we call $\mathcal{T}$ on $\texttt{ST.infl}_1$. In that case $\mathcal{T}$ must eventually return the call on $\texttt{ST.leak}_1$. Before doing so, it is allowed to do recursive calls to `N`. It is, however, not allowed to do recursive calls to $\mathcal{S}$, as it is connected to $\mathcal{S}$ by an outport that is a leakage and not an influence port, so this would be against the rule *return or clock*.

We can extend the definition of a clocked entity to an interactive system $\mathcal{IS}$ composed of several clocked entities. A system $\mathcal{IS}$ composed of clocked entities is defined to be active if and only if at least one of its constituents is active. It is calling if one its constituents is calling and has called an agent that it not part of $\mathcal{IS}$. It is now trivial to check that if $\mathcal{IS}$ consists of clocked entities, then $\mathcal{IS}$ itself also obeys all the above rules for clocked entities. That is, we have

[3]Note that $\mathcal{S}$ is *not* allowed to send the activation on any other outport, as it is only allowed to do recursive calls on ports ending in `infl` or $\texttt{infl}_i$.

[4]Note that $\mathcal{T}$ is *not* allowed to send the activation on any other outport.

**Lemma 4.1** *A composition of clocked entities is a clocked entity.*

Another advantage of only having simple recursive calls is that we can define a notion of polynomial time which is maintained under composition.

**Definition 4.1** *We say that a clocked entity is* recursive poly-time *if all its internal computation is expected polynomial time (in the security parameter $\kappa$) and it does at most an expected polynomial (in $\kappa$) number of recursive calls before it returns it own call.*

**Lemma 4.2** *A composition of recursive poly-time clocked entities is a recursive poly-time clocked entity.*

*Proof* Consider a system $\mathcal{IS}$ composed of recursive poly-time agents. It follows from Lemma 4.1 that $\mathcal{IS}$ is a clocked entity. It is also clear that $\mathcal{IS}$ is responsive, i.e., an activation is eventually returned. Now let us see that the total number of internal activations in any execution of $\mathcal{IS}$ is expected polynomial. This follows by induction on the number of agents. If $\mathcal{IS}$ has only 1 agent, the claim is trivial. So assume we have $i$ agents and consider an activation of $\mathcal{IS}$ and say agent $\mathtt{A}$ is the first one activated. Note that by the clocking rules the activation of $\mathcal{IS}$ ends when $\mathtt{A}$ returns its call. The expected number of calls $\mathtt{A}$ makes is bounded by a polynomial $p(\kappa)$ by definition. By the clocking rules, each such call is actually an activation of a composed clocked entity with $i-1$ agents. By induction hypothesis the expected number of internal calls in the smaller system is bounded by a polynomial $q(\kappa)$. This means that the total expected number of calls in $\mathcal{IS}$ is at most $p(\kappa)q(\kappa)$, which is polynomial. The fact that $\mathcal{IS}$ is responsive and has expected polynomial number of internal activations exactly means that $\mathcal{IS}$ is a poly-time interactive system in the sense we defined in Chapter 2. Hence, by Proposition 2.6, the expected running spent in any activation of $\mathcal{IS}$ is polynomial. □

In the following, we describe various types of agents we need in our framework, namely players in protocols, ideal functionalities and simulators. These will all be clocked entities and are therefore assumed to follow the rules for clocked entities. A typical behavior for such an agent when it is activated is to empty all its message queues and then return the call on the appropriate leakage port. When we describe concrete examples, we will not always say explicitly that calls are returned this way, as this is required behavior for any clocked entity.

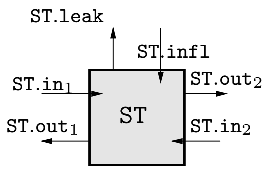
### 4.2.2 Ideal Functionalities

Formally an ideal functionality will be an interactive agent, as defined in Section 2.4. To name ports uniquely throughout the framework, each ideal functionality has a name $\mathtt{F}$. The interface of $\mathsf{F}$ is as follows: $\mathsf{F}$ has $n$ inports named $\mathtt{F.in}_1, \dots, \mathtt{F.in}_n$ and $n$ outports named $\mathtt{F.out}_1, \dots, \mathtt{F.out}_n$. These $2n$ ports are called the protocol ports. In addition to the protocol ports, $\mathsf{F}$ has two special ports, an inport $\mathtt{F.infl}$ called the influence port and an outport $\mathtt{F.leak}$ called the leakage port.

**Example 4.5** As an example, we specify an ideal functionality $\mathsf{F}_{\mathsf{ST}}$ for secure transfer as in Agent $\mathrm{F}_{\mathrm{ST}}$. Only the port structure for two parties is shown. The code is general enough to handle any number of parties. Each time the ideal functionality is clocked, we apply the rules until all messages queues are empty and then return on $\mathtt{ST.leak}$. The commands on the influence port is used to determine in which order the messages are delivered. The leakage of $(mid, i, j, |m|)$ specifies that also an implementation of $\mathsf{F}_{\mathsf{ST}}$ is allowed to leak the message identifier and the length of $m$. Leaking $|m|$ is important, as no cryptosystem can fully hide the size of the information being encrypted. Leaking $mid$ allows implementations to do the same, which might make implementation easier. △

Agent $\mathrm{F}_{\mathtt{ST}}$

The ideal functionality for secure transfer between $n$ parties.



- On input $(mid, j, m)$ on $\mathtt{ST.in}_i$ (where $j \in \{1, \dots, n\}$), output $(mid, i, j, |m|)$ on $\mathtt{ST.leak}$ and store $(mid, i, j, m)$. Ignore any later input of the form $(mid, j, \cdot)$ on $\mathtt{ST.in}_i$. Here $mid$ is a message identifier used to distinguish different messages from $i$ to $j$.

- On input $(\mathtt{deliver}, mid, i, j)$ on $\mathtt{ST.infl}$, where some $(mid, i, j, m)$ is stored, delete $(mid, i, j, m)$ and output $(mid, i, m)$ on $\mathtt{ST.out}_j$.

**Modeling Corruption**

The special ports of an ideal functionality are also used to model which information is allowed to leak when a party is corrupted and which control an adversary gets over a party when that party is corrupted. There are many choices, but we will assume that all ideal functionalities have the following standard corruption behavior.

- On input $(\mathtt{passive\ corrupt}, i)$ on $\mathtt{F.infl}$, the ideal functionality $\mathsf{F}$ outputs $(\mathtt{state}, i, \sigma)$ on $\mathtt{F.leak}$, where $\sigma$ is called the ideal internal state of party $i$ and is defined to be all previous inputs on $\mathtt{F.in}_i$ plus those on the message queue of $\mathtt{F.in}_i$, along with all previous outputs on $\mathtt{F.out}_i$.[5]

- On input $(\mathtt{active\ corrupt}, i)$ on $\mathtt{F.infl}$, the ideal functionality $\mathsf{F}$ records that the party $i$ is corrupted and outputs the ideal internal state of party $i$ on $\mathtt{F.leak}$. Then it starts ignoring all inputs on $\mathtt{F.in}_i$ and stops giving outputs on $\mathtt{F.out}_i$. Instead, whenever it gets an input $(\mathtt{input}, i, x)$ on $\mathtt{F.infl}$, it behaves exactly as if $x$ had arrived on $\mathtt{F.in}_i$, and whenever $\mathsf{F}$ is about to output some value $y$ on $\mathtt{F.out}_i$, it instead outputs $(\mathtt{output}, i, y)$ on $\mathtt{F.leak}$.

One way to think of a passive corruption is that if a party is corrupted, the adversary only learns the inputs and outputs of that party.[6]

One way to think about active corruption is that after a party has become actively corrupted, all its inputs are chosen by the adversary (via $\mathtt{F.infl}$) and all its outputs are leaked to the adversary (via $\mathtt{F.leak}$). Since it is impossible to protect against input substitution (even an otherwise honest party could do this) and it is inevitable that outputs intended for some party is seen by an adversary controlling that party, the standard corruption behavior models an ideal situation where an adversary gets only these inevitable powers.

[5]Formally we do not require that the ideal functionality empties its input queues and outputs all messages on the leakage tape, as this would not be poly-time behaviour, and in some settings we need that ideal functionalities are poly-time. We can solve this technical problem by instead saying that it outputs only the next message from $\mathtt{F.in}_i$. The adversary can then just input $(\mathtt{passive\ corrupt}, i)$ several times in a row to empty the input queue, if so desired. This is considered poly-time, as the ideal functionality responds in poly-time on each activation. Similar tricks are used later, but we stop mentioning them explicitly.

[6]A non-standard corruption behavior could be to only leak the last input or output. This would model an even more ideal situation, where an adversary cannot learn previous inputs and outputs when it breaks into a party. This is known as forward security and is a desirable property in some cases, but not a concern we will have in this book.

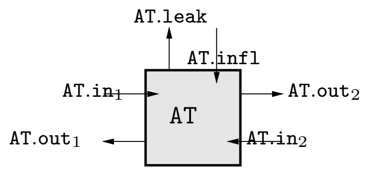
### 4.2.3 Protocols

A simple protocol $\pi$ consists of just $n$ parties, $\mathsf{P}_1, \ldots, \mathsf{P}_n$, where each $\mathsf{P}_i$ is an agent, i.e., the protocol is the interactive system $\pi = \{\mathsf{P}_1, \ldots, \mathsf{P}_n\}$. A simple protocol has a protocol name $\mathtt{F}$. For reasons that will become obvious, we let a protocol $\pi$ and the ideal functionality $\mathsf{F}$ that $\pi$ is supposed to implement have the same name. A simple protocol also has a resource name $\mathtt{R}$. This is the name of the resource that $\pi$ uses for communication. The agent $\mathsf{P}_i$ is called a simple party, and it has six ports. The port structure of $\mathsf{P}_i$ is derived from the names $\mathtt{F}$ and $\mathtt{R}$. It has an inport $\mathtt{F.in}_i$ and an outport $\mathtt{F.out}_i$, exactly as the ideal functionality $\mathsf{F}$ that $\pi$ will be compared to later. Those two ports are called the protocol ports. In addition it has an outport named $\mathtt{R.in}_i$ and an inport named $\mathtt{R.out}_i$. Those two ports are called the resource ports. Finally $\mathsf{P}_i$ has an inport name $\mathtt{R.infl}_i$ and an outport named $\mathtt{R.leak}_i$. These are called the special ports, and are used to model corruption of $\mathsf{P}_i$ (and to clock $\mathsf{P}_i$). This is detailed below, but first we discuss how the parties in a protocol communicate.

Let $\pi = \{\mathsf{P}_1, \ldots, \mathsf{P}_n\}$ be a protocol with resource name $\mathtt{R}$ and let $\mathcal{R}$ be an ideal functionality with name $\mathtt{R}$. Then $\mathcal{R}$ has an inport named $\mathtt{R.in}_i$ and $\mathsf{P}_i$ has an outport named $\mathtt{R.in}_i$, and $\mathcal{R}$ has an outport named $\mathtt{R.out}_i$ and $\mathsf{P}_i$ has an inport named $\mathtt{R.out}_i$. This means that in the system $\pi \diamond \mathcal{R} = \{\mathsf{P}_1, \ldots \mathsf{P}_n, \mathcal{R}\}$, the resource ports of the parties are connected to the protocol ports of $\mathcal{R}$. Hence the only open ports in $\pi \diamond \mathcal{R}$ are the protocol ports of $\pi$ and the special ports of $\pi$ and $\mathcal{R}$. We call $\pi \diamond \mathcal{R}$ a protocol using resource $\mathcal{R}$. Notice that this way ideal functionalities can play two roles, they can be specification of intended behavior, but they can also play the role as network resources, i.e., the means by which parties communicate. As we will see, this duality is central in formulating the UC Theorem.

Agent $\mathrm{F}_{\mathtt{AT}}$

The ideal functionality for authenticated transfer between $n$ parties.



- On receiving a value $(mid, j, m)$ on $\mathtt{AT.in}_i$ (where $j \in \{1, \ldots, n\}$), output $(mid, i, j, m)$ on $\mathtt{AT.leak}$ and store $(mid, i, j, m)$. Ignore any later input of the form $(mid, j, \cdot)$ on $\mathtt{AT.in}_i$.
- On an input $(\mathtt{deliver}, mid, i, j)$ on $\mathtt{AT.infl}$, where some $(mid, i, j, m)$ is stored, delete $(mid, i, j, m)$ and output $(mid, i, m)$ on $\mathtt{AT.out}_j$.

Agent $\mathrm{P}_1$

1. On input $(mid, 2, m)$ on $\mathtt{ST.in}_1$, output $(mid, 2, \mathtt{hello})$ on $\mathtt{AT.in}_1$.
2. On a later input $(mid, 2, ek)$ on $\mathtt{AT.out}_1$, sample a random encryption $c \leftarrow E_{ek}(m)$ and output $(mid, 2, c)$ on $\mathtt{AT.in}_1$.

**Example 4.6** We continue with the secure transfer example. We want to implement $\mathsf{F}_{\mathtt{ST}}$ using an authenticated channel and a public-key encryption scheme. For starters, let us consider a

Agent $\mathrm{P}_2$

1. On input $(mid, 1, \mathtt{hello})$ on $\mathtt{AT.out}_2$, sample a random key pair $(ek, dk)$ for an IND-CPA secure public-key cryptosystem and output $(mid, 1, ek)$ on $\mathtt{AT.in}_2$.
2. On a later input $(mid, 1, c)$ on $\mathtt{AT.out}_2$, output $(mid, 1, D_{dk}(c))$ on $\mathtt{ST.out}_2$.

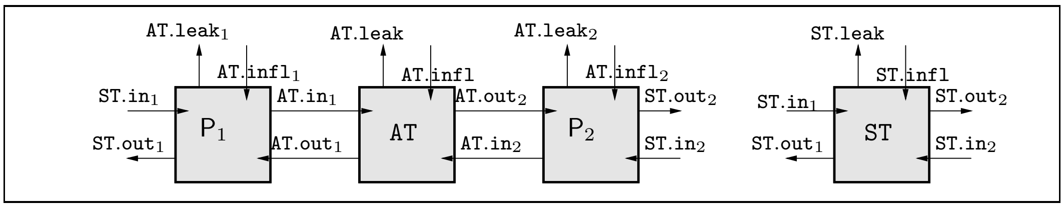


Figure 4.2: The protocol $\pi_{\mathtt{ST}}$ for secure transfer and the ideal functionality $\mathsf{F}_{\mathtt{ST}}$ for secure transfer

sender $\mathsf{P}_1$ and a receiver $\mathsf{P}_2$. These will communicate using an authenticated channel. To do a secure transfer from $\mathsf{P}_1$ to $\mathsf{P}_2$ one can use the following protocol:

1. First $\mathsf{P}_1$ announces over the authenticated channel to $\mathsf{P}_2$ that it wants to send a message securely.
2. Then $\mathsf{P}_2$ samples a key pair $(ek, dk)$ and sends the encryption key $ek$ to $\mathsf{P}_1$ over the authenticated channel.
3. Then $\mathsf{P}_1$ encrypts the message, $c \leftarrow E_{ek}(m)$, and returns $c$ over the authenticated channel.
4. Then $\mathsf{P}_2$ outputs $m = D_{dk}(c)$.

We want to formally model this protocol within the UC model.

To model the above protocol for secure transfer we need an ideal functionality $\mathsf{F}_{\mathtt{AT}}$ for authenticated transfer, to use as resource for the protocol, see Agent $\mathrm{F}_{\mathtt{AT}}$. The only difference from $\mathsf{F}_{\mathtt{ST}}$ is that $m$ is leaked, and not just $|m|$. This models that $m$ is allowed to leak in an authenticated channel. The protocol $\pi_{\mathtt{ST}}$ is given by Agent $\mathrm{P}_1$ and Agent $\mathrm{P}_2$. Similar code is included for the direction from $\mathsf{P}_2$ to $\mathsf{P}_1$, and similarly between each other pair of parties. In Fig. 4.2 we show the protocol $\pi_{\mathtt{ST}} \diamond \mathsf{F}_{\mathtt{AT}}$ next to the ideal functionality $\mathsf{F}_{\mathtt{ST}}$ that it tries to implement. $\triangle$

### Modeling Corruption

Back to how corruption is modeled. Again there are many choices, but we assume that all parties have the following standard corruption behavior:

- If a party $\mathsf{P}_i$ receives a special symbol (`passive corrupt`) on $\mathtt{R.infl}_i$, then $\mathsf{P}_i$ returns its internal state $\sigma$ on $\mathtt{R.leak}_i$. The internal state $\sigma$ consists of all randomness used by the party so far along with all inputs sent and received on its ports and the messages in the message queues of its inports.
- If a party $\mathsf{P}_i$ receives a special symbol (`active corrupt`) on $\mathtt{R.infl}_i$, then $\mathsf{P}_i$ outputs it current state on $\mathtt{R.leak}_i$ and starts executing the following rules, and only these rules:
  - On input $(\mathtt{read}, \mathtt{P})$ on $\mathtt{R.infl}_i$, where $\mathtt{P}$ is an inport of $\mathsf{P}_i$, it reads the next message $m$ on $\mathtt{P}$ and returns $m$ on $\mathtt{R.leak}_i$.
  - On input $(\mathtt{send}, \mathtt{P})$ on $\mathtt{R.infl}_i$, where $\mathtt{P}$ is an outport of $\mathsf{P}_i$, it sends $m$ on $\mathtt{P}$.

By passive corrupting a party in a protocol $\pi \diamond \mathcal{R}$ using resource $\mathcal{R}$, we mean that (`passive corrupt`) is input on $\texttt{R.infl}_i$ and then ($\texttt{passive corrupt}, i$) is input on `R.infl` on the communication device $\mathcal{R}$. By active corrupting a party in $\pi \diamond \mathcal{R}$ we mean that (`active corrupt`) is input on $\texttt{R.infl}_i$ and then ($\texttt{active corrupt}, i$) is input on `R.infl` on $\mathcal{R}$. After that $\mathsf{P}_i$ can be controlled using read and send commands.

### 4.2.4 The Simulator

If we inspect Fig. 4.2 we see that we have the problem discussed in the introduction to this chapter that a protocol, like $\pi_{\texttt{ST}} \diamond \mathsf{F}_{\texttt{AT}}$, and the ideal functionality specifying its intended behavior, like $\mathsf{F}_{\texttt{ST}}$, have different open port structures and hence cannot be behaviorally equivalent. We also discussed that we solve this be introducing a simulator $\mathcal{S}$ which gives $\pi_{\texttt{ST}} \diamond \mathsf{F}_{\texttt{AT}}$ and $\mathsf{F}_{\texttt{ST}} \diamond \mathcal{S}$ the same open port structure. An example is shown in Fig. 4.3.

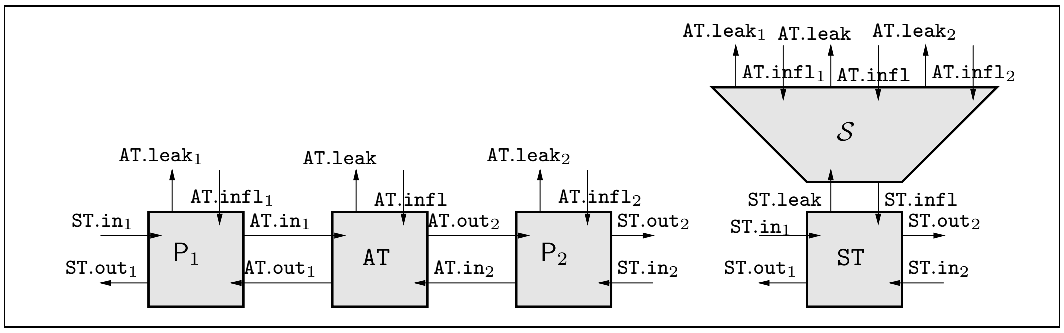


Figure 4.3: Port structure of the protocol $\pi_{\texttt{ST}} \diamond \mathsf{F}_{\texttt{AT}}$ for secure transfer and the ideal functionality $\mathsf{F}_{\texttt{ST}}$ for secure transfer composed with a simulator $\mathcal{S}$ for $\pi_{\texttt{ST}}$.

In general, a simulator $\mathcal{S}$ for a simple protocol $\pi$ with protocol name `F` and resource name `R` is defined as follows:

It is a poly-time[7] interactive system with an open inport named `F.leak` and an open outport named `F.infl`. These two ports allows it to connect to an ideal functionality $\mathsf{F}$ with name `F`, as such an $\mathsf{F}$ has an open outport named `F.leak` and an open inport named `F.infl`. In addition to these two ports $\mathcal{S}$ has ports corresponding to the special ports of $\pi$, i.e., for $i = 1, \ldots, n$ it has inports named $\texttt{R.infl}_i$ and outports named $\texttt{R.leak}_i$, like the parties of $\pi$; finally $\mathcal{S}$ has an inport named `R.infl` and an outport named `R.leak`, like a resource $\mathcal{R}$ used by $\pi$. As a consequence, $\mathsf{F} \diamond \mathcal{S}$ and $\pi \diamond \mathcal{R}$ has the same open ports.

We call the open ports of $\mathcal{S}$ which connects to $\mathsf{F}$ the ideal functionality ports of $\mathcal{S}$. We call the other open ports of $\mathcal{S}$ the simulation ports of $\mathcal{S}$.

We additionally require that $\mathcal{S}$ is corruption preserving in the sense that it does not corrupt a party on $\mathsf{F}$ unless that party was corrupted in the simulation. I.e., $\mathcal{S}$ does not output ($\texttt{passive corrupt}, i$) on `F.infl` unless it at some point received (`passive corrupt`) on $\texttt{R.infl}_i$, and $\mathcal{S}$ does not output ($\texttt{active corrupt}, i$) on `F.infl` unless it at some point received (`active corrupt`) on $\texttt{R.infl}_i$. We also require that if $\mathcal{S}$ receives (`passive corrupt`) on $\texttt{R.infl}_i$, then its next action is to output ($\texttt{passive corrupt}, i$) on `F.infl`, and similarly for active corruptions. This last requirement is a technicality needed when we later define so-called static security.

Finally we require that $\mathcal{S}$ is clock preserving. This is a technicality needed when we later define synchronous security. What we need is that $\mathcal{S}$ outputs ($\texttt{clockin}, i$) on `F.infl` if and only

[7]A poly-time interactive system is a system which returns on an open outport withing expected poly-time when it is activated on an open inport. For the interested reader, a formal definition is given in Definition 2.11.

if it just received $(\texttt{clockin}, i)$ on $\texttt{R.infl}_i$, and $\mathcal{S}$ outputs $(\texttt{clockout}, i)$ on $\texttt{F.infl}$ if and only if it just received $(\texttt{clockout}, i)$ on $\texttt{R.infl}_i$. We shall return to the use of these commands later.

### 4.2.5 The Environment

Having introduced the simulator we can now compare protocol $\pi$ using resource $\mathcal{R}$, to an ideal functionality $\mathsf{F}$ with an attached simulator $\mathcal{S}$ for the protocol. We will that by requiring that $\pi \diamond \mathcal{R}$ is "hard" to distinguish from from $\mathsf{F} \diamond \mathcal{S}$. Indistinguishability of interactive systems was defined in Definition 2.12, in Chapter 2. The technical details can be found there, here we remind the reader of the main ideas, this will be sufficient to understand our discussion in this chapter.

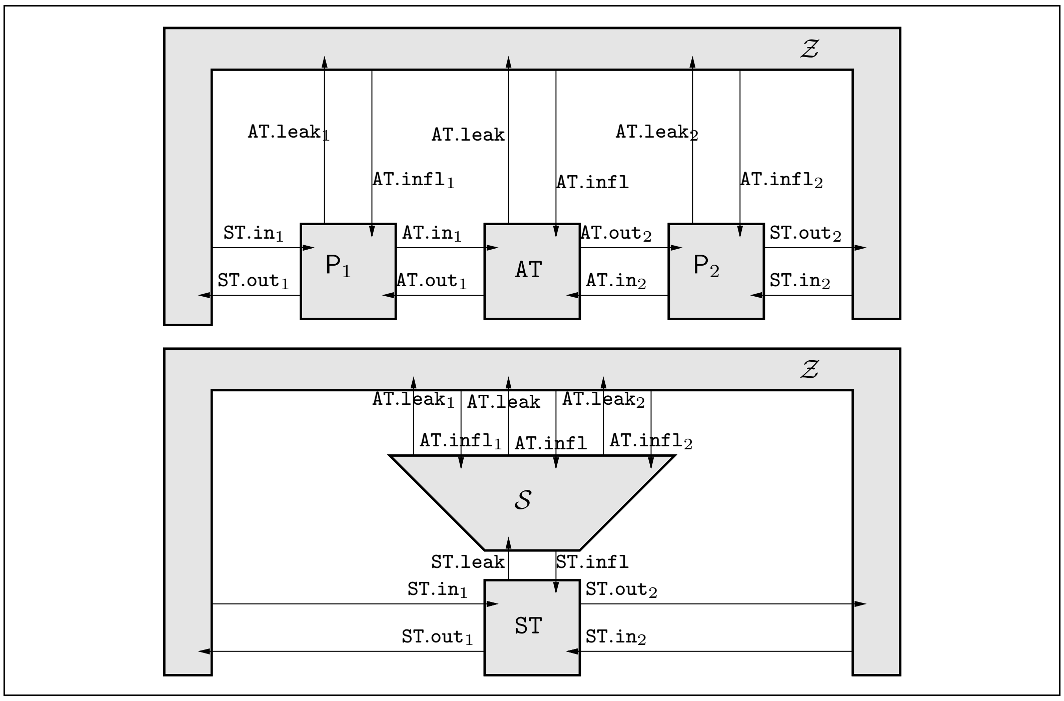


Figure 4.4: Example of two systems closed using the same environment $\mathcal{Z}$

The notion of indistinguishability requires that we consider an interactive system $\mathcal{Z}$ that must be an environment for $\pi \diamond \mathcal{R}$ and $\mathsf{F} \diamond \mathcal{S}$. This notion was defined for general interactive systems in Definition 2.12, and we now flesh out what it means in our concrete context of protocols and resources:

An environment $\mathcal{Z}$ for a simple protocol $\pi$ with protocol name $\texttt{F}$ and resource name $\texttt{R}$ is an interactive system $\mathcal{Z}$ which have the dual open ports of $\pi \diamond \mathcal{R}$ where $\mathcal{R}$ is a resource with name $\texttt{R}$. Concretely, for each open inport $\texttt{P}$ of $\pi \diamond \mathcal{R}$, the system $\mathcal{Z}$ has an open outport named $\texttt{P}$ and for each open outport $\texttt{P}$ of $\pi \diamond \mathcal{R}$, the system $\mathcal{Z}$ has an open inport named $\texttt{P}$. As a consequence $\pi \diamond \mathcal{R} \diamond \mathcal{Z}$ is a closed systems, and so is $\mathsf{F} \diamond \mathcal{S} \diamond \mathcal{Z}$. A closed system is just one with no open ports. See Fig. 4.4 for an example. Note that $\mathcal{S}$, being a simulator must be such that the port structure of $\pi \diamond \mathcal{R}$ and $\mathsf{F} \diamond \mathcal{S}$ are the same and therefore $\mathcal{Z}$ is an environment for both systems.

We call the open ports of $\mathcal{Z}$ which connects to the protocol ports of $\pi$ the protocol ports of $\mathcal{Z}$. We call the other open ports of $\mathcal{Z}$ the special ports of $\mathcal{Z}$.

Two systems are indistinguishable to an environment $\mathcal{Z}$ if $\mathcal{Z}$ cannot tell them apart (except with negligible advantage) by sending and receiving messages on the open ports of the systems. Two systems such as $\pi \diamond \mathcal{R}$ and $\mathsf{F} \diamond \mathcal{S}$ are called indistinguishable to a class $Z$ of environments if they are indistinguishable to all $\mathcal{Z} \in Z$. This is written $\pi \diamond \mathcal{R} \overset{Z}{\equiv} \mathsf{F} \diamond \mathcal{S}$. If, for instance, $Z$ is the class of polynomial time environments, we speak about computational indistinguishability and write $\pi \diamond \mathcal{R} \overset{\text{comp}}{\equiv} \mathsf{F} \diamond \mathcal{S}$. If $Z$ contains all environments (that activate the system only a polynomial number of times) we speak about statistical indistinguishability (written $\pi \diamond \mathcal{R} \overset{\text{stat}}{\equiv} \mathsf{F} \diamond \mathcal{S}$). Finally, if all environments have zero advantage in distinguishing, we speak about perfect indistinguishability (written $\pi \diamond \mathcal{R} \overset{\text{perf}}{\equiv} \mathsf{F} \diamond \mathcal{S}$).

### 4.2.6 Comparing Protocols to the Ideal Functionalities

We are now finally ready to define what it means that a protocol $\pi$ using, say, resource $\mathsf{F}_{\mathsf{AT}}$ does "the same" as an ideal functionality, say $\mathsf{F}_{\mathsf{ST}}$. The definition is general, but we reuse the names from the example for concreteness.

**Definition 4.2 (Security for simple protocols)** *Let $\mathsf{F}_{ST}$ be any ideal functionality with name $\mathsf{ST}$, let $\pi_{ST}$ be any simple protocol with protocol name $\mathsf{ST}$ and resource name $\mathsf{AT}$, and let $\mathsf{F}_{AT}$ be any ideal functionality with name $\mathsf{AT}$. We say that $\pi_{ST} \diamond \mathsf{F}_{AT}$ securely implements $\mathsf{F}_{ST}$ in environments $Z$ if there exists a simulator $\mathcal{S}$ for $\pi_{ST}$ such that $\pi_{ST} \diamond \mathsf{F}_{AT} \overset{Z}{\equiv} \mathsf{F}_{ST} \diamond \mathcal{S}$.*

*We will also write this as $\pi_{ST} \diamond \mathsf{F}_{AT} \overset{Z}{\geq} \mathsf{F}_{ST}$. We will sometimes say that $\pi_{ST} \diamond \mathsf{F}_{AT}$ is at least as secure as $\mathsf{F}_{ST}$ for environments in $Z$.*

By defining $Z$ appropriately, we can use the definiiton to talk talk about perfect, statistical, computational security of $\pi_{\mathsf{ST}}$.

For simplicity, this definition only considers protocols using a single resource. It is trivial to extend to several resources by allowing multiple resource names as long the port structure of simulators and environments are required to match this.

Agent $\mathcal{S}$

A simulator for $\pi_{\mathsf{ST}}$. This is how to simulate a secure transfer from $\mathsf{P}_1$ to $\mathsf{P}_2$. All other directions are handled similarly.

1. On input $(mid, 1, 2, l)$ on `ST.leak` (from $\mathsf{F}_{\mathsf{ST}}$, and with $l = |m|$) it outputs $(mid, 1, 2, \texttt{hello})$ on `AT.leak`.
2. On a later input $(\texttt{deliver}, mid, 1, 2)$ on `AT.infl` it samples a key pair $(ek, dk)$ and outputs $(mid, 2, 1, ek)$ on `AT.leak`.
3. On a later input $(\texttt{deliver}, mid, 2, 1)$ on `AT.infl` it samples a random encryption $c \leftarrow E_{ek}(m')$ and outputs $(mid, 1, 2, C)$ on `AT.leak`. Here $m' = 0^l$ is an all-zero message of length $l$.
4. On a later input $(\texttt{deliver}, mid, 1, 2)$ on `AT.infl` it outputs $(\texttt{deliver}, mid, 1, 2)$ on `ST.infl`, which makes $\mathsf{F}_{\mathsf{ST}}$ output $(mid, 1, m)$ on $\texttt{ST.out}_2$.

**Example 4.7** As an example, we prove that $\pi_{\mathsf{ST}} \diamond \mathsf{F}_{\mathsf{AT}}$ securely implements $\mathsf{F}_{\mathsf{ST}}$ in polynomial time environments. As a first step, we first consider a poly-time environment $\mathcal{Z}$ which does not corrupt any of the parties.

As we describe the simulator $\mathcal{S}$, it is instructive to look at Fig. 4.4. Recall that we try to construct some $\mathcal{S}$ for that setting such that $\mathcal{Z}$ cannot see which of the two systems it is playing with. The simulator is described in Agent $\mathcal{S}$.

The reason why $\mathcal{S}$ uses $m'$ and not the real $m$ is that $\mathsf{F}_{\mathsf{ST}}$ only outputs $l = |m|$ to $\mathcal{S}$. Giving $m$ to $\mathcal{S}$ would make the simulation trivial, but remember that the whole idea of $\mathcal{S}$ is to demonstrate that the real leakage can be simulated given only the leakage allowed by the ideal functionality, and $m$ is not allowed to leak in a secure transfer.

Consider now some distinguisher $\mathcal{Z}$ which gets to play with either $\pi_{\mathtt{ST}} \diamond \mathsf{F}_{\mathtt{AT}}$ or $\mathsf{F}_{\mathtt{ST}} \diamond \mathcal{S}$. For now we assume that $\mathcal{Z}$ does not use the special ports of the parties — i.e., it makes no corruptions. Furthermore, for simplicity, we only allow $\mathcal{Z}$ to do one secure transfer and to do it only from $\mathsf{P}_1$ to $\mathsf{P}_2$. Then $\mathcal{Z}$ works as follows:

1. It picks some message $m$ and inputs $(mid, 2, m)$ on $\mathtt{ST.in}_1$.
2. Then it sees $(mid, 1, 2, \mathtt{hello})$ on $\mathtt{AT.leak}$ and inputs $(\mathtt{deliver}, mid, 1, 2)$ on $\mathtt{AT.infl}$.
3. Then it sees some $(mid, 2, 1, ek)$ on $\mathtt{AT.leak}$ and inputs $(\mathtt{deliver}, mid, 2, 1)$ on $\mathtt{AT.infl}$.
4. Then it sees some $(mid, 1, 2, c'')$ on $\mathtt{AT.leak}$ and inputs $(\mathtt{deliver}, mid, 1, 2)$ on $\mathtt{AT.infl}$.
5. In response to this it sees some $(mid, 1, m'')$ output on $\mathtt{ST.out}_2$.

It could, of course, refuse some of the deliveries. This would, however, only have $\mathcal{Z}$ see less messages and thus make the distinguishing of the systems harder.

Note that by design of the simulator $\mathcal{S}$, the distinguisher $\mathcal{Z}$ will see both system behave as specified above. The only difference between the two systems is that

- If $\mathcal{Z}$ is playing with $\mathsf{F}_{\mathtt{ST}} \diamond \mathcal{S}$, then $c'' \leftarrow E_{ek}(0^{|m|})$ and $m''$ is the message $m'' = m$ output by $\mathsf{F}_{\mathtt{ST}}$.
- If it is playing with $\pi_{\mathtt{ST}} \diamond \mathsf{F}_{\mathtt{AT}}$, then $c'' \leftarrow E_{ek}(m)$ and $m'' = D_{dk}(c'')$ is the message output by $\mathsf{P}_2$.

If the encryption scheme has perfect correctness, then $D_{dk}(E_{ek}(m)) = m$, so $\pi_{\mathtt{ST}} \diamond \mathsf{F}_{\mathtt{AT}}$ and $\mathsf{F}_{\mathtt{ST}} \diamond \mathcal{S}$ always output the same $m''$. So, the only difference between the two systems is that $c'' \leftarrow E_{ek}(m)$ or $c'' \leftarrow E_{ek}(0^{|m|})$. So, a distinguisher $\mathcal{Z}$ essentially has the following job: Pick $m$ and receive $(ek, E_{ek}(m'))$, where $ek$ is random and $m' = m$ or $m' = 0^{|m|}$. Then try to distinguish which $m'$ was used. The definition of IND-CPA security more or less exactly says that no poly-time system can distinguisher $(ek, E_{ek}(m))$ and $(ek, E_{ek}(0^{|m|}))$. Hence it follows that $\pi_{\mathtt{ST}} \diamond \mathsf{F}_{\mathtt{AT}} \diamond \mathcal{Z} \overset{\text{stat}}{\equiv} \mathsf{F}_{\mathtt{ST}} \diamond \mathcal{S} \diamond \mathcal{Z}$ for all poly-time environments $\mathcal{Z}$ which do not corrupt parties and which do only one transfer, from $\mathsf{P}_1$ to $\mathsf{P}_2$. More formally, we can turn a poly-time system $\mathcal{Z}$ which distinguishes the two systems $\pi_{\mathtt{ST}} \diamond \mathsf{F}_{\mathtt{AT}}$ and $\mathsf{F}_{\mathtt{ST}} \diamond \mathcal{S}$ into a poly-time system $\mathcal{Z}'$ which wins in the IND-CPA game in Section 2.5:

1. The system $\mathcal{Z}'$ will play with $\mathsf{A}_0$ or $\mathsf{A}_1$ as defined in Section 2.5.
2. First $\mathcal{Z}'$ receives $ek$ from $\mathsf{A}_b$ (here $b = 0$ or $b = 1$).
3. Then $\mathcal{Z}'$ runs $\mathcal{Z}$ to see which message $(mid, 2, m)$ it outputs on $\mathtt{ST.in}_1$.
4. Then $\mathcal{Z}'$ inputs $(m, 0^{|m|})$ to $\mathsf{A}_b$ and gets back an encryption $c^*$, where $c^*$ is an encryption of $m$ if $b = 0$ and $c^*$ is an encryption of $0^{|m|}$ if $b = 1$.
5. Then $\mathcal{Z}'$ runs $\mathcal{Z}$ and shows it the messages $(mid, 1, 2, \mathtt{hello})$, $(mid, 2, 1, ek)$, $(mid, 1, 2, c^*)$ on $\mathtt{AT.leak}$ and $(mid, 1, m)$ on $\mathtt{ST.out}_2$.
6. Then $\mathcal{Z}'$ runs $\mathcal{Z}$ until it outputs its guess $c$, and then $\mathcal{Z}'$ outputs $c$.

If $b = 0$, then $c^*$ is a random encryption of $m$, and if $b = 1$, then $c^*$ is a random encryption of $0^{|m|}$. So, if $b = 0$, then $\mathcal{Z}$ see exactly the interaction it would see when interacting with $\pi_{\mathtt{ST}} \diamond \mathsf{F}_{\mathtt{AT}}$, and if $b = 1$, then $\mathcal{Z}$ sees exactly the interaction it would see when interacting with $\mathsf{F}_{\mathtt{ST}} \diamond \mathcal{S}$. So, if $\mathcal{Z}$ can distinguish $\pi_{\mathtt{ST}} \diamond \mathsf{F}_{\mathtt{AT}}$ and $\mathsf{F}_{\mathtt{ST}} \diamond \mathcal{S}$, then $c = b$ with probability significantly better than $\frac{1}{2}$. But this exactly means that $\mathcal{Z}'$, who also outputs $c$, will guess $b$ with probability significantly better than $\frac{1}{2}$. If the encryption scheme is IND-CPA secure, then this is a contradiction, as $\mathcal{Z}'$ is poly-time when $\mathcal{Z}$ is poly-time and IND-CPA security means that no poly-time system can distinguish $\mathsf{A}_0$ and $\mathsf{A}_1$.

It is fairly easy to extend the argument to poly-time environments $\mathcal{Z}$ which do not corrupt parties but are allowed to do any number of transfers. Each transfer is simulated as above, and the security follows from the fact that IND-CPA security is maintained even if you get so see many encryptions. Later we will extend the analysis also to poly-time environments which are allowed to corrupt parties. △

### 4.2.7 Composed Protocols

An important property of the UC framework is that when $\pi_{\mathtt{ST}} \diamond \mathsf{F}_{\mathtt{AT}}$ implements $\mathsf{F}_{\mathtt{ST}}$ with computational security, then $\mathsf{F}_{\mathtt{ST}}$ can securely be replaced by $\pi_{\mathtt{ST}} \diamond \mathsf{F}_{\mathtt{AT}}$ in any poly-time protocol.

Consider some third ideal functionality $\mathsf{F}_{\mathtt{N}}$ doing some interesting task ideally secure. Assume that we can design a protocol $\pi_{\mathtt{N}}$ with protocol name N and resource name ST, which securely implements $\mathsf{F}_{\mathtt{N}}$ when it uses $\mathsf{F}_{\mathtt{ST}}$ as communication resource. Designing a secure implementation of $\mathsf{F}_{\mathtt{N}}$ using secure transfer as communication resource is potentially much easier than designing a protocol using only authenticated transfer as communication resource. The structure of such a protocol is shown in the top row of Fig. 4.5, along with $\mathsf{F}_{\mathtt{N}}$.

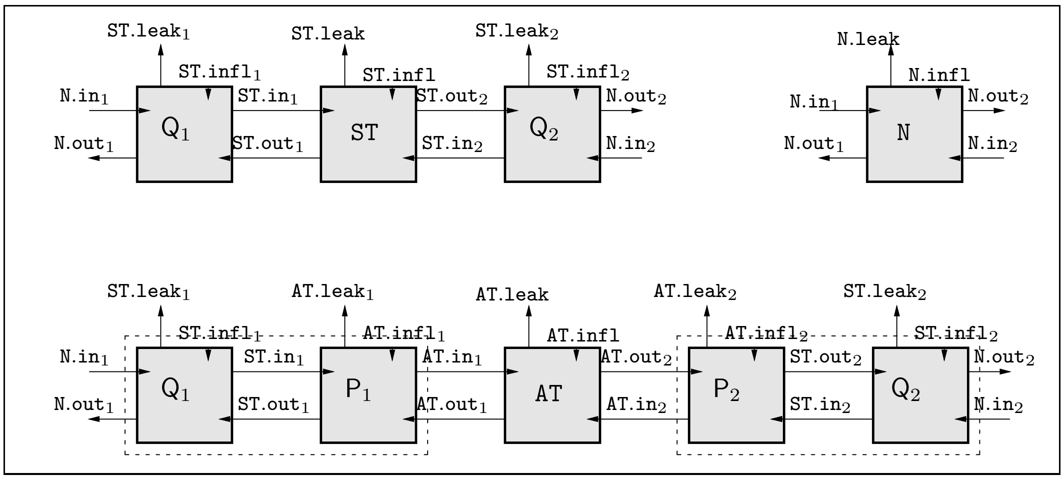


Figure 4.5: The protocol $\pi_{\mathtt{N}} \diamond \mathsf{F}_{\mathtt{ST}}$, the ideal functionality $\mathsf{F}_{\mathtt{N}}$, and the protocol $\pi_{\mathtt{N}} \diamond (\pi_{\mathtt{ST}} \diamond \mathsf{F}_{\mathtt{AT}})$

To get an implementation using only authenticated transfer, we can replace the use of $\mathsf{F}_{\mathtt{ST}}$ in $\pi_{\mathtt{N}} \diamond \mathsf{F}_{\mathtt{ST}}$ by the use of the protocol $\pi_{\mathtt{ST}} \diamond \mathsf{F}_{\mathtt{AT}}$, resulting in the protocol $\pi_{\mathtt{N}} \diamond (\pi_{\mathtt{ST}} \diamond \mathsf{F}_{\mathtt{AT}})$. This is possible as $\mathsf{F}_{\mathtt{ST}}$ and $\pi_{\mathtt{ST}} \diamond \mathsf{F}_{\mathtt{AT}}$ have the same structure of protocol ports. The result is shown in the bottom row in Fig. 4.5.

We have that $\pi_{\mathtt{N}} \diamond (\pi_{\mathtt{ST}} \diamond \mathsf{F}_{\mathtt{AT}}) = (\pi_{\mathtt{N}} \diamond \pi_{\mathtt{ST}}) \diamond \mathsf{F}_{\mathtt{AT}}$, as interactive systems are just sets of agents and $\diamond$ just the union operator on sets. We call $\pi_{\mathtt{N}} \diamond \pi_{\mathtt{ST}}$ a composed protocol. In general a composed protocol is just an interactive system composed of simple protocols. We use the term protocol to cover both simple protocols and composed protocols.

The protocol name of a composed protocol is the protocol name of the outer simple protocol, as it is this outer protocol which provides the open protocol ports. As an example, the protocol name of $\pi_{\mathtt{N}} \diamond \pi_{\mathtt{ST}}$ is N.

The resource name of a composed protocol is the protocol name of the inner simple protocol, as it is this inner protocol which provides the open resource ports. As an example, the resource name of $\pi_{\mathtt{N}} \diamond \pi_{\mathtt{ST}}$ is AT.

We can also compose protocols which are already composed, as long as the resource name of the outer composed protocol matches the protocol name of the inner composed protocol.

### Composed Parties

We consider $\mathsf{Q}_1$ and $\mathsf{P}_1$ as one party and consider $\mathsf{P}_2$ and $\mathsf{Q}_2$ as one party. This leads to a notion of a composed party, which are just interactive systems consisting of simple parties. In these words, a composed protocol $\pi$ simply consist of $n$ composed parties $\mathcal{P}_1, \dots, \mathcal{P}_n$, i.e., $\pi = \mathcal{P}_1 \diamond \cdots \diamond \mathcal{P}_n$. We use the term party to cover both simple parties and composed parties.

As an example, $\mathcal{P}_1 = \{\mathsf{Q}_1, \mathsf{P}_1\}$ is just an interactive system with open ports $\mathtt{AT.in}_1$ and $\mathtt{AT.out}_1$ connecting it to the protocol's communication device $\mathsf{F}_{\mathtt{AT}}$ and with open protocol ports $\mathtt{N.in}_1$ and $\mathtt{N.out}_1$ named as the ideal functionality $\mathsf{F}_{\mathtt{N}}$ that the protocol is trying to implement. The ports inside $\mathcal{P}$ are just particularities of how the party is implemented.

In addition $\mathcal{P}_1$ has some special ports which allow to corrupt it. We could insist on somehow joining these ports, but allowing to corrupt the components of $\mathcal{P}_1$ separately just gives more power to the attacker. A passive corruption of $\mathcal{P}_1$ is done by inputting (passive corrupt) on both $\mathtt{ST.infl}_1$ and $\mathtt{AT.infl}_1$, and inputting (passive corrupt, 1) on AT.infl, in response to which one receives the internal state of both components of the party plus the internal state of the party on the communication device. An active corruption of $\mathcal{P}_1$ is done by inputting (active corrupt) on both $\mathtt{ST.infl}_1$ and $\mathtt{AT.infl}_1$, and inputting (active corrupt, 1) on AT.infl.

### Security of Composed Protocols

We define security of composed protocols as we did for simple protocols. Consider the protocol $\pi_{\mathtt{N}} \diamond \pi_{\mathtt{ST}} \diamond \mathsf{F}_{\mathtt{AT}}$ in the bottom row of Fig. 4.5. If we want to ask if it is a secure implementation of $\mathsf{F}_{\mathtt{N}}$ in the upper right corner of Fig. 4.5, we have the usual problem that the port structures are different. We therefore introduce a simulator $\mathcal{U}$. As usual, this simulator has an open outport N.infl and an open inport N.leak, so it can connect to $\mathsf{F}_{\mathtt{N}}$ to exploit the allowed leakage and influence of this ideal functionality. Its job is then to simulate the leakage and influence of the protocol $\pi_{\mathtt{N}} \diamond \pi_{\mathtt{ST}} \diamond \mathsf{F}_{\mathtt{AT}}$. Therefore $\mathcal{U}$ must have an open port for each of the special ports in $\pi_{\mathtt{N}} \diamond \pi_{\mathtt{ST}} \diamond \mathsf{F}_{\mathtt{AT}}$, see Fig. 4.6.

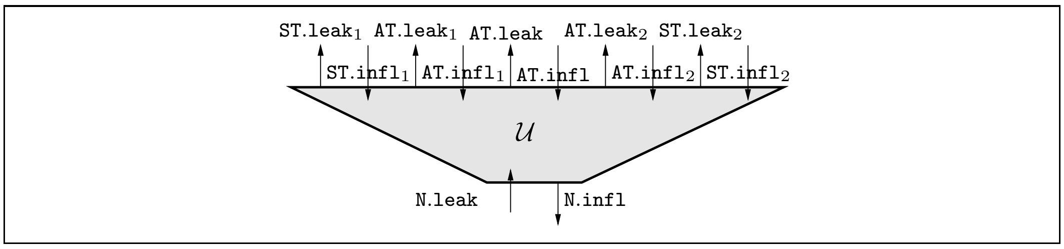


Figure 4.6: Open port structure of a simulator for the composed protocol $\pi_{\mathtt{N}} \diamond \pi_{\mathtt{ST}} \diamond \mathsf{F}_{\mathtt{AT}}$.

In general a simulator for a composed protocol $\pi$ with protocol name F and resource name R is just an interactive system $\mathcal{S}$ with an open outport F.infl and an open inport F.leak, so it can connect to the special ports of $\mathsf{F}_{\mathtt{F}}$. It must further have open ports that match the special ports of all simple parties and resources used in the composed protocol. With this extension of the simulator, we can reuse Definition 4.2 to say what it means for a composed protocol to implement a functionality.

## 4.2.8 The UC Theorem

We are now ready to phrase and prove the UC Theorem. Before doing so, it is, however, instructive to give a pictorial proof using our secure transfer example.

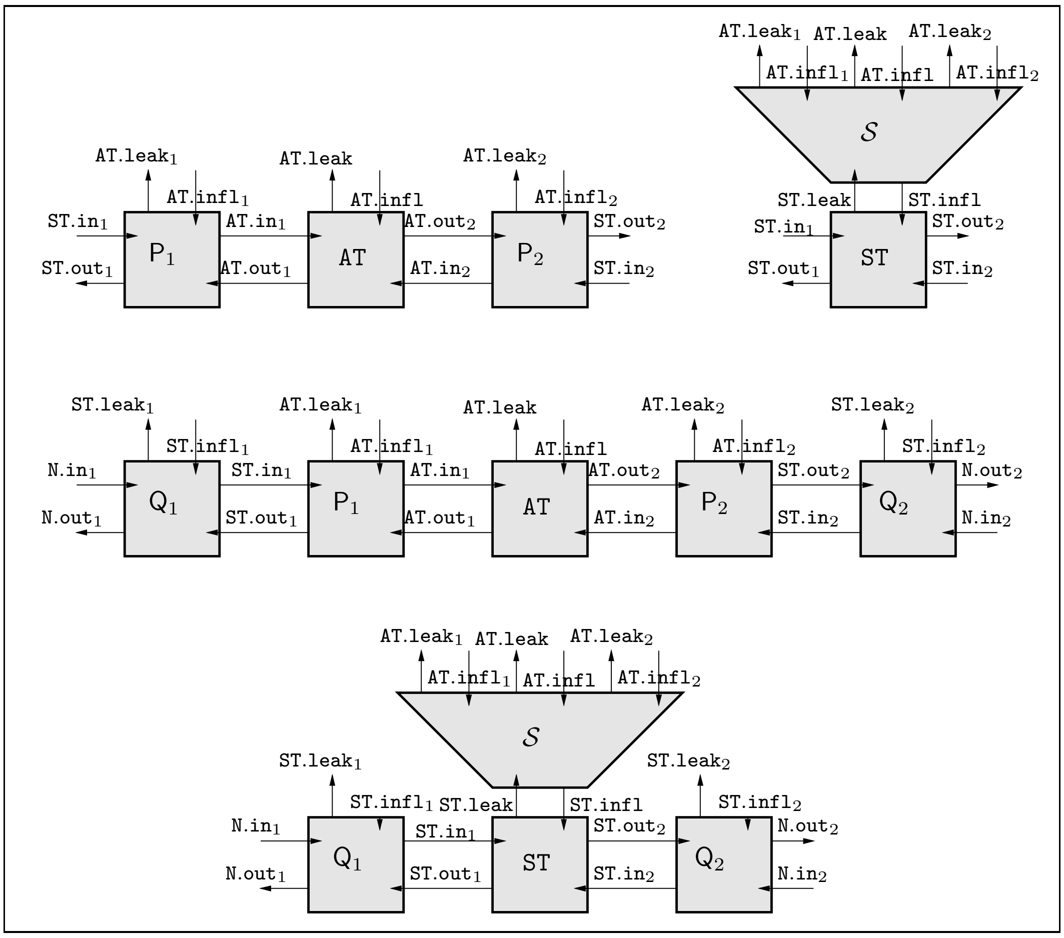


Figure 4.7: Extending $\pi_{\mathtt{ST}}$ and $\mathsf{F}_{\mathtt{ST}} \diamond \mathcal{S}$ with $\mathsf{Q}_1$ and $\mathsf{Q}_2$

**Example 4.8** Let $\mathcal{Z}$ be the set of poly-time environments. We assumed that $\pi_{\mathtt{N}} \diamond \mathsf{F}_{\mathtt{ST}} \overset{\text{Env}}{\geqslant} \mathsf{F}_{\mathtt{N}}$ and we argued that $\pi_{\mathtt{ST}} \diamond \mathsf{F}_{\mathtt{AT}} \overset{\text{Env}}{\geqslant} \mathsf{F}_{\mathtt{ST}}$, though we still did not consider all cases in the analysis. Those two security guarantees allows us to conclude that $\pi_{\mathtt{N}} \diamond \pi_{\mathtt{ST}} \diamond \mathsf{F}_{\mathtt{AT}} \overset{\text{Env}}{\geqslant} \mathsf{F}_{\mathtt{N}}$. Notice that if all the systems where real numbers and $\diamond$ was addition and $\overset{\text{Env}}{\geqslant}$ was $\geq$, then our assumptions would be that $\pi_{\mathtt{N}} + \mathsf{F}_{\mathtt{ST}} \geq \mathsf{F}_{\mathtt{N}}$ and $\pi_{\mathtt{ST}} + \mathsf{F}_{\mathtt{AT}} \geq \mathsf{F}_{\mathtt{ST}}$. From this we could conclude as follows: $\pi_{\mathtt{N}} + \pi_{\mathtt{ST}} + \mathsf{F}_{\mathtt{AT}} = \pi_{\mathtt{N}} + (\pi_{\mathtt{ST}} + \mathsf{F}_{\mathtt{AT}}) \geq \pi_{\mathtt{N}} + \mathsf{F}_{\mathtt{ST}} \geq \mathsf{F}_{\mathtt{N}}$. Then it follows from the transitivity of $\geq$ that $\pi_{\mathtt{N}} + \pi_{\mathtt{ST}} + \mathsf{F}_{\mathtt{AT}} \geq \mathsf{F}_{\mathtt{N}}$, which would correspond to $\pi_{\mathtt{N}} \diamond \pi_{\mathtt{ST}} \diamond \mathsf{F}_{\mathtt{AT}} \overset{\text{Env}}{\geqslant} \mathsf{F}_{\mathtt{N}}$. The proof of the UC Theorem follows this line of arguing. The proof goes as follows.

From $\pi_{\mathtt{ST}} \diamond \mathsf{F}_{\mathtt{AT}} \overset{\text{Env}}{\geqslant} \mathsf{F}_{\mathtt{ST}}$ we know that there exists a simulator $\mathcal{S}$ such that the two systems in the top row of Fig. 4.7 cannot be told apart by playing with their open ports. So, if we connect $\pi_{\mathtt{N}}$ to both systems, where $\pi_{\mathtt{N}}$ is represented by $\mathsf{Q}_1$ and $\mathsf{Q}_2$ in the figure, we know that the two bottom systems in Fig. 4.7 cannot be told apart. Namely, if some environment $\mathcal{Z}$ plays with the open ports of one of the bottom systems, it is really just playing with one of the system in the top row, via $\pi_{\mathtt{N}}$. I.e., $\mathcal{Z} \diamond \pi_{\mathtt{N}}$ is playing with one of the system in the top row. So, if $\mathcal{Z}$ distinguishes the bottom systems, then $\mathcal{Z} \diamond \pi_{\mathtt{N}}$ distinguishes the top systems. Since these systems cannot be told apart by any environment, it follows that no $\mathcal{Z}$ can distinguish the bottom systems.

From $\pi_{\mathtt{N}} \diamond \mathsf{F}_{\mathtt{ST}} \overset{\text{Env}}{\geqslant} \mathsf{F}_{\mathtt{N}}$ we know that there exists a simulator $\mathcal{T}$ such that the two systems in

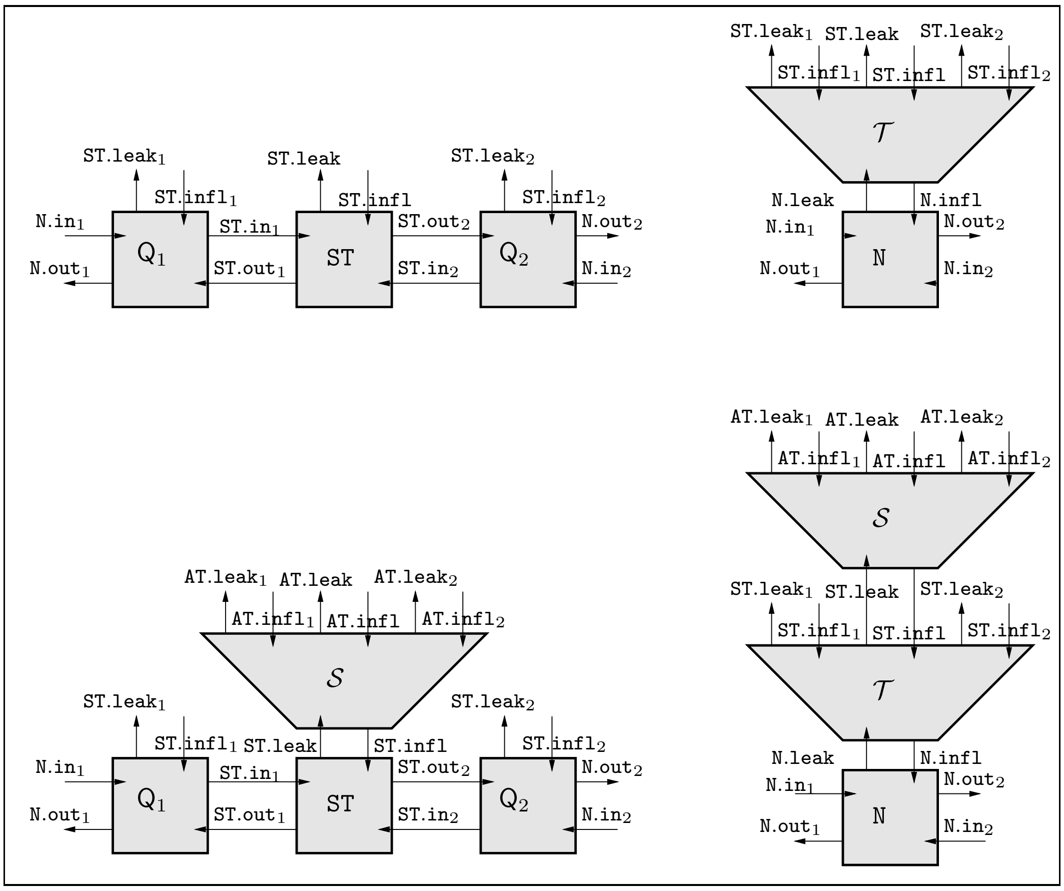


Figure 4.8: Extending $\pi_{\mathtt{N}}$ and $\mathsf{F}_{\mathtt{N}} \diamond \mathcal{T}$ with $\mathcal{S}$

the top row of Fig. 4.8 cannot be told apart by playing with their open ports. So, if we connect $\mathcal{S}$ to both systems, we know that the two bottom systems cannot be told apart by playing with their open ports, using the same logic as above. Here we turn a successful distinguisher $\mathcal{Z}$ for the bottom systems into a successful distinguisher $\mathcal{Z} \diamond \mathcal{S}$ for the top systems and reach a contradiction.

Now, if $\mathcal{IS}_0$ cannot be told apart from $\mathcal{IS}_1$ and $\mathcal{IS}_1$ cannot be told apart from $\mathcal{IS}_2$, it follows that $\mathcal{IS}_0$ cannot be told apart from $\mathcal{IS}_2$. This follows from transitivity of indistinguishability as shown in Chapter 2. We use this transitivity to finish the proof. The protocol $\pi_{\mathtt{N}} \diamond \pi_{\mathtt{ST}} \diamond \mathsf{F}_{\mathtt{AT}}$ is the middle system in Fig. 4.7. We concluded that this system cannot be told apart from the system in the bottom row of Fig. 4.7. The same system is found as the bottom, left system in Fig. 4.8, and we concluded that this system cannot be told apart from the bottom right system in Fig. 4.8. The bottom, right system in Fig. 4.8 is $\mathsf{F}_{\mathtt{N}} \diamond \mathcal{T} \diamond \mathcal{S}$. So, we can conclude that $\pi_{\mathtt{N}} \diamond \pi_{\mathtt{ST}} \diamond \mathsf{F}_{\mathtt{AT}}$ cannot be told apart from $\mathsf{F}_{\mathtt{N}} \diamond \mathcal{T} \diamond \mathcal{S}$. If we let $\mathcal{U} = \mathcal{T} \diamond \mathcal{S}$, then it follows that $\pi_{\mathtt{N}} \diamond \pi_{\mathtt{ST}} \diamond \mathsf{F}_{\mathtt{AT}}$ cannot be told apart from $\mathsf{F}_{\mathtt{N}} \diamond \mathcal{U}$. But that means that $\mathcal{U}$ successfully simulates $\pi_{\mathtt{N}} \diamond \pi_{\mathtt{ST}} \diamond \mathsf{F}_{\mathtt{AT}}$ given $\mathsf{F}_{\mathtt{N}}$,[8] so $\pi_{\mathtt{N}} \diamond \pi_{\mathtt{ST}} \diamond \mathsf{F}_{\mathtt{AT}}$ is a secure implementation of $\mathsf{F}_{\mathtt{N}}$. △

We are now almost ready to phrase and prove the UC Theorem. The last definition we need is the notion of a class of environments. The purpose of this notion is clear from the above example. Recall that we said that if some environment $\mathcal{Z}$ can tell apart the two bottom systems Fig. 4.7,

[8]Notice, in particular, that $\mathcal{T} \diamond \mathcal{S}$ is a systems which can connect to the special ports of $\mathsf{F}_{\mathtt{N}}$ and which has open special ports corresponding to $\pi n \diamond \pi st$.

then $\pi_{\mathtt{N}} \diamond \mathcal{Z}$ can tell apart the two systems in the top row of Fig. 4.7. We said that this leads to a contradiction as no environment can tell apart the two systems in the top row of Fig. 4.7 by assumption. Notice, however, that this requires that $\pi_{\mathtt{N}} \diamond \mathcal{Z}$ *is an environment.* We assume that *no environment* can tell apart the two systems in the top row of Fig. 4.7. So, if $\pi_{\mathtt{N}} \diamond \mathcal{Z}$ is not an environment, we do not get a contradiction. So, we need to require that if we take a protocol $\pi$ and an environment $\mathcal{Z}$ for $\pi$, then $\mathcal{Z} \diamond \pi$ is again an environment. For similar reasons we need to assume that if we take an environment $\mathcal{Z}$ and a simulator $\mathcal{S}$, then $\mathcal{Z} \diamond \mathcal{S}$ is again an environment. As an example, if we work with the class of environments which are poly-time, then we need that if we take such an environment $\mathcal{Z}$ and a protocol $\pi$, then $\mathcal{Z} \diamond \pi$ is again poly-time. If we define poly-time environments in a proper way and only consider poly-time protocols, then this will be the case, as we demonstrate below.

**Definition 4.3** *We let* Pro *be the set of simple and composed protocols where all simple parties follows the rules for clocked entities and are recursive poly-time.*

It is straight-forward to verify the following lemma.

**Lemma 4.3** *If* $\pi_{\mathtt{F}} \in \mathrm{Pro}$ *is a protocol with protocol name* $\mathtt{F}$ *and resource name* $\mathtt{R}$ *and* $\pi_{\mathtt{R}} \in \mathrm{Pro}$ *is a protocol with protocol name* $\mathtt{R}$ *and* $\pi_{\mathtt{F}} \diamond \pi_{\mathtt{R}} \neq \perp$*, then* $\pi_{\mathtt{F}} \diamond \mathtt{R} \in \mathrm{Pro}$*.*

**Definition 4.4** *We let* Sim *be the set of interactive systems* $\mathcal{S}$ *which are a simulator for some protocol. We remind the reader that this means that for* $\mathcal{S} \in \mathrm{Sim}$ *it holds that*

1. *There exists a protocol* $\pi_{\mathtt{F}}$ *(composed or simple) with protocol name* $\mathtt{F}$ *and an ideal functionality* $\mathsf{F}_{\mathtt{F}}$ *with name* $\mathtt{F}$ *such that* $\pi_{\mathtt{F}}$ *and* $\mathsf{F}_{\mathtt{F}} \diamond \mathcal{S}$ *have the same special ports.*

2. $\mathcal{S}$ *follows the rules for clocked entities and is recursive poly-time.*

3. $\mathcal{S}$ *is corruption preserving.*

4. $\mathcal{S}$ *is clock preserving.*

It is straight-forward to verify the following lemma.

**Lemma 4.4** *If* $\mathcal{S} \in \mathrm{Sim}$ *and* $\mathcal{T} \in \mathrm{Sim}$ *and* $\mathcal{T}$ *has two open special ports which connects it to the ideal functionality ports of* $\mathcal{S}$ *and* $\mathcal{S} \diamond \mathcal{T} \neq \perp$*, then* $\mathcal{S} \diamond \mathcal{T} \in \mathrm{Sim}$*.*

**Definition 4.5** *We say that* Env *is an* environment class *if the following holds.*

1. *Each* $\mathcal{Z} \in \mathrm{Env}$ *have the open port structure of an environment for some simple or composed protocol.*

2. *For all* $\mathcal{Z} \in \mathrm{Env}$ *and all* $\pi \in \mathrm{Pro}$ *where* $\pi \diamond \mathcal{Z} \neq \perp$ *and the protocol ports of* $\pi$ *connect to the protocol ports of* $\mathcal{Z}$ *in* $\pi \diamond \mathcal{Z}$ *it holds that* $\pi \diamond \mathcal{Z} \in \mathrm{Env}$*.*

3. *For all* $\mathcal{Z} \in \mathrm{Env}$ *and all* $\mathcal{S} \in \mathrm{Sim}$ *where* $\mathcal{S} \diamond \mathcal{Z} \neq \perp$ *and the simulation ports of* $\mathcal{S}$ *connect to the special ports of* $\mathcal{Z}$*, it holds that* $\mathcal{S} \diamond \mathcal{Z} \in \mathrm{Env}$*.*

In the following we call an environment $\mathcal{Z}$ recursive poly-time if it is poly-time and it makes at most an expected polynomial number of calls before it makes its guess.

**Proposition 4.5** *Let* $\mathrm{Env}^{poly}$ *be the set of all recursive poly-time systems which have an open port structure of an environment for some simple or composed protocol. Then* $\mathrm{Env}^{poly}$*is an environment class.*

*Proof* 1) The first property of an environment class is fulfilled by definition.

2) If $\mathcal{Z} \in \text{Env}^{\texttt{poly}}$ and $\pi \in \text{Pro}$ and the protocol ports of $\pi$ connect to the protocol ports of $\mathcal{Z}$ in $\pi \diamond \mathcal{Z}$, then $\pi \diamond \mathcal{Z}$ has an open port structure of an environment for some simple or composed protocol. Note that this is true because we compose $\mathcal{Z}$ with only $\pi$, we do not include the resource used by $\pi$. Hence, if $\pi$ has resource name $\texttt{R}$, $\pi \diamond \mathcal{Z}$ has an open port structure matching a protocol with protocol name $\texttt{R}$. So, to show that $\pi \diamond \mathcal{Z} \in \text{Env}^{\texttt{poly}}$ we just have to show that $\pi \diamond \mathcal{Z}$ is recursive poly-time. But since $\mathcal{Z} \in \text{Env}^{\texttt{poly}}$ it is recursive poly-time. Moreover $\pi \in \text{Pro}$ is also recursive poly-time by definition, so $\pi \diamond \mathcal{Z}$ is recursive poly-time, as desired.

3) If $\mathcal{Z} \in \text{Env}^{\texttt{poly}}$ and $\mathcal{S} \in \text{Sim}$ and the simulation ports of $\mathcal{S}$ connect to the special ports of $\mathcal{Z}$, it holds that $\mathcal{S} \diamond \mathcal{Z}$ has an open port structure of an environment for some simple or composed protocol. So, to show that $\mathcal{S} \diamond \mathcal{Z} \in \text{Env}^{\texttt{poly}}$ we just have to show that $\mathcal{S} \diamond \mathcal{Z}$ is recursive poly-time. This follows as for $\pi \diamond \mathcal{Z}$ as $\mathcal{S} \in \text{Sim}$ implies that $\mathcal{S}$ is recursive poly-time. □

**Exercise 4.1** Show that if $\text{Env}_1$ and $\text{Env}_2$ are environment classes, then $\text{Env}_1 \cap \text{Env}_2$ is an environment class and $\text{Env}_1 \cup \text{Env}_2$ is an environment class.

The following definition is very similar to Definition 4.2 but we now consider also composed protocols, and we make all the demands on environments that are necessary to prove the UC Theorem.

**Definition 4.6 (UC security for protocols)** *Let $\mathsf{F}_{\texttt{F}}$ be an ideal functionality with name $\texttt{F}$, let $\pi_{\texttt{F}}$ be a protocol with protocol name $\texttt{F}$ and resource name $\texttt{R}$, and let $\mathsf{F}_{\texttt{R}}$ be an ideal functionality with name $\texttt{R}$. Let* Env *be an environment class. We say that $\pi_{\texttt{F}} \diamond \mathsf{F}_{\texttt{R}}$ securely implements $\mathsf{F}_{\texttt{F}}$ in environments* Env *if there exists a simulator $\mathcal{S} \in \text{Sim}$ for $\pi_{\texttt{F}}$ such that $\pi_{\texttt{ST}} \diamond \mathsf{F}_{\texttt{AT}} \overset{\text{Env}}{\equiv} \mathsf{F}_{\texttt{ST}} \diamond \mathcal{S}$.*

*We will also write this as $\pi_{\texttt{F}} \diamond \mathsf{F}_{\texttt{R}} \overset{\text{Env}}{\geqslant} \mathsf{F}_{\texttt{F}}$, and we will sometimes say that $\pi_{\texttt{ST}} \diamond \mathsf{F}_{\texttt{AT}}$ is at least as secure as $\mathsf{F}_{\texttt{ST}}$ for environments in $Z$.*

**Theorem 4.6 (The UC Theorem)** *Let* Env *be an environment class. Let $\pi_{\texttt{F}} \in$* Pro *be a protocol with protocol name $\texttt{F}$ and resource name $\texttt{G}$. Let $\pi_{\texttt{G}}$ be a protocol with protocol name $\texttt{G}$ and resource name $\texttt{H}$ and for which $\pi_{\texttt{F}} \diamond \pi_{\texttt{G}} \neq \perp$. Let $\mathsf{F}_{\texttt{F}}$, $\mathsf{F}_{\texttt{G}}$ and $\mathsf{F}_{\texttt{H}}$ be ideal functionalities with names $\texttt{F}$, $\texttt{G}$ respectively $\texttt{H}$. If $\pi_{\texttt{F}} \diamond \mathsf{F}_{\texttt{G}} \overset{\text{Env}}{\geqslant} \mathsf{F}_{\texttt{F}}$ and $\pi_{\texttt{G}} \diamond \mathsf{F}_{\texttt{H}} \overset{\text{Env}}{\geqslant} \mathsf{F}_{\texttt{G}}$, then $(\pi_{\texttt{F}} \diamond \pi_{\texttt{G}}) \diamond \mathsf{F}_{\texttt{H}} \overset{\text{Env}}{\geqslant} \mathsf{F}_{\texttt{F}}$.*

*Proof* We start by writing out the premises of the theorem, to make them easier to use. We assume that $\pi_{\texttt{F}} \diamond \mathsf{F}_{\texttt{G}} \overset{\text{Env}}{\geqslant} \mathsf{F}_{\texttt{F}}$. This means that there exists a simulator $\mathcal{S} \in \text{Sim}$ for $\pi_{\texttt{F}}$ such that

$$\pi_{\texttt{F}} \diamond \mathsf{F}_{\texttt{G}} \diamond \mathcal{Z}_1 \overset{\text{stat}}{\equiv} \mathsf{F}_{\texttt{F}} \diamond \mathcal{S} \diamond \mathcal{Z}_1 \tag{4.1}$$

for all $\mathcal{Z}_1 \in \text{Env}$ for which the compositions make sense. Note that this is just a convenient (and equivalent) way to say that $\pi_{\texttt{F}} \diamond \mathsf{F}_{\texttt{G}} \overset{\text{Env}}{\equiv} \mathsf{F}_{\texttt{F}} \diamond \mathcal{S}$

We also assume that $\pi_{\texttt{G}} \diamond \mathsf{F}_{\texttt{H}} \overset{\text{Env}}{\geqslant} \mathsf{F}_{\texttt{G}}$. This means that there exists a simulator $\mathcal{T} \in \text{Sim}$ for $\pi_{\texttt{G}}$ such that

$$\pi_{\texttt{G}} \diamond \mathsf{F}_{\texttt{H}} \diamond \mathcal{Z}_2 \overset{\text{stat}}{\equiv} \mathsf{F}_{\texttt{G}} \diamond \mathcal{T} \diamond \mathcal{Z}_2 \tag{4.2}$$

for all $\mathcal{Z}_2 \in \text{Env}$ for which the compositions make sense.

We then write out the conclusion, to better see what we have to prove. We want to prove that $\pi_{\texttt{F}} \diamond \pi_{\texttt{G}} \diamond \mathsf{F}_{\texttt{H}} \overset{\text{Env}}{\geqslant} \mathsf{F}_{\texttt{H}}$. This means that there should exist a simulator $\mathcal{U} \in \text{Sim}$ for $\pi_{\texttt{F}} \diamond \pi_{\texttt{G}}$ such that

$$\pi_{\texttt{F}} \diamond \pi_{\texttt{G}} \diamond \mathsf{F}_{\texttt{H}} \diamond \mathcal{Z} \overset{\text{stat}}{\equiv} \mathsf{F}_{\texttt{F}} \diamond \mathcal{U} \diamond \mathcal{Z} \tag{4.3}$$

for all $\mathcal{Z} \in \text{Env}$ for which the compositions make sense.

The flow of the proof is as follows. We first show that

$$\pi_{\texttt{F}} \diamond \pi_{\texttt{G}} \diamond \mathsf{F}_{\texttt{H}} \diamond \mathcal{Z} \overset{\text{stat}}{\equiv} \pi_{\texttt{F}} \diamond \mathsf{F}_{\texttt{G}} \diamond \mathcal{T} \diamond \mathcal{Z} \ . \tag{4.4}$$

Then we show that

$$\pi_{\mathtt{F}} \diamond \mathsf{F}_{\mathtt{G}} \diamond \mathcal{T} \diamond \mathcal{Z} \overset{\text{stat}}{\equiv} \mathsf{F}_{\mathtt{F}} \diamond \mathcal{S} \diamond \mathcal{T} \diamond \mathcal{Z} \ . \tag{4.5}$$

Then we use transitivity of $\overset{\text{stat}}{\equiv}$ on Eq. 4.4 and Eq. 4.5 to conclude that

$$\pi_{\mathtt{F}} \diamond \pi_{\mathtt{G}} \diamond \mathsf{F}_{\mathtt{H}} \diamond \mathcal{Z} \overset{\text{stat}}{\equiv} \mathsf{F}_{\mathtt{F}} \diamond \mathcal{S} \diamond \mathcal{T} \diamond \mathcal{Z} \ . \tag{4.6}$$

Then we observe that if we $\mathcal{U} \overset{\text{def}}{=} \mathcal{S} \diamond \mathcal{T}$, then $\mathcal{U} \in \text{Sim}$. Plugging this into Eq. 4.6, we get exactly Eq. 4.3. What remains is therefore just to show Eq. 4.4 and Eq. 4.5.

Consider first Eq. 4.4 and let $\mathcal{Z}_2 \overset{\text{def}}{=} \pi_{\mathtt{F}} \diamond \mathcal{Z}$. Then

$$\pi_{\mathtt{G}} \diamond \mathsf{F}_{\mathtt{H}} \diamond \mathcal{Z}_2 = \pi_{\mathtt{G}} \diamond \mathsf{F}_{\mathtt{H}} \diamond \pi_{\mathtt{F}} \diamond \mathcal{Z} = \pi_{\mathtt{F}} \diamond \pi_{\mathtt{G}} \diamond \mathsf{F}_{\mathtt{H}} \diamond \mathcal{Z}$$
$$\mathsf{F}_{\mathtt{G}} \diamond \mathcal{T} \diamond \mathcal{Z}_2 = \mathsf{F}_{\mathtt{G}} \diamond \mathcal{T} \diamond \pi_{\mathtt{F}} \diamond \mathcal{Z} = \pi_{\mathtt{F}} \diamond \mathsf{F}_{\mathtt{G}} \diamond \mathcal{T} \diamond \mathcal{Z}$$

So, to prove Eq. 4.4 we just have to prove that

$$\pi_{\mathtt{G}} \diamond \mathsf{F}_{\mathtt{H}} \diamond \mathcal{Z}_2 \overset{\text{stat}}{\equiv} \mathsf{F}_{\mathtt{G}} \diamond \mathcal{T} \diamond \mathcal{Z}_2 \ . \tag{4.7}$$

This, however, follows directly from Eq. 4.2, as $\mathcal{Z}_2 \in \text{Env}$ when $\pi_{\mathtt{F}} \in \text{Pro}$ and $\mathcal{Z} \in \text{Env}$.

Consider then Eq. 4.5 and let $\mathcal{Z}_2 \overset{\text{def}}{=} \mathcal{T} \diamond \mathcal{Z}$. Then to prove Eq. 4.4 we just have to prove that

$$\pi_{\mathtt{F}} \diamond \mathsf{F}_{\mathtt{G}} \diamond \mathcal{Z}_1 \overset{\text{stat}}{\equiv} \mathsf{F}_{\mathtt{F}} \diamond \mathcal{S} \diamond \mathcal{Z}_1 \ . \tag{4.8}$$

This, however, follows directly from Eq. 4.1, as $\mathcal{Z}_1 \in \text{Env}$ when $\mathcal{S} \in \text{Sim}$ and $\mathcal{Z} \in \text{Env}$. □

### 4.2.9 Extensions

It is possible to extend the UC Theorem in several ways. As an example, we say (with the same notation as in the UC theorem) that $\pi_{\mathtt{F}}$ is a perfect secure implementation of $\mathsf{F}_{\mathtt{F}}$ in environments Env if there exist a simulator $\mathcal{S} \in \text{Sim}$ for $\pi_{\mathtt{F}}$ such that

$$\pi_{\mathtt{F}} \diamond \mathsf{F}_{\mathtt{R}} \diamond \mathcal{Z} \overset{\text{perf}}{\equiv} \mathsf{F}_{\mathtt{F}} \diamond \mathcal{S} \diamond \mathcal{Z}$$

for all $\mathcal{Z} \in \text{Env}$ instead of just

$$\pi_{\mathtt{F}} \diamond \mathsf{F}_{\mathtt{R}} \diamond \mathcal{Z} \overset{\text{stat}}{\equiv} \mathsf{F}_{\mathtt{F}} \diamond \mathcal{S} \diamond \mathcal{Z} \ .$$

The following theorem follows directly from the proof of Theorem 4.6.

**Theorem 4.7** *Let* Env *be an environment class. Let $\pi_{\mathtt{F}} \in \text{Pro}$ be a protocol with protocol name $\mathtt{F}$ and resource name $\mathtt{G}$. Let $\pi_{\mathtt{G}}$ be a protocol with protocol name $\mathtt{G}$ and resource name $\mathtt{H}$ and for which $\pi_{\mathtt{F}} \diamond \pi_{\mathtt{G}} \neq \bot$. Let $\mathsf{F}_{\mathtt{F}}$, $\mathsf{F}_{\mathtt{G}}$ and $\mathsf{F}_{\mathtt{H}}$ be ideal functionalities with names $\mathtt{F}$, $\mathtt{G}$ respectively $\mathtt{H}$. If $\pi_{\mathtt{F}} \diamond \mathsf{F}_{\mathtt{G}}$ is a perfect secure implementation of $\mathsf{F}_{\mathtt{F}}$ in the environments* Env *and $\pi_{\mathtt{G}} \diamond \mathsf{F}_{\mathtt{H}}$ is a perfect secure implementation of $\mathsf{F}_{\mathtt{G}}$ in the environments* Env*, then $(\pi_{\mathtt{F}} \diamond \pi_{\mathtt{G}}) \diamond \mathsf{F}_{\mathtt{H}}$ is a perfect secure implementation of $\mathsf{F}_{\mathtt{F}}$ in the environments* Env*.*

It is also possible to show the following theorem.

**Theorem 4.8** *Let* $\text{Env}_1$ *and* $\text{Env}_2$ *be environment classes. Let $\pi_{\mathtt{F}} \in \text{Pro}$ be a protocol with protocol name $\mathtt{F}$ and resource name $\mathtt{G}$. Let $\pi_{\mathtt{G}}$ be a protocol with protocol name $\mathtt{G}$ and resource name $\mathtt{H}$. Let $\mathsf{F}_{\mathtt{F}}$, $\mathsf{F}_{\mathtt{G}}$ and $\mathsf{F}_{\mathtt{H}}$ be ideal functionalities with names $\mathtt{F}$, $\mathtt{G}$ respectively $\mathtt{H}$. Assume that if $\mathcal{Z} \in \text{Env}_1$, then $\mathcal{Z} \diamond \pi_{\mathtt{F}} \in \text{Env}_2$. If $\pi_{\mathtt{F}} \diamond \mathsf{F}_{\mathtt{G}} \overset{\text{Env}_1}{\geqslant} \mathsf{F}_{\mathtt{F}}$ and $\pi_{\mathtt{G}} \diamond \mathsf{F}_{\mathtt{H}} \overset{\text{Env}_2}{\geqslant} \mathsf{F}_{\mathtt{G}}$, then $(\pi_{\mathtt{F}} \diamond \pi_{\mathtt{G}}) \diamond \mathsf{F}_{\mathtt{H}} \overset{\text{Env}_1}{\geqslant} \mathsf{F}_{\mathtt{F}}$.*

Intuitively $\pi_{\mathtt{G}} \diamond \mathsf{F}_{\mathtt{H}} \overset{\text{Env}_2}{\geqslant} \mathsf{F}_{\mathtt{G}}$ say that it is secure to use $\pi_{\mathtt{G}} \diamond \mathsf{F}_{\mathtt{H}}$ instead of $\mathsf{F}_{\mathtt{G}}$ in environments from the class $\text{Env}_2$. And the assumption that if $\mathcal{Z} \in \text{Env}_1$, then $\mathcal{Z} \diamond \pi_{\mathtt{F}} \in \text{Env}_2$ says that if $\pi_{\mathtt{F}}$ is run in an environment from the class $\text{Env}_1$, then it provides and environment from the class $\text{Env}_2$ for its resource. So, replacing $\mathsf{F}_{\mathtt{G}}$ by $\pi_{\mathtt{G}} \diamond \mathsf{F}_{\mathtt{H}}$ inside $\pi_{\mathtt{F}}$ is secure as long as $\pi_{\mathtt{F}}$ is run in an environment from the class $\text{Env}_1$.

**Exercise 4.2** Give a formal proof of Theorem 4.8 along the lines of the proof of Theorem 4.6.

Another, important, generalization is that we can consider protocols $\pi$ which use more than one communication resource. In that case $\pi$ has one set of protocol ports, named like the ideal functionality $\mathsf{F}$ it tries to implement, but it has a separate set of resource ports for each of the ideal functionalities $\mathsf{F}_{\mathtt{R,1}}, \ldots, \mathsf{F}_{\mathtt{R,N}}$ which it uses as communication resources. It is still secure to replace some resource by a secure implementation of the resource. I.e., $\pi \diamond \mathsf{F}_{\mathtt{R,1}} \diamond \cdots \diamond \mathsf{F}_{\mathtt{R,N}} \overset{\text{Env}}{\geqslant} \mathsf{F}$ and $\pi_{\mathtt{R,I}} \overset{\text{Env}}{\geqslant} \mathsf{F}_{\mathtt{R,I}}$ implies that $\pi \diamond \mathsf{F}_{\mathtt{R,1}} \diamond \cdots \diamond \mathsf{F}_{\mathtt{R,I-1}} \diamond \pi_{\mathtt{R,I}} \diamond \mathsf{F}_{\mathtt{R,I+1}} \diamond \cdots \diamond \mathsf{F}_{\mathtt{R,N}} \overset{\text{Env}}{\geqslant} \mathsf{F}$. Using this $N$ times, we can replace all the resources by secure implementations.

For the above to hold some changes has to be made to the definition of Env, as $\mathcal{Z}_2$ in the proof will be $\pi \diamond \mathsf{F}_{\mathtt{R,1}} \diamond \cdots \diamond \mathsf{F}_{\mathtt{R,I-1}} \diamond \mathsf{F}_{\mathtt{R,I+1}} \diamond \cdots \diamond \mathsf{F}_{\mathtt{R,N}} \diamond \mathcal{Z}$. We therefore need that this $\mathcal{Z}_2$ is in Env. If Env is the set of all recursive poly-time environments and each $\mathsf{F}_{\mathtt{R,J}}$ for $J = 1, \ldots, n$, $J \neq I$ are recursive poly-time, then it can be seen that Env is an environment class under this definition.

**Exercise 4.3** Prove the UC Theorem for protocols using several resources, and make the needed modifications to the definitions involved if you have to. Argue that all the changes are needed.

## 4.3 Adversaries and Their Powers

In secure multiparty computation it is often impossible to prove security in all environments. We therefore work with a large number of restricted classes of environments. Below we list some of these restrictions, discuss why they have been considered, and model them using an environment class. In doing that we always let Env be the environment class consisting of all interactive systems, and then discuss how to restrict it.

### 4.3.1 Threshold Security

Our protocol Protocol CEPS (Circuit Evaluation with Passive Security) from Chapter 3 is only secure against $t < n/2$ corrupted parties, as $n/2$ parties have enough shares in all secret sharings to allow them to reconstruct. This is called threshold security, and in this case the threshold is $\lfloor n/2 - 1 \rfloor$. For any $t$ we let $\text{Env}^t$ be the set of $\mathcal{Z} \in \text{Env}$ which corrupts at most $t$ parties. To prove that this is an environment class, the crucial observation is that if $\mathcal{Z} \in \text{Env}^t$ and $\mathcal{S} \in \text{Sim}$, then also $\mathcal{Z} \diamond \mathcal{S}$ corrupts at most $t$ parties, as $\mathcal{S}$ is corruption preserving.

### 4.3.2 Adaptive Security versus Static Security

Environments in Env are allowed to corrupt parties when they desire. This is called adaptive corruption and the environment is called an adaptive adversary, as it can adapt its corruption pattern to the communication that it observes in the protocol. Protocols which can be proven secure against adaptive adversaries are called adaptively secure.

Sometimes adaptive corruption makes security proofs hard or impossible. We therefore also work with the notion of a static adversary which must specify which parties it is going to corrupt *before the protocol execution starts.* This is called static corruption and a protocol which is only proven secure against static adversaries are called statically secure.

Recall that when we proved Protocol CEPS (Circuit Evaluation with Passive Security) secure we gave the set of corrupted parties as input to the simulator $\mathsf{S}$, as we gave it $\{x_i, y_i\}_{\mathsf{P}_i \in C}$. So what we did in Chapter 3 was actually a static simulation. It turns out that Protocol CEPS (Circuit Evaluation with Passive Security) is also adaptively secure, we just did not want to go into the details of this in Chapter 3. However, as we will see in a later example, there exist protocols which are statically but not adaptively secure. For now we concentrate on modeling the notion of a static adversary.

Technically, what we want is that the simulator knows which parties are going to be corrupted before the simulation starts. We ensure this by restricting Env to the set of $\mathcal{Z} \in$ Env which behave as follows. Before $\mathcal{Z}$ sends any other messages, it does a corruption of some subset $C$ of the parties, either passively or actively. This is called the preamble. In the simulation, $\mathcal{S}$ sees the initial corruptions done by $\mathcal{Z}$, and can therefore learn the set $C$ before it has to simulate any leakage or influence. We use $\text{Env}^{\texttt{static}}$ to denote this set of environments. To prove that this is an environment class, the crucial observation is that if $\mathcal{Z} \in \text{Env}^{\texttt{static}}$ and $\mathcal{S} \in \text{Sim}$, then also $\mathcal{Z} \diamond \mathcal{S}$ does static corruptions, as $\mathcal{S}$ is corruption preserving.

**Example 4.9** To demonstrate the difference between adaptive corruption and static corruption we return to our secure transfer example, Example 4.7, and consider how we simulate corruption. Assume first that we allow one static, active corruption. This means that at most one party gets corrupted, and that the simulator $\mathcal{S}$ is told which party it is before the protocol is run.

Assume first that it is $\mathsf{P}_1$ who is corrupted. Technically, this means that before it does anything else, the environment $\mathcal{Z}$ outputs ($\texttt{active corrupt}$) on $\texttt{AT.infl}_1$ and then outputs $(\texttt{active corrupt}, 1)$ on $\texttt{AT.infl}$. Now, since $\mathcal{S}$ is corruption preserving it will then output $(\texttt{active corrupt}, 1)$ on $\texttt{ST.infl}$. More importantly, $\mathcal{S}$ is now allowed to send $(\texttt{send}, 1, (mid, 2, m'))$ on $\texttt{ST.infl}$ at any point, in response to which $\mathsf{F}_{\texttt{ST}}$ stores $(mid, 1, 2, m')$ as if $(mid, 2, m')$ had arrived on $\texttt{ST.in}_1$. Then $\mathcal{S}$ can send $(\texttt{deliver}, mid, 1, 2)$ on $\texttt{ST.infl}$, in response to which $\mathsf{F}_{\texttt{ST}}$ outputs $(mid, 1, m')$ on $\texttt{ST.out}_2$. So, all in all, $\mathcal{S}$ can make $\mathsf{F}_{\texttt{ST}}$ output any message of the form $(mid, 1, m')$ on $\texttt{ST.out}_i$. The simulator uses this to simulate as follows: It runs a copy of $\pi_{\texttt{ST}} \diamond \mathsf{F}_{\texttt{AT}}$ internally, where it corrupts party 1. It connects the special ports of $\pi_{\texttt{ST}} \diamond \mathsf{F}_{\texttt{AT}}$ to $\mathcal{Z}$, such that $\mathcal{Z}$ is interacting with a copy of $\pi_{\texttt{ST}} \diamond \mathsf{F}_{\texttt{AT}}$ exactly as in the real world. Whenever the party $\mathsf{P}_2$ in the internal copy of $\pi_{\texttt{ST}} \diamond \mathsf{F}_{\texttt{AT}}$ outputs a message of the form $(mid, 1, m')$ on $\texttt{ST.out}_2$, then $\mathcal{S}$ makes $\mathsf{F}_{\texttt{ST}}$ output $(mid, 1, m')$ on $\texttt{ST.out}_2$. As a result $\mathcal{Z}$ will receive $(mid, 1, m')$ on $\texttt{ST.out}_2$ exactly as in the real world where it interacts with $\pi_{\texttt{ST}} \diamond \mathsf{F}_{\texttt{AT}}$. It is easy to see that this gives a perfect simulation.[9]

If it is $\mathsf{P}_2$ who is corrupted, then $\mathcal{S}$ simulates as follows: Whenever $\mathcal{S}$ is activated it checks if it received some $(mid, 1, 2, |m|)$ on $\texttt{ST.leak}$. If so, it inputs $(\texttt{deliver}, mid, 1, 2)$ on $\texttt{ST.infl}$. In response to this $\mathsf{F}_{\texttt{ST}}$ outputs $(mid, 1, m)$ on $\texttt{ST.infl}$, as outputs of corrupted parties are redirected to the simulator. This means that as soon as $\mathcal{Z}$ sends $(mid, 2, m)$ to $\mathsf{F}_{\texttt{ST}}$ on $\texttt{ST.in}_1$, the simulator can learn $m$. This again makes the simulation trivial, as $\mathcal{S}$ now knows the input, so it can just run the protocol on $m$ and connect the special ports of the protocol to $\mathcal{Z}$. △

The above example shows that $\pi_{\texttt{ST}} \diamond \mathsf{F}_{\texttt{AT}}$ securely implements $\mathsf{F}_{\texttt{ST}}$ in poly-time environments doing one active corruption. It is trivial to give a simulator for environments that do a static corruption of both $\mathsf{P}_1$ and $\mathsf{P}_2$, as the simulator then sees all inputs of $\mathsf{P}_1$ and $\mathsf{P}_2$ and determines which outputs $\mathsf{F}_{\texttt{ST}}$ gives on behalf of $\mathsf{P}_1$ and $\mathsf{P}_2$, as in a complete break down. So,

$$\pi_{\texttt{ST}} \diamond \mathsf{F}_{\texttt{AT}} \overset{\text{Env}^{\texttt{poly,static}}}{\geqslant} \mathsf{F}_{\texttt{ST}} \ .$$

To demonstrate that there is a difference between static and adaptive security we now argue that $\pi_{\texttt{ST}} \diamond \mathsf{F}_{\texttt{AT}}$ does not implement $\mathsf{F}_{\texttt{ST}}$ even against just one adaptive corruption, at least if the encryption scheme is a normal encryption scheme like RSA. Assume namely that both parties are honest from the beginning of the protocol and that $\mathcal{Z}$ sends a uniformly random bit string $m \in_{\text{R}} \{0,1\}^k$ and finishes the protocol, and then corrupts $\mathsf{P}_1$. When playing with $\pi_{\texttt{ST}} \diamond \mathsf{F}_{\texttt{AT}}$, it will see $pk$ and $c = E_{pk}(m; r)$ during the execution of the protocol, and when it corrupts $\mathsf{P}_1$ it receives the internal state of $\mathsf{P}_1$, which includes $m$ and $r$. Then it checks if $c = E_{pk}(m; r)$. If

[9]The argument here was easy because we only consider two parties. If we look at the larger protocol for $n$ parties, other things could go wrong. If $\mathcal{Z}$ for instance could make $\mathsf{P}_2$ output a message of the form $(mid, 3, m')$ in a setting where $\mathsf{P}_3$ is honest and where $(mid, 3, m')$ was not input on $\texttt{ST.in}_3$,[10] then $\mathcal{S}$ cannot simulate, as it cannot make $\mathsf{F}_{\texttt{ST}}$ output a message of the form $(mid, 3, m)$ on $\texttt{ST.out}_2$ unless $\mathsf{F}_{\texttt{ST}}$ received $(mid, 2, m)$ on $\texttt{ST.in}_3$ or $\mathsf{P}_3$ is corrupted. It is, however, easy to see that $\mathcal{Z}$ cannot create such a situation, as the protocol uses authenticated transfer as resource.

so, it outputs 1, otherwise it outputs 0. It is clear that $\Pr[(\pi_{\mathtt{ST}} \diamond \mathsf{F}_{\mathtt{AT}}) \diamond \mathcal{Z} = 1] = 1$. Consider then the simulation, $\mathsf{F}_{\mathtt{ST}} \diamond \mathcal{S}$. As before $\mathcal{Z}$ sends $m \in_{\mathrm{R}} \{0,1\}^\kappa$. Since $\mathsf{P}_1$ is honest during the execution, $\mathcal{S}$ will only learns $|m| = \kappa$. Then $\mathcal{S}$ must show some $pk$ and some $c$ to $\mathcal{Z}$. Then $\mathcal{Z}$ corrupts $\mathsf{P}_1$ and $\mathcal{S}$ must simulate this by giving an internal state of $\mathsf{P}_1$ which includes some $m$ and $r$. Sending the correct $m$ is easy, as $\mathcal{S}$ receives $m$ from $\mathsf{F}_{\mathtt{ST}}$ when $\mathsf{P}_1$ is corrupted. The value $r$, the simulator must cook up itself. Now $\mathcal{Z}$ checks if $c = E_{pk}(m; r)$. The claim is that the probability that $c = E_{pk}(m; r)$ is at most $2^{-\kappa}$. So, $\Pr[(\mathsf{F}_{\mathtt{ST}} \diamond \mathcal{S}) \diamond \mathcal{Z} = 1] = 2^{-\kappa}$, no matter which strategy $\mathcal{S}$ is using. The reason is that as soon as $\mathcal{S}$ sends $pk$ and $c$ there exists at most one $m'$ for which there exists some $r'$ such that $c = E_{pk}(m'; r')$, as an ciphertext $c$ can be an encryption of at most one plaintext. When $\mathcal{S}$ sent $(pk, c)$ it did not know $m$, only $|m|$. So, the $m'$ which is fixed by $(pk, c)$ is independent of the uniformly random $m$ sent by $\mathcal{Z}$, hence the probability that $m = m'$ is exactly $2^{-\kappa}$, and if $m \neq m'$, then $\mathcal{S}$ cannot produce $r$ such that $c = E_{pk}(m; r)$, even if it had infinite computing power — such an $r$ simply does not exist. If $\mathcal{Z}$ does an adaptive corruption of $\mathsf{P}_2$ instead of $\mathsf{P}_1$, the situation gets even worse, as the simulator would also have to show the secret key to $\mathcal{Z}$.

**Non-Committing Encryption**

The basic problem demonstrated above is that as soon as $\mathcal{S}$ shows the simulated $pk$ and $c$ to $\mathcal{Z}$, it is committed to the $m'$ inside $c$ and hence it gets caught if $m'$ is not the message that $\mathcal{Z}$ gave to $\mathsf{F}_{\mathtt{ST}}$. There actually exist encryption schemes which avoid this problem, called non-committing encryption schemes. A non-committing encryption scheme is an encryption scheme which in normal mode works as a normal encryption scheme. It is, however, possible to produce a simulated public key $pk$ and a simulated ciphertext $c$, which are indistinguishable from a real public key and a real ciphertext, but where it is possible to take any message $m$ and then efficiently compute $r$ and $sk$ such that: $r$ is indistinguishable from a uniformly random randomizer for the encryption scheme, $sk$ is indistinguishable from a secret for $pk$ and $c = E_{pk}(m; r)$ and $D_{sk}(c) = m$. Using a non-committing encryption scheme in $\pi_{\mathtt{ST}}$ will produce a protocol which is secure also against adaptive corruptions. We do, however, not know any efficient construction of non-committing encryption — all known schemes use at least $\kappa$ bits to encrypt just one plaintext bit, where $\kappa$ is the security parameter. Finding a non-committing encryption scheme which can encrypt $\kappa$ bits using in the order of $\kappa$ bits of ciphertext is an important open problem in the theory of adaptive secure multiparty computation.

### 4.3.3 Active Security versus Passive Security

We already saw that protocols can be secure against passive corruptions but not active corruptions, where the environment takes complete control over the corrupted party. By definition environments $\mathcal{Z} \in \mathrm{Env}$ are allowed active corruptions. We use $\mathrm{Env}^{\mathsf{passive}}$ to denote the set of $\mathcal{Z} \in \mathrm{Env}$ which only does passive corruptions. That it is an environment class follows from all simulators being corruption preserving. This defines notions of a passive adversary and passive secure and active adversary and active secure. We also call a protocol which is active secure robust and we call a protocol which is passive secure private.

### 4.3.4 Unconditional Security versus Computational Security

Recall that our protocol Protocol CEPS (Circuit Evaluation with Passive Security) from Chapter 3 was proven perfectly secure. This means that even an environment $\mathcal{Z}$ with unbounded computing power cannot distinguish the protocol from the simulation. When we use cryptography in a protocol, as in $\pi_{\mathtt{ST}}$, then we can, however, only expect to prove security against poly-time environments, as stronger environments, e.g., can distinguish encryptions of different messages. When security can be proven against a computationally unbounded adversary, we talk about unconditional security. When security can only be proven against a poly-time adversary, we talk about computational security.

We use $\text{Env}^{\texttt{poly}}$ to denote the set of recursive poly-time environments. We already saw that this is an environment class.

If it is possible to prove security against unbounded environments and it in addition holds for all $\mathcal{Z}$ that

$$\pi_{\mathsf{F}} \diamond \mathsf{F}_{\mathtt{R}} \diamond \mathcal{Z} \overset{\text{perf}}{\equiv} \mathsf{F}_{\mathtt{F}} \diamond \mathcal{S} \diamond \mathcal{Z}$$

instead of just

$$\pi_{\mathsf{F}} \diamond \mathsf{F}_{\mathtt{R}} \diamond \mathcal{Z} \overset{\text{stat}}{\equiv} \mathsf{F}_{\mathtt{F}} \diamond \mathcal{S} \diamond \mathcal{Z} \ ,$$

then we talk about perfect security.

## 4.3.5 Synchronous versus Asynchronous

In our framework, we have chosen to clock the parties via the influence ports, which means that it is the environment which determines in which order the parties are clocked. The environment can therefore either choose to clock the parties an equal number of times, modeling that they have synchronized clocks, or it can choose to clock parties a different number of times, modeling that the parties do not have synchronized clocks. This, in particular, means that we have to model clock synchronization simply as a restriction on the environment. Another approach could have been to add clock synchronization as an ideal functionality $\mathsf{F}_{\mathtt{CLOCK\text{-}SYNC}}$. This would have allowed us to talk about securely realizing clock synchronization, by giving a protocol securely realizing $\mathsf{F}_{\mathtt{CLOCK\text{-}SYNC}}$. One disadvantage of our approach of letting clock synchronization being a restriction on the environment is that we cannot talk about how to securely realize it, at least not via the UC Theorem. An advantage of our approach is that it is technically much simpler. Specifying $\mathsf{F}_{\mathtt{CLOCK\text{-}SYNC}}$ in a usable and implementable way is technically very challenging. Since the topic of securely realizing clock synchronization is not a topic of this book, we have chosen the simpler approach.

When talking about synchronous computation we talk in the terms of rounds. Each round will actually consist of two clockings of each party. The first clocking, called inwards clocking, allows the party to send its messages to the ideal functionality for that round. The second clocking, called outwards clocking, allows the party to receive its messages from the ideal functionality for that round.

We explain inwards clocking and outwards clocking using an example. Consider the composed party $\mathcal{P}_1 = \{\mathsf{Q}_1, \mathsf{P}_1\}$ in Fig. 4.5 using the resource $\mathsf{F}_{\mathtt{AT}}$.

When we say that $\mathcal{P}_1$ is inwards clocked, we mean that $\mathcal{Z}$ first clocks $\mathsf{Q}_1$ and then $\mathsf{P}_1$ and then $\mathcal{Z}$ sends $(\mathtt{inclock}, i)$ on $\mathtt{AT.infl}$ and clocks $\mathsf{F}_{\mathtt{AT}}$. Note that this allows $\mathcal{P}_1$ to deliver a message to $\mathsf{F}_{\mathtt{AT}}$ and allows $\mathsf{F}_{\mathtt{AT}}$ to process it.

When we say that $\mathcal{P}_1$ is outwards clocked, we mean that $\mathcal{Z}$ first sends $(\mathtt{outclock}, i)$ on $\mathtt{AT.infl}$, then clocks $\mathsf{F}_{\mathtt{AT}}$, then clocks $\mathsf{P}_1$ and then clocks $\mathsf{Q}_1$. Note that this allows $\mathsf{F}_{\mathtt{AT}}$ to deliver a message to $\mathcal{P}_1$.

We say that $\mathcal{Z}$ is a synchronous environment if it proceeds in rounds, where in each round it behaves as follows:

1. First it does an inwards clocking of all parties which are not actively corrupted. It is up to $\mathcal{Z}$ in which order the parties are inwards clocked.

2. Then it possibly interacts with $\mathsf{F}_{\mathtt{AT}}$ by sending messages on $\mathtt{AT.infl}$, clocking $\mathsf{F}_{\mathtt{AT}}$ and looking at the messages output on $\mathtt{AT.leak}$.

3. Then it does an outwards clocking of all parties which are not actively corrupted. It is up to $\mathcal{Z}$ in which order the parties are outwards clocked.

In each round we call the phase from the point at which the first party $\mathsf{P}_i$ which is not actively corrupted is clocked in until the point at which the last party $\mathsf{P}_i$ which is not actively corrupted is clocked in the clock-in phase. We call the phase from the point at which the last party $\mathsf{P}_i$ which is not actively corrupted is clocked in until the point at which the first party $\mathsf{P}_i$ which is not actively corrupted is clocked out the negotiation phase. We call the phase from the point at which the first party $\mathsf{P}_i$ which is not actively corrupted is clocked out until the point at which the last party $\mathsf{P}_i$ which is not actively corrupted is clocked out the clock-out phase. We call the phase from the point in round $r$ at which the last party $\mathsf{P}_i$ which is not actively corrupted is clocked out until the point in round $r+1$ at which the first party $\mathsf{P}_i$ which is not actively corrupted is clocked in the transition phase. We say that the transition phase belongs to round $r$.

The environment $\mathcal{Z}$ is allowed to do corruptions in all phases. Notice that a phase can end by a corruption. Assume, e.g., that all parties which are not actively corrupted have been clocked in, except $\mathsf{P}_1$. If then $\mathsf{P}_1$ is actively corrupted by $\mathcal{Z}$ it now holds that *all* parties which are not actively corrupted have been clocked in, so the clock-in phase ended and the negotiation phase started.

We use $\text{Env}^{\mathsf{sync}}$ to denote the class of synchronous environments. It is not hard to see that if $\mathcal{Z} \in \text{Env}^{\mathsf{sync}}$ and $\pi \in \text{Pro}$ and $\pi \diamond \mathcal{Z}$ connects the protocols ports of the two systems, then $\pi \diamond \mathcal{Z} \in \text{Env}^{\mathsf{sync}}$. It is also true that if $\mathcal{Z} \in \text{Env}^{\mathsf{sync}}$ and $\mathcal{S} \in \text{Sim}$ and $\mathcal{S} \diamond \mathcal{Z}$ connects the special ports of the two systems, then $\mathcal{S} \diamond \mathcal{Z} \in \text{Env}^{\mathsf{sync}}$. This follows from $\mathcal{S}$ being clock preserving.

**Synchronous Communication**

If $n$ parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$ are using $\mathsf{F}_{\mathtt{AT}}$ as communication resource, then even though the parties are synchronized, the communication is not. It might happen that $\mathcal{P}_1$ sends a message to $\mathcal{P}_2$ in round $r$ during its inwards clocking and still $\mathcal{P}_2$ does not receive it in round $r$ during its outwards clocking. This is because we formulated $\mathsf{F}_{\mathtt{AT}}$ in an inherently asynchronous manner, which made it easier to talk about how to securely implement it, as we did not have to deal with synchronization issues. If we also want to guarantee that messages sent in round $r$ are received in round $r$, then we talk about synchronous communication, which is technically different from clock synchronization. In fact, implementing synchronous communication securely require that you both have good clock synchronization and that you have an upper bound on message delivery time on the network, such that you can use timeouts. Again, we are not going to talk about how to implement synchronous communication, we are simply going to assume it when we need it. It is simple to turn $\mathsf{F}_{\mathtt{AT}}$ into an ideal functionality for synchronous communication by adding the following two rules:

1. On $(\mathtt{inclock}, i)$ process all messages on $\mathtt{AT.in}_i$ as $\mathsf{F}_{\mathtt{AT}}$ does.
2. On $(\mathtt{outclock}, i)$, for all stored values $(mid, j, i, m)$, delete $(mid, j, i, m)$ and output $(mid, j, m)$ on $\mathtt{AT.out}_i$.

The first rule ensures that when $\mathcal{P}_i$ is inwards clocked, all the messages $(mid, j, m)$ sent in that round will be processed and hence $(mid, i, j, m)$ will be stored. The second rule ensures that when $\mathcal{P}_i$ is outwards clocked, all messages sent to it in that round will be delivered first, such that they are available to $\mathcal{P}_i$. In a synchronous environment, this implies that all honest parties get to deliver all their messages to all parties in all rounds.

When we talk about a synchronous protocol, we mean a protocol using an ideal functionality for synchronous communication run in an synchronous environment, i.e., a synchronous protocol uses both clock synchronization and synchronous communication.

In this book, we will mainly work with synchronous protocols. Assuming synchronous communication is not a very good model of reality, but we allow ourselves to assume it because the problems we look at are hard enough already in this model, and because most of the techniques of multiparty computation are easier to present in the synchronous model.

As an example of an advantage of assuming synchronous communication, assume that we want to tolerate that up to $t$ of the $n$ parties are corrupted and assume that we only have asynchronous communication. Since $t$ parties can be corrupted it is easy to see that if an honest party at some point in the protocol waits for messages from more than $n - t$ parties, then it might potentially be waiting for a message from a corrupted party. If this corrupted party is Byzantine, it might not have sent its message at all, and since no lower bound on message delivery is guaranteed, an unsent message cannot be distinguished from a slow message.[11] The party might therefore end up waiting forever for the unsent message, and the protocol deadlocks. So, in an asynchronous protocol which must tolerate $t$ corruptions and must be dead-lock free, the honest parties cannot wait for messages from more than $n - t$ parties in each round. But this means that *some of the honest parties might not even be able to send their inputs to the other honest parties*, let alone having their inputs securely contribute to the result. Therefore any asynchronous protocol allows input deprivation in the sense that in some executions up to $t$ of the honest parties might not be able to make the result depend on their inputs. In some application, like voting, this is of course intolerable.

### 4.3.6 Consensus Broadcast versus Point-to-Point Communication

For active adversaries, there is a problem with broadcasting, namely if a protocol requires a player to broadcast a message to everyone, it does not suffice to just ask him to send the same message to all players. If he is corrupt, he may say different things to different players, and it may not be clear to the honest players if he did this or not. And, in some protocols, if a party sends different messages to different parties when it should not, it can make the protocol give incorrect results or make it fail in such a way that the adversary learns more than it should. Hence we sometimes need a mechanism which forces a party to send the same message to all parties. This is known as consensus broadcast or Byzantine agreement.[12]

In a consensus broadcast, all honest receivers are guaranteed to receive the same message *even if the sender and some of the other parties are corrupted.* This problem is so hard that sometimes it is impossible to implement. One therefore in general has to make a distinction between the case where a consensus broadcast channel is given for free as a part of the model, or whether such a channel has to be simulated by a sub-protocol. We return to this issue in more detail later, and look at an implementation of consensus broadcast.

What is important here is that consensus broadcast is not hard to model as an ideal functionality. We therefore do not model it as an environment class. We mention the problem in this section only because it fits into the discussion of the powers of adversaries.

### 4.3.7 Combining Environment Classes

We can combine the above notions, and, e.g., talk about a synchronous, poly-time bounded, $t$-threshold adversary only doing static (and active) corruptions. The corresponding environment class is

$$\mathrm{Env}^{\mathtt{sync},\mathtt{poly},t,\mathtt{static}} \stackrel{\mathrm{def}}{=} \mathrm{Env}^{\mathtt{sync}} \cap \mathrm{Env}^{\mathtt{poly}} \cap \mathrm{Env}^{t} \cap \mathrm{Env}^{\mathtt{static}} \ .$$

Since the intersection of environment classes produces an environment class, we have that $\mathrm{Env}^{\mathtt{sync},\mathtt{poly},t,\mathtt{static}}$ is an environment class.

## 4.4 Some Ideal Functionalities

In this section we describe some standard ideal functionalities that we will use in the following.

[11]In fact, it is easy to see that the ability to distinguish an unsent message from a sent-but-slow message requires that you know an upper bound on delivery time.

[12]Sometimes it is also just known by broadcast, but in the distributed computing literature the term *broadcast* is sometimes used to refer to a communication mechanisms which does not necessarily guarantee consistency if the sender is corrupted. We want to avoid this possible confusion, and therefore use the term *consensus broadcast.*

### 4.4.1 Complete Break Down

Before describing the functionalities, it is useful to introduce the notion of a complete break down. When we say that an ideal functionality $\mathsf{F}_{\mathtt{F}}$ does a complete break down, then we mean that it starts behaving as if all parties were actively corrupted. More specifically it starts ignoring all its other code and only executes the following rules:

- It first outputs the ideal internal state of *all* parties on $\mathtt{F.leak}$.
- On all subsequent inputs $(\mathtt{send}, i, m)$ on $\mathtt{F.infl}$, it outputs $m$ on $\mathtt{F.out}_i$.
- On all subsequent inputs $m$ on $\mathtt{F.in}_i$, it outputs $(i, m)$ on $\mathtt{F.leak}$.

Notice that if $\mathsf{F}_{\mathtt{F}}$ is used as a communication resource and does a complete break down, then the adversary sees all inputs given to $\mathsf{F}_{\mathtt{F}}$ and it is the adversary who specifies all outputs delivered by $\mathsf{F}_{\mathtt{F}}$. This is an ultimately useless and insecure communication resource. On the other hand, if $\mathsf{F}_{\mathtt{F}}$ is acting as ideal functionality and does a complete break down, then the simulator sees all inputs given to $\mathsf{F}_{\mathtt{F}}$ and it is the simulator $\mathcal{S}$ who specifies all outputs delivered by $\mathsf{F}_{\mathtt{F}}$. This makes simulation trivial. Assume namely that $\mathcal{S}$ is trying to simulate a protocol $\pi_{\mathtt{F}} \diamond \mathcal{R}$. The simulator $\mathcal{S}$ will simply run a copy of $\pi_{\mathtt{F}} \diamond \mathcal{R}$ internally and connect the special ports of $\pi_{\mathtt{F}} \diamond \mathcal{R}$ to the environment $\mathcal{Z}$. Whenever $\mathcal{Z}$ inputs $m$ on some inport $\mathtt{F.in}_i$ of $\mathsf{F}_{\mathtt{F}}$, then $\mathcal{S}$ is given $(i, m)$. In response to this, $\mathcal{S}$ inputs $m$ on the port $\mathtt{F.in}_i$ of its internal copy of $\pi_{\mathtt{F}} \diamond \mathcal{R}$. And, whenever the copy of $\pi_{\mathtt{F}} \diamond \mathcal{R}$ that $\mathcal{S}$ is running outputs a value $m$ on some port $\mathtt{F.out}_i$, then $\mathcal{S}$ inputs $(\mathtt{send}, i, m)$ to $\mathsf{F}_{\mathtt{F}}$, in response to which $\mathsf{F}_{\mathtt{F}}$ sends $m$ to $\mathcal{Z}$ on $\mathtt{F.out}_i$. This means that $\mathcal{Z}$ is essentially just playing with the copy of $\pi_{\mathtt{F}} \diamond \mathcal{R}$ run by $\mathcal{S}$, so clearly $\mathcal{Z}$ cannot distinguish this from a situation where it is playing with $\pi_{\mathtt{F}} \diamond \mathcal{R}$. In fact, $\pi_{\mathtt{F}} \diamond \mathcal{R}$ and $\mathsf{F}_{\mathtt{F}} \diamond \mathcal{S}$ will be perfectly indistinguishable to any $\mathcal{Z}$.

With the above discussion in mind, we can think of a complete break down of an ideal functionality as a technical way to say that from the point of the complete break down, we require no security properties of an implementation of $\mathsf{F}_{\mathtt{F}}$.

### 4.4.2 Secure Synchronous Communication and Broadcast

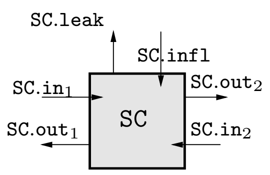
In Agent $\mathrm{F}_{\mathtt{SC}}$ we give an ideal functionality for secure *syncronous* communication and *consensus* broadcast. In each round each party $\mathsf{P}_i$ specifies one message $m_{i,j}$ for each of the other parties $\mathsf{P}_j$ plus a message $m_i$ that $\mathsf{P}_i$ wants to broadcast. At the end of the round each $\mathsf{P}_j$ receives the messages $m_{i,j}$ and $m_i$ from each $\mathsf{P}_i$. The message $m_i$ output to different parties $\mathsf{P}_j$ and $\mathsf{P}_k$ are guaranteed to be the same. This means that all messages are guaranteed to be delivered and that there is no way to send different broadcast messages to different parties.

Notice that we allow the corrupted parties to see their messages first and we allow them to change their mind on their own messages until the round is over. This allows them to choose their own messages based on what the honest parties are sending. This is called rushing. If we do not allow rushing, then it is very hard and sometimes impossible to securely implement $\mathsf{F}_{\mathtt{SC}}$, the reason essentially being that some party must send its messages first, and the corrupted parties can always wait a little longer than the honest parties.

Notice that if a party fails to send a message in some round, we simply set all its messages to be the empty string $\epsilon$. This holds for both the honest parties and the corrupted parties. In terms of implementation, this just means that if you do not receive a message from some party before the timeout for a given round, you define the message to be $\epsilon$. For an implementation to be secure it is therefore essential that the timeout is so long that any honest party who tries to deliver a message will always have it delivered before the timeout, and this should hold in the worst case and except with negligible probability. This can make a secure implementation of synchronous communication very inefficient, as each round suffer a delay which is at least as long as the worst case delivery time of the network.

Agent $\mathrm{F}_{\mathtt{SC}}$

The ideal functionality for secure communication and consensus broadcast between $n$ parties (pictures shows only 2 for simplicity).



**initialization** The ideal functionality keeps track of three set, $A$, $P$ and $C$, which are all initially empty. When party $i$ is actively corrupted it adds $i$ to $A$ and $C$. When party $i$ is passively corrupted it adds $i$ to $P$ and $C$.

**honest inputs** On input $(\mathtt{clockin}, i)$ on $\mathtt{SC.infl}$, read a message from $\mathtt{SC.in}_i$. If there was a message on $\mathtt{SC.in}_i$, then parse it on the form $(m_{i,1}, \ldots, m_{i,n}, m_i)$. Here $m_{i,j}$ is the message that $\mathsf{P}_i$ sends to $\mathsf{P}_j$ and $m_i$ is the message $\mathsf{P}_i$ is broadcasting. It then outputs $(\{m_{i,j}\}_{j \in C}, m_i)$ on $\mathtt{SC.leak}$ (this means that as soon as a honest party sends messages, the adversary learns the messages intended for the corrupted parties). Finally it stores $(m_{i,1}, \ldots, m_{i,n}, m_i)$. Furthermore, if at some point during the round the set $C$ grows, then output $(i, j, m_{i,j})$ on $\mathtt{SC.leak}$ for the new $j \in C$.

**corrupted inputs** On input $(\mathtt{change}, i, (m_{i,1}, \ldots, m_{i,n}, m_i))$ on $\mathtt{SC.infl}$, where $i \in A$ and at a point before the clock-out phase started, store $(m_{i,1}, \ldots, m_{i,n}, m_i)$, overriding any previous input from the actively corrupted party $i$ in this round (this means that as long as no passively corrupted party or honest party received outputs, the corrupted parties are allowed to change their inputs).

**delivery** On input $(\mathtt{clockout}, i)$ on $\mathtt{SC.infl}$, if there is a party $\mathsf{P}_i$ for which no $(m_{i,1}, \ldots, m_{i,n}, m_i)$ is stored, then store $(m_{i,1}, \ldots, m_{i,n}, m_i) = (\epsilon, \ldots, \epsilon)$ for all such parties. Then output $(m_{1,i}, \ldots, m_{n,i}, (m_1, \ldots, m_n))$ on $\mathtt{SC.out}_i$.

### 4.4.3 Secure Synchronous Function Evaluation

In Agent $\mathrm{F}^f_{\mathtt{SFE}}$ we give an ideal functionality for secure synchronous function evaluation. For simplicity we focus on the case where each party gives just one input, and we give an ideal functionality which can be used only once.

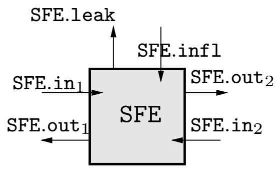
As for $\mathsf{F}_{\mathtt{SC}}$ we allow that actively corrupted parties can change their inputs as long as the function was not evaluated, and we give all corrupted parties their outputs as soon as they are defined. This makes it easier to implement $\mathsf{F}^f_{\mathtt{SFE}}$ as a protocol is allowed to have the same influence and leakage.

We allow that the adversary determines when the function is evaluated, but we require that it waits until all parties which are not actively corrupted have given their inputs, or we would implicitly have allowed the adversary to set the inputs of honest parties or passively corrupted parties to 0.

Notice that until the ideal functionality provides outputs, it sends no messages on its outports $\mathtt{SFE.out}_i$. This means that some larger protocol using $\mathsf{F}^f_{\mathtt{SFE}}$ will just observe silence from $\mathsf{F}^f_{\mathtt{SFE}}$ during most of the rounds.

Agent $\mathrm{F}^f_{\texttt{SFE}}$

The ideal functionality for one secure function evaluation between $n$ parties of a function $f : \mathbb{F}^n \to \mathbb{F}^n$ for a finite field $\mathbb{F}$ (picture shows only 2 parties for simplicity).



**initialize** The ideal functionality keeps track of three sets, $A$, $P$ and $C$ as $\mathsf{F}_{\texttt{SC}}$. It also keeps bits `delivery-round`, `evaluated`, `inputs-ready`, $\texttt{input-ready}_1, \dots, \texttt{input-ready}_n \in \{0,1\}$, initially set to 0.

**honest inputs** On input $(\texttt{clockin}, i)$ on `SFE.infl` for $i \notin A$, read a message from $\texttt{SFE.in}_i$. If there was a message $x_i$ on $\texttt{SFE.in}_i$ and $x_i \in \mathbb{F}$ and $\texttt{input-ready}_i = 0$, then set $\texttt{input-ready}_i \leftarrow 1$, store $(i, x_i)$ and output $(\texttt{input}, i)$ on `SFE.leak`.

**corrupted inputs** On input $(\texttt{change}, i, x_i)$ on `SFE.infl`, where $i \in A$ and $x_i \in \mathbb{F}$ and $\texttt{evaluated} = 0$, set $\texttt{input-ready}_i \leftarrow 1$ and store $(i, x_i)$, overriding any such previous value stored for party $i$ (this means that as long as the function has not been evaluated on the inputs, the corrupted parties are allowed to change their inputs).

**simultaneous inputs** If it holds in some round that after the clock-in phase ends there exist $i, j \notin A$ such that $\texttt{input-ready}_i = 0$ and $\texttt{input-ready}_j = 1$, then do a complete break down.

If it happens in some round that after the clock-in phase ends that $\texttt{input-ready}_i = 1$ for all $i \notin A$ and $\texttt{inputs-ready} = 0$, then set $\texttt{inputs-ready} \leftarrow 1$ and for each $i \in A$ where $\texttt{input-ready}_i = 0$, store $(i, x_i) = (i, 0)$.

**evaluate function** On input $(\texttt{evaluate})$ on `SFE.infl` where $\texttt{inputs-ready} = 1$ and $\texttt{evaluated} = 0$, set $\texttt{evaluated} \leftarrow 1$, let $(x_1, \dots, x_n)$ be the values defined by the stored values $(i, x_i)$ for $i = 1, \dots, n$ and compute $(y_1, \dots, y_n) = f(x_1, \dots, x_n)$. Then output $\{(i, y_i)\}_{i \in C}$ on `SFE.leak`, and if $C$ later grows, then output $(j, x_j)$ on `SFE.leak` for the new $j \in C$.

**simultaneous output** On input $(\texttt{delivery-round})$ on `SFE.infl`, where $\texttt{evaluated} = 1$ and $\texttt{delivery-round} = 0$, proceed as follows: if we are at a point where not party $i \notin A$ was clocked out yet, then set $\texttt{delivery-round} \leftarrow 1$.

**delivery** On input $(\texttt{clockout}, i)$ on `SFE.infl`, where $\texttt{delivery-round} = 1$, output $y_i$ on $\texttt{SFE.out}_i$.

### Simultaneous Input

Notice also that we implicitly require that all honest parties provide their inputs in the same round. Namely, if they don't do this, we make $\mathsf{F}^f_{\texttt{SFE}}$ do a complete break down. This allows a secure implementation of $\mathsf{F}^f_{\texttt{SFE}}$ to do the same. In other words, we do not require any security properties from an implementation if the parties in the protocol do not get inputs in the same round. This makes implementation much easier, as the implementation can just assume that all parties get inputs from the environment in the same round, and ignore what happens in case this is not true: any behavior of the protocol in case the simultaneous input assumption breaks can be trivially simulated as the ideal functionality does a complete break down.

Assuming that all parties which are not actively corrupted get their inputs in the same round is called simultaneous input. The reason why we do not require security of a protocol when there is non-simultaneous input is that it is very hard to securely handle this, and sometimes impossible.

As an example of this, consider the protocol Protocol CEPS (Circuit Evaluation with Passive Security). Assume that honest parties only start doing the protocol when they get inputs from the environment, and say that for instance party $\mathsf{P}_1$ gets input in round 42 and party $\mathsf{P}_2$ gets input in round 1042. This means that when $\mathsf{P}_1$ starts running the protocol $\pi_{\mathsf{CEPS}}$, the party $\mathsf{P}_2$ will not be running the protocol yet, so $\mathsf{P}_1$ will not receive shares of the input of $\mathsf{P}_2$ and $\mathsf{P}_2$ will not be ready to receive and store the share of the input of $\mathsf{P}_1$. And it is provably impossible to do anything about this if we don't want input deprivation and if we want to tolerate active corruptions and want to guarantee that the protocol terminates. Namely, from the point of view of $\mathsf{P}_1$, the party $\mathsf{P}_2$ might either be an honest party which did not get its input yet or $\mathsf{P}_2$ might be a corrupted party who is deviating from the protocol. Any sub-protocol which tries to resolve this must have the property that it does not start the protocol until all honest parties received their inputs. On the other hand, it should allow the honest parties to proceed even if some corrupted party claims that it is honest but just did not get its input yet. These two requirements are clearly contradictory.

**Simultaneous Output**

Finally, notice that the way we handle delivery means that all parties which are not actively corrupted will receive their outputs in the *same* round. This is called simultaneous output.

Having simultaneous output is very convinient in a synchronous environment. To see why, consider a synchronous protocol $\pi_{\mathrm{N}}$ which uses $\mathsf{F}^f_{\mathsf{SFE}}$ as a sub routine in some larger protocol. Consider the following plausible behavior: At some point the parties $\mathsf{Q}_i$ calls $\mathsf{F}^f_{\mathsf{SFE}}$. Then they wait until $\mathsf{F}^f_{\mathsf{SFE}}$ delivers outputs, and in that round they then continue with the outer protocol, where the next step is to invoke some other sub protocol $\pi_{\mathsf{SUB}}$. Now, if $\mathsf{F}^f_{\mathsf{SFE}}$ delivers outputs to different parties in different rounds, then the parties $\mathsf{Q}_i$ would continue with the outer protocol in different rounds. This would make them give inputs to $\pi_{\mathsf{SUB}}$ in different rounds. So, if $\pi_{\mathsf{SUB}}$ assumes simultaneous inputs, the overall protocol will now break down. Put briefly, we need simultaneous outputs to guarantee simultaneous inputs in later parts of the protocol. This means that simultaneous outputs is an important *security property.*

It is instructive to see how that property is captured by the UC framework. Let us say that $\pi_{\mathrm{N}}$ is secure when it use $\mathsf{F}^f_{\mathsf{SFE}}$, and that $\pi_{\mathrm{N}}$ depends on $\mathsf{F}^f_{\mathsf{SFE}}$ having simultaneous output. If we then replaces $\mathsf{F}^f_{\mathsf{SFE}}$ by a protocol $\pi_{\mathsf{SFE}}$ which does not have simultaneous output, then the overall protocol would break.

Fortunately our framework would exactly catch such a bad protocol before we substitute it for $\mathsf{F}^f_{\mathsf{SFE}}$, as it cannot be proven to be a secure implementation of $\mathsf{F}^f_{\mathsf{SFE}}$. Assume namely that $\mathsf{F}^f_{\mathsf{SFE}}$ has simultaneous output and $\pi_{\mathsf{SFE}}$ does not, then it will be very easy to distinguish the two: No matter which simulator we attach to $\mathsf{F}^f_{\mathsf{SFE}}$ the system $\mathsf{F}^f_{\mathsf{SFE}} \diamond \mathcal{S}$ will have simultaneous output when run in a synchronous environment, as $\mathcal{S}$ is clock preserving. And we assumed that the protocol $\pi_{\mathsf{SFE}}$ does not have simultaneous output. So, $\mathcal{Z}$ just looks at whether the outputs are delivered in the same round, and outputs 1 if and only if they are. Then $\pi_{\mathsf{SFE}} \diamond \mathcal{Z}$ will output 0 and $\mathsf{F}^f_{\mathsf{SFE}} \diamond \mathcal{S} \diamond \mathcal{Z}$ will outout 1. The same distinguishing strategy works even if $\pi_{\mathsf{SFE}}$ fails to have simultaneous outputs with some non-negligible probability.

It might seem that simultaneous output is easy to achieve. That is true, but only if consensus broadcast is given for free by the communication model. If consensus broadcast is to be implemented using a secure protocol which only uses point-to-point communication, then it is possible to prove the following two claims:

1. Any secure realization of consensus broadcast which can tolerate up to $t$ actively corrupted parties and which has simultaneous output uses at least $t + 1$ communication rounds, even

when no parties are corrupted.

2. There exist a secure realization of consensus broadcast which can tolerate up to $t$ actively corrupted parties and which has non-simultaneous output which uses only $\min(t+1, f+2)$ communication rounds, where $f \in \{0, \ldots, t\}$ is the number of parties who were actually corrupted during the execution of the protocol.

So, if we for instance have $n = 1001$ parties and want to tolerate $t = 500$ actively corrupted parties, but in typical executions of the protocol no parties are actually deviating from the protocol, then requiring simultaneous output would make each consensus broadcast cost 501 communication rounds, whereas tolerating non-simultaneous output would typically allow to run each consensus broadcast in 2 communication rounds. If we are in a network where the worst case delivery time of a message is one minute, this would make the difference between each consensus broadcast taking two minutes or more than eight hours. I.e., with non-simultaneous output we only pay a high price when there are many parties who deviate from the protocol. If we insist on simultaneous output, then we pay a high price even if no parties deviate from the protocol.

**Example 4.10** We conclude this chapter by arguing that our protocol Protocol CEPS (Circuit Evaluation with Passive Security) is secure in the UC model. Let $\pi^f_{\mathsf{CEPS}}$ denote the protocol Protocol CEPS (Circuit Evaluation with Passive Security) for a function $f$ with the addition that all parties wait with outputting their output $y_i$ until the results $y_j$ have been reconstructed towards all parties. Then all parties output their $y_i$ in the same round. We show that

$$\pi^f_{\mathsf{CEPS}} \diamond \mathsf{F}_{\mathsf{SC}} \overset{\mathrm{Env}^{t,\mathtt{static},\mathtt{sync}}}{\geqslant} \mathsf{F}^f_{\mathsf{SFE}} ,$$

for all $t < n/2$.

For this purpose we construct the simulator Agent $\mathcal{S}_{\mathsf{CEPS}}$ shown in the box.

By inspecting $\mathcal{S}_{\mathsf{CEPS}}$ we can see that the produced views are distributed as in the real world. Actually, $\mathcal{S}_{\mathsf{CEPS}}$ is producing the exact same views as the simulator $\mathsf{S}$ from Section 3.3.3. (Repeating the argument, $\mathcal{S}_{\mathsf{CEPS}}$ runs the corrupt players on their true inputs, so this has the exact same distribution. During the protocol, the environment will never see more than $t$ shares of values belonging to honest players. No information on these values is given since they are shared using random polynomials of degree $t$. In output reconstruction, the environment sees for each corrupted player's output $y_i$ all the shares sent by the honest parties. But these are already given as those consistent with the output being $y_i$ and the $t$ shares held by the corrupted parties.) Furthermore, both $\pi^f_{\mathsf{CEPS}} \diamond \mathsf{F}_{\mathsf{SC}}$ and $\mathsf{F}^f_{\mathsf{SFE}}$ output the correct $y_i$, and hence *the same* $y_i$. Finally, $\mathcal{S}_{\mathsf{CEPS}}$ makes $\mathsf{F}^f_{\mathsf{SFE}}$ give output on $\mathtt{SFE.out}_i$ exactly when $\mathsf{P}_i$ would give output on $\mathtt{SFE.out}_i$. Hence, the perfect security follows.

We now describe how to modify $\mathcal{S}_{\mathsf{CEPS}}$ to handle the case where the environment does not give inputs to all parties in the same round. Again $\mathcal{S}_{\mathsf{CEPS}}$ learns the set of corrupted parties $C$ in the preamble. If, in some round $\mathcal{Z}$ gives inputs to some parties and not others, then $\mathsf{F}^f_{\mathsf{SFE}}$ does a complete break down, so $\mathcal{S}$ learns all inputs $(x_1, \ldots, x_n)$. It then runs $\pi^f_{\mathsf{CEPS}} \diamond \mathsf{F}_{\mathsf{SC}}$ on these inputs and connects the special ports of $\pi^f_{\mathsf{CEPS}} \diamond \mathsf{F}_{\mathsf{SC}}$ to $\mathcal{Z}$. Whenever $\mathcal{Z}$ makes a party $\mathsf{P}_i$ output some value $y'_i$ on $\mathtt{SFE.out}_i$, the simulator instructs $\mathsf{F}^f_{\mathsf{SFE}}$ to do the same, which it is allowed to do as $\mathsf{F}^f_{\mathsf{SFE}}$ is in a complete break down. This perfectly simulates the protocol, no matter how it behaves when there is non-simultaneous input. There is one little twist though. In the round where $\mathcal{Z}$ gives inputs $x_i$ to some $\mathsf{P}_i$ and not to others, $\mathsf{F}^f_{\mathsf{SFE}}$ does not do a complete break down until there is some $\mathsf{P}_i$ which does not get an input, and $\mathcal{Z}$ might choose to, e.g., first give input to $\mathsf{P}_2$ and then $\mathsf{P}_5$ and then fail to give input to $\mathsf{P}_3$ during the clock-in of $\mathsf{P}_3$. In this case the simulator would have to simulate $\mathsf{P}_2$ and $\mathsf{P}_5$ while $\mathsf{F}^f_{\mathsf{SFE}}$ is not in complete break down. This is, however, easy. If, e.g., $\mathsf{P}_2$ is corrupted, then $\mathcal{S}$ learns $x_2$ from $\mathsf{F}^f_{\mathsf{SFE}}$ and inputs $x_2$ to $\mathsf{P}_2$ in $\pi^f_{\mathsf{CEPS}} \diamond \mathsf{F}_{\mathsf{SC}}$. If, e.g., $\mathsf{P}_5$ is honest, then $\mathcal{S}$ does not learn $x_5$ from $\mathsf{F}^f_{\mathsf{SFE}}$, so it just inputs 0 to $\mathsf{P}_5$ in

Agent $\mathcal{S}_{\mathtt{CEPS}}$

A simulator for $\pi^f_{\mathtt{CEPS}} \diamond \mathsf{F}_{\mathtt{SC}}$ for the static case where the environment gives inputs to all parties in the same round.

**Initialize:** In the preamble, the environment $\mathcal{Z}$ specifies the set $C$ of passively corrupted parties. After this, the simulator sets up a copy of $\pi^f_{\mathtt{CEPS}} \diamond \mathsf{F}_{\mathtt{SC}}$ where it passively corrupts the players in $C$ and connects the special ports of $\pi^f_{\mathtt{CEPS}} \diamond \mathsf{F}_{\mathtt{SC}}$ to $\mathcal{Z}$.

**Input Sharing:** After the clock-in phase in the round where all inputs are given, $\mathcal{S}_{\mathtt{CEPS}}$ does passive corruptions of $\mathsf{P}_i \in C$ on $\mathsf{F}^f_{\mathtt{SFE}}$, which then makes $\mathsf{F}^f_{\mathtt{SFE}}$ leak the corresponding inputs $x_i$ to $\mathcal{S}_{\mathtt{CEPS}}$.

Now $\mathcal{S}_{\mathtt{CEPS}}$ does the input sharing steps of $\pi^f_{\mathtt{CEPS}}$ with the true inputs for the corrupted parties and with 0 as input for honest parties. That is, for each player $\mathsf{P}_j$, $\mathcal{S}_{\mathtt{CEPS}}$ distributes $[x^{(0)}_j; f_{x^{(0)}_j}]_t$, where $x^{(0)}_j = x_j$ for $\mathsf{P}_j \in C$ and $x^{(0)}_j = 0$ for $\mathsf{P}_j \notin C$.

**Computation Phase:** In the computation phase the simulator simply follows the protocol steps of $\pi^f_{\mathtt{CEPS}}$ according to the gates that need to be processed.

**Output Reconstruction:** First $\mathcal{S}_{\mathtt{CEPS}}$ instructs $\mathsf{F}^f_{\mathtt{SFE}}$ to evaluate the function, thereby learning the outputs $y_j$ for all $\mathsf{P}_j \in C$.

For each $\mathsf{P}_j \in C$, $\pi^f_{\mathtt{CEPS}}$ takes the $t$ shares that the corrupted parties hold in the output of $\mathsf{P}_j$ and defines $f_{y_j}(0) \overset{\text{def}}{=} y_j$. After this, $f_{y_j}(\mathtt{X})$ is defined in $t+1$ points and hence $\mathcal{S}_{\mathtt{CEPS}}$ can use Lagrange interpolation to compute the unique degree-$t$ polynomial consistent with these points. Then, for $\mathsf{P}_i \notin C$ it computes the share $f_{y_j}(i)$ and sends it to $\mathsf{P}_j$ on behalf of $\mathsf{P}_i$.

When $\mathcal{Z}$ clocks out $\mathsf{P}_i$ in the copy of $\pi^f_{\mathtt{CEPS}} \diamond \mathsf{F}_{\mathtt{SC}}$ so that $\mathsf{P}_i$ should produce an output, the simulator first inputs (`delivery-round`) to $\mathsf{F}^f_{\mathtt{SFE}}$ if it has not done so already. Then $\mathcal{S}_{\mathtt{CEPS}}$ inputs (`clock-out`, $i$) to $\mathsf{F}^f_{\mathtt{SFE}}$ to make it output $y_i$ on $\mathtt{SFE.out}_i$, exactly as $\mathsf{P}_i$ would have done in the protocol.

$\pi^f_{\mathtt{CEPS}} \diamond \mathsf{F}_{\mathtt{SC}}$, which makes $\mathsf{P}_5$ do a sharing of 0, and this will show at most $t$ shares to $\mathcal{Z}$, namely those sent to $\mathsf{P}_i$ for $i \in C$. When $\mathcal{Z}$ then fails to give input to $\mathsf{P}_3$, the complete break down will give the true value of $x_5$ to $\mathcal{S}$. Then $\mathcal{S}$ computes the secret sharing of $x_5$ which is consistent with $f(0) = x_5$ and the $t$ shares already shown to $\mathcal{Z}$. From $f(\mathtt{X})$ it computes the shares that $\mathsf{P}_5$ would have sent to the other honest parties and then it updates the internal state of $\mathsf{P}_5$ and $\mathsf{F}_{\mathtt{SC}}$ to be consistent with $\mathsf{P}_5$ having sent these shares — this is possible as the internal state of the honest $\mathsf{P}_5$ was not shown to $\mathcal{Z}$ at this point and because $\mathsf{F}_{\mathtt{SC}}$ only leaked what $\mathsf{P}_5$ sent to the corrupted parties. Now the state of $\pi^f_{\mathtt{CEPS}} \diamond \mathsf{F}_{\mathtt{SC}}$ is consistent with $\mathsf{P}_5$ having run with $x_5$. The simulator patches the state this way for all honest parties which received inputs before the complete break down, and then $\mathcal{S}_{\mathtt{CEPS}}$ finishes the simulation as above, by simply running the protocol on the true inputs of the parties. △

**Exercise 4.4** Argue that

$$\pi^f_{\mathtt{SFE}} \diamond \mathsf{F}_{\mathtt{SC}} \overset{\mathrm{Env}^{t,\mathrm{sync}}}{\geqslant} \mathsf{F}^f_{\mathtt{SFE}} \ ,$$

for all $t < n/2$. I.e., argue that the protocol is also adaptively secure. [Hint: Look at how the simulator in the example above handled the case where it first had to simulate $\mathsf{P}_5$ without knowing its input and then had to patch the state of $\mathsf{P}_5$ to be consistent with $x_5$ when the total break down occurred.]

# Chapter 5

# Information Theoretic Robust MPC Protocols

**Contents**

## 5.1 Introduction

In this chapter, we show how to modify the protocol from Chapter 3 to make it secure also against active attacks. In the terms introduced in the previous chapter, the goal is to implement $\mathsf{F}^f_{\mathtt{SFE}}$, where we will assume that we have available the ideal functionality $\mathsf{F}_{\mathtt{SC}}$ for secure communication and consensus broadcast, and also another ideal commitment functionality $\mathsf{F}_{\mathtt{COM}}$ that we explain in detail below. In section 5.4 we show how the commitment functionality can be implemented based only on $\mathsf{F}_{\mathtt{SC}}$ under appropriate assumptions on $t$, the number of corrupted players.

The final results are collected in Section 5.5. We note already now, however, that if $t < n/3$ then $\mathsf{F}^f_{\mathtt{SFE}}$ can be implemented with perfect security without using broadcast, but assuming secure channels between each pair of players. If we additionally assume broadcast and we allow a non-zero error probability, then $t < n/2$ will be sufficient. These bounds are tight, as we shall see later.

## 5.2 Model for Homomorphic Commitments and some Auxiliary Protocols

We will assume that each player $\mathsf{P}_i$ can commit to a value $a \in \mathbb{F}$, while keeping his choice secret. He may, however, choose to reveal $a$ later. We shall see how this can be implemented by distributing and/or broadcasting some information to other players. We model it here by

Agent $F_{\texttt{COM}}$

Ideal functionality for homomorphic commitment.

**commit:** This command is executed if in some round player $\mathsf{P}_i$ sends $(\texttt{commit}, i, cid, a)$ and in addition all honest players send $(\texttt{commit}, i, cid, ?)$. In this case $\mathsf{F}_{\texttt{COM}}$ records the triple $(i, cid, a)$. Here, $cid$ is just a commitment identifier, and $a$ is the value committed to.[a] The output to all parties is $(\texttt{commit}, i, cid, \texttt{success})$.

If $\mathsf{P}_i$ is corrupted and does not input some $(\texttt{commit}, i, cid, a)$, then no tuple is stored, and instead $(\texttt{commit}, i, cid, \texttt{fail})$ is output to all parties.

**public commit:** This command is executed if in some round all parties send $(\texttt{pubcommit}, i, cid, a)$. In this case $\mathsf{F}_{\texttt{COM}}$ records the triple $(i, cid, a)$. The output to all parties is $(\texttt{pubcommit}, i, cid, \texttt{success})$.[b]

**open:** This command is executed if in some round all honest players send $(\texttt{open}, i, cid)$ and some $(i, cid, a)$ is stored. The output to all parties is $(\texttt{open}, i, cid, a)$.

If $\mathsf{P}_i$ is corrupted and does not input $(\texttt{open}, i, cid)$, then $(\texttt{open}, i, cid, \texttt{fail})$ is output to all parties.

**designated open:** This command is executed if in some round all honest players send $(\texttt{open}, i, cid, j)$ and some $(i, cid, a)$ is stored. The output to $\mathsf{P}_j$ is $(\texttt{open}, i, cid, j, a)$ and the output to all other parties is $(\texttt{open}, i, cid, j, \texttt{success})$.

If $\mathsf{P}_i$ is corrupted and does not input $(\texttt{open}, i, cid, j)$, then $(\texttt{open}, i, cid, \texttt{fail})$ is output to all parties.

**add:** This command is executed if in some round all honest players send $(\texttt{add}, i, cid_1, cid_2, cid_3)$ and some $(i, cid_1, a_1)$ is stored and some $(i, cid_2, a_2)$ is stored. As a result $\mathsf{F}_{\texttt{COM}}$ stores $(i, cid_3, a_1 + a_2)$. The output to all parties is $(\texttt{add}, i, cid_1, cid_2, cid_3, \texttt{success})$.

**mult by constant:** This command is executed if in some round all honest players send $(\texttt{mult}, i, \alpha, cid_2, cid_3)$ and $\alpha \in \mathbb{F}$ and some $(i, cid_2, a_2)$ is stored. As a result $\mathsf{F}_{\texttt{COM}}$ stores $(i, cid_3, \alpha a_2)$. The output to all parties is $(\texttt{mult}, i, \alpha, cid_2, cid_3, \texttt{success})$.

[a]We require that all honest players agree to the fact that a commitment should be made because an implementation will require the active participation of all honest players.

[b]The difference here is that all parties input $a$ and that $\mathsf{P}_i$ is forced to commit. In an implementation the other parties can in principle just remember that $\mathsf{P}_i$ is committed to the known $a$, but it is convenient to have an explicit command for this.

assuming that we have an ideal functionality $\mathsf{F}_{\texttt{COM}}$. To commit, one simply sends $a$ to $\mathsf{F}_{\texttt{COM}}$, who will then keep it until $\mathsf{P}_i$ asks to have it revealed. Formally, we assume $\mathsf{F}_{\texttt{COM}}$ is equipped with commands `commit` and `open` as described in the box.

Some general remarks on the definition of $\mathsf{F}_{\texttt{COM}}$: since the implementation of any of the commands may require all (honest) players to take part actively, we require that all honest players in a given round send the same commands to $\mathsf{F}_{\texttt{COM}}$ in order for the command to be executed. In some cases, such as a commitment, we can of course not require that all players send exactly the same information since only the committing players knows the value to be committed to. So in such a case, we require that the committer sends the command and his secret input, while the others just send the command. If $\mathsf{F}_{\texttt{COM}}$ is not used as intended, e.g., the honest players do not agree on the command to execute, $\mathsf{F}_{\texttt{COM}}$ will do a complete breakdown. It will also do a complete breakdown if an honest party commits under the same $cid$ twice or uses the same $cid_3$ twice in an addition command or a multiplication command. This is to ensure that all values are stored under unique identifiers. It will also do a complete breakdown if the honest parties use some $cid_2$ or $cid_3$ which has not been defined yet, i.e., if they for instance input $(\texttt{add}, i, cid_1, cid_2, cid_3)$, but no $(i, cid_1, a)$ is stored.

Below we will use the following short-hand for describing interactions with $\mathsf{F}_{\texttt{COM}}$. The symbol $\langle\cdot\rangle_i$ denotes a variable in which $\mathsf{F}_{\texttt{COM}}$ keeps a committed value received from player $\mathsf{P}_i$. Thus when we write $\langle a\rangle_i$, this means that player $\mathsf{P}_i$ has committed to $a$. We also need the following notation:

$\langle a\rangle_i \leftarrow a$: the `commit` command is executed to let $\mathsf{P}_i$ commit to $a$.

$\langle a\rangle_i \Leftarrow a$: the `pubcommit` command is used to force $\mathsf{P}_i$ to commit to $a$.

$a \leftarrow \langle a\rangle_i$: the `open` command is executed to let all parties learn $a$.

$(\mathsf{P}_j)a \leftarrow \langle a\rangle_i$: the designated `open` command is executed to let $\mathsf{P}_j$ learn $a$.

It is clear from the description that all players know at any point which committed values have been defined. Of course, the value committed to is not known to the players (except the committer), but nevertheless, they can ask $\mathsf{F}_{\texttt{COM}}$ to manipulate committed values, namely to add committed values and multiply them by public constants, as long as the involved variables belong to the same party. This is done by issuing the `add` or `mult` commands. The following notation will stand for execution of these commands:

$$\langle a_3\rangle_i \leftarrow \langle a_1\rangle_i + \langle a_2\rangle_i \quad \langle a_3\rangle_i \leftarrow \alpha\langle a_2\rangle_i,$$

where it is understood that $a_3 = a_1 + a_2$ respectively $a_3 = \alpha a_2$.

Agent $\mathrm{F}_{\texttt{COM}}$ (CONTINUED)

Advanced manipulation commands for the functionality $\mathsf{F}_{\texttt{COM}}$:

**transfer:** This command is executed if in some round all honest players send $(\texttt{transfer}, i, cid, j)$ and some $(i, cid, a)$ is stored. As a result $\mathsf{F}_{\texttt{COM}}$ stores $(j, cid, a)$. The output to all parties is $(\texttt{transfer}, i, cid, j, \texttt{success})$.

If $\mathsf{P}_i$ is corrupted and does not input $(\texttt{transfer}, i, cid, j)$, then no value is stored and the output to all parties is $(\texttt{transfer}, i, cid, j, \texttt{fail})$.

If $\mathsf{P}_j$ is corrupted $a$ is leaked immediately (even before the transfer is delivered).

**deliver transfer:** This command is executed if in some round all honest players send $(\texttt{deltransfer}, i, cid, j)$ and some $(j, cid, a)$ is stored. As a result $\mathsf{F}_{\texttt{COM}}$ outputs to all parties $(\texttt{deltransfer}, i, cid, j, \texttt{success})$, except $\mathsf{P}_j$ which gets $(\texttt{deltransfer}, i, cid, j, a)$.

**mult:** This command is executed if in some round all honest players send $(\texttt{mult}, i, cid_1, cid_2, cid_3)$ and some $(i, cid_1, a_1)$ is stored and some $(i, cid_2, a_2)$ is stored. As a result $\mathsf{F}_{\texttt{COM}}$ stores $(i, cid_3, a_1a_2)$. The output to all parties is $(\texttt{mult}, i, cid_1, cid_2, cid_3, \texttt{success})$.

If $\mathsf{P}_i$ is corrupted and does not input $(\texttt{mult}, i, cid_1, cid_2, cid_3)$, then no value is stored and the output to all parties is $(\texttt{transfer}, i, cid, j, \texttt{fail})$.

The final part of the description of $\mathsf{F}_{\texttt{COM}}$ includes two *advanced manipulation* commands. The `transfer` command transfers a committed value from one party to another. The idea is that even if $\mathsf{P}_i$ has committed to a value $a$, after a transfer to $\mathsf{P}_j$, we are in a situation equivalent to what we would have if $\mathsf{P}_j$ had committed to $a$ in the first place – except, of course, that $\mathsf{P}_i$ also knows $a$. The `mult` command is used by a player to create, given commitments to $a_1, a_2$ a new commitment that is guaranteed to contain $a_1 a_2$. We write

$$\langle a\rangle_j \leftarrow \langle a\rangle_i, \quad \langle a_3\rangle_i \leftarrow \langle a_1\rangle_i\langle a_2\rangle_i$$

to denote executions of these commands.

The first step towards using $\mathsf{F}_{\texttt{COM}}$ for secure function evaluation, is to implement the advanced manipulation commands from the basic set of operations. Formally speaking, we can think of this as follows: we are given a functionality $\mathsf{F}_{\texttt{COM-SIMPLE}}$ that has all the commands of $\mathsf{F}_{\texttt{COM}}$, except the the advanced manipulation commands. We then build protocols that will implement $\mathsf{F}_{\texttt{COM}}$ when given access to $\mathsf{F}_{\texttt{COM-SIMPLE}}$. Once we show that these protocols work as defined in the previous chapter, the UC theorem allows us to assume in the following that we have $\mathsf{F}_{\texttt{COM}}$ available. On the other hand, when it comes to implementing commitments, we will only need to implement $\mathsf{F}_{\texttt{COM-SIMPLE}}$.

Protocol Transfer

If any command fails in Steps 1-3, it will be clear that either $\mathsf{P}_i$ or $\mathsf{P}_j$ is corrupt. In this case, go to Step 4

1. $(\mathsf{P}_j)a \leftarrow \langle a\rangle_i$.
2. $\langle a\rangle_j \leftarrow a$.
3. For $k = 1..n$ do:[a]
   (a) $\mathsf{P}_i$ picks a uniformly random $r \in_R \mathbb{F}$ and sends it privately to $\mathsf{P}_j$.
   (b) Execute $\langle r\rangle_i \leftarrow r$ and $\langle r\rangle_j \leftarrow r$.
   (c) $\mathsf{P}_k$ broadcasts a random challenge $e \in \mathbb{F}$.
   (d) The parties execute $\langle s\rangle_i \leftarrow e\langle a\rangle_i + \langle r\rangle_i$ and $\langle s\rangle_j \leftarrow e\langle a\rangle_j + \langle r\rangle_j$.
   (e) $s \leftarrow \langle s\rangle_i$ and $s' \leftarrow \langle s\rangle_j$ (we use $s'$ here to indicate that the opened values may be different if $\mathsf{P}_i$ or $\mathsf{P}_j$ is corrupt).
   (f) If $s \neq s'$, all players go to step 4.

   If all $n$ iterations of the loop were successful, parties output `success`, except $\mathsf{P}_j$ who outputs $a$.
4. This is the "exception handler". It is executed if it is known that $\mathsf{P}_j$ or $\mathsf{P}_i$ is corrupt. First execute $a \leftarrow \langle a\rangle_i$. If this fails, all parties output `fail`. Otherwise do $\langle a\rangle_j \Leftarrow a$ (this cannot fail). Parties output `success`, except $\mathsf{P}_j$ who outputs $a$.

[a]The description of this step assumes that $1/|\mathbb{F}|$ is negligible in the security parameter. If that is not the case, we repeat the step $u$ times in parallel, where $u$ is chosen such that $(1/|\mathbb{F}|)^u$ is negligible.

### 5.2.1 Implementations of the `transfer` Command

We will present two implementations of this command. One is only statistically secure but makes no assumption on $t$, the number of corrupted players, while the other is perfectly secure but requires $t < n/2$.

The basic idea is in both cases that $\langle a\rangle_i$ will be opened to $\mathsf{P}_j$ only, and then $\mathsf{P}_j$ commits to $a$. If $\mathsf{P}_j$ is corrupt and fails to do a commitment, then we make $a$ public and force a commitment to

Protocol Perfect Transfer

If any command fails in Steps 1-4, it will be clear that either $\mathsf{P}_i$ or $\mathsf{P}_j$ is corrupt. In this case, go to Step 5

1. $(\mathsf{P}_j)a \leftarrow \langle a \rangle_i$.
2. $\langle a \rangle_j \leftarrow a$.
3. To check that $\mathsf{P}_i$ and $\mathsf{P}_j$ are committed to the same value, we do the following:

   $\mathsf{P}_i$ selects a polynomial $f(\mathtt{X}) = a + \alpha_1 \mathtt{X} + ... + \alpha_t \mathtt{X}^t$, for uniformly random $\alpha_1, ..., \alpha_t$, and executes $\langle \alpha_l \rangle_i \leftarrow \alpha_l$ for $l = 1..t$. We invoke the `add` and `mult` commands so all players collaborate to compute, for $k = 1...n$:

   $$\langle f(k) \rangle_i = \langle a \rangle_i + k\langle \alpha_1 \rangle_i + ... + k^t \langle \alpha_t \rangle_i$$

   and then $(\mathsf{P}_k)f(k) \leftarrow \langle f(k) \rangle_i$.

   Then $\mathsf{P}_i$ sends $\alpha_1, ..., \alpha_t$ privately to $\mathsf{P}_j$ and we execute $\langle \alpha_l \rangle_j \leftarrow \alpha_l$ for $l = 1..t$. In the same way as above, we compute $\langle f(k) \rangle_j$ and execute $(\mathsf{P}_k)f(k) \leftarrow \langle f(k) \rangle_j$.
4. For $k = 1...n$: $\mathsf{P}_k$ compares the two values that were opened towards him. If they agree, he broadcasts `accept`, else he broadcasts `reject`.

   For every $\mathsf{P}_k$ who said reject, execute $a_k \leftarrow \langle f(k) \rangle_i$ and $a'_k \leftarrow \langle f(k) \rangle_j$. If $a_k = a'_k$ for all $k$, all players output `success`, except $\mathsf{P}_j$ who outputs $a$. Otherwise go to step 5.
5. This is the "exception handler". It is executed if it is known that $\mathsf{P}_j$ or $\mathsf{P}_i$ is corrupt. First execute $a \leftarrow \langle a \rangle_i$. If this fails, all parties output `fail`. Otherwise do $\langle a \rangle_j \Leftarrow a$ (this cannot fail). Parties output `success`, except $\mathsf{P}_j$ who outputs $a$.

$a$ for $\mathsf{P}_j$. This is secure, as when $\mathsf{P}_j$ is corrupted, then $a$ becomes know to the adversary in the ideal world also (as $a$ is output to $\mathsf{P}_j$). So, the simulator learns $a$ and can simulate the protocol simply by running it on the correct $a$.

This is, however, not sufficient. If $\mathsf{P}_j$ is corrupted it could do $\langle a' \rangle_j \leftarrow a'$ for $a' \neq a$. This cannot be solved by doing $(\mathsf{P}_i)a' \leftarrow \langle a' \rangle_j$ and let $\mathsf{P}_i$ check that $a' = a$, because $\mathsf{P}_i$ could be corrupted as well. The ideal functionality ensures that $a' = a$ even if $\mathsf{P}_i$ and $\mathsf{P}_j$ are corrupted, therefore this should hold also for the protocol.

The final step in both protocols is therefore that $\mathsf{P}_i$ and $\mathsf{P}_j$ prove to the other parties that $a' = a$.

We sketch the argument why Protocol Transfer is secure: It is clear that if $\mathsf{P}_i$ and $\mathsf{P}_j$ are honest, then all proofs will be accepting. Second, if $\mathsf{P}_i$ and $\mathsf{P}_j$ are honest, then $r$ is random and the only value leaked to the other parties is $ea + r$, which is just a uniformly random field element. This is secure, as the simulator can simulate $ea + r$ using a uniformly random value. If either $\mathsf{P}_i$ or $\mathsf{P}_j$ is corrupted, there is nothing to simulate as the simulator learns $a$ in the ideal world. What remains is to argue if that if $a' \neq a$, then the proof will fail with high probability. This ensures that a case where $a' \neq a$, but still parties output `success`, will occur with only negligible probability. This is necessary, as such a case would constitute a chance to distinguish between the ideal case and the protocol execution.

So, assume that $a' \neq a$. I.e., $\mathsf{P}_j$ made a commitment $[a + \Delta_a]$ for $\Delta_a \neq 0$. Then $\mathsf{P}_i$ and $\mathsf{P}_j$ makes commitments $\langle r \rangle_i$ respectively $[r + \Delta_r]_i$. Again $\mathsf{P}_j$ could pick $\Delta_r \neq 0$, but could also use $\Delta_r = 0$ — we do not know. We do however know that $s = ea + r$ and that

$s' = e(a + \Delta_a) + (r + \Delta_r) = s + (e\Delta_a + \Delta_r)$. Therefore the proof is accepted if and only if $e\Delta_a + \Delta_r = 0$, which is equivalent to $e = \Delta_a^{-1}(-\Delta_r)$ (recall that $\Delta_a \neq 0$ and thus invertible.) Since $e$ is uniformly random when $\mathsf{P}_k$ is honest and picked after $\Delta_a$ and $\Delta_r$ are fixed, the probability that $e = \Delta_a^{-1}(-\Delta_r)$ is exactly $1/|\mathbb{F}|$. If that is not negligible the whole process can be repeated a number of times in parallel as mentioned in the protocol description.

Note that if the proof fails, it could be due to $\mathsf{P}_j$ being corrupted, so we run the "exception handler" to give $\mathsf{P}_i$ a chance to reveal $a$ and let the other parties do a forced commitment of $\mathsf{P}_j$ to $a$.

In Protocol Perfect Transfer, the idea is to check in a different way that the values committed by $\mathsf{P}_i$ and $\mathsf{P}_j$ are the same, namely we ask them to commit also to two polynomials that should be the same, furthermore these polynomial both determine the value $a$ in question. We then check that the two polynomials agree in so many points that they must be equal and this in turn implies that the values committed by $\mathsf{P}_i$ and $\mathsf{P}_j$ are indeed the same. A more detailed argument can be found in the proof of Theorem 5.1.

Protocol Commitment Multiplication

If any command used during this protocol fails, all players output `fail`.

1. $\mathsf{P}_i : \langle c\rangle_i \leftarrow ab$.
2. For $k = 1...n$, do:[a]
   (a) $\mathsf{P}_i$ chooses a uniformly random $\alpha \in \mathbb{F}$.
   (b) $\langle\alpha\rangle_i \leftarrow \alpha$; $\langle\gamma\rangle_i \leftarrow \alpha b$.
   (c) $\mathsf{P}_k$ broadcasts a uniformly random challenge $e \in \mathbb{F}$.
   (d) $\langle A\rangle_i \leftarrow e\langle a\rangle_i + \langle\alpha\rangle_i$; $A \leftarrow \langle A\rangle_i$.
   (e) $\langle D\rangle_i \leftarrow A\langle b\rangle_i - e\langle c\rangle_i - \langle\gamma\rangle_i$; $D \leftarrow \langle D\rangle_i$.
   (f) If $D \neq 0$ all players output `fail`.
3. If we arrive here, no command failed during the protocol and all $D$-values were 0. All players output `success`.

[a]The description of this step assumes that $1/|\mathbb{F}|$ is negligible in the security parameter. If that is not the case, we repeat the step $u$ times in parallel, where $u$ is chosen such that $(1/|\mathbb{F}|)^u$ is negligible.

### 5.2.2 Implementations of the `mult` Command

For the `mult`-command, we will follow a similar pattern as for `transfer`: we present an implementation that is only statistically secure but makes no assumption on $t$, the number of corrupted players, while the other implementation is perfectly secure but requires $t < n/3$.

The idea in Protocol Commitment Multiplication is that $\mathsf{P}_i$ commits to what he claims is the product $ab$ and then proves to each other player $\mathsf{P}_k$ that this was done correctly. It is easy to show that if $\mathsf{P}_i$ remains honest, then the proof succeeds and all values opened are random (or fixed to 0). So they reveal no extra information to the adversary and are easy to simulate. Using an analysis similar to the one for Protocol Transfer, it can be shown that if $c = ab + \Delta$ for $\Delta \neq 0$, then for each $\mathsf{P}_k$ the proof fails except with probability $1/|\mathbb{F}|$.

The idea in Protocol Perfect Commitment Multiplication is to have $\mathsf{P}_i$ commit to polynomials $f_a, f_b$ (determining $a, b$) and to what he claims is the product $h = f_a f_b$ of the polynomials. Then players check that $h(k) = f_a(k)f_b(k)$ in a number of points sufficient to

Protocol PERFECT COMMITMENT MULTIPLICATION

If any command used during this protocol fails, all players output `fail`.

1. $\langle c \rangle_i \leftarrow ab$.

2. $\mathsf{P}_i$ chooses polynomials $f_a(\mathtt{X}) = a + \alpha_1 \mathtt{X} + ... + \alpha_t \mathtt{X}^t$ and $f_b(\mathtt{X}) = b + \beta_1 \mathtt{X} + ... + \beta_t \mathtt{X}^t$, for random $\alpha_j, \beta_j$. He computes $h(\mathtt{X}) = f_a(\mathtt{X}) f_b(\mathtt{X})$, and writes $h(\mathtt{X})$ as $h(\mathtt{X}) = c + \gamma_1 \mathtt{X} + ... + \gamma_{2t} \mathtt{X}^{2t}$. Then establish commitments as follows:

$$\langle \alpha_j \rangle_i \leftarrow \alpha_j \text{ for } j = 1...t$$
$$\langle \beta_j \rangle_i \leftarrow \beta_j \text{ for } j = 1...t$$
$$\langle \gamma_j \rangle_i \leftarrow \gamma_j \text{ for } j = 1...2t$$

3. The `add` and `mult` commands are invoked to compute, for $k = 1...n$:

$$\langle f_a(k) \rangle_i = \langle a \rangle_i + k \langle \alpha_1 \rangle_i + ... + k^t \langle \alpha_t \rangle_i$$
$$\langle f_b(k) \rangle_i = \langle b \rangle_i + k \langle \beta_1 \rangle_i + ... + k^t \langle \beta_t \rangle_i$$
$$\langle h(k) \rangle_i = \langle b \rangle_i + k \langle \gamma_1 \rangle_i + ... + k^{2t} \langle \gamma_{2t} \rangle_i$$

   Then $(\mathsf{P}_k) a_k \leftarrow \langle f_a(k) \rangle_i$, $(\mathsf{P}_k) b_k \leftarrow \langle f_b(k) \rangle_i$ and $(\mathsf{P}_k) c_k \leftarrow \langle h(k) \rangle_i$ are executed.

4. For $k = 1..n$, do: $\mathsf{P}_k$ checks that $a_k b_k = c_k$, and broadcasts `accept` if this is the case, else broadcast `reject`.

5. For each $\mathsf{P}_k$ who said `reject`, do $a_k \leftarrow \langle f_a(k) \rangle_i$, $b_k \leftarrow \langle f_b(k) \rangle_i$ and $c_k \leftarrow \langle h(k) \rangle_i$, all players check whether $a_k b_k = c_k$ holds. If this is not the case, all players output `fail`.

6. If we arrive here, no command failed during the protocol and all relations $a_k b_k = c_k$ that were checked, hold. All players output `success`.

guarantee that indeed $h = f_a f_b$. A more detailed argument can be found in the proof of Theorem 5.1.

Let $\pi_{\texttt{TRANSFER,MULT}}$ be the protocol which implements the basic commands by simply relaying the commands to $\mathsf{F}_{\texttt{COM-SIMPLE}}$ and which implements the transfer and multiplication commands by running Protocol TRANSFER respectively Protocol COMMITMENT MULTIPLICATION on $\mathsf{F}_{\texttt{COM-SIMPLE}}$. Let $\pi_{\texttt{PTRANSFER,PMULT}}$ be the protocol which implements the basic commands by relaying the commands to $\mathsf{F}_{\texttt{COM-SIMPLE}}$ and which implements the transfer and multiplication commands by running Protocol PERFECT TRANSFER respectively Protocol PERFECT COMMITMENT MULTIPLICATION on $\mathsf{F}_{\texttt{COM-SIMPLE}}$.[1] We can then summarize the protocols and results we have shown so far as follows:

**Theorem 5.1** *Let $\mathsf{F}_{\texttt{COM-SIMPLE}}$ be a functionality that has all the commands of $\mathsf{F}_{\texttt{COM}}$, except the advanced manipulation commands. Then $\pi_{\texttt{TRANSFER,MULT}} \diamond \mathsf{F}_{\texttt{SC}} \diamond \mathsf{F}_{\texttt{COM-SIMPLE}}$ implements $\mathsf{F}_{\texttt{COM}}$ in $\mathrm{Env}^{t,sync}$ with statistical security for all $t < n$. Moreover, $\pi_{\texttt{PTRANSFER,PMULT}} \diamond \mathsf{F}_{\texttt{SC}} \diamond \mathsf{F}_{\texttt{COM-SIMPLE}}$ implements $\mathsf{F}_{\texttt{COM}}$ in $\mathrm{Env}^{t,sync}$ with perfect security for all $t < n/3$.*

*Proof* We prove here in detail the case for perfect security. The case of statistical security can be proved along the same lines, based on the arguments we gave above. This is straightforward

[1]The protocols for advanced commands should pick commitment identifiers for intermediary values as to not collide with the identifiers used by the basic commands.

(but quite tedious) and is left to the reader.

The first step is constructing the simulator $\mathcal{S}_{\texttt{COM}}$. On a high level, the idea is that the simulator will relay messages connected to the basic commitment commands while for `transfer` and `mult`, it will simulate a copy of $\pi_{\texttt{PTRANSFER,PMULT}} \diamond \mathsf{F}_{\text{SC}} \diamond \mathsf{F}_{\texttt{COM-SIMPLE}}$, where it uses $\mathsf{F}_{\texttt{COM-SIMPLE}}$ for basic commitment commands related to executing $\pi_{\texttt{PTRANSFER,PMULT}}$. Here, the simulator will as usual play the roles of the honest parties.

Agent $\mathcal{S}_{\texttt{COM}}$

A simulator for $\pi_{\texttt{PTRANSFER,PMULT}} \diamond \mathsf{F}_{\text{SC}} \diamond \mathsf{F}_{\texttt{COM-SIMPLE}}$ for the static case.

**Basic Commands:** When any of the basic commands are executed, the simulator will just forward messages back and forth between the environment and $\mathsf{F}_{\texttt{COM}}$[a]. When a corrupted player commits to a value, $\mathcal{S}_{\texttt{COM}}$ will store it.

**Transfer:** When $\mathsf{F}_{\texttt{COM}}$ is executing a `transfer` of some value $a$ from $\mathsf{P}_i$ to $\mathsf{P}_j$, then $\mathsf{P}_i$ has already committed to that value. Therefore, if $\mathsf{P}_i$ is corrupted, the simulator has stored $a$. On the other hand, if $\mathsf{P}_j$ is corrupt, then $a$ is leaked from $\mathsf{F}_{\texttt{COM}}$ as soon as `transfer` is initiated. Now, $\mathcal{S}_{\texttt{COM}}$ can run $\pi_{\texttt{PTRANSFER}} \diamond \mathsf{F}_{\text{SC}} \diamond \mathsf{F}_{\texttt{COM-SIMPLE}}$ with the environment using as input the correct value $a$, and where it simulates the parts of the honest players.

During this execution, if $\mathsf{P}_i$ is corrupt and makes some command fail, the simulator will not input $(\texttt{transfer}, i, cid, j)$ on `COM.infl`, which would make $\mathsf{F}_{\texttt{COM}}$ output fail, matching the real scenario. Otherwise $\mathcal{S}_{\texttt{COM}}$ inputs $(\texttt{transfer}, i, cid, j)$ on `COM.infl`, followed by $(\texttt{deltransfer}, i, cid, j)$.

If both parties are honest, then the transfer protocol is simulated with some random value.

**Multiplication:** When $\mathsf{F}_{\texttt{COM}}$ is executing a `mult` for $\mathsf{P}_i$ of some values $a, b$, then $\mathsf{P}_i$ has already committed to these values. Hence, if $\mathsf{P}_i$ is corrupted, the simulator has already stored $a, b$ and can do $\pi_{\texttt{PMULT}} \diamond \mathsf{F}_{\text{SC}} \diamond \mathsf{F}_{\texttt{COM-SIMPLE}}$ with the correct values.

During this execution, if $\mathsf{P}_i$ makes some command fail, the simulator will not input $(\texttt{mult}, i, cid_1, cid_2, cid_3)$ on `COM.infl`, which would make $\mathsf{F}_{\texttt{COM}}$ output fail, matching the real scenario. Otherwise it inputs $(\texttt{mult}, i, cid_1, cid_2, cid_3)$ on `COM.infl`.

If $\mathsf{P}_i$ is honest, $\pi_{\texttt{PMULT}} \diamond \mathsf{F}_{\text{SC}} \diamond \mathsf{F}_{\texttt{COM-SIMPLE}}$ is done with random values.

[a]Note here, that this does *not* include the basic commands needed for executing $\pi_{\texttt{PTRANSFER,PMULT}}$. This is only for the basic commands that $\mathsf{F}_{\texttt{COM}}$ will execute.

We see that the distribution produced by $\mathcal{S}_{\texttt{COM}}$ is the same as in the real case: Since messages concerning the basic commands are just relayed between the environment and $\mathsf{F}_{\texttt{COM}}$, this is exactly the same. During `transfer` if either $\mathsf{P}_i$ or $\mathsf{P}_j$ is corrupt, the simulator learns the value to be transferred. Similarly in `mult`, if $\mathsf{P}_i$ is corrupt, the simulator knows the values to be multiplied. Hence in these cases the protocol $\pi_{\texttt{PTRANSFER,PMULT}}$ is run on the true values. If, on the other hand, the players are honest then for both `transfer` and `mult` the environment sees at most $t$ points of polynomials of degree at least $t$. Thus, no information on the actual values is given.

What remains to argue is that the protocol execution is consistent with the inputs/outputs to/from $\mathsf{F}_{\texttt{COM}}$ . That is, if the protocol finishes successfully, then the output is correct, meaning that the players have indeed transferred a commitment or multiplied values correctly. On the other hand, if the simulated protocol fails then this is also consistent with the state of $\mathsf{F}_{\texttt{COM}}$.

First during `transfer`, if $\mathsf{P}_i$ and $\mathsf{P}_j$ are honest it is clear that all the proofs will be accepting. Moreover, if one or both are corrupted but we still reach the point where values are compared, no command failed earlier. At this point we know we have strictly more than $2t$ honest players. Hence, if at most $t$ players said `reject` then at least $t+1$ honest players said `accept` which implies that indeed $\mathsf{P}_i$ and $\mathsf{P}_j$ are committed to the same polynomials. In particular, their degree-0 terms are also the same, so it follows that $\langle a \rangle_i$ and $\langle a \rangle_j$ do contain the same value.

As for `mult`, it is clear from the way the polynomials are committed to, that even if $\mathsf{P}_i$ is corrupt, we have $f_a(0) = a, f_b(0) = b, h(0) = c$, and that $\deg(f_a), \deg(f_b) \leq t$ and $\deg(h) \leq 2t$. Moreover, if the players output `success` at the end, it follows that $f_a(k)f_b(k) = h(k)$ for every player $\mathsf{P}_k$ who remains honest. The assumption that $t < n/3$ implies that there are at least $2t+1$ honest players. Since $2t+1$ points determine any polynomial of degree at most $2t$ uniquely, it follows that $f_a(\mathtt{X})f_b(\mathtt{X}) = h(\mathtt{X})$, and hence that $ab = f_a(0)f_b(0) = h(0) = c$. It is also easy to check that if $\mathsf{P}_i$ remains honest, the protocol will always be successful.

Finally, if the simulated protocol fails at some point because of a corrupted $\mathsf{P}_i$ the simulator outputs `fail` which is consistent with the internal state of $\mathsf{F}_{\mathtt{COM}}$ since, in this case, $\mathcal{S}_{\mathtt{COM}}$ also makes $\mathsf{F}_{\mathtt{COM}}$ fail.

**Adaptive Corruption** We now show how to modify $\mathcal{S}_{\mathtt{COM}}$ to handle adaptively corrupted players. Hence, we allow a player $\mathsf{P}_k$ to be corrupted in the beginning of any round during the protocol. If this happens, the simulator learns the true values of $\mathsf{P}_k$'s inputs and then it has to show the internal state of $\mathsf{P}_k$ to the environment. For this, the simulator must patch the state so that it is consistent with the rest of the values shown to environment during the execution.

After creating a correctly patched state of the simulation we give the state of the newly corrupted $\mathsf{P}_k$ to the environment and then simulation continues exactly as described in $\mathcal{S}_{\mathtt{COM}}$.

We describe how to patch up for the advanced commands. The basic commands are executed by an ideal functionality, so nothing needs to be handled there.

First assume a player different from $\mathsf{P}_i$ or $\mathsf{P}_j$ for `transfer` or different from $\mathsf{P}_i$ for `mult` becomes corrupted. Here, nothing needs to be patched, the simulator simply shows the environment the values that were used in the execution. This is perfectly fine, since for both `transfer` and `mult`, the environment has seen less than $t$ points of polynomials of degree at least $t$. Therefore, showing the set of points which was used is still consistent with what has already been shown, and since at most $t$ points are shown for each polynomial, still no information is given on the actual inputs to the command.

Similarly for `transfer`, no patching is needed if at a point where one of $\mathsf{P}_i$ and $\mathsf{P}_j$ is corrupted, the other player gets corrupted. The protocol $\pi_{\mathtt{PTRANSFER}}$ was run with the correct value, so the simulator simply shows the environment the values that were used in the execution.

However, if in `mult`, $\mathsf{P}_i$ gets corrupted or in `transfer` either $\mathsf{P}_i$ or $\mathsf{P}_j$ gets corrupted where before both were honest, then the state needs to be patched. When the player becomes corrupted the simulator will learn the true value of the player's input. That is, for `transfer`, $\mathcal{S}_{\mathtt{COM}}$ will learn the true value to be transferred and for `mult`, $\mathcal{S}_{\mathtt{COM}}$ will learn the values to be multiplied.

Now $\mathcal{S}_{\mathtt{COM}}$ patches the state by recomputing the steps of the protocol with the true values. For the steps that involve secret sharing, $\mathcal{S}_{\mathtt{COM}}$ recomputes new polynomials which are consistent with the degree-0 term equalling the true values and the (strictly less than $t$) shares already shown to the environment. For example for `transfer` this means that the polynomial $f(X)$ from step 3 is recomputed such that it is consistent with the shares already shown to the environment and $f(0) = a$, where $a$ is the true value to be transferred. Note that this patching can easily be done even if a player gets corrupted in the middle of a command. The steps are simply recomputed up the point where the player gets corrupted.

□

Protocol CEAS (Circuit Evaluation with Active Security)

The protocol assumes that $t < n/3$ and proceeds in three phases: the input sharing, computation and output reconstruction phases.

**Input Sharing:** Each player $\mathsf{P}_i$ holding input $x_i \in \mathbb{F}$ distributes $[\![x_i; f_{x_i}]\!]_t$. If this fails, $\mathsf{P}_i$ is corrupt, so we will instead assign 0 as default input for $\mathsf{P}_i$: execute $\langle 0 \rangle_k \Leftarrow 0$ for $k = 1..n$, thus creating $[\![0; o]\!]_t$ where $o$ is the zero-polynomial.

We then go through the circuit and process the gates one by one in the computational order. Just after the input sharing phase, we consider all input gates as being processed. We will maintain the following:

**Invariant:** Consider an input or an output wire for any gate, and let $a \in \mathbb{F}$ be the value assigned to this wire. Then, if the gate has been processed, the players hold $[\![a; f_a]\!]_t$ for a polynomial $f_a$.

We then continue with the last two phases of the protocol:

**Computation Phase:** Repeat the following until all gates have been processed (then go to the next phase): Consider the first gate in the computational order that has not been processed yet. According to the type of gate, do one of the following

**Addition gate:** The players hold $[\![a; f_a]\!]_t, [\![b; f_b]\!]_t$ for the two inputs $a, b$ to the gate. The players compute $[\![a + b; f_a + f_b]\!]_t = [\![a; f_a]\!]_t + [\![b; f_b]\!]_t$.[a]

**Multiply-by-constant gate:** The players hold $[\![a; f_a]\!]_t$ for the input $a$ to the gate. The players compute $[\![\alpha a; \alpha f_a]\!]_t = \alpha[\![a; f_a]\!]_t$.

**Multiplication gate:** The players hold $[\![a; f_a]\!]_t, [\![b; f_b]\!]_t$ for the two inputs $a, b$ to the gate.

1. The players compute $[\![ab; f_a f_b]\!]_{2t} = [\![a; f_a]\!]_t * [\![b; f_b]\!]_t$. Note that this involves each $\mathsf{P}_j$ committing to $f_a(j)f_b(j)$ and proving that this was done correctly. So this may fail for up to $t$ players.
2. Define $h \overset{\text{def}}{=} f_a f_b$. Then $h(0) = f_a(0)f_b(0) = ab$ and the parties hold $[\![ab; h]\!]_{2t}$, where $\mathsf{P}_i$ is committed by $\langle h(i) \rangle_i$. Each $\mathsf{P}_i$ distributes $[\![h(i); f_i]\!]_t$ from $\langle h(i) \rangle_i$. Also this step may fail for up to $t$ players.
3. Let $S$ be the indices of at the (at least $n-t$) players for which the previous two steps did not fail. Note that since $t < n/3$, $\deg(h) = 2t < n - t$. Let $\mathbf{r}_S$ be a recombination vector for $S$, that is, a vector $\mathbf{r}_S = (r_1, \ldots, r_n)$ for which it holds that $h(0) = \sum_{i \in S} r_i h(i)$ for any polynomial $h$ of degree $\leq 2t$. The players compute

$$\sum_{i \in S} r_i [\![h(i); f_i]\!]_t = [\![\sum_{i \in S} r_i h(i); \sum_{i \in S} r_i f_i]\!]_t = [\![h(0); \sum_{i \in S} r_i f_i]\!]_t = [\![ab; \sum_{i \in S} r_i f_i]\!]_t \, .$$

**Output Reconstruction:** At this point all gates, including the output gates have been processed. So do the following for each output gate (labeled $i$): The players hold $[\![y; f_y]\!]_t$ where $y$ is the value assigned to the output gate. Now we open $[\![y; f_y]\!]_t$ towards $\mathsf{P}_i$.

[a]Whenever we say that the parties compute a command, we implicitly say that they wait until $\mathsf{F}_{\mathsf{COM}}$ returns `success` or `fail` before the proceed with the next command.

## 5.3 A Secure Function Evaluation Protocol for Active Adversaries

In this section we show how to implement $\mathsf{F}^f_{\mathtt{SFE}}$, given access to $\mathsf{F}_{\mathtt{SC}}$ and $\mathsf{F}_{\mathtt{COM}}$. The basic idea is to do the same as in our passively secure protocol, but in addition make sure that all parties are committed to all shares they hold.

We note for future reference that the definition of $\mathsf{F}_{\mathtt{COM}}$ implies that if one of its commands fails, this is always because a particular corrupt player has caused this to happen by not providing the correct input. Therefore the honest players can always conclude from failure of a command that some particular player is corrupt.

We will need the following notation for $a \in \mathbb{F}$ and $f$ a polynomial of degree at most $t$ over $\mathbb{F}$:

$$[\![a; f]\!]_t = (\langle f(1)\rangle_1, ..., \langle f(n)\rangle_n) \ .$$

Thus this notation refers to $n$ variables held by $\mathsf{F}_{\mathtt{COM}}$ and claims that the $i$'th variable contains the appropriate value $f(i)$, owned by $\mathsf{P}_i$.

When we say that $\mathsf{P}_i$ distributes $[\![a; f_a]\!]_t$, this means the following: $\mathsf{P}_i$ chooses a polynomial $f_a(\mathtt{X}) = a + \alpha_1\mathtt{X} + ... + \alpha_t\mathtt{X}^t$ with $\alpha_j \in \mathbb{F}$ at random, then $\mathsf{P}_i$ commits to the coefficients of $f_a$, the addition and multiplication commands are used to compute a commitment to $f_a(k)$ owned by $\mathsf{P}_i$, for $k = 1..n$, and finally this commitment is transferred to $\mathsf{P}_k$. More formally, the following sequence of commands of $\mathsf{F}_{\mathtt{COM}}$ are executed:

1. $\langle a\rangle_i \leftarrow a$, $\langle \alpha_j\rangle_i \leftarrow \alpha_j, j = 1..t$.
2. $\langle f_a(k)\rangle_i \leftarrow \langle a\rangle_i + \sum_{j=1}^t k^j \langle \alpha_j\rangle_i, k = 1..n$
3. $\langle f_a(k)\rangle_k \leftarrow \langle f_a(k)\rangle_i, k = 1..n$

It is very easy to see that when $\mathsf{P}_i$ distributes $[\![a; f_a]\!]_t$, this does in fact create the object $[\![a; f_a]\!]_t$, or all players agree that one of the commands failed, in which case they can conclude that $\mathsf{P}_i$ is corrupt. Note that this fact is not a result of proving the above sequence of commands secure as a protocol, it simply follows by definition of $\mathsf{F}_{\mathtt{COM}}$. Above, we said that $\mathsf{P}_i$ should commit to $a$ as a part of the process. But $\mathsf{P}_i$ could instead use an already existing commitment $\langle a\rangle_i$ to some value $a$. In this case, we say that $\mathsf{P}_i$ distributes $[\![a; f_a]\!]_t$ from $\langle a\rangle_i$. Again by definition of $\mathsf{F}_{\mathtt{COM}}$, it is guaranteed that the new object determines the same value that the existing commitment contains.

Distributing $[\![a; f_a]\!]_t$ is actually a concept known from the literature as Verifiable Secret Sharing: a dealer (here $\mathsf{P}_i$) distributes shares of a secret to the players in such a way that the honest players are guaranteed to get consistent shares of a well defined secret, or they will agree that the dealer failed. Moreover, the correct secret can always be reconstructed, even if the dealer does not participate. This is easily seen to be the case for $[\![a; f_a]\!]_t$ if $t < n/2$: In the following, when we say that $[\![a; f_a]\!]$ is opened, this means we execute

$$f_a(k) \leftarrow \langle f_a(k)\rangle_k, \text{ for } k = 1..n \ .$$

This will result in up to $t$ failures, but also in at least $n - t > t$ correct values, from which all players can compute $a$ by Lagrange interpolation (see Section 3.2). We can also open $[\![a; f_a]\!]$ towards $\mathsf{P}_j$. This means we execute $(\mathsf{P}_j)f_a(k) \leftarrow \langle f_a(k)\rangle_k$, for $k = 1..n$, so that only $\mathsf{P}_j$ learns $a$.

We can also do arithmetic on objects of form $[\![a; f_a]\!]_t$, in much the same way as we did for $[a, f_a]_t$ when we did passively secure protocols. Now, however we have to do arithmetic on committed values. Hence, for $\alpha, a, b \in \mathbb{F}$ and $f_a, f_b$ polynomials of degree at most $t$, we define $[\![a; f_a]\!]_t + [\![b; f_b]\!]_t$, $\alpha[\![a; f_a]\!]_t$ and $[\![a; f_a]\!]_t * [\![b; f_b]\!]_t$ to denote sequences of commands from $\mathsf{F}_{\mathtt{COM}}$, as follows

$[\![a; f_a]\!]_t + [\![b; f_b]\!]_t$ means that for $k = 1, \ldots, n$ we execute: $\langle f_a(k) + f_b(k)\rangle_k \leftarrow \langle f_a(k)\rangle_k + \langle f_b(k)\rangle_k$.

$\alpha[\![a; f_a]\!]_t$ means that for $k = 1, \ldots, n$ we execute: $\langle \alpha f_a(k)\rangle_k \leftarrow \alpha\langle f_a(k)\rangle_k$.

$[\![a; f_a]\!]_t * [\![b; f_b]\!]_t$ means that for $k = 1, \ldots, n$ we execute: $\langle f_a(k) f_b(k)\rangle_k \leftarrow \langle f_a(k)\rangle_k \langle f_b(k)\rangle_k$.

Again by definition of $\mathsf{F}_{\mathtt{COM}}$, it is easy to see that these command sequences create objects of the form we naturally would expect, namely $[\![a + b; f_a + f_b]\!]_t$, $[\![\alpha a; \alpha f_a]\!]_t$, and $[\![ab; f_a f_b]\!]_{2t}$, respectively, again unless one of the commands fail. Therefore, by a slight abuse of notation, in the following we will write:

$$[\![a + b; f_a + f_b]\!]_t = [\![a; f_a]\!]_t + [\![b; f_b]\!]_t,$$

$$[\![\alpha a; \alpha f_a]\!]_t = \alpha[\![a; f_a]\!]_t,$$

$$[\![ab; f_a f_b]\!]_{2t} = [\![a; f_a]\!]_t * [\![b; f_b]\!]_t$$

to denote that we execute the command sequence on the right side to create the object on the left side.

The notation we have defined is used in Protocol CEAS, whose security follows almost directly from the security of the passive secure protocol. In particular, since the manipulation commands of $\mathsf{F}_{\mathtt{COM}}$ are used, all parties compute all shares exactly as in the passive secure protocol, and $\mathsf{F}_{\mathtt{COM}}$ keeps the shares as secret. Therefore, as we see below, essentially the same simulator works in this case as well. Letting $\pi_{\mathtt{CEAS}}$ denote Protocol CEAS, we have

**Theorem 5.2** *$\pi^f_{CEAS} \diamond F_{SC} \diamond F_{COM}$ implements $F^f_{SFE}$ in* $\mathrm{Env}^{t,sync}$ *with perfect security for any $f$ and for all $t < n/3$.*

*Proof* We construct a simulator for $\pi^f_{\mathtt{CEAS}} \diamond \mathsf{F}_{\mathtt{SC}} \diamond \mathsf{F}_{\mathtt{COM}}$ where the idea is very similar to the simulator, $\mathcal{S}_{\mathtt{CEPS}}$, for the passive case. The simulator runs a copy of $\pi^f_{\mathtt{CEAS}} \diamond \mathsf{F}_{\mathtt{SC}} \diamond \mathsf{F}_{\mathtt{COM}}$ where it plays the role of the honest parties and executes the protocol with the environment which is controlling the actively corrupted players. Since the simulator is running $\mathsf{F}_{\mathtt{COM}}$ it is able to keep track of all the corrupted parties' values. We present first the simulator Agent $\mathcal{S}_{\mathtt{CEAS}}$ for the static case, where the environment provides inputs to all parties in the same round. The case where not all parties receive inputs in the same round is handled as in the passive case. After arguing static security, we show how to modify the simulator to achieve adaptive security as well.

Perfect security is argued similarly to the passive case. First, the shares sent by honest players to corrupt players in the input distribution and computation phase are distributed exactly as in the simulation, namely $t$ uniformly random shares for each shared value. Furthermore, since the commands of $\mathsf{F}_{\mathtt{COM}}$ are used, the shares computed by local computation are the same as in the passively secure protocol. This follows from the definition of $\mathsf{F}_{\mathtt{COM}}$ which guarantees that either the players are committed to the correct shares or they all agree that a command failed because a particular player $\mathsf{P}_i$ is corrupt. However, this can happen for at most $t$ players which we by construction can tolerate when $t < n/3$. Therefore, the invariant in the protocol holds, which implies that the output is correct and hence *the same* in both $\pi^f_{\mathtt{CEAS}} \diamond \mathsf{F}_{\mathtt{SC}} \diamond \mathsf{F}_{\mathtt{COM}}$ and $\mathsf{F}^f_{\mathtt{SFE}}$. Finally, $\mathcal{S}_{\mathtt{CEAS}}$ makes $\mathsf{F}^f_{\mathtt{SFE}}$ give output on $\mathtt{SFE.out}_i$ at the same point when $\mathsf{P}_i$ would. We conclude that the views produced by $\mathcal{S}_{\mathtt{CEAS}}$ are distributed exactly as in the real case.

**Adaptive Corruption** We now show how to modify $\mathcal{S}_{\mathtt{CEAS}}$ such that it can handle adaptively corrupted players. Hence, we allow a player $\mathsf{P}_k$ to be corrupted in the beginning of any round during the protocol. If this happens, the simulator learns the true value of $x_k$ and then it has to show the internal state of $\mathsf{P}_k$ to the environment. For this, the simulator must patch the state of $\mathsf{P}_k$ so that it is consistent with $x_k$ and the rest of the values shown to environment during the execution. After creating a correctly patched state of the simulation we give the state of the newly corrupted $\mathsf{P}_k$ to the environment and then simulation continues exactly as described in $\mathcal{S}_{\mathtt{CEAS}}$.

Agent $\mathcal{S}_{\texttt{CEAS}}$

A simulator for $\pi^f_{\texttt{CEAS}} \diamond \mathsf{F}_{\texttt{SC}} \diamond \mathsf{F}_{\texttt{COM}}$ for the static case where the environment gives inputs to all parties in the same round.

**Initialize:** In the preamble, the environment $\mathcal{Z}$ specifies the set $C$ of actively corrupted parties. After this, the simulator sets up a copy of $\pi^f_{\texttt{CEAS}} \diamond \mathsf{F}_{\texttt{SC}} \diamond \mathsf{F}_{\texttt{COM}}$ where it actively corrupts the players in $C$ and connects the special ports of $\pi^f_{\texttt{CEAS}} \diamond \mathsf{F}_{\texttt{SC}} \diamond \mathsf{F}_{\texttt{COM}}$ to $\mathcal{Z}$. $\mathcal{S}_{\texttt{CEAS}}$ also does active corruptions of $\mathsf{P}_i \in C$ on $\mathsf{F}^f_{\texttt{SFE}}$, which means that $\mathsf{F}^f_{\texttt{SFE}}$ will ignore all inputs on $\texttt{SFE.in}_i$ and stop giving outputs on $\texttt{SFE.out}_i$. Instead the simulator must now provide the input to $\mathsf{F}^f_{\texttt{SFE}}$ on behalf of the corrupt players.

**Input Sharing:** The simulator performs the steps of input sharing with the environment (which runs the corrupted parties). That is, $\mathcal{S}_{\texttt{CEAS}}$ runs the honest parties, so it will for each $\mathsf{P}_i \notin C$ distribute a sharing of 0.

When the corrupt parties share their inputs the simulator will receive all the values they want to commit to (since the simulator is running $\mathsf{F}_{\texttt{COM}}$). In particular $\mathcal{S}_{\texttt{CEAS}}$ will for each $\mathsf{P}_j \in C$ receive $x_j$ which it can then pass on to $\mathsf{F}^f_{\texttt{SFE}}$.

**Computation Phase:** In the computation phase the simulator simply performs the protocol steps of $\pi^f_{\texttt{CEAS}}$ together with the environment according to the gates that need to be processed.

**Output Reconstruction:** First $\mathcal{S}_{\texttt{CEAS}}$ instructs $\mathsf{F}^f_{\texttt{SFE}}$ to evaluate the function, thereby learning the outputs $y_j$ for all $\mathsf{P}_j \in C$.

For each output of a corrupted party $\mathsf{P}_j$, it takes the $t$ shares that the corrupted parties hold and defines $f_{y_j}(0) \stackrel{\text{def}}{=} y_j$. After this, $f_{y_j}(\texttt{X})$ is defined in $t+1$ points and hence $\mathcal{S}_{\texttt{CEAS}}$ can compute the unique degree-$t$ polynomial consistent with these points. Then, for $\mathsf{P}_i \notin C$ it computes a commitment to the share $f_{y_j}(i)$ and opens it to $\mathsf{P}_j$ on behalf of $\mathsf{P}_i$.

When $\mathcal{Z}$ clocks out $\mathsf{P}_i$ in the copy of $\pi^f_{\texttt{CEAS}} \diamond \mathsf{F}_{\texttt{SC}} \diamond \mathsf{F}_{\texttt{COM}}$ so that $\mathsf{P}_i$ should produce an output, the simulator first inputs ($\texttt{delivery-round}$) to $\mathsf{F}^f_{\texttt{SFE}}$ if it has not done so already. Then $\mathcal{S}_{\texttt{CEAS}}$ inputs $(\texttt{clock-out}, i)$ to $\mathsf{F}^f_{\texttt{SFE}}$ to make it output $y_i$ on $\texttt{SFE.out}_i$, exactly as $\mathsf{P}_i$ would have done in the protocol. (Since the environment is running the corrupted parties this will only happen for the honest parties.)

We describe here the case where $\mathsf{P}_k$ gets corrupted after the protocol is completed. Then, handling corruption at an earlier stage will simply consist of only doing a smaller part of the patching steps (namely up to the point where the player gets corrupted). Basically, the way to patch the state is to recompute every honest player's share which is affected by the input of $\mathsf{P}_k$. This is done as follows [2]:

- *Input Sharing.* When $\mathcal{S}_{\texttt{CEAS}}$ learns $x_k$ it can compute a new random secret sharing of $x_k$ which is consistent with the (strictly less than $t$) shares shown to the environment and $f_{x_k}(0) = x_k$. From $f_{x_k}(\texttt{X})$ the simulator recomputes the steps of input distribution, not involving other corrupt parties. That is, first $\mathcal{S}_{\texttt{CEAS}}$ computes the commitments to the new coefficients of $f_{x_k}(\texttt{X})$. Then it computes new commitments of shares to honest parties and transfers the commitments to them, obtaining $\langle f_{x_k}(i) \rangle_i$ .

- *Addition or Multiplication by a constant.* Here the players only do local computations so

[2] It might be a good idea to have the protocol $\pi_{\texttt{CEAS}}$ in mind when reading this.

$\mathcal{S}_{\texttt{CEAS}}$ can simply recompute commitments of all honest parties' shares which were affected by $f_{x_k}(\texttt{X})$.

- *Multiplication.* The first round in the processing of a multiplication gate only has local computations and hence, the simulator recomputes as above commitments of the affected shares of honest parties. Then in the second round, new shares are distributed of the product of two polynomials $f_a f_b$. If $f_{x_k}(\texttt{X})$ is involved in one of these polynomials, $\mathcal{S}_{\texttt{CEAS}}$ must compute a new random secret sharing of $f_a(0)f_b(0)$ as it did for $x_k$ in input sharing. In the third and last round the simulator is again able to recompute the shares of the honest parties by local computations. Recall that the recombination vector is independent of $f_a f_b(\texttt{X})$; it is the same for all polynomials of degree at most $n-t$, so it is indeed possible to redo this step with the new product.

- *Output Reconstruction.* This step can also simply be redone, making the honest parties open the new commitments of shares that were recomputed above.

As mentioned, if a player gets corrupted earlier, patching is done only up to that point. Note that this is also possible even if the player gets corrupted before a round in the middle of some command which has more than one round. Patching is simply done up to the round just before the player gets corrupted and then the simulation proceeds as in the static case. Also, if a player gets corrupted just before the input sharing phase but after the environment has provided the inputs, the simulation is essentially the same as before. The only difference is that giving $\mathsf{F}^f_{\texttt{SFE}}$ the input of a corrupted party now means overriding any previous value stored for that party. □

We note that it is possible to modify Protocol CEAS to make it work also for $t < n/2$. The difficulty in this case is that if a player fails during processing of a multiplication gate, we cannot proceed since all players are needed to determine a polynomial of degree $2t$, when we only assume $t < n/2$. The simplest way to handle such failures is to go back to the start of the computation and open the input values of the players that have just been disqualified. Now we restart the protocol, while the honest players simulate the disqualified players. Note that we can use default random choices for these players, so there is no communication needed for the honest players to agree on the actions of the simulated ones. This allows the adversary to slow down the protocol by a factor at most linear in $n$. In doing this, it is important that we do not have to restart the protocol after some party received its output — this is ensured by starting the Output Reconstruction phase only after all multiplication gates have been handled. If we allowed some party $\mathsf{P}_i$ to see its output before all multiplication gates were handled, then it could for instance first run with $x_i = 1$ to let the corrupted parties learn their parts of the output $f(x_1, \ldots, x_n)$ when $x_i = 1$. Then it could make the next multiplication gate fail. Then the parties would rerun, but now using the dummy input $x_i = 0$ for $\mathsf{P}_i$. This would let the corrupted parties learn their parts of the output $f(x_1, \ldots, x_n)$ when $x_i = 0$. Seeing $f$ evaluated on two points might let the corrupted learn more about the inputs of the honest parties than is possible in the ideal world. It is, however, clear that it is secure to rerun as long as no party received its output, as the prefix of the protocol not including the Reconstruction phase leaks no information to any party.

**Exercise 5.1** The above way to handle corrupted parties allows the corrupted parties to delay the computation by a factor $t$, as they might one by one refuse to participate in the last multiplication gate, forcing yet another rerun. It is possible to ensure that they can delay by at most a factor $O(\log(t))$. We sketch the first part of how this is done, and the exercise is to realize how to generalize. Assume that in the first execution of the protocol all parties follow the protocol, except that $\mathsf{P}_n$ at some point refuses to participate in the execution of the protocol for a multiplication gate. Then we can let $n' = n-1$, $t' = t-1$ and $f'(x_1, \ldots, x_{n'}) = (y_1, \ldots, y_{n'})$, where the outputs are given by $(y_1, \ldots, y_{n'}, y_n) = f(x_1, \ldots, x_{n'}, 0)$. Then we can let the parties $\mathsf{P}_1, \ldots, \mathsf{P}_{n'}$ run the protocol for securely evaluating $f'$ among themselves using secret sharings of degree $t'$

instead of $t$. It is secure to only use degree $t'$ as at most $t'$ of the remaining parties are corrupted, as we started with at most $t$ corrupted parties and just excluded $\mathsf{P}_n$. This approach also leads to a correct result, as the honest parties learn their part of the output $f(x_1, \dots, x_{n-1}, 0)$. So, all $\mathsf{P}_n$ got out of cheating was that it was forced to use input $x_n = 0$ and that it does not learn its output. Notice, however, that if we started with $2t + 1 \leq n$ we now have that $2t' + 1 \leq n' - 1$, as $2t' + 1 = 2(t - 1) + 1 = (2t + 1) - 2 \leq n - 2$ and $n' = n - 1$. Since $2t' + 1 \leq n' - 1$ and we are running the protocol with polynomials of degree at most $t'$ and has $n'$ parties which are holding shares, it follows that we can run the protocol for a multiplication gate even if 1 party do not share its $h(i)$ value — we will still have $n' - 1$ of the values $h(i)$ which is enough to interpolate $h(0)$ as $h(\mathtt{X})$ has degree at most $2t'$. This means that to force a rerun of the protocol, at least 2 of the remaining corrupted parties have to stop participating. In that case we exclude these 2 parties and restart with $n'' = n' - 2$ parties and at most $t'' = t' - 2$ corrupted parties. So, we can use polynomials of degree at most $t'' = t - 3$ and hence tolerate that even more corrupted parties stop participating before we need to do the next rerun — we can in fact tolerate that up to 3 of the $n''$ parties do not help in the multiplications.

1. In the first execution we can tolerate that 0 corrupted parties refuse to participate in the protocol for a multiplication gate. In the second execution we can tolerate that 1 corrupted party refuses to participate in the protocol for a multiplication gate. Argue that in the third execution we can tolerate that 3 corrupted parties refuse to participate in the protocol for a multiplication gate and that we hence can exclude 4 more parties before the fourth execution if we are forced to do a fourth execution.

2. How many refusing parties can we tolerate in the fourth execution?

3. How many refusing parties can we tolerate in the $i$'th execution?

4. If we start with $2t + 1 = n$, what is the worst-case number of reruns which the corrupted parties can force?

Let $\pi^f_{\mathtt{CEAS\text{-}N/2}}$ be $\pi^f_{\mathtt{CEAS}}$ modified to handle $t < n/2$ as we just discussed. We then have the following theorem, the proof of which we leave to the reader:

**Theorem 5.3** *$\pi^f_{\mathtt{CEAS\text{-}N/2}} \diamond \mathsf{F}_{\mathtt{SC}} \diamond \mathsf{F}_{\mathtt{COM}}$ implements $\mathsf{F}^f_{\mathtt{SFE}}$ in* $\mathrm{Env}^{t,sync}$ *with perfect security for any $f$ and for all $t < n/2$.*

### 5.3.1 Reactive Functionalities

So far, our results for general secure computing have only been about secure function evaluation. It is easy to define a *reactive functionality* that is much more general than $\mathsf{F}^f_{\mathtt{SFE}}$, namely a functionality that keeps internal state and offers to repeatedly do the following: compute some function that depends on both the state and inputs from the players, and update the state as well as deliver outputs to players.

Using the way we represent data in $\pi_{\mathtt{CEAS}}$, it is easy to design a secure implementation of such a functionality: we can represent the state as a number of objects of form $[\![a; f_a]\!]$. We can assume we get the inputs from players represented in the same form, and then we evaluate whatever function we need using the subprotocols for addition and multiplication. When we have representations of the outputs, we keep those that represent the new state, and open those that contain output to the players.

It is straightforward to formalize this construction, we leave the details to the reader.

## 5.4 Realization of Homomorphic Commitments

We assume throughout this section that we are in the information theoretic scenario with secure point-to-point channels and consensus broadcast. Most of the time we assume that $t < n/3$, though we will look at the case $t < n/2$ at the end of the section. We show how to implement a commitment scheme with the desired basic commands and simple manipulation commands.

The idea that immediately comes to mind in order to have a player $D$ commit to $a$ is to ask him to secret share $a$. At least this will hide $a$ from the adversary if $D$ is honest, and will immediately ensure the homomorphic properties we need, namely to add commitments, each player just adds his shares, and to multiply by a constant, all shares are multiplied by the constant.

However, if $D$ is corrupt, he can distribute inconsistent shares, and can then easily "open" a commitment in several ways, as detailed in the exercise below.

**Exercise 5.2** A player $D$ sends a value $a_i$ to each player $\mathsf{P}_i$ (also to himself). $D$ is supposed to choose these such that $a_i = f(i)$ for all $i$, for some polynomial $f(\mathtt{X})$ of degree at most $t$ where $t < n/3$ is the maximal number of corrupted players. At some later time, $D$ is supposed to reveal the polynomial $f(\mathtt{X})$ he used, and each $\mathsf{P}_i$ reveals $a_i$, The polynomial is accepted if values of at most $t$ players disagree with $f(\mathtt{X})$ (we cannot demand fewer disagreements, since the corrupted parties might send wrong values, so we may get $t$ of them even if $D$ was honest).

1. We assume here (for simplicity) that $n = 3t + 1$. Suppose the adversary corrupts $D$. Show how to choose two different polynomials $f(\mathtt{X}), f'(\mathtt{X})$ of degree at most $t$ and values $\tilde{a}_i$ for $D$ to send, such that $D$ can later reveal and have accepted both $f(\mathtt{X})$ and $f'(\mathtt{X})$.

2. Suppose for a moment that we would settle for computational security, and that $D$ must send to $\mathsf{P}_i$, not only $a_i$, but also his digital signature $s_i$ on $a_i$. We assume that we can force $D$ to send a valid signature even if he is corrupt. We can now demand that to be accepted, a polynomial must be consistent with *all* revealed and properly signed shares. Show that now, the adversary cannot have two different polynomials accepted, even if up to $t \leq n/3$ players may be corrupted before the polynomial is to be revealed. [Hint: First argue that the adversary must corrupt $D$ before the $a_i, s_i$ are sent out (this is rather trivial). Then, assume $f_1(\mathtt{X})$ is later successfully revealed and let $C_1$ be the set that is corrupted when $f_1$ is revealed. Assume the adversary could also choose to let $D$ reveal $f_2(\mathtt{X})$, in which case $C_2$ is the corrupted set. Note that if we assume the adversary is adaptive, you cannot assume that $C_1 = C_2$. But you can still use the players outside $C_1, C_2$ to argue that $f_1(\mathtt{X}) = f_2(\mathtt{X})$.]

3. (Optional) Does the security proved above still hold if $t > n/3$? Why or why not?

To prevent the problems outlined above, we must find a mechanism to ensure that the shares of all uncorrupted players after committing consistently determine a polynomial $f(\mathtt{X})$ of degree at most $t$, without harming privacy of course.

### Minimal Distance Decoding

Before we do so, it is important to note that $n$ shares out of which at most $t$ are corrupted still uniquely determine the committed value $a$, even if we don't know which $t$ of them are corrupted. This is based on an observation also used in error correcting codes.

Concretely, let $f(\mathtt{X})$ be a polynomial of degree at most $t$ and define the shares

$$\mathbf{s}_f \stackrel{\text{def}}{=} (f(1), \ldots, f(n)) ,$$

and let $\mathbf{e} \in \mathbb{F}^n$ be an arbitrary "error vector" subject to

$$w_H(\mathbf{e}) \leq t ,$$

where $w_H$ denotes the Hamming-weight of a vector (i.e., the number of its non-zero coordinates), and define

$$\tilde{\mathbf{s}} \stackrel{\text{def}}{=} \mathbf{s} + \mathbf{e} .$$

Then $a$ is uniquely defined by $\tilde{\mathbf{s}}$.

In fact, more is true, since the entire polynomial $f(\mathtt{X})$ is. This is easy to see from Lagrange Interpolation and the fact that $t < n/3$.

Namely, suppose that $\tilde{\mathbf{s}}$ can also be "explained" as originating from some other polynomial $g(\mathtt{X})$ of degree at most $t$ together with some other error vector $\mathbf{u}$ with Hamming-weight at most $t$. In other words, suppose that

$$\mathbf{s}_f + \mathbf{e} = \mathbf{s}_g + \mathbf{u} \ .$$

Since $w_H(\mathbf{e}), w_H(\mathbf{u}) \leq t$ we have that $w_H(\mathbf{e}+\mathbf{y}) \leq 2t$, so since $n > 3t$ there are at $\geq n-2t > t$ positions in which the coordinates of both are simultaneously zero. Thus, for at least $t+1$ values of $i$ we have

$$f(i) = g(i) \ .$$

Since both polynomials have degree at most $t$, this means that

$$f(\mathtt{X}) = g(\mathtt{X}) \ .$$

The conclusion is that if a player $\mathsf{P}_i$ has distributed shares of some value $a \in \mathbb{F}$, and we can somehow guarantee that the shares are consistent with one polynomial $f$ of degree at most $t < n/3$, then this can serve as a commitment to $a$. $\mathsf{P}_i$ can open by broadcasting the polynomial, and then each player can state whether his share agrees with the polynomial claimed by $\mathsf{P}_i$. The opening is accepted is at most $t$ players disagree. By the above discussion, if $\mathsf{P}_i$ tries to open a polynomial different from $f$, at least $t+1$ players would disagree.

Note that we do not rely on Lagrange interpolation or any efficient method for decoding in presence of errors: we rely on the committer to supply the correct polynomial, we are fine as long as it is uniquely defined.

**Exercise 5.3** This exercises asks you to realize how minimal distance decoding can be used to make a noisy channel reliable. Assume that we have a sender $\mathsf{S}$ and a receiver $\mathsf{R}$ which are connected be a noisy channel which allows to transfer bit-strings. The effect of the noise is that sometimes a bit sent as 0 will be received as 1 and vice versa. If the errors are probabilistic in nature and not too common, then the first step in handling the errors is to send long bit strings. Then, even though individual bits might flip, most of the bit string will be received correctly. The next step is then to send a redundant string, such that receiving most of the string correctly will allow to compute the original string. This exercise focuses on the second step.

Let $t$ be some fixed parameter, let $n = 3t+1$ and let $\ell = \lceil \log_2(n) \rceil$, and let $\mathbb{F}$ be the finite field with $2^\ell$ elements. This allows to consider a bit string $m_i \in \{0,1\}^\ell$ as an element from $\mathbb{F}$, and it allows to consider a bit string $m \in \{0,1\}^{(t+1)\ell}$ as an element from $\mathbb{F}^{t+1}$, as follows: split $m$ into $t+1$ blocks $m = (m_0, \ldots, m_t)$, each being $\ell$ bits long. Then consider $m_i$ as an element from $\mathbb{F}$. Now $m \in \mathbb{F}^{t+1}$. Finally, given an element $(m_0, \ldots, m_t) \in \mathbb{F}^{t+1}$ we can define the polynomial $f_m(\mathtt{X}) \overset{\text{def}}{=} \sum_{i=0}^{t} m_i \mathtt{X}^i$ over $\mathbb{F}$, i.e., for each bit string $m \in \{0,1\}^{(t+1)\ell}$ we have a unique polynomial $f_m(\mathtt{X})$ of degree at most $t$ and for each polynomial $f(\mathtt{X})$ of degree at most $t$ we have a bit string $m \in \{0,1\}^{(t+1)\ell}$. To send a bit string from $\{0,1\}^{(t+1)\ell}$ we can hence focus on sending a polynomial of degree at most $t$.

Show that when $\mathsf{S}$ is given a bit string $m \in \{0,1\}^{(t+1)\ell}$ it can send a bit string $c \in \{0,1\}^{n\ell}$ to $\mathsf{R}$ which allows $\mathsf{R}$ to recover $m$ even if up to $t$ of the bits of $c$ are flipped during transmission.

### A Perfect Commitment Protocol

Based on the above observation, what we need is a protocol ensuring that all honest parties end up holding consistent shares of some value, of course while preserving privacy if the committer is (remains) honest.

The idea for ensuring consistent behavior is to have the committer $\mathsf{P}_i$ not only secret share the value he commits to, but also distribute some extra "redundant" information that the other players can check.

**Redundant sharing using bivariate polynomials.** To do a redundant sharing, $\mathsf{P}_i$ chooses a polynomial in two variables:

$$f_a(\mathtt{X}, \mathtt{Y}) = \sum_{\sigma,\tau=0}^{t} \alpha_{\sigma,\tau} \mathtt{X}^\sigma \mathtt{Y}^\tau,$$

where the $\alpha_{\sigma,\tau}$ are random, subject to two constraints: First $\alpha_{0,0} = a$, or put differently $f_a(0,0) = a$. Second the polynomial is *symmetric* i..e, $\alpha_{\sigma,\tau} = \alpha_{\tau,\sigma}$. The committer will then send a polynomial in one variable to each player $\mathsf{P}_k$, namely

$$f_k(\mathtt{X}) = f_a(\mathtt{X}, k) = \sum_{\sigma=0}^{t} (\sum_{\tau=0}^{t} \alpha_{\sigma,\tau} k^\tau) \ \mathtt{X}^\sigma,$$

where by sending a polynomial, we simply means sending its coefficients.

**Interpretation in terms of standard secret sharing.** Let us first explain how this connects to the more standard form of secret sharing we have seen before: consider the polynomial

$$g_a(\mathtt{X}) = f_a(\mathtt{X}, 0) = \sum_{\sigma=0}^{t} \alpha_{\sigma,0} \mathtt{X}^\sigma,$$

which is clearly a polynomial of degree at most $t$ such that $g_a(0) = a$. Furthermore, because $f_a$ is symmetric, we have $g_a(k) = f_a(k,0) = f_a(0,k) = f_k(0)$. In other words, by distributing the information we described, the committer has in particular secret shared $a$ in the standard way, using the polynomial $g_a(\mathtt{X})$, where the $k$'th share is $f_k(0)$.

Furthermore, the coefficients in $f_k(\mathtt{X})$ are $c_\sigma = \sum_{\tau=0}^{t} \alpha_{\sigma,\tau} k^\tau$, for $\sigma = 0, ..., t$. Note that each $c_\sigma$ can be interpreted as the value of a degree $t$ polynomial evaluated in point $k$, namely the polynomial with coefficients $\alpha_{\sigma,0}, ..., \alpha_{\sigma,t}$. So each $c_\sigma$ is in fact also a share of a secret, namely the degree-0 coefficient $\alpha_{\sigma,0}$. Note that $\alpha_{\sigma,0}$ is also a coefficient in $g_a(\mathtt{X})$.

This means that the process of choosing $f_a(\mathtt{X}, \mathtt{Y})$ and sending $f_k(\mathtt{X})$ to $\mathsf{P}_k$ can also be interpreted as follows: first secret share $a$ in the standard way using polynomial $g_a(\mathtt{X})$, and then secret share all the coefficients in $g_a(\mathtt{X})$, also in the standard way, and send shares to the respective players. To make this be exactly equivalent to what we described above, the polynomials for the last step must be chosen such that their coefficients satisfy the symmetry condition. One might think that this could hurt privacy, since then coefficients are not independent. But this is not the case, as we show below.

**Consistency Check** The key to checking consistency of the information $\mathsf{P}_i$ distributes is to observe that by symmetry of $f_a(\mathtt{X}, \mathtt{Y})$, we have for any two players $\mathsf{P}_k, \mathsf{P}_j$ that

$$f_k(j) = f_a(j,k) = f_a(k,j) = f_j(k).$$

The idea for the protocol is then as follows: after each player $\mathsf{P}_k$ has received $f_k(\mathtt{X})$, he will check with each other player $\mathsf{P}_j$ whether it is the case that $f_k(j) = f_j(k)$. If all pairwise checks go through for a set of at least $t + 1$ honest players, their information is sufficient to determine a polynomial of degree at most $t$ that all honest players eventually agree on. Of course the protocol has to take care of what should happen if the checks are not satisfied, here the idea is to ask $\mathsf{P}_i$ to help resolve any conflicts.

The complete protocol for implementing the `commit`-command from $\mathsf{F}_{\texttt{COM-SIMPLE}}$ is described in Protocol Commit. The implementation of the other commands is specified in Protocol Perfect-Com-Simple.

Before arguing the security of the protocols, we show two simple results:

Protocol Commit

1. On input $(\texttt{commit}, i, cid, a)$, $\mathsf{P}_i$ samples a bivariate symmetric polynomial $f_a(\texttt{X}, \texttt{Y})$ of degree at most $t$, such that $f_a(0,0) = a$. He sends the polynomial $f_k(\texttt{X}) = f_a(\texttt{X}, k)$ to each $\mathsf{P}_k$ (therefore $\mathsf{P}_k$ also learns $\beta_k = f_k(0)$).
2. Each $\mathsf{P}_j$ computes $\beta_{k,j} = f_j(k)$ and sends $\beta_{k,j}$ to each $\mathsf{P}_k$.
3. Each $\mathsf{P}_k$ checks that $\deg(f_k) \le t$ and that $\beta_{k,j} = f_k(j)$ for $j = 1, \ldots, n$. If so, he broadcasts `success`. Otherwise, he broadcasts $(\texttt{dispute}, k, j)$ for each inconsistency.
4. For each dispute reported in the previous step, $\mathsf{P}_i$ broadcasts the correct value of $\beta_{k,j}$.
5. If any $\mathsf{P}_k$ finds a disagreement between what $\mathsf{P}_i$ has broadcast and what he received privately from $\mathsf{P}_i$, he knows $\mathsf{P}_i$ is corrupt and broadcasts $(\texttt{accuse}, k)$.
6. For any accusation from $\mathsf{P}_k$ in the previous step, $\mathsf{P}_i$ broadcasts $f_k(\texttt{X})$.
7. If any $\mathsf{P}_k$ finds a new disagreement between what $\mathsf{P}_i$ has now broadcast and what he received privately from $\mathsf{P}_i$, he knows $\mathsf{P}_i$ is corrupt and broadcasts $(\texttt{accuse}, k)$.
8. If the information broadcast by $\mathsf{P}_i$ is not consistent, or if more than $t$ players have accused $\mathsf{P}_i$, players output `fail`.

   Otherwise, players who accused $\mathsf{P}_i$ and had a new polynomial $f_k(\texttt{X})$ broadcast will accept it as their polynomial. All others keep the polynomial they received in the first step. Now each $\mathsf{P}_k$ outputs `success` and stores $(cid, i, \beta_k)$. In addition $\mathsf{P}_i$ stores the polynomial $g_a(\texttt{X}) = f_a(\texttt{X}, 0)$.

**Lemma 5.4** *If $P_i$ remains honest throughout Protocol* Commit*, the view of any $t$ corrupted players is independent of the committed value $a$, and all players who are honest at the end of the protocol will output shares consistent with $a$ and the polynomial $g_a(X)$ that $P_i$ distributed.*

*Proof* We first argue that the information revealed in the first step is independent of $a$: by Lagrange interpolation, let $h(\texttt{X})$ be a polynomial of degree at most $t$, such that $h(0) = 1$ and $h(k) = 0$ for all corrupt $\mathsf{P}_k$. Then $h(\texttt{X})h(\texttt{Y})$ is a symmetric polynomial with value 1 in $(0,0)$. Suppose $\mathsf{P}_i$ used polynomial $f_a(\texttt{X}, \texttt{Y})$. Let $f_0(\texttt{X}, \texttt{Y}) = f_a(\texttt{X}, \texttt{Y}) - ah(\texttt{X})h(\texttt{Y})$. Clearly, $f_0(0,0) = 0$ and each corrupt $\mathsf{P}_k$ will receive the same information whether $f_a$ or $f_0$ is used. Moreover, adding $-ah(\texttt{X})h(\texttt{Y})$ is a bijection between symmetric polynomials consistent with $a$ and those consistent with 0. Since $\mathsf{P}_i$ chooses a uniform polynomial consistent with $a$, it follows that any $t$ corrupt players in Step 1 see the same distribution regardless of $a$, namely the one that follows from sharing 0.

In the following steps, one easily sees that the set of corrupt players is told nothing they did not already know: there can be no dispute unless at least one of $\mathsf{P}_j, \mathsf{P}_k$ is corrupt, so they already know all $\beta_{k,j}$ broadcast. Likewise, only a corrupt $\mathsf{P}_k$ will accuse an honest $\mathsf{P}_i$, so the corrupt players already know all polynomials and values broadcast. The fact that an honest player never accuses $\mathsf{P}_i$ also means that honest players will always output the shares they got in the first step. This implies the last conclusion of the lemma because $g_a(k) = f_a(k,0) = f_a(0,k) = f_k(0) = \beta_k$ and $g_a(0) = f_a(0,0) = a$. □

**Lemma 5.5** *If $t < n/3$, then no matter how corrupt players behave in Protocol* Commit*, players who are honest at the end of the protocol will all output `fail` or will output a set of shares in*

Protocol Perfect-Com-Simple

**commit:** See Protocol Commit.

**public commit:** On input $(\texttt{pubcommit}, i, cid, a)$ a party $\mathsf{P}_k$ lets $a_k = a$ (this is a share of the polynomial $a(\mathtt{X}) = a$), outputs $(\texttt{pubcommit}, i, cid, \texttt{success})$ and stores $(cid, i, a_k)$.

**open:** On input $(\texttt{open}, i, cid)$ where some $(cid, i, a(\mathtt{X}))$ is stored, the party $\mathsf{P}_i$ broadcasts $(cid, i, a(\mathtt{X}))$.

On input $(\texttt{open}, i, cid)$ where some $(cid, i, a_k)$ is stored a party $\mathsf{P}_k \neq \mathsf{P}_i$ broadcasts $(cid, i, a_k)$.

If $\deg(a(\mathtt{X})) \leq t$ and $a_k = a(k)$ for at least $n - t$ parties, then all parties output $(\texttt{open}, i, cid, a(0))$. Otherwise they output $(\texttt{open}, i, cid, \texttt{fail})$.

**designated open:** As above, but the polynomial and the shares are only sent to $\mathsf{P}_j$. If $\mathsf{P}_j$ rejects the opening, it broadcasts a public complaint, and $\mathsf{P}_i$ must then do a normal opening. If that one fails too, all parties output `fail`.

**add:** On input $(\texttt{add}, i, cid_1, cid_2, cid_3)$ where some $(i, cid_1, a(\mathtt{X}))$ and $(i, cid_2, b(\mathtt{X}))$ are stored the party $\mathsf{P}_i$ stores $(i, cid_3, a(\mathtt{X}) + b(\mathtt{X}))$ and outputs $(\texttt{add}, i, cid_1, cid_2, cid_3, \texttt{success})$.

On input $(\texttt{add}, i, cid_1, cid_2, cid_3)$ where some $(i, cid_1, a_k)$ and $(i, cid_2, b_k)$ are stored a party $\mathsf{P}_k \neq \mathsf{P}_i$ stores $(i, cid_3, a_k + b_k)$ and outputs $(\texttt{add}, i, cid_1, cid_2, cid_3, \texttt{success})$.

**multiplication by constant:** On input $(\texttt{mult}, i, \alpha, cid_2, cid_3)$ where some $(i, cid_2, b(\mathtt{X}))$ is stored the party $\mathsf{P}_i$ stores $(i, cid_3, \alpha b(\mathtt{X}))$ and outputs $(\texttt{mult}, i, \alpha, cid_2, cid_3, \texttt{success})$.

On input $(\texttt{mult}, i, \alpha, cid_2, cid_3)$ where some $(i, cid_2, b_k)$ is stored a party $\mathsf{P}_k \neq \mathsf{P}_i$ stores $(i, cid_3, \alpha b_k)$ and outputs $(\texttt{mult}, i, \alpha, cid_2, cid_3, \texttt{success})$.

*some value $a'$ all consistent with a polynomial $g_{a'}(\mathtt{X})$ of degree at most $t$.*

*Proof* The decision to output fail is based on public information, so it is clear that players who remain honest will either all output `fail` or all output some value, so assume the latter case occurs. This means that at least $n - t$ players did not accuse $\mathsf{P}_i$, and hence by assumption in the lemma at least $n - 2t \geq t + 1$ players who remained honest did not accuse $\mathsf{P}_i$. Let $S$ be the set of indices of such honest players, and consider an honest $\mathsf{P}_j$. We claim that for every $k \in S$ with $\mathsf{P}_k$ holding polynomial $f_k(\mathtt{X})$ we have that the $\beta_{k,j}$ that follows from the polynomial $f_j(\mathtt{X})$ held by $\mathsf{P}_j$ at the end satisfies

$$f_k(j) = \beta_{k,j}.$$

We argue this as follows: if $\mathsf{P}_j$ had a new polynomial broadcast for him in Step 6, then he keeps that one, and the claim is true, since otherwise $\mathsf{P}_k$ would have accused in Step 7. On the other hand, suppose no new values were broadcast for $\mathsf{P}_j$. Then he keeps the values he was given in the Step 1, as does $\mathsf{P}_k$, and we can also assume that neither $\mathsf{P}_j$ nor $\mathsf{P}_k$ accused in Step 5. But our claim holds for the values sent in Step 1, for if not, a $(\texttt{dispute}, k, j)$ would have been reported, and either $\mathsf{P}_j$ or $\mathsf{P}_k$ would have accused $\mathsf{P}_i$, and by assumption this did not happen.

Now, let $(r_k)_{k \in S}$ be the reconstruction vector for reconstructing a secret from shares of players in $S$. That is, for any polynomial $f$ of degree at most $t$ we have $f(0) = \sum_{k \in S} r_k f(k)$, such a vector exists because $|S| \geq t + 1$.

Now define the polynomial $g_{a'}(\mathtt{X})$ as

$$g_{a'}(\mathtt{X}) = \sum_{k \in S} r_k f_k(\mathtt{X})$$

and set $a' = g_{a'}(0)$. Since an honest $\mathsf{P}_j$ outputs $\beta_j$, we see that in order to show the lemma, we need to show that $g_{a'}(j) = \beta_j$ for all honest $\mathsf{P}_j$. Recall that $\mathsf{P}_j$ computes $\beta_{k,j}$ as $\beta_{k,j} = f_j(k)$. Now, by the claim above we can compute as follows:

$$g_{a'}(j) = \sum_{k \in S} r_k f_k(j) = \sum_{k \in S} r_k \beta_{k,j} = \sum_{k \in S} r_k f_j(k) = f_j(0) = \beta_j$$

$\square$

From the above lemmas, we get the following theorem:

**Theorem 5.6** *$\pi_{\mathtt{PERFECT\text{-}COM\text{-}SIMPLE}} \diamond \mathsf{F}_{\mathtt{SC}}$ implements $\mathsf{F}_{\mathtt{COM\text{-}SIMPLE}}$ in $\mathrm{Env}^{t,sync}$ with perfect security for all $t < n/3$.*

*Proof* A simulator for the case of static corruption is given as $\mathcal{S}_{\mathtt{PERFECT\text{-}COM\text{-}SIMPLE}}$. We argue that this is a perfect simulator: all `commit` commands are simulated perfectly, this follows from Lemma 5.4 if the committer is honest and from that fact the we just follow the protocol if the committer is corrupt. Furthermore, each `open` command will reveal the shares the environment already knew in case the committer is corrupt (note that the existence of these shares for all honest players is guaranteed by Lemma 5.5). If the committer is honest, it will be a set of shares consistent with what the corrupt players have and the opened value. This holds in both real protocol and simulation.

It follows that the view of corrupt players is identically distributed in real protocol and in execution, so the only way the environment can distinguish is if the input/output behavior of honest players is different in simulation and in the real execution. This cannot happen for an honest committer, since the simulator gets the correct values to open and the open protocol always succeeds for an honest committer. It cannot happen for a corrupt committer either: if the open protocol fails, the simulator makes the functionality fail as well, and if it succeeds, the opened value must be the one that follows from earlier committed values and therefore equals the one output by the functionality. This is because Lemma 5.5 guarantees that all honest players have consistent shares of each committed value, therefore by our discussion on minimal distance decoding, an incorrect polynomial will disagree with at least $t+1$ honest players and the opening will fail.

**Adaptive Corruption.** To simulate for adaptive corruption, we start $\mathcal{S}_{\mathtt{PERFECT\text{-}COM\text{-}SIMPLE}}$ in the state where all players are honest. Recall that $\mathcal{S}_{\mathtt{PERFECT\text{-}COM\text{-}SIMPLE}}$ works with virtual players $\bar{\mathsf{P}}_i$. When we get to a point where a new player $\mathsf{P}$ is corrupted, we corrupt $\mathsf{P}$ on $\mathsf{F}_{\mathtt{COM\text{-}SIMPLE}}$ and get the set of values he committed to from its leakage port. Using this data we then adjust (if needed) the views of the virtual players so they are consistent with the new values we learned and everything the environment has seen so far. We give the view of $\bar{\mathsf{P}}$ to the environment, and continue the simulation following the algorithm of $\mathcal{S}_{\mathtt{PERFECT\text{-}COM\text{-}SIMPLE}}$, giving it the adjusted state of the virtual players (except for that of $\mathsf{P}$ who is now corrupt).

We will see that the view of $\mathsf{P}$ that we give to the environment always has exactly the correct distribution, and furthermore the state of virtual players is distributed exactly as if we had run $\mathcal{S}_{\mathtt{PERFECT\text{-}COM\text{-}SIMPLE}}$ with $\mathsf{P}$ being corrupt from the start, but where the environment let it behave honestly up to this point. Therefore it follows from the result for static corruption, that the simulation after the corruption is perfect.

We now describe the procedure for adjusting views: we will be given every value committed by $\mathsf{P}$. For every such value $a$, the state of $\mathcal{S}_{\mathtt{PERFECT\text{-}COM\text{-}SIMPLE}}$ contains a polynomial $f_{\bar{a}}(\mathtt{X},\mathtt{Y})$ that is part of its state for the virtual player $\bar{\mathsf{P}}$. We now replace $f_{\bar{a}}(\mathtt{X},\mathtt{Y})$ by $f_{\bar{a}}(\mathtt{X},\mathtt{Y}) + (a - \bar{a})h(\mathtt{X})h(\mathtt{Y})$, where

Agent $\mathcal{S}_{\texttt{PERFECT-COM-SIMPLE}}$

Simulator for $\pi_{\texttt{PERFECT-COM-SIMPLE}}$ for the case of static corruption.

**Intialize.** In the preamble, the environment $\mathcal{Z}$ specifies the set $C$ of actively corrupted players. Then the simulator sets up an internal copy of $\mathsf{F}_{\texttt{SC}}$, where it corrupts players in $C$ and connects the special ports of $\mathsf{F}_{\texttt{SC}}$ to $\mathcal{Z}$. It also corrupts $C$ on $\mathsf{F}_{\texttt{COM-SIMPLE}}$. Finally, it sets up a copy of every honest player. These "virtual" players are called $\bar{\mathsf{P}}_i, \bar{\mathsf{P}}_k$ and $\bar{\mathsf{P}}_j$; their internal variables will be called $\bar{a}, \bar{\beta}_k$ etc. The basic idea is to let the virtual players execute the protocol with the corrupt players (controlled by $\mathcal{Z}$), letting the virtual players commit to dummy values. Below, we describe in more detail how this is done for each command.

For every committed value $a$ (dummy or not), the simulator stores a polynomial $f_a(\mathtt{X}, \mathtt{Y})$ determining the complete set of shares of $a$ held by the (corrupt and virtual) players.

For every honest player $\mathsf{P}_i$ we maintain an invariant: Note that every value $b$ that $\mathsf{P}_i$ opened via an `open` command is a linear combination of all his committed values so far $a_1, ..., a_T$: $b = \gamma_1 a_1 + ... + \gamma_T a_T$, where the $\gamma_i$'s are public as they follow from the executed sequence of `add` and `mult` commands. The invariant now is: the set of "committed" dummy values $\{\bar{a}_j\}$ held by $\bar{\mathsf{P}}_i$ is a random set of values consistent with every $b$ that has been opened, i.e., it holds that $b = \gamma_1 \bar{a}_1 + ... + \gamma_T \bar{a}_T$, and furthermore each polynomial $f_{\bar{a}_j}$ held by the virtual player is a random symmetric polynomial of degree at most $t$ that is consistent with $\bar{a}_j$ and the shares held by corrupt players.

`commit` If $\mathsf{P}_i$ is honest, execute the `commit` protocol giving a random $\bar{a}$ as input to $\bar{\mathsf{P}}_i$. This maintains the invariant as no opened value depends on value we commit now. The simulator stores the polynomial $f_{\bar{a}}$ used by $\bar{\mathsf{P}}_i$. If $\mathsf{P}_i$ is corrupt, we execute the `commit` protocol, if it fails, we make $\mathsf{F}_{\texttt{COM-SIMPLE}}$ fail as well, otherwise we use Lemma 5.5 to compute the committed value $a'$ and give this as input to $\mathsf{F}_{\texttt{COM-SIMPLE}}$. The simulator also stores the polynomial $g_{a'}(\mathtt{X})$ guaranteed by the lemma.

`add`, `mult` There is no communication to simulate, we just let the virtual players do the local computation prescribed in the protocol.

`open` If the committer is honest, we get the value $z$ to be revealed from leakage of $\mathsf{F}_{\texttt{COM-SIMPLE}}$. We now adjust the views of the virtual players so that it is consistent with the value $z$ and hence the invariant is maintained. We do this by considering all equations of form $b = \gamma_1 a_1 + ... + \gamma_T a_T$ for revealed values $b$ (including the new equation for $z$), and set $(\bar{a}_1, ..., \bar{a}_t)$ to be a random solution to this system of equations. Note that a solution must exist, namely $(\bar{a}_1, ..., \bar{a}_t) = (a_1, ..., a_t)$, the set of values actually input by $\mathcal{Z}$. The set of committed values held by the virtual player is set to be the new solution we found, and we adjust each polynomial $f_{\bar{a}_j}(\mathtt{X}, \mathtt{Y})$ so that the invariant is maintained. This is done by computing the new polynomial as $f_{\bar{a}_{j,new}}(\mathtt{X}, \mathtt{Y}) = f_{\bar{a}_{j,old}}(\mathtt{X}, \mathtt{Y}) + (\bar{a}_{j,new} - \bar{a}_{j,old})h(\mathtt{X})h(\mathtt{Y})$, where $h$ is a degree at most $t$ polynomial that is 1 in 0 and 0 in points of corrupt players. This always works as there are at most $t$ corrupt players. Now all virtual players are still in a consistent state, and we can execute the `open` protocol.

If the committer is corrupt, we simply execute the `open` protocol, if it fails, we make $\mathsf{F}_{\texttt{COM-SIMPLE}}$ fail as well.

$h()$ is a polynomial that is 1 in 0 and 0 in all points of players that were corrupted earlier, and we adjust the view of all virtual players accordingly. By the invariant of $\mathcal{S}_{\texttt{PERFECT-COM-SIMPLE}}$, $f_{\bar{a}}(\texttt{X}, \texttt{Y})$ is a random polynomial consistent with $\bar{a}$ and the views of corrupt players. The adjustment we do will maintain this, but now with $\bar{a}$ replaced by $a$. □

Agent $\mathrm{F}_{\texttt{ICSIG}}$

Ideal functionality for IC signatures.Each instance of this functionality provides service to two special players $\mathsf{P}_D$ and $\mathsf{P}_{INT}$. If both are corrupt, the functionality does a complete breakdown.

**sign:** This command is executed if in some round player $\mathsf{P}_D$ sends $(\texttt{sign}, sid, a)$ and in addition all honest players send $(\texttt{sign}, i, sid, ?)$. In this case $\mathsf{F}_{\texttt{ICSIG}}$ records the triple $(sid, a)$. Here, $sid$ is just an identifier, and $a$ is the value that is "signed".[a] The output to all parties is $(\texttt{sign}, sid, \texttt{success})$, in addition $a$ is sent to $\mathsf{P}_{INT}$.

**reveal:** This command is executed if in some round all honest players send $(\texttt{reveal}, sid)$ and some $(sid, a)$ is stored. The output to all parties is $(\texttt{reveal}, sid, a)$.

If $\mathsf{P}_{INT}$ is corrupted and does not input $(\texttt{reveal}, sid)$, then $(\texttt{reveal}, sid, \texttt{fail})$ is output to all parties.

**designated reveal:** This command is executed if in some round all honest players send $(\texttt{reveal}, j, sid)$ and some $(sid, a)$ is stored. The command behaves exactly as `reveal`, except that only $\mathsf{P}_j$ gets output.

**add:** This command is executed if in some round all honest players send $(\texttt{add}, sid_1, sid_2, sid_3)$ and some $(sid_1, a_1)$ is stored and some $(sid_2, a_2)$ is stored. As a result $\mathsf{F}_{\texttt{ICSIG}}$ stores $(sid_3, a_1 + a_2)$. The output to all parties is $(\texttt{add}, sid_1, sid_2, sid_3, \texttt{success})$.

**mult by constant:** This command is executed if in some round all honest players send $(\texttt{mult}, c, sid_2, sid_3)$ and $c \in \mathbb{F}$ and some $(i, sid_2, a_2)$ is stored. As a result $\mathsf{F}_{\texttt{ICSIG}}$ stores $(i, sid_3, ca_2)$. The output to all parties is $(\texttt{mult}, c, sid_2, sid_3, \texttt{success})$.

[a]We require that all honest players agree to the fact that a value should be signed because an implementation will require the active participation of all honest players.

### 5.4.1 Commitments in Case of Honest Majority

In this section, we present a protocol that implements $\mathsf{F}_{\texttt{COM-SIMPLE}}$ with statistical security for the case of honest majority, i.e., when $t < n/2$. We will use the same idea as before of committing by secret sharing the value to commit to. But the problem we face is that even if we force the committer to give us consistent shares, the resulting commitment will not be binding.

The problem is that there may only be $t + 1$ honest players. Even if they all have consistent shares, the adversary can easily cook up a different polynomial that is consistent with $t$ of these players, and any value of the secret he desires. If he would open such a value using the `open` protocol we have seen, only one honest player would disagree, and this may as well be a corrupt player who is complaining for no good reason.

The problem could be solved if the unhappy honest player could *prove* that his inconsistent share was in fact what he received from the committer. This would require something similar to a signature scheme.

Protocol ICSign

**initialize:** This step is carried out before any commands are executed. For $i = 1...n$, $\mathsf{P}_D$ chooses $\alpha_i \in \mathbb{F}$ a random and sends $\alpha_i$ to $\mathsf{P}_i$, both players store the value.

**sign:** On input $(\texttt{sign}, sid, a)$ to $\mathsf{P}_D$ and $(\texttt{sign}, sid, ?)$ to other players, do the following:

For $i = 1...n$, $\mathsf{P}_D$ chooses $\beta_{i,a} \in \mathbb{F}$ at random and sets $K_{i,a} = (\alpha_i, \beta_{i,a})$. He sends $\beta_{i,a}$ to $\mathsf{P}_i$ and $a, \{m_{i,a} = MAC_{K_{i,a}}(a)\}_{i=1..n}$ to $\mathsf{P}_{INT}$.

We now check that $\mathsf{P}_D$ has sent correctly formed data. First, $\mathsf{P}_D$ chooses $a' \in \mathbb{F}$ at random and sends $a', \{m_{i,a'} = MAC_{K_{i,a'}}(a')\}_{i=1..n}$ to $\mathsf{P}_{INT}$. $\mathsf{P}_{INT}$ chooses $e \in \mathbb{F}$ at random and broadcasts $e, a' + ea$.

Then do the following loop for $i = 1...n$:

1. $\mathsf{P}_D$ chooses $\beta_{i,a'} \in \mathbb{F}$ at random, sets $K_{i,a'} = (\alpha_i, \beta_{i,a'})$ and sends $\beta_{i,a'}$ to $\mathsf{P}_i$.
2. $\mathsf{P}_{INT}$ broadcasts $m_{i,a'} + em_{i,a} = MAC_{K_{i,a'}+eK_{i,a}}(a' + ea)$.
3. $\mathsf{P}_D$ checks the broadcast message from $\mathsf{P}_{INT}$. If it is correct, he broadcasts `accept`, else he broadcasts "$\mathsf{P}_{INT}$ is corrupt" and we exit the loop.
4. $\mathsf{P}_i$ checks that $m_{i,a'}+em_{i,a} = MAC_{K_{i,a'}+eK_{i,a}}(a'+ea)$, and broadcasts `accept` or `reject` accordingly. $\mathsf{P}_D$ checks that $\mathsf{P}_i$ acted correctly, if so he broadcasts `accept` else he broadcasts "$\mathsf{P}_i$ is corrupt: $K_{i,a}$". This broadcasted key value will be used in the following and $\mathsf{P}_{INT}$ adjusts his values so it holds that $m_{i,a} = MAC_{K_{i,a}}(a)$. If $\mathsf{P}_i$ said `reject` and $\mathsf{P}_D$ said `accept`, we exit the loop (this can only happen if $\mathsf{P}_D$ is corrupt).

If the above loop was terminated by an exit, $\mathsf{P}_D$ broadcasts $a, \{m_{i,a}\}_{i=1..n}$. These values will be used in the following and each $\mathsf{P}_i$ adjusts his $\beta_{i,a}$-value so it holds that $m_{i,a} = MAC_{K_{i,a}}(a)$.

$\mathsf{P}_{INT}$ outputs $(\texttt{sign}, sid, a, \texttt{success})$ where he uses the last value for $a$ he received from $\mathsf{P}_D$, and stores $a, \{m_{i,a}\}_{i=1..n}$ under $sid$. Each $\mathsf{P}_i$ outputs $(\texttt{sign}, sid, \texttt{success})$ and stores $K_{i,a}$ under $sid$.

### IC Signatures

We therefore begin with an important tool, known as Information Checking or IC-signatures. The idea is to provide a functionality similar to digital signatures, but with information theoretic security. The basic version of this idea involves two players that we call $\mathsf{P}_D, \mathsf{P}_{INT}$ for Dealer and Intermediary. The basic idea is that $\mathsf{P}_D$ can give a secret message $s$ to $\mathsf{P}_{INT}$, and at any later time $\mathsf{P}_{INT}$ can choose to reveal $s$ and prove to the other players that $s$ really was the value he got from $\mathsf{P}_D$. We specify what we want more formally as a functionality $\mathsf{F}_{\texttt{ICSIG}}$. The main properties we get are that if both $\mathsf{P}_D, \mathsf{P}_{INT}$ are honest, then $s$ will always be successfully revealed, but no corrupt player knows $s$ before it is revealed. If $\mathsf{P}_{INT}$ is corrupt (but $\mathsf{P}_D$ is honest), he can refuse to reveal $s$ but cannot reveal a wrong value successfully. Finally, if $\mathsf{P}_D$ is corrupt (but $\mathsf{P}_{INT}$ is honest), it will still be possible for $\mathsf{P}_{INT}$ to convince the other players about the correct value of $s$.

In addition we want that linear operations can be done on signed values, so that if $\mathsf{P}_D$ has given $a, b \in \mathbb{F}$ to $\mathsf{P}_{INT}$, then $\mathsf{P}_{INT}$ can later convince everyone about the correct value of $ca + b$ for a publicly known constant $c$.

The protocol for implementing $\mathsf{F}_{\texttt{ICSIG}}$ is based on the well-known notion of unconditionally secure message authentication codes (MACs). Such a MAC uses a key $K$, a random pair $K = (\alpha, \beta) \in \mathbb{F}^2$, and the authentication code for a value $a \in \mathbb{F}$ is $\texttt{MAC}_K(a) = \alpha a + \beta$.

Protocol ICSig (Continued)

**(designated) reveal:** On input $(\texttt{reveal}, sid)$ to all players, $\mathsf{P}_{INT}$ broadcasts $a, \{m_{i,a}\}_{i=1...n}$ as stored under *sid*. Each $\mathsf{P}_i$ retrieves $K_{i.a}$ stored under *sid* and checks that $m_{i,a} = MAC_{K_{i,a}}(a)$. If this holds, he outputs $(\texttt{reveal}, sid, a)$ else he output $(\texttt{reveal}, sid, \texttt{fail})$.

If the command was designated, $(\texttt{reveal}, j, sid)$, $\mathsf{P}_{INT}$ only sends $a, m_{j,a}$ to $\mathsf{P}_j$ who checks the MAC as in the normal `reveal` command.

**add:** On input $(\texttt{add}, sid_1, sid_2, sid_3)$ to all players, $\mathsf{P}_{INT}$ retrieves the values $(a_1, m_{1,a_1}, ..., m_{n,a_1}), (a_2, m_{1,a_2}, ..., m_{n,a_2})$, that were stored under $sid_1, sid_2$ and stores $(a_1 + a_2, m_{1,a_1} + m_{1,a_2}, ..., m_{n,a_1} + m_{n,a_2})$ under $sid_3$.

Each $\mathsf{P}_i$ retrieves the values $\beta_{i,a_1}, \beta_{i,a_2}$ stored under $sid_1, sid_2$ and stores $\beta_{i,a_1} + \beta_{i,a_2}$ under $sid_3$.

All players output $(\texttt{add}, sid_1, sid_3, sid_3, \texttt{success})$.

**multiplication by constant:** On input $(\texttt{mult}, c, sid_2, sid_3)$ to all players, $\mathsf{P}_{INT}$ retrieves the values $(a_2, m_{1,a_2}, ..., m_{n,a_2})$, that were stored under $sid_2$ and stores $(c \cdot a_2, c \cdot m_{1,a_2}, ..., c \cdot m_{n,a_2})$ under $sid_3$.

Each $\mathsf{P}_i$ retrieves the value $\beta_{i,a_2}$ stored under $sid_2$ and stores $c \cdot \beta_{i,a_2}$ under $sid_3$.

All players output $(\texttt{mult}, c, sid_2, sid_3, \texttt{success})$.

We will apply the MACs by having $P_D$ give a key $K_{i,a} = (\alpha_i, \beta_{i,a})$ to each $\mathsf{P}_i$ and a set of values $a, \mathtt{MAC}_{K_{i,a}}(a), i = 1..n$ to $\mathsf{P}_{INT}$. The idea is to use the MACs to prevent $\mathsf{P}_{INT}$ from lying about $a$ when he broadcasts it. Given $a$ and $K_{i,a}$, each $\mathsf{P}_i$ can compute the MAC value he expects to see and compare to the MAC sent by $\mathsf{P}_{INT}$.

It will be very important in the following that if we keep $\alpha$ constant over several different MAC keys, then one can add two MACs and get a valid authentication code for the sum of the two corresponding messages. More concretely, two keys $K = (\alpha, \beta), K' = (\alpha', \beta')$ are said to be *consistent* if $\alpha = \alpha'$. For consistent keys, we define $K + K' = (\alpha, \beta + \beta')$, it is then trivial to verify that $\mathtt{MAC}_K(a) + \mathtt{MAC}_{K'}(a') = \mathtt{MAC}_{K+K'}(a + a')$.

If $\mathsf{P}_{INT}$ is corrupt (but $\mathsf{P}_D$ is honest) it is also easy to see (as detailed in the proof below) that $\mathsf{P}_{INT}$ will be able to lie about the values he was given (or about any linear combination of them) with probability at most $1/|\mathbb{F}|$. Also, a key clearly reveals no information on the authenticated value. On the other hand, if If $\mathsf{P}_D$ is corrupt (but $\mathsf{P}_{INT}$ is honest), he may not give consistent information to players, so the protocol must do an interactive verification when a value is signed, in order to check this.

**Theorem 5.7** *$\pi_{\mathtt{ICSIG}} \diamond \mathsf{F}_{\mathtt{SC}}$ implements $\mathsf{F}_{\mathtt{ICSIG}}$ in* $\mathrm{Env}^{t,sync}$ *with statistical security for all $t < n$.*

*Proof* As usual, we construct a simulator $\mathcal{S}_{\mathtt{ICSIG}}$ for the protocol. The simulator shown deals with static corruption, and considers separately the three cases where: both $\mathsf{P}_D, \mathsf{P}_{INT}$ are honest, $\mathsf{P}_D$ is corrupt and $\mathsf{P}_{INT}$ is corrupt. At the same time any number of other players may be corrupt. If $\mathsf{P}_D$ and $\mathsf{P}_{INT}$ are both corrupt, there is nothing to show as the functionality does a complete breakdown in that case. We show below how to deal with adaptive corruption, but first argue why the simulation is statistically close to a real execution:

**Both $\mathsf{P}_D, \mathsf{P}_{INT}$ are honest.** In this case, $\mathcal{S}_{\mathtt{ICSIG}}$ runs the protocol on behalf of the honest players using random dummy inputs. Note that this also involves generating keys to be held by each player $\mathsf{P}_i$ (whether he is corrupt or not). Now, by the random choice of $a'$, the view of the

Agent $\mathcal{S}_{\texttt{ICSIG}}$

Simulator for $\pi_{\texttt{ICSIG}}$ for the case of static corruption.

**intialize** In the preamble, the environment $\mathcal{Z}$ specifies the set $C$ of actively corrupted players. Then the simulator sets up an internal copy of $\mathsf{F}_{\texttt{SC}}$, where it corrupts players in $C$ and connects the special ports of $\mathsf{F}_{\texttt{SC}}$ to $\mathcal{Z}$. It also corrupts $C$ on $\mathsf{F}_{\texttt{ICSIG}}$. Finally, it sets up a copy of every honest player. These "virtual" players are called $\bar{\mathsf{P}}_i, \bar{\mathsf{P}}_D$ and $\bar{\mathsf{P}}_{INT}$; their internal variables will be called $\bar{a}, \bar{a'}$ etc. The basic idea is to let the virtual players execute the protocol with the corrupt players (controlled by $\mathcal{Z}$). Below, we describe in more detail how this is done for each command.

We consider three "modes": 1) both $\mathsf{P}_D, \mathsf{P}_{INT}$ are honest, 2) $\mathsf{P}_D$ is corrupt, and 3) $\mathsf{P}_{INT}$ is corrupt. If both $\mathsf{P}_D, \mathsf{P}_{INT}$ are corrupt, simulation becomes trivial as $\mathsf{F}_{\texttt{ICSIG}}$ does a complete breakdown in that case.

In mode 1), we maintain an invariant: Note that every value $b$ that is revealed in a `reveal` command is a linear combination of all signed values so far $a_1, ..., a_t$: $b = \gamma_1 a_1 + ... + \gamma_t a_t$, where the $\gamma_i$'s are public as they follow from the executed sequence of `add` and `mult` commands. The invariant now is: the set of "signed" values $\{\bar{a}_j\}$ held by $\bar{\mathsf{P}}_D$ is a random set of values consistent with every $b$ that has been revealed, i.e., it holds that $b = \gamma_1 \bar{a}_1 + ... + \gamma_t \bar{a}_t$.

`sign` In mode 1), execute the `sign` protocol giving a random $\bar{a}$ as input to $\bar{\mathsf{P}}_D$. This maintains the invariant as no revealed value depends on the input we sign now. In mode 2), we get a value $a$ that the corrupt $\mathsf{P}_D$ sends via $\mathsf{F}_{\texttt{SC}}$. We give $a$ as input to $\mathsf{F}_{\texttt{ICSIG}}$ on behalf of $\mathsf{P}_D$ and then execute the `sign` protocol. In mode 3), we get a value $a$ from leakage of $\mathsf{F}_{\texttt{ICSIG}}$ (since $\mathsf{P}_{INT}$ is corrupt) and we execute the `sign` protocol giving $a$ as input to $\bar{\mathsf{P}}_D$.

`add`, `mult` There is no communication to simulate, we just let the virtual players do the local computation prescribed in the protocol.

**(designated)** )`reveal` For a public `reveal` in mode 1), we get the value $z$ to be revealed from leakage of $\mathsf{F}_{\texttt{ICSIG}}$. We now adjust the views of the virtual players so that it is consistent with the value $z$ and hence the invariant is maintained. We do this by considering all equations of form $b = \gamma_1 a_1 + ... + \gamma_t a_t$ for revealed values $b$ (including $z$), and set $(\bar{a}_1, ..., \bar{a}_t)$ to be a random solution to this system of equations. Note that a solution must exist, namely $(\bar{a}_1, ..., \bar{a}_t) = (a_1, ..., a_t)$ the set of values actually input by $\mathcal{Z}$. If this requires a previous value of $\bar{a}_j$ to be changed, we compensate for this by adjusting the value of $\bar{a}'_j$ (and hence the MAC on $\bar{a}'_j$) such that all broadcasted values remain the same. Now all virtual players are still in a consistent state, and we can execute the `reveal` command. The same is done for a designated reveal where the receiver is corrupt. If the receiver is honest, the corrupt players see no communication, the simulator just executes the designated `reveal` internally to keep a consistent state.

In modes 2) and 3), we have executed the protocol with the correct signed values and we simply therefore simply execute the `reveal` or designated `reveal` protocol.

`sign` protocol is independent of the signed values, so the simulation of this part is perfect. So is the simulation of the `add` and `mult` commands since no communication is done. Furthermore, when a value $b$ is opened, the simulator adjusts the views of its virtual players so they are consistent with $b$. This includes computing from $K_{i,b}$ a MAC on $b$ to show to each $\mathsf{P}_i$. Since the key held by $\mathsf{P}_i$ has the correct distribution and the MAC follows deterministically from $a$ and $K_{i,b}$, the joint distribution of everything seen by $\mathcal{Z}$ in the `reveal` command is correct.

**$\mathsf{P}_{INT}$ is corrupt.** In this case, $\mathcal{S}_{\texttt{ICSIG}}$ knows each $a$ to be signed from the start and now simulates by simply following the protocol. Therefore the only way in which the simulation can deviate from the real execution comes from the fact that in the simulation, when a value is revealed, honest players get the value from the functionality, which by definition always reveals the value sent by $\mathsf{P}_D$, while this may not be the case in the real protocol. It *is* the case except with negligible probability, however: if $\mathsf{P}_{INT}$ is about to reveal a value $a$ to honest player $\mathsf{P}_i$, he knows the correct MAC $m_{i,a} = \alpha_i a + \beta_{i,a}$. This reveals no information on $\alpha_i$ by random choice of $\beta_{i,a}$, and neither does any other MACs known to $\mathsf{P}_{INT}$ by the same argument. Now, if $\mathsf{P}_{INT}$ produces $a' \neq a$ and $m_{i,a'}$ that $\mathsf{P}_i$ would accept, we know that $m_{i,a} = \alpha_i a + \beta_{i,a}$ and $m_{i,a'} = \alpha_i a + \beta_{i,a}$. Subtracting one equation from the other we obtain $\alpha_i = (m_{i,a} - m_{i,a'})/(a - a')$, which makes sense because $a - a' \neq 0$. In other words $\mathsf{P}_{INT}$ must guess $\alpha_i$ to cheat and this can done with probability at most $1/|\mathbb{F}|$. If a MAC is rejected and the protocol continues, $\mathsf{P}_{INT}$ knows that a certain value is not the correct $\alpha_i$. But in a polynomial time protocol, he will only be able to exclude a polynomial number of values, so the probability to cheat remains negligible. The simulation is therefore statistically close in this case.

**$\mathsf{P}_D$ is corrupt.** In this case, $\mathcal{S}_{\texttt{ICSIG}}$ knows each $a$ to be signed from the start and simulates by following the protocol. This means that the only way the simulation can deviate from the real protocol comes from the fact that in the simulation, the functionality on a `reveal` command always reveals the value it got from $\mathsf{P}_D$ to the honest players. In the protocol it may happen that the honest $\mathsf{P}_{INT}$ attempts to reveal the correct value but has it rejected by an honest $\mathsf{P}_i$, however, this only happens with negligible probability: in the cases where in the `sign` phase, $\mathsf{P}_D$ broadcasts $a$ and a set of MACs, or broadcasts the key for some $\mathsf{P}_i$, the problem never occurs because all honest $\mathsf{P}_i$ are guaranteed to have data that are consistent with $\mathsf{P}_{INT}$. Therefore, the only remaining case is where $\mathsf{P}_i$ says `accept` in the sign phase and $\mathsf{P}_D$ does not claim that $\mathsf{P}_i$ or $\mathsf{P}_{INT}$ are corrupt. Now, $\mathsf{P}_i$ only accepts if it holds that

$$m_{i,a'} + e m_{i,a} = MAC_{K_{i,a'} + eK_{i,a}}(a' + ea) = \alpha_i(a' + ea) + \beta_{i,a'} + e\beta_{i,a}.$$

If $\mathsf{P}_D$ gave an incorrect MAC to $\mathsf{P}_{INT}$ in the first place then $m_{i,a} \neq \alpha_i a + \beta_{i,a}$. This would mean that we could rearrange the above equation as follows

$$e = (\alpha_i a' + \beta_{i,a'} - m_{i,a'})/(m_{i,a} - \alpha_i a - \beta_{i,a})$$

and compute $e$ from the values generated by $\mathsf{P}_D$ before $e$ is chosen, in other words $\mathsf{P}_D$ must guess $e$ to successfully distribute inconsistent data, so this happen with only negligible probability $1/|\mathbb{F}|$, and the simulation is statistically close also in this case.

**Adaptive Corruption.** To construct a simulator for adaptive corruption, we start $\mathcal{S}_{\texttt{ICSIG}}$ in the mode where all players are honest. Recall that $\mathcal{S}_{\texttt{ICSIG}}$ works with virtual players $\bar{\mathsf{P}}_i, \bar{\mathsf{P}}_D, \bar{\mathsf{P}}_{INT}$. When a new player $\mathsf{P}$ is corrupted, we corrupt $\mathsf{P}$ on $\mathsf{F}_{\texttt{ICSIG}}$ and get some data from the leakage port of $\mathsf{F}_{\texttt{ICSIG}}$. Using this data we then adjust (if needed) the views of the virtual players so they are consistent with everything the environment has seen so far, we give the view of $\bar{\mathsf{P}}$ to the environment, and continue the simulation following the algorithm of $\mathcal{S}_{\texttt{ICSIG}}$. If $\mathsf{P} = \mathsf{P}_D$ or $\mathsf{P}_{INT}$, we switch to the simulation mode where $\mathsf{P}_D$ or $\mathsf{P}_{INT}$ is corrupt. We will see that the view of $\mathsf{P}$ that we give to the environment always has exactly the correct distribution. Therefore we can argue, as for $\mathcal{S}_{\texttt{ICSIG}}$, that the only case where simulation and real execution differ is if a corrupt $\mathsf{P}_D$ or $\mathsf{P}_{INT}$ succeeds in cheating which happens with negligible probability as we saw above.

We now describe the procedure for adjusting views: if $\mathsf{P}$ is not $\mathsf{P}_D$ or $\mathsf{P}_{INT}$, we use the view of $\bar{\mathsf{P}}$ with no change. This works because the simulator gets no new input/output information when such a player is corrupted, and the view of such a player consists of keys (that we chose with the right distribution) and possibly some revealed values and MACs that were already broadcast.

If if $\mathsf{P}$ is not $\mathsf{P}_D$ or $\mathsf{P}_{INT}$, we are given all values that were privately revealed to this player in designated `reveal`. If $\mathsf{P} = \mathsf{P}_D$ or $\mathsf{P}_{INT}$, we are given every signed value. In any case, we then run the procedure used by $\mathcal{S}_{\texttt{ICSIG}}$ during the simulation of `reveal`, to adjust the views of $\bar{\mathsf{P}}_D$ and $\bar{\mathsf{P}}_{INT}$ to be consistent with each value we got from $\mathsf{F}_{\texttt{ICSIG}}$ (by doing the same as if each value was revealed). This gives a correctly distributed history for every value we consider: given the correct set of $a$-values to sign (and random $e$-values), one can either choose the $a'$-values at random and compute the broadcasted values from this (this is what the real protocol does) – or one can choose the broadcasted values with the correct distribution and compute matching $a'$-values. This is what the simulation does. Both methods clearly lead to the same distribution. Note that all keys are chosen independently and the MAC's follow deterministically from keys and authenticated values. Keys and MACs therefore make no difference to this argument.

Finally, if both $\mathsf{P}_D$ and $\mathsf{P}_{INT}$ are corrupted, we do as just explained, give the views of $\bar{\mathsf{P}}_D$ and $\bar{\mathsf{P}}_{INT}$ to $\mathcal{Z}$ and let $\mathcal{Z}$ decide what happens hereafter (since $\mathsf{F}_{\texttt{ICSIG}}$ now breaks down). □

Protocol Statistical-Commit

1. On input $(\texttt{commit}, i, cid, a)$, $\mathsf{P}_i$ samples polynomials $g_a(\texttt{X}) = a + \alpha_1\texttt{X} + ... + \alpha_t\texttt{X}^t$ and $f_{\alpha_\tau}(\texttt{X})$, as described in the text.

   For $j = 1...n$ and $\tau = 1...t$ he sends $\alpha_{\tau,j} = f_{\alpha_\tau}(j)$ to $\mathsf{P}_j$. Players call `sign` on $\mathsf{F}_{\texttt{ICSIG}}$ with $\mathsf{P}_D = \mathsf{P}_i, \mathsf{P}_{INT} = \mathsf{P}_j$ and $\alpha_{\tau,j}$ being the value to sign.

   Finally, for $k = 1...n$, he sends the polynomial $f_{\beta_k}(\texttt{X}) = g_a(\texttt{X}) + \sum_{\tau=1}^{t} f_{\alpha_\tau}(\texttt{X})k^\tau$ to $\mathsf{P}_k$ (therefore $\mathsf{P}_k$ also learns $\beta_k = f_{\beta_k}(0)$). Players call `sign` on $\mathsf{F}_{\texttt{ICSIG}}$ $t+1$ times with $\mathsf{P}_D = \mathsf{P}_i, \mathsf{P}_{INT} = \mathsf{P}_k$ and each coefficent of $f_{\beta_k}(\texttt{X})$ as the value to sign.

2. Each $\mathsf{P}_j$ computes $\beta_{k,j} = \beta_j + \sum_{\tau=1}^{t} \alpha_{\tau,j}k^\tau$ and sends $\beta_{k,j}$ to each $\mathsf{P}_k$.

3. Each $\mathsf{P}_k$ checks that $\deg(f_{\beta_k}) \leq t$ and that $\beta_{k,j} = f_{\beta_k}(j)$ for $j = 1, \dots, n$. If so, it broadcasts `success`. Otherwise, it broadcasts $(\texttt{dispute}, k, j)$ for each inconsistency.

4. For each dispute reported in the previous step, $\mathsf{P}_i$ broadcasts the correct value of $\beta_{k,j}$.

5. If any $\mathsf{P}_k$ finds a disagreement between what $\mathsf{P}_i$ has broadcast and what he received privately from $\mathsf{P}_i$, he knows $\mathsf{P}_i$ is corrupt and broadcasts $(\texttt{accuse}, k)$ as well as the offending value $\beta'$. Note that this value must always be a linear combination of values received directly from $\mathsf{P}_i$.

6. For any accusation from $\mathsf{P}_k$ in the previous step, players call an appropriate sequence of `add` and `mult` commands to have $\mathsf{F}_{\texttt{ICSIG}}$ compute $\beta'$ internally. Then they call `reveal` to make the value public. If the revealed value is indeed $\beta'$ and $\beta'$ differs from the corresponding value broadcast by $\mathsf{P}_i$, players output `fail` and the protocol ends here.

7. If we reach this step, no accusations were proved. Each $\mathsf{P}_j$ outputs `success` and stores $(cid, i, \beta_j)$. $\mathsf{P}_i$ in addition stores the polynomial $g_a(\texttt{X})$

Protocol Com-Simple

**commit:** See Protocol Statistical-Commit.

**public commit:** On input $(\texttt{pubcommit}, i, cid, a)$ a party $\mathsf{P}_k$ lets $a_k = a$ (this is a share on the polynomial $a(\texttt{X}) = a$), outputs $(\texttt{pubcommit}, i, cid, \texttt{success})$ and stores $(cid, i, a_k)$. Players call `sign` on $\mathsf{F}_{\texttt{ICSIG}}$ with $\mathsf{P}_D = \mathsf{P}_i, \mathsf{P}_{INT} = \mathsf{P}_k$ and $a$ as the value to sign (this is just for consistency so that every value stored after a commit is signed by the committer).

**open:** On input $(\texttt{open}, i, cid)$ where some $(cid, i, a(\texttt{X}))$ is stored, the party $\mathsf{P}_i$ broadcasts $(cid, i, a(\texttt{X}))$.

On input $(\texttt{open}, i, cid)$ where some $(cid, i, a_k)$ is stored each party $\mathsf{P}_k \neq \mathsf{P}_i$ broadcasts $(cid, i, a_k)$. Players call `reveal` on $\mathsf{F}_{\texttt{ICSIGN}}$ to confirm that $a_k$ is correct. If the `reveal` fails or the revealed value is not $a_k$, $\mathsf{P}_k$ is *ignored.*

If $\deg(a(\texttt{X})) \leq t$ and $a_k = a(k)$ for all parties that were not ignored then all parties output $(\texttt{open}, i, cid, a(0))$. Otherwise they output $(\texttt{open}, i, cid, \texttt{fail})$.

**designated open:** As above, but the polynomial and the shares are only sent to $\mathsf{P}_j$, and designated `reveal` is called on $\mathsf{F}_{\texttt{ICSIG}}$. If $\mathsf{P}_j$ rejects the opening, it broadcasts a public complaint, and $\mathsf{P}_i$ must then do a normal opening. If that one fails too, all parties output `fail`.

**add:** On input $(\texttt{add}, i, cid_1, cid_2, cid_3)$ where some $(i, cid_1, a(\texttt{X}))$ and $(i, cid_2, b(\texttt{X}))$ are stored the party $\mathsf{P}_i$ stores $(i, cid_3, a(\texttt{X}) + b(\texttt{X}))$ and outputs $(\texttt{add}, i, cid_1, cid_2, cid_3, \texttt{success})$.

On input $(\texttt{add}, i, cid_1, cid_2, cid_3)$ where some $(i, cid_1, a_k)$ and $(i, cid_2, b_k)$ are stored a party $\mathsf{P}_k \neq \mathsf{P}_i$ stores $(i, cid_3, a_k + b_k)$. Players call `add` on $\mathsf{F}_{\texttt{ICSIG}}$ to have it add $a_k$ and $b_k$ internally. $\mathsf{P}_k$ outputs $(\texttt{add}, i, cid_1, cid_2, cid_3, \texttt{success})$.

**multiplication by constant:** On input $(\texttt{mult}, i, \alpha, cid_2, cid_3)$ where some $(i, cid_2, b(\texttt{X}))$ is stored the party $\mathsf{P}_i$ stores $(i, cid_3, \alpha b(\texttt{X}))$ and outputs $(\texttt{mult}, i, \alpha, cid_2, cid_3, \texttt{success})$.

On input $(\texttt{mult}, i, \alpha, cid_2, cid_3)$ where some $(i, cid_2, b_k)$ is stored a party $\mathsf{P}_k \neq \mathsf{P}_i$ stores $(i, cid_3, \alpha b_k)$. Players call `mult` on $\mathsf{F}_{\texttt{ICSIG}}$ to have it multiply $\alpha$ and $b_k$ internally. $\mathsf{P}_k$ outputs $(\texttt{mult}, i, \alpha, cid_2, cid_3, \texttt{success})$.

### A Statistically Secure Commitment Scheme

The basic idea in the implementation of $\mathsf{F}_{\texttt{COM-SIMPLE}}$ for $t < n/2$ is to do essentially the same protocol as for $t < n/3$, but ask the committer to sign every value he sends to players. This actually makes the `commit` protocol simpler because accusations against the committer can be proved: the accuser can demonstrate that the message he claims to have received from the committer is correct, and everyone can then check that it indeed contradicts what the committer has broadcast.

Protocol Statistical-Commit and Protocol Com-Simple specify the solution in detail. The protocols assume access to $\mathsf{F}_{\texttt{ICSIG}}$.

The following theorem is easy to show, using the same principles we have seen many times by now. We leave the details to the reader.

**Theorem 5.8** *$\pi_{\texttt{COM-SIMPLE}} \diamond \pi_{\texttt{ICSIG}} \diamond \mathsf{F}_{\texttt{SC}}$ implements $\mathsf{F}_{\texttt{COM-SIMPLE}}$ in $\mathrm{Env}^{t,sync}$ with statistical security for all $t < n/2$.*

## 5.5 Final Results and Optimality of Corruption Bounds

In this section, we put together all the results we have seen in this chapter and show that the resulting bounds on the number of actively corrupted parties we can tolerate with our protocols are in fact optimal. We emphasize that all the negative results hold even if we only require security for static corruption, while the positive results hold for adaptive security.

Let $\mathsf{F}_{\mathsf{PPC}}$ be a functionality that provides a secure point-to-point channel between every pair of players. That is, it is defined exactly as $\mathsf{F}_{\mathsf{SC}}$, except that broadcast is not provided. A first important fact is that Lamport et al. [LSP82] have shown a result which in our terminology is as follows:

**Theorem 5.9** *There exists a protocol $\pi_{\mathtt{BROADCAST}}$ such that $\pi_{\mathtt{BROADCAST}} \diamond \mathsf{F}_{\mathsf{PPC}}$ implements $\mathsf{F}_{\mathsf{SC}}$ in* $\mathrm{Env}^{t,sync}$ *with perfect security for all $t < n/3$.*

They have also shown that the bound on $t$ is optimal, i.e.,

**Theorem 5.10** *There exists no protocol $\pi_{\mathtt{BROADCAST}}$ such that $\pi_{\mathtt{BROADCAST}} \diamond \mathsf{F}_{\mathsf{PPC}}$ implements $\mathsf{F}_{\mathsf{SC}}$ in* $\mathrm{Env}^{t,sync}$ *with perfect or statistical security if $t \geq n/3$.*

We do not give the proofs here, as the techniques are somewhat out of scope for this book. But the intuition is not hard to understand. Consider that case where $n = 3$, $t = 1$ and $\mathsf{P}_1$ wants to broadcast a value $x$. Let us assume that $\mathsf{P}_2$ is honest. Now, if $\mathsf{P}_1$ is honest, $\mathsf{P}_2$ must output $x$. On the other hand, if $\mathsf{P}_1$ is corrupt and may send different values to different players, $\mathsf{P}_2$ must agree with $\mathsf{P}_3$ on some value. Of course, we can ask $\mathsf{P}_1$ to send $x$ to both $\mathsf{P}_2$ and $\mathsf{P}_3$. But now the only thing $\mathsf{P}_2$ can do to ensure he agrees with $\mathsf{P}_3$ is to ask him what he heard from $\mathsf{P}_1$. If he is told a value $x' \neq x$, he knows of course that one of $\mathsf{P}_1, \mathsf{P}_3$ is corrupt, but there is no way to tell who is lying, and therefore no way to decide if the output should be $x$ or $x'$. The proof of Theorem 5.10 is basically a formalization of this argument. On the other hand, if $n = 4$ and $t = 1$, there are 2 other players $\mathsf{P}_2$ can ask what they heard from $\mathsf{P}_1$, and if $\mathsf{P}_1$ is honest we are guaranteed that the correct output can be found by taking majority decision among the 3 values $\mathsf{P}_2$ has seen. This is the basic reason why a broadcast protocol as claimed in Theorem 5.9 can be built.

Putting all our results for the case of $t < n/3$ together using the UC theorem we get that Theorems 5.1, 5.2, 5.6 and 5.9 imply that any function can be computed with perfect, active security if less than $n/3$ players are corrupt, given access to secure point to point channels. More formally, we have:

**Theorem 5.11** *$\pi^{f}_{\mathtt{CEAS}} \diamond \pi_{\mathtt{PTRANSFER,PMULT}} \diamond \pi_{\mathtt{PCOM\text{-}SIMPLE}} \diamond \pi_{\mathtt{BROADCAST}} \diamond \mathsf{F}_{\mathsf{PPC}}$ implements $\mathsf{F}^{f}_{\mathsf{SFE}}$ in* $\mathrm{Env}^{t,sync}$ *with perfect security for all $t < n/3$.*

We now argue that the bound $t < n/3$ is optimal. First, note that the problem of implementing broadcast can be phrased as a problem of computing a function securely, namely a function $f$ that takes an input $x$ from a single player and outputs $x$ to all players. In this language, Theorem 5.10 says that if $t \geq n/3$, there exists functions (such as $f$) that cannot be computed given only $\mathsf{F}_{\mathsf{PPC}}$, not even if we want only statistical rather than perfect security.

In other words, Theorem 5.11 is false for $t \geq n/3$ even if we only ask for statistical security. We shall see in a moment that if we assume broadcast is given (i.e., access to $\mathsf{F}_{\mathsf{SC}}$ rather than $\mathsf{F}_{\mathsf{PPC}}$) and only ask for statistical security we can tolerate $t < n/2$. So the obvious remaining question is what happens if we have access to $\mathsf{F}_{\mathsf{SC}}$ but insist on perfect security. The answer turns out to be negative:

Let $\mathsf{F}_{\mathsf{COMMIT}}$ be the functionality defined as $\mathsf{F}_{\mathsf{COM\text{-}SIMPLE}}$, but with only the `commit` and `open` commands.

**Theorem 5.12** *There exists no protocol $\pi_{\mathtt{COMMIT}}$ such that $\pi_{\mathtt{COMMIT}} \diamond \mathsf{F}_{\mathsf{SC}}$ implements $\mathsf{F}_{\mathsf{COMMIT}}$ in* $\mathrm{Env}^{t,sync}$ *with perfect security if $t \geq n/3$.*

*Proof* We demonstrate that no such protocol exists for $n = 3$ and $t = 1$. Assume for contradiction that we have a protocol $\pi_{\texttt{COMMIT}}$. Suppose $\mathsf{P}_1$ commits to a bit $b$. Let $V_i(b, R)$ be the view of the execution of the `commit` command seen by player $\mathsf{P}_i$ where we assume all players follow the protocol and $R$ is the random coins used by all players. By perfect security, the distribution of $V_2(b, R)$ is independent of $b$ ($\mathsf{P}_2$ learns nothing about $b$ before opening). It follows that given a sample $V_2(0, R)$, there must exist $R'$ such that $V_2(0, R) = V_2(1, R')$. Now consider the following two cases:

**Case 0:** $\mathsf{P}_1$ is corrupt. He commits to $b = 0$ following the protocol, so he sees $V_1(0, R)$ for some $R$. Now, he tries to guess $R'$ such that $V_2(0, R) = V_2(1, R')$, and computes $V_1(1, R')$. He then executes the `open` command following the protocol, but pretending that his view of the `open` command was $V_1(1, R')$.

**Case 1:** $\mathsf{P}_1$ is honest, but $\mathsf{P}_3$ is corrupt. $\mathsf{P}_1$ commits to $b = 1$, and $\mathsf{P}_3$ follows the protocol during the execution of the `commit` command. $R'$ are the random coins used. Now $\mathsf{P}_3$ tries to guess $R$ as defined in Case 0, computes $V_3(0, R)$, and then executes the `open` command following the protocol, but pretending that his view of the `open` command was $V_3(0, R)$.

Obviously both cases may happen with (possibly small but) non-zero probability. Furthermore the views that are input to the `open` command are exactly the same in the two cases, so the distribution of the bit that $\mathsf{P}_2$ will output is the same in both cases. However, by perfect security, $\mathsf{P}_2$ must *always* output 0 in Case 0 and 1 in Case 1, which is clearly not possible. □

The reader might complain that what we have shown impossible here is a reactive functionality, and this is a more advanced tool than secure evaluation of a function which is what the positive results talk about. However, first the protocols we have shown can also be used to implement general reactive functionalities as discussed earlier. Second, Theorem 5.12 can also be shown for secure evaluation of a function, although this is a bit more cumbersome. Consider the function that takes inputs $x$ from $\mathsf{P}_1$, $y$ from $\mathsf{P}_2$ and outputs $x, y$ to all players. If we require that the ideal functionality for computing this function does not output anything before its gets both inputs, we can show a result similar to Theorem 5.12 for this function. The point is that even if $\mathsf{P}_1$ is corrupt, he must decide on $x$ before he gets any output, and the same holds for $\mathsf{P}_2$. Hence the protocol must commit $\mathsf{P}_1$ to $x$ but without revealing any information on $x$ to the corrupt players if $\mathsf{P}_1$ is honest. This is the same property we need for commitments, so the result can be shown along the same lines. We leave the details to the reader.

The final result in this chapter concerns the case where we assume access to $\mathsf{F}_{\texttt{SC}}$ and go for statistical security. In this case, putting previous results together using the UC theorem, we get that Theorems 5.1, 5.3 and 5.8 imply that any function can be computed with statistical, active security if less than $n/2$ players are corrupt, given access to secure point to point channels and broadcast. More formally, we have:

**Theorem 5.13** *$\pi^f_{\texttt{CEAS-N/2}} \diamond \pi_{\texttt{TRANSFER,MULT}} \diamond \pi_{\texttt{COM-SIMPLE}} \diamond \mathsf{F}_{\texttt{SC}}$ implements $\mathsf{F}^f_{\texttt{SFE}}$ in $\mathrm{Env}^{t,sync}$ with statistical security for all $t < n/2$.*

Here, $\pi^f_{\texttt{CEAS-N/2}}$ is $\pi^f_{\texttt{CEAS}}$ modified to handle $t < n/2$ as discussed in Section 5.3. The bound $t < n/2$ is optimal: we have already seen in Chapter 3 that secure computation with information theoretic security is not even possible with passive corruption if $t \geq n/2$.

# Chapter 6

# MPC from General Linear Secret Sharing Schemes

**Contents**

## 6.1 Introduction

In this chapter we will show how the protocols we have seen so far can be generalized to be based on general linear secret sharing schemes. Doing this generalization has several advantages: first, it allows us to design protocols that protect against general adversary structures, a concept we explain below, and second, it allows us to consider protocols where the field that is used in defining the secret sharing scheme can be of size independent of the number of players. This turns out to be an advantage for the applications of multiparty computation we consider later, but cannot be done for the protocols we have seen so far: to use Shamir's scheme for $n$ players based on polynomials as we have seen, the field must be of size at least $n + 1$.

For now, we will only consider the theory for linear secret sharing that we need to construct the protocols in this chapter. However, we note already now that to get full milage from the more general applications of secret sharing and MPC, we need additional theory, in particular about the asymptotic behavior of secret sharing schemes. This material can be found in Chapter 10.

## 6.2 General Adversary Structures

The corruption tolerance of the protocols we have seen so far have been characterized by only a single number $t$, the maximal number of corruptions that can be tolerated. This is also known as the threshold-$t$ model. One way to motivate this might be to think of corruption of a player as something that requires the adversary to invest some amount of resource, such as time or money. If the adversary has only bounded resources, there should be a limit on the corruptions he can do. However, characterizing this limit by only an upper bound implicitly assumes that all players are equally hard to corrupt. In reality this may be completely false: some players may have much better security than others, so a more realistic model might be that the adversary can corrupt a small number of well protected players or a large number of poorly protected players. However, the threshold model does not allow us to express this.

Another way to interpret corruptions is to see them as a result of some subset of players working together to cheat the others. Also from this point of view, it seems quite restrictive to assume that any subset of a given size might form such a collaboration. Which subsets we should actually worry about may depend on the composition of the set of players and their relation to each other, more than on the size of the subsets.

To solve these issues, we consider the more general concept of an Adversary Structure. This is a family $\mathcal{A}$ of subsets of the set of players $\mathsf{P} = \{\mathsf{P}_1, ..., \mathsf{P}_n\}$. The idea is that this is a complete list of the subsets that the adversary is able to corrupt. In the threshold-$t$ model, $\mathcal{A}$ would contain all subsets of size at most $t$. But in general any list of subsets is possible, with one constraint, however: we require that $\mathcal{A}$ be anti-monotone, namely $A \subseteq B$ and $B \in \mathcal{A}$ implies that $A \in \mathcal{A}$. The meaning of this is that if the adversary is powerful enough to corrupt set $B$, he can also choose to corrupt any smaller set. We characterize two important classes of adversary structures:

**Definition 6.1** *An anti-monotone adversary structure $\mathcal{A}$ is said to be $Q2$ if for all $A_1, A_2 \in \mathcal{A}$ it holds that $A_1 \cup A_2 \neq \mathsf{P}$. It is said to be $Q3$ if for all $A_1, A_2, A_3 \in \mathcal{A}$ it holds that $A_1 \cup A_2 \cup A_3 \neq \mathsf{P}$.*

The $Q2$ and $Q3$ conditions are natural generalizations of the threshold-$t$ model, for $t < n/2$ and $t < n/3$, respectively. For instance, the union of two sets that are both smaller than $n/2$ cannot be the entire player set $\mathsf{P}$.

We can then define what it means that a protocol for implements some functionality securely against an adversary structure $\mathcal{A}$. This is done in exactly the same way as we defined security for at most $t$ corruptions, except that we now consider environments that corrupt only subsets in $\mathcal{A}$.

We have seen that multiparty computation with perfect, passive security in the threshold-$t$ model is possible if and only if $t < n/2$ and with active security if and only if $t < n/3$. The following is then easy to prove:

**Theorem 6.1** *For any adversary structure $\mathcal{A}$ that is not $Q2$, there are functions that cannot be securely computed with passive and statistical security against $\mathcal{A}$. Moreover, for any adversary structure $\mathcal{A}$ that is not $Q3$, there are functions that cannot be securely computed with active and statistical security against $\mathcal{A}$.*

To prove this, one notices that the corresponding proofs for $t \geq n/2$ and $t \geq n/3$ actually only use the fact the the underlying adversary structures are not $Q2$, respectively $Q3$. This is straightforward and is left to the reader.

We shall see later that he converse is also true: multiparty computation with perfect, passive security against adversary structure $\mathcal{A}$ is possible if $\mathcal{A}$ is $Q2$ and with active security if $\mathcal{A}$ is $Q3$.

The following examples illustrate that we can achieve strictly more with general adversary structures that what can be done in the threshold case:

**Example 6.1** Suppose we have 6 players, and consider the adversary structure containing the sets $\{\mathsf{P}_1\}, \{\mathsf{P}_2, \mathsf{P}_4\}, \{\mathsf{P}_2, \mathsf{P}_5, \mathsf{P}_6\}, \{\mathsf{P}_3, \mathsf{P}_5\}, \{\mathsf{P}_3, \mathsf{P}_6\}, \{\mathsf{P}_4, \mathsf{P}_5, \mathsf{P}_6\}$, and all smaller sets. It is trivial to see that this structure is $Q3$, so we can do multiparty computation with active security in this case. Note that if the threshold-$t$ model with $t < n/3$ was all we could do, we could only tolerate corruption of a single player. $\triangle$

**Example 6.2** Here is an infinite family of examples: suppose our player set is divided into two disjoint sets $X$ and $Y$ of $m$ players each ($n = 2m$) where the players are on friendly terms within each group but tend to distrust players in the other group. Hence, a coalition of active cheaters might consist of almost all players from $X$ or from $Y$, whereas a mixed coalition with players from both groups is likely to be quite small. Concretely, suppose we assume that a corrupted set can consist of at most $9m/10$ players from only $X$ or only $Y$, *or* it can consist of less than $m/5$ players coming from both $X$ and $Y$. This defines a $Q3$ adversary structure, and so multi-party computation with active security is possible in this scenario. Nevertheless, no threshold solution exists, since the largest coalitions of corrupt players have size more than $n/3$. The intuitive reason why multiparty computation is nevertheless possible is that although some corruptible sets are larger than $n/3$, we do not need to protect against corruption of *all* such sets. $\triangle$

## 6.3 Linear Secret-Sharing

Let us consider how we can realize multiparty computation that is secure if the adversary may corrupt subsets of players in some adversary structure, that is not necessarily threshold. One might think that a solution could be to use Shamir secret sharing, generate more shares than there are players, and give a different number of shares to different players. At least this will imply that we can tolerate corruption of a larger number of those players that received only a small number of shares. This approach would allow us to handle some, but not all relevant adversary structures, and it will often be suboptimal in terms of efficiency.

A much better solution is to generalize Shamir's scheme further: Say we share the secret $s$ among $n$ players in the usual way using a polynomial $f_s(\mathtt{X})$. We compute shares by evaluating $f_s$ in points $1, 2..., n$. This computation of the shares can be rephrased as follows: Consider a Van der Monde matrix $M$ which by definition is a matrix where the rows are of form $(i^0, i^1, i^2, ..., i^t)$ for $i = 1, ..., n$. Now define a column vector $\mathbf{r}_s$ whose entries are the coefficients of $f_s$, with $s$ (the degree-0 coefficient) as the first entry. Now, computing the product $M \cdot \mathbf{r}_s$ is clearly equivalent to evaluating $f_s$ in points $1, ...., n$. Therefore we can rephrase Shamir's scheme as follows: choose a random vector with $t + 1$ entries subject to the secret $s$ appearing as first coordinate. Compute $M\mathbf{r}_s$ and give the $i$'th entry of the result to player $\mathsf{P}_i$.

A generalization now quite naturally suggests itself: first, why not consider other matrices than Van der Monde, in particular with more than $n$ rows? second, once we do this, we can assign more than one row to each player, so they may receive more than one value in the field. This generalization exactly leads to the linear secret sharing schemes we consider in this chapter.

Before we define these schemes formally, some notation: for consistency in the following, vectors are column vectors unless we state otherwise, and $\cdot$ denotes matrix multiplication throughout. Thus we will write the inner product of $\mathbf{a}, \mathbf{b}$ as $\mathbf{a}^\mathsf{T} \cdot \mathbf{b}$.

**Definition 6.2** *A linear secret sharing scheme $\mathcal{S}$ over a field $\mathbb{F}$ for $n$ players is defined by a matrix $M$ and a function $\phi$. $M$ has $m \geq n$ rows and $t + 1$ columns, and $\phi$ is a* labeling function *$\phi : \{1, ..., m\} \mapsto \{1, ..., n\}$. We say $M$ is* the matrix for $\mathcal{S}$. *We can apply $\phi$ to the rows of $M$ in a natural way, and we say that player $\mathsf{P}_{\phi(i)}$* owns *the i'th row of $M$. For a subset $A$ of the players, we let $M_A$ be the matrix consisting of the rows owned by players in $A$.*

To secret-share $s \in \mathbb{F}$ using $\mathcal{S}$, one forms a column vector $\mathbf{r}_s$ with $s$ as the first coordinate, while the other coordinates are $r_1, .., r_t$ where each $r_i$ is uniformly random in $\mathbb{F}$. Finally one computes the vector of shares $M\mathbf{r}_s$ and distributes them among players, by giving $(M\mathbf{r}_s)_i$ to player $P_{\phi(i)}$. Note that in this way, a player may get more than one value as his share. Note also that for a subset $A$ of players, $M_A\mathbf{r}_s$ is the set of values received by players in $A$ when $s$ is shared.

**Definition 6.3** *The* adversary structure *of $\mathcal{S}$ is a family of subsets of players. A set $A$ is in the adversary structure if and only if the distribution of $M_A\mathbf{r}_s$ is independent of $s$. Such a set is* *called unqualified. The access structure of $\mathcal{S}$ is also a family of subsets of players. A set $A$ is in the access structure if and only if $s$ is uniquely determined from $M_A\mathbf{r}_s$. Such a set is called qualified.*

It follows form Theorem 6.3 below that every player subset is either qualified or unqualified.

We will let $\mathbf{m}_k$ denote the $k$'th row of $M$. When we need to talk about individual entries in $M$, it turns out to be convenient to number the columns from 0, and hence the $k$'th row of $M$ can also be written as

$$\mathbf{m}_k = (M_{k,0}, M_{k,1}, ..., M_{k,t})$$

This also gives an alternative way to address single shares that will sometimes be convenient, namely we have:

$$(M\mathbf{r}_a)_k = \mathbf{m}_k \cdot \mathbf{r}_a$$

We will need the following basic fact from linear algebra, where we recall that $Im(N)$ for a matrix $N$ is the set of all vectors of form $N\mathbf{v}$, or put differently; the set of vectors that are obtained as linear combinations of the columns of $N$.

**Lemma 6.2** *For any matrix $M$ and vector $\mathbf{e}$ we have that $\mathbf{e} \notin Im(M^\mathsf{T})$ if and only if there exists $\mathbf{w}$ with $\mathbf{w} \in \ker(M)$ and $\mathbf{w}^\mathsf{T} \cdot \mathbf{e} \neq 0$.*

*Proof* For a subspace $V$ of any vector space over $\mathbb{F}$, $V^\perp$ denotes the orthogonal complement of $V$, the subspace of vectors that are orthogonal to all vectors in $V$. Now, notice that $Im(M^\mathsf{T})$ is the space spanned by the rows of $M$. Therefore, it is clear that $\ker(M) = Im(M^\mathsf{T})^\perp$, since computing $M \cdot \mathbf{a}$ for some $\mathbf{a}$ is equivalent to computing inner product of $\mathbf{a}$ with every row of $M$. From this and $V = (V^\perp)^\perp$ for any subspace $V$ follows that $\ker(M)^\perp = Im(M^\mathsf{T})$. So $\mathbf{e} \notin Im(M^\mathsf{T})$, if and only if $\mathbf{e} \notin \ker(M)^\perp$, and this holds if and only if $\mathbf{w}$ as in the statement of lemma exists. □

**Theorem 6.3** *Let $\mathcal{S}$ be a linear secret sharing scheme and let $M$ be the matrix for $\mathcal{S}$. A player subset $A$ is qualified in $\mathcal{S}$ if and only if $\mathbf{e} = (1, 0, ..., 0)^\mathsf{T} \in Im(M_A^\mathsf{T})$, i.e., the subspace spanned by the rows of $M_A$ contains $\mathbf{e}$. In this case there exists a reconstruction vector $\mathbf{u}$ such that $\mathbf{u}^\mathsf{T}(M_A\mathbf{r}_s) = s$ for all $s$. $A$ is unqualified in $\mathcal{S}$ if and only if $\mathbf{e} \notin Im(M_A^\mathsf{T})$. In this case there exists a vector $\mathbf{w}$ that is orthogonal to all rows in $M_A$ and has first coordinate 1.*

*Proof* If $\mathbf{e} \in Im(M_A^\mathsf{T})$ there exists a vector $\mathbf{u}$ such that $M_A^\mathsf{T} \cdot \mathbf{u} = \mathbf{e}$, or $\mathbf{u}^T \cdot M_A = \mathbf{e}^\mathsf{T}$. But then we have

$$\mathbf{u}^\mathsf{T} \cdot (M_A \cdot \mathbf{r}_s) = (\mathbf{u}^\mathsf{T} \cdot M_A) \cdot \mathbf{r}_s = \mathbf{e}^\mathsf{T} \cdot \mathbf{r}_s = s.$$

If $\mathbf{e}$ is not in the span of the rows of $M_A$, then by Lemma 6.2, there exists a vector $\mathbf{w}$ that is orthogonal to all rows of $M_A$ but not orthogonal to $\mathbf{e}$. This can be used as the $w$ claimed in the theorem since we may assume without loss of generality that the inner product $\mathbf{w}^\mathsf{T} \cdot \mathbf{e}$ is 1. Let $s, s'$ be any two possible values of the secret to be shared. For any $\mathbf{r}_s$, let $\mathbf{r}_{s'} = \mathbf{r}_s + (s' - s)\mathbf{w}$. Then the first coordinate of $\mathbf{r}_{s'}$ is $s'$ and we have

$$M_A \cdot \mathbf{r}'_s = M_A \cdot \mathbf{r}_s$$

That is, for each set of random coins used to share $s$ there is exactly one set of coins for sharing $s'$ such that the players in $A$ receive the same shares. Since the randomness is chosen uniformly it follows that the distribution of shares received by $A$ is the same for all values of the secret, and so $A$ is in the adversary structure. □

In the following, we will refer to $m$, the number of rows in $M$ as the size of the secret sharing scheme defined by $M$. The reason it makes sense to ignore $t$ (the number of columns) in this connection is that we can assume without loss of generality that $t + 1 \leq m$: given any $M$ we could always consider a new matrix $M'$ containing only a maximal independent set of columns from $M$. Because the column-rank equals the row-rank for any matrix, such a set will have size at most $m$. Moreover, it will lead to a secret sharing scheme with the same access structure. This follows from Theorem 6.3 which says that a set $A$ is qualified exactly if the rows of $M_A$ span the first basis vector. This happens if and only if the rows of $M'_A$ span the first basis vector, because the columns in $M_A$ can all be obtained as linear combinations of the columns in $M'_A$.

In the following we will reuse some notation from earlier chapters and write

$$M\mathbf{r}_s = [s; \mathbf{r}_s]_{\mathcal{S}}$$

Using the standard entrywise addition and multiplication by a scalar and Schur product the following vectors are well defined (for $\alpha, a, b \in \mathbb{F}$):

$$[a; \mathbf{r}_a]_{\mathcal{S}} + [b; \mathbf{r}_b]_{\mathcal{S}}, \quad \alpha[a; \mathbf{r}_a]_{\mathcal{S}}$$

We now want to consider multiplication of shared values. To understand how this works, recall how we handled secure multiplication with Shamir's secret sharing scheme in previous chapters: say $[a; f_a]_t, [b; f_b]_t$ have been distributed and we want to compute $ab$ in shared form. The main observation that we used for this was that if each player locally multiplies his share of $a$ and of $b$, then the local products form a set of shares in $ab$, namely they form $[ab; f_a f_b]_{2t}$. This new sharing is of different type, however, it is formed by a polynomial of degree $2t$ and not $t$.

It turns out that something similar happens for general linear secret-sharing. Assume that $[a; \mathbf{r}_a]_{\mathcal{S}}, [b; \mathbf{r}_b]_{\mathcal{S}}$ have been distributed. We now want to define what it means for players to locally multiply their shares of $a$ and of $b$. This gets more complicated than in the case of polynomials because a player may have more than one field value as his share of $a$ (and of $b$). Therefore there may be several different pairwise products he can compute, and we need notation to capture all of these. We therefore define a special type of product, written

$$[a; \mathbf{r}_a]_{\mathcal{S}} \odot [b; \mathbf{r}_b]_{\mathcal{S}}$$

This vector is constructed so it contains all products of a share of $a$ and a share of $b$ that some player can compute locally. More formally

$$[a; \mathbf{r}_a]_{\mathcal{S}} \odot [b; \mathbf{r}_b]_{\mathcal{S}} = (([a; \mathbf{r}_a]_{\mathcal{S}})_{\{\mathsf{P}_1\}} \otimes ([b; \mathbf{r}_b]_{\mathcal{S}})_{\{\mathsf{P}_1\}}, \ldots, (([a; \mathbf{r}_a]_{\mathcal{S}})_{\{\mathsf{P}_n\}} \otimes ([b; \mathbf{r}_b]_{\mathcal{S}})_{\{\mathsf{P}_n\}})$$

where for $\mathbf{a}, \mathbf{b} \in \mathbb{F}^{\ell}$, $\mathbf{a} \otimes \mathbf{b}$ is the tensor product of vectors, i.e., $\mathbf{a} \otimes \mathbf{b} \in \mathbb{F}^{\ell^2}$ and contains all (ordered) pairwise products of entries in the two vectors.

Note that if $n = m$ and every player owns one value we can set $\phi(i) = i$. In this case, $[a; \mathbf{r}_a]_{\mathcal{S}} \odot [b; \mathbf{r}_b]_{\mathcal{S}}$ is just the Schur product $[a; \mathbf{r}_a]_{\mathcal{S}} * [b; \mathbf{r}_b]_{\mathcal{S}}$. In the following, $\hat{m}$ will denote the length of $[a; \mathbf{r}_a]_{\mathcal{S}} \odot [b; \mathbf{r}_b]_{\mathcal{S}}$.

### 6.3.1 Multiplicative Secret-Sharing

As we hinted at above, if values $a, b$ have been secret shared using $\mathcal{S}$, it turns out that the vector $[a; \mathbf{r}_a]_{\mathcal{S}} \odot [b; \mathbf{r}_b]_{\mathcal{S}}$ forms a set of shares of $ab$, albeit in a different linear secret-sharing scheme, which we will denote $\hat{\mathcal{S}}$. We specify $\hat{\mathcal{S}}$ by directly constructing its matrix $\hat{M}$: Consider the rows of $M$ owned by $\mathsf{P}_i$, i.e., $M_{\{\mathsf{P}_i\}}$. We now define $\hat{M}_{\{\mathsf{P}_i\}}$ to be a matrix consisting of all rows of form $\mathbf{u} \otimes \mathbf{v}$ where $\mathbf{u}$ and $\mathbf{v}$ are rows in $M_{\{\mathsf{P}_i\}}$. Then $\hat{M}$ is built by putting all the $\hat{M}_{\{\mathsf{P}_i\}}$ together in one matrix. The labeling function is defined such that $\mathsf{P}_i$ owns exactly the rows that came from $\hat{M}_{\{\mathsf{P}_i\}}$.

With this extension of the labeling function, we get the following simple fact that we will use later:

**Lemma 6.4** *Let $c_k$ be the k'th entry in $[a; \mathbf{r}_a]_{\mathcal{S}} \odot [b; \mathbf{r}_b]_{\mathcal{S}}$. Then there exists (fixed) indices $u, v$ such that $c_k = a_u b_v$ where $a_u$ is the u'th entry in $[a; \mathbf{r}_a]_{\mathcal{S}}$, $b_v$ is the v'th entry in $[b; \mathbf{r}_b]_{\mathcal{S}}$ and $\phi(k) = \phi(u) = \phi(v)$.*

The main reason we introduce $\hat{\mathcal{S}}$ is that it can be used to implement multiplication of shared secrets:

**Lemma 6.5** *For any linear secret sharing scheme $\mathcal{S}$ we have*

$$[a; \mathbf{r}_a]_{\mathcal{S}} \odot [b; \mathbf{r}_b]_{\mathcal{S}} = [ab; \mathbf{r}_a \otimes \mathbf{r}_b]_{\hat{\mathcal{S}}}$$

*Proof* Notice first that if $\mathbf{u}, \mathbf{v}$ are rows in $M_{\{\mathsf{P}_i\}}$, then $\mathbf{u} \cdot \mathbf{r}_a, \mathbf{v} \cdot \mathbf{r}_b$ are values $\mathsf{P}_1$ will receive when secrets $a, b$ are shared, i.e., they are entries in $([a; \mathbf{r}_a]_{\mathcal{S}})_{\{\mathsf{P}_1\}}$ and $([b; \mathbf{r}_b]_{\mathcal{S}})_{\{\mathsf{P}_1\}}$, respectively.

Therefore, the entries in $([a; \mathbf{r}_a]_{\mathcal{S}})_{\{\mathsf{P}_1\}} \otimes ([b; \mathbf{r}_b]_{\mathcal{S}})_{\{\mathsf{P}_1\}}$ are of form

$$(\mathbf{u} \cdot \mathbf{r}_a) \cdot (\mathbf{v} \cdot \mathbf{r}_b) = (\mathbf{u} \otimes \mathbf{v}) \cdot (\mathbf{r}_a \otimes \mathbf{r}_b).$$

The lemma now follows by definition of $\hat{M}$. $\square$

Note that the adversary structure of $\hat{\mathcal{S}}$ contains that of $\mathcal{S}$. This is because any player set $A$ can locally compute their part of $[ab; \mathbf{r}_a \otimes \mathbf{r}_b]_{\hat{\mathcal{S}}}$ after secrets $a, b$ have been shared. If players in $A$ could reconstruct $ab$ from this, this reveals information on $a$ and $b$, which cannot be the case if $A$ is in the adversary structure of $\mathcal{S}$.

**Definition 6.4** *If the access structure of $\hat{\mathcal{S}}$ contains the set of all players, $\mathcal{S}$ is said to be* multiplicative. *If the access structure of $\hat{\mathcal{S}}$ contains the complement of every set in the adversary structure $\mathcal{A}$ of $\mathcal{S}$, then $\mathcal{S}$ is said to be* strongly multiplicative.

Note that if $\mathcal{S}$ is multiplicative then there exists a reconstruction vector $\mathbf{u}$ such that

$$\mathbf{u} \cdot ([a; \mathbf{r}_a]_{\mathcal{S}} \odot [b; \mathbf{r}_b]_{\mathcal{S}}) = \mathbf{u} \cdot [ab; \mathbf{r}_a \otimes \mathbf{r}_b]_{\hat{\mathcal{S}}} = ab$$

A way to think about the strong multiplication property is as follows: if an adversary is able to corrupt a set of players in $\mathcal{A}$ (but not more than that), then the set of honest players is always the complement of a set in $\mathcal{A}$. And then strong multiplication says that the set of honest players can on their own reconstruct $ab$ from $[ab; \mathbf{r}_a \otimes \mathbf{r}_b]_{\hat{\mathcal{S}}}$.

The reader should take a moment to understand that the multiplication property indeed generalizes in a natural way what we know about secret sharing using polynomials, as we promised above: the secret-sharing scheme $\mathcal{S}$ corresponds to sharing using polynomials of degree at most $t$, whereas $\hat{\mathcal{S}}$ corresponds to sharing using polynomials of degree at most $2t$. Likewise, $\mathbf{r}_s$ corresponds to the coefficients of a polynomial that evaluates to $s$ in 0, and $\mathbf{r}_a \otimes \mathbf{r}_b$ corresponds to multiplication of polynomials.

Summarizing what we have seen, by linearity of the function $g$ and what we just stated on $\hat{\mathcal{S}}$, we immediately get the following:

**Lemma 6.6** *The following holds for any $a, b, \alpha \in \mathbb{F}$ and any multiplicative linear secret-sharing scheme $\mathcal{S}$:*

$$[a; \mathbf{r}_a]_{\mathcal{S}} + [b; \mathbf{r}_b]_{\mathcal{S}} = [a + b; \mathbf{r}_a + \mathbf{r}_b]_{\mathcal{S}}, \tag{6.1}$$

$$\alpha[a; \mathbf{r}_a]_{\mathcal{S}} = [\alpha a; \alpha \mathbf{r}_a]_{\mathcal{S}}, \tag{6.2}$$

$$[a; \mathbf{r}_a]_{\mathcal{S}} \odot [b; \mathbf{r}_b]_{\mathcal{S}} = [ab; \mathbf{r}_a \otimes \mathbf{r}_b]_{\hat{\mathcal{S}}}. \tag{6.3}$$

### 6.3.2 On Existence of Linear Secret Sharing Schemes

We start by the most basic facts on existence of linear secret sharing schemes:

**Theorem 6.7** *Linear secret sharing schemes of the types we discussed exist for all adversary structures:*

- *For every anti-monotone structure $\mathcal{A}$, there exists a linear secret sharing-scheme $\mathcal{S}$ whose adversary structure is $\mathcal{A}$.*
- *For every $Q2$-structure $\mathcal{A}$, there exists a multiplicative linear secret sharing-scheme $\mathcal{S}$ whose adversary structure is $\mathcal{A}$.*
- *For every $Q3$-structure $\mathcal{A}$, there exists a strongly multiplicative linear secret sharing-scheme $\mathcal{S}$ whose adversary structure is $\mathcal{A}$.*

This theorem can be shown using just a single scheme, namely so-called replicated secret sharing scheme which works as follows. Given adversary structure $\mathcal{A}$ and secret $s \in \mathbb{F}$ to share, do the following: for each set $A \in \mathcal{A}$, choose $r_A \in \mathbb{F}$ at random, subject to the constraint that $s = \sum_{A \in \mathcal{A}} r_A$. Then for all $A \in A$, give $r_A$ to all players *not* in $A$. We will call this scheme $\mathcal{R}_{\mathcal{A}}$. Note that it can be defined over any field, independently of the number of players.

**Exercise 6.1** Show that $\mathcal{R}_{\mathcal{A}}$ is a linear secret sharing scheme with adversary structure $\mathcal{A}$.

**Exercise 6.2** Show that $\mathcal{R}_{\mathcal{A}}$ is multiplicative if $\mathcal{A}$ is $Q2$.

**Exercise 6.3** Show that $\mathcal{R}_{\mathcal{A}}$ is strongly multiplicative if $\mathcal{A}$ is $Q3$.

Of course, $\mathcal{R}_{\mathcal{A}}$ is in general very inefficient in the sense that most adversary structures contain a number of sets that is exponential in the number of players and so players will receive a very large number of shares. On the other hand, Shamir's scheme with threshold $t$ is linear, and is easily seen to be multiplicative, respectively strongly multiplicative if $t < n/2$, respectively $t < n/3$. In this scheme players get only a single field element as their share, so we see that the share size does not have to depend on the size of the adversary structure, even if we also require (strong) multiplication.

It would therefore be great if we could characterize those access structures that admit efficient linear schemes, i.e., of size polynomial in the number of players. However, this question is completely open. But we can at least say that asking for multiplication does not incur a large cost over just asking for a linear scheme. This is formalized in Theorem 6.9 below. However, we first need a new notion:

**Definition 6.5** *Let $\mathcal{F}$ be an access structure. Then the* dual *access structure $\bar{\mathcal{F}}$ is defined as follows: a set $A$ is in $\bar{\mathcal{F}}$ if and only if the complement $\bar{A}$ is not in $\mathcal{F}$.*

For instance, if $\mathcal{F}$ contains all sets of size larger than $t$, then $\bar{\mathcal{F}}$ contains all sets of size larger than $n - t$.

**Exercise 6.4** Let $\mathcal{S}$ be a secret sharing scheme with adversary structure $\mathcal{A}$ and access structure $\mathcal{F}$. Show that if $\mathcal{A}$ is $Q2$, then $\bar{\mathcal{F}} \subseteq \mathcal{F}$.

**Lemma 6.8** *Given any linear secret sharing scheme $\mathcal{S}$ with access structure $\mathcal{F}$, one can construct a linear secret sharing scheme $\bar{\mathcal{S}}$ with access structure $\bar{\mathcal{F}}$ in time polynomial in the size of $\mathcal{S}$. Moreover, the matrices $M, \bar{M}$ of $\mathcal{S}$ and $\bar{\mathcal{S}}$ satisfy $M^{\mathsf{T}}\bar{M} = E$, where $E$ is a matrix with 1 in the top-left corner and 0 elsewhere.*

*Proof* Let $M$ be the matrix of $\mathcal{S}$, let $\mathbf{v}$ be a solution to $M^{\mathsf{T}} \cdot \mathbf{v_0} = \mathbf{e}$, and let $\mathbf{v}_1, ..., \mathbf{v}_{m-(t+1)}$ be a basis for $Ker(M^{\mathsf{T}})$.

We now let the matrix $\bar{M}$ for $\bar{\mathcal{S}}$ be the matrix with columns $\mathbf{v_0}, \mathbf{v}_1, ..., \mathbf{v}_{m-(t+1)}$, and with the same labeling function as for $\mathcal{S}$. We also define for later $\bar{\mathbf{e}} \in \mathbb{F}^{m-(t+1)+1}$ to be $(1, 0, ..., 0)^{\mathsf{T}}$. Then $M^{\mathsf{T}}\bar{M} = E$ follows immediately by construction of $\bar{M}$.

We now want to show that $\bar{\mathcal{F}}$ is indeed the access structure of $\bar{\mathcal{S}}$, i.e., $A \in \mathcal{F}$ if and only if $\bar{A}$ is not in the access structure of $\bar{\mathcal{S}}$.

If $A \in \mathcal{F}$, the rows of $M_A$ span $\mathbf{e}$, or equivalently, there exists $\mathbf{v}$ such that $\mathbf{v}_{\bar{A}} = 0$ and $M^{\mathsf{T}} \cdot \mathbf{v} = \mathbf{e}$. By construction of $\mathbf{v_0}, \mathbf{v}_1, ..., \mathbf{v}_{m-(t+1)}$ any such $\mathbf{v}$ can be written as $\mathbf{v_0}$ plus some linear combination of $\mathbf{v}_1, ..., \mathbf{v}_{m-(t+1)}$, or equivalently there exists a vector $\mathbf{w}$ with 1 as its first coordinate such that $\mathbf{v} = \bar{M} \cdot \mathbf{w}$. But then $\bar{M}_{\bar{A}} \cdot \mathbf{w} = \mathbf{v}_{\bar{A}} = 0$, and $\mathbf{w}^{\mathsf{T}} \cdot \bar{\mathbf{e}} = 1 \neq 0$. By Lemma 6.2, the existence of such a $\mathbf{w}$ shows that the rows of $\bar{M}_{\bar{A}}$ do not span $\bar{\mathbf{e}}$, and so $\bar{A}$ is not in the access structure of $\bar{\mathcal{S}}$.

Conversely, if $\bar{A}$ is not in the access structure of $\bar{\mathcal{S}}$, there exists $\mathbf{w}$ with $\bar{M}_{\bar{A}} \cdot \mathbf{w} = 0$ and $\mathbf{w}^{\mathsf{T}} \cdot \bar{\mathbf{e}} = 1$. If we set $\mathbf{v} = \bar{M} \cdot \mathbf{w}$ then $\mathbf{v}_{\bar{A}} = 0$, and so

$$M_A^{\mathsf{T}} \cdot \mathbf{v}_A = M^{\mathsf{T}} \cdot \mathbf{v} = M^{\mathsf{T}} \cdot \bar{M} \cdot \mathbf{w} = E \cdot \mathbf{w} = \mathbf{e}$$

This shows that $A$ is in $\mathcal{F}$. □

**Theorem 6.9** *From every linear secret sharing scheme $\mathcal{S}$ with $Q2$-adversary structure $\mathcal{A}$, we can construct a multiplicative linear secret sharing scheme $\mathcal{S}'$ whose adversary structure is $\mathcal{A}$ in time polynomial in the size of $\mathcal{S}$. The size of $\mathcal{S}'$ is at most twice that of $\mathcal{S}$.*

*Proof* We begin by using Lemma 6.8 to construct $\bar{\mathcal{S}}$ with access structure $\bar{\mathcal{F}}$.

We can now start describing the desired new scheme $\mathcal{S}'$. As a first step consider a secret sharing scheme where we share a secret $s$ according to both $\mathcal{S}$ and $\bar{\mathcal{S}}$. More formally, we generate vectors $\mathbf{r}_s$ and $\bar{\mathbf{r}}_s$, at random but with $s$ as first coordinate, compute $M \cdot \mathbf{r}_s$, $\bar{M} \cdot \bar{\mathbf{r}}_s$ and distribute shares according to the labeling function. Now, from Exercise 6.4, we know that $\bar{\mathcal{F}} \subseteq \mathcal{F}$ and this implies that the access structure of our new scheme is $\mathcal{F}$.

Now assume we have shared secrets $a, b$ according to this scheme. In particular this means that players hold $M \cdot \mathbf{r}_a$ and $\bar{M} \cdot \bar{\mathbf{r}}_b$. Note that because we use the same labeling function for $M$ and $\bar{M}$, the entries in the vector $(M \cdot \mathbf{r}_a) * (\bar{M} \cdot \bar{\mathbf{r}}_b)$ can be computed locally by the players. We claim that the sum of all these entries is actually $ab$. Since the sum of the entries in the $*$-product is just the inner product, this can be seen from:

$$(M \cdot \mathbf{r}_a)^{\mathsf{T}} \cdot (\bar{M} \cdot \bar{\mathbf{r}}_b) = \mathbf{r}_a^{\mathsf{T}} \cdot (M^{\mathsf{T}} \cdot \bar{M}) \cdot \bar{\mathbf{r}}_b = \mathbf{r}_a^{\mathsf{T}} \cdot E \cdot \bar{\mathbf{r}}_b = ab \tag{6.4}$$

by Lemma 6.8

We can now finally specify $\mathcal{S}'$ by directly constructing its matrix $M'$, which will have $2m$ rows and $2(t+1)$ columns. First build a matrix $M''$ filled in with $M$ in the upper left corner and $\bar{M}$ in the lower right corner. Let $\mathbf{k}$ be the column in $M''$ that passes through the first column of $\bar{M}$. Add $\mathbf{k}$ to the first column of $M''$ and delete $\mathbf{k}$ from $M''$. Let $M'$ be the result of this. The labeling of $M'$ is carried over in the natural way from $M$ and $\bar{M}$.

It is now clear that secret sharing according to $\mathcal{S}'$ will generate shares from both $\mathcal{S}$ and $\bar{\mathcal{S}}$ as described above and hence by (6.4), the set of all players will be qualified in $\hat{\mathcal{S}}'$. □

We end this section by showing that multiplicative schemes can only exist for $Q2$ structures:

**Theorem 6.10** *Let $\mathcal{S}$ be a multiplicative linear secret sharing scheme. Then the adversary structure for $\mathcal{S}$ is $Q2$.*

*Proof* Suppose for contradiction that the adversary structure $\mathcal{A}$ is not $Q2$. Then there are two sets $A_1, A_2 \in \mathcal{A}$ such that $A_1 \cup A_2 = \mathsf{P}$. By Lemma 6.2 there exists vectors $\mathbf{w}_1, \mathbf{w}_2$ with $\mathbf{w}_1 \in Ker(M_{A_1})$ and $\mathbf{w}_2 \in Ker(M_{A_2})$ and with non-zero first coordinates, that we can assume without loss of generality are both 1. We can think of $M \cdot \mathbf{w}_1$ and $M \cdot \mathbf{w}_2$ as two sets of shares of the secret 1. Hence by the multiplicative property we should be able to reconstruct the product 1 as a linear combination of the entries in $(M \cdot \mathbf{w}_1) \odot (M \cdot \mathbf{w}_2)$. But since $(M \cdot \mathbf{w}_1)_{A_1}$ and $(M \cdot \mathbf{w}_2)_{A_2}$ are both 0 and $A_1 \cup A_2 = \mathsf{P}$, $(M \cdot \mathbf{w}_1) \odot (M \cdot \mathbf{w}_2)$ is in fact the all-0 vector and we have a contradiction. □

**Exercise 6.5** Let $\mathcal{S}$ be a strongly multiplicative linear secret sharing scheme. Show that the adversary structure for $\mathcal{S}$ is $Q3$.

An obvious question is whether a result similar to Theorem 6.9 holds for strongly multiplicative secret sharing, i.e., from a linear secret sharing scheme for a $Q3$ adversary structure, can one construct a strongly multiplicative scheme of similar size? This question is wide open, it has resisted attempts to solve it for more than a decade, and no clear evidence is known either way as to what the answer might be.

Protocol GCEPS (General Circuit Evaluation with Passive Security)

The protocol proceeds in three phases: the input sharing, computation and output reconstruction phases.

**Input Sharing:** Each player $\mathsf{P}_i$ holding input $x_i \in \mathbb{F}$ distributes $[x_i; \mathbf{r}_{x_i}]_{\mathcal{S}}$.

We then go through the circuit and process the gates one by one in the computational order defined above. Just after the input sharing phase, we consider all input gates as being processed. We will maintain the following:

**Invariant:** Recall that computing with the circuit on inputs $x_1, ..., x_n$ assigns a unique value to every wire. Consider an input or an output wire for any gate, and let $a \in \mathbb{F}$ be the value assigned to this wire. Then, if the gate has been processed, the players hold $[a; \mathbf{r}_a]_{\mathcal{S}}$ for some vector $\mathbf{r}_a$.

We then continue with the last two phases of the protocol:

**Computation Phase:** Repeat the following until all gates have been processed (then go to the next phase): Consider the first gate in the computational order that has not been processed yet. According to the type of gate, do one of the following

**Addition gate:** The players hold $[a; \mathbf{r}_a]_{\mathcal{S}}$, $[b; \mathbf{r}_b]_{\mathcal{S}}$. for the two inputs $a, b$ to the gate. The players compute $[a; \mathbf{r}_a]_{\mathcal{S}} + [b; \mathbf{r}_b]_{\mathcal{S}} = [a + b; \mathbf{r}_a + \mathbf{r}_b]_{\mathcal{S}}$.

**Multiply-by-constant gate:** The players hold $[a; \mathbf{r}_a]_{\mathcal{S}}$ for the inputs $a$ to the gate. The players compute $\alpha[a; \mathbf{r}_a]_{\mathcal{S}} = [\alpha a; \alpha \mathbf{r}_a]_{\mathcal{S}}$.

**Multiplication gate:** The players hold $[a; \mathbf{r}_a]_{\mathcal{S}}$, $[b; \mathbf{r}_b]_{\mathcal{S}}$ for the two inputs $a, b$ to the gate.

1. The players compute $[a; \mathbf{r}_a]_{\mathcal{S}} \odot [b; \mathbf{r}_b]_{\mathcal{S}} = [ab; \mathbf{r}_a \otimes \mathbf{r}_b]_{\hat{\mathcal{S}}}$
2. Define $\mathbf{s} \overset{\text{def}}{=} \mathbf{r}_a \otimes \mathbf{r}_b$. Then the parties hold $[ab; \mathbf{s}]_{\hat{\mathcal{S}}}$. Let $[ab; \mathbf{s}]_{\hat{\mathcal{S}}} = (c_1, ...., c_{\hat{m}})$. For each $\mathsf{P}_i$ and each $c_j$ he owns, $\mathsf{P}_i$ distributes $[c_j; \mathbf{r}_{c_j}]_{\mathcal{S}}$.
3. Let $\mathbf{u} = (u_1, ..., u_{\hat{m}})$ be the reconstruction vector defined above and guaranteed by the fact that $\mathcal{S}$ is multiplicative. The players compute

$$\sum_j u_j [c_j; \mathbf{r}_{c_j}]_{\mathcal{S}} = [\sum_j u_j c_j; \sum_j r_j \mathbf{r}_{c_j}]_{\mathcal{S}} = [\sum_j ab; \sum_j u_j \mathbf{r}_{c_j}]_{\mathcal{S}} \ .$$

**Output Reconstruction:** At this point all gates, including the output gates have been processed. So do the following for each output gate (labeled $i$): The players hold $[y; \mathbf{r}_y]_{\mathcal{S}}$ where $y$ is the value assigned to the output gate. Each $\mathsf{P}_j$ securely sends every share he holds of $y$ to $\mathsf{P}_i$, who can then compute $y = \omega_P \cdot [y; \mathbf{r}_y]_{\mathcal{S}}$, where $\omega_P$ is the reconstruction vector for the set of all players $P$.

## 6.4 A Passively Secure Protocol

In this section we show a natural generalization of the passively secure protocol, we have seen earlier, Protocol CEPS (Circuit Evaluation with Passive Security). We basically replace polynomials by linear secret sharing and do "the same thing". First a definition:

**Definition 6.6** *When we say that a player $P_i$ distributes $[s; \mathbf{r}_s]_{\mathcal{S}}$, we means that he calculates $[s; \mathbf{r}_s]_{\mathcal{S}}$ as described above and distributes the shares to the players according to the labelling function $\phi$. Whenever players have obtained shares of a value $a$, based on polynomial $\mathbf{r}_a$, we say that the players hold $[s; \mathbf{r}_s]_{\mathcal{S}}$.*

*Suppose the players hold $[a; \mathbf{r}_a]_{\mathcal{S}}$, $[b; \mathbf{r}_b]_{\mathcal{S}}$. Then, when we say that the players compute $[a; \mathbf{r}_a]_{\mathcal{S}} + [b; \mathbf{r}_b]_{\mathcal{S}}$, this means that each player $P_i$ computes the sum of corresponding shares of $a$ and $b$ that he holds, and by Lemma 6.6, this means that the players now hold $[a+b; \mathbf{r}_a + \mathbf{r}_b]_{\mathcal{S}}$. In a similar way we define what it means for the players to compute $[a; \mathbf{r}_a]_{\mathcal{S}} \odot [b; \mathbf{r}_b]_{\mathcal{S}}$ and $\alpha \cdot [a; \mathbf{r}_a]_{\mathcal{S}}$.*

We can now describe the protocol for securely evaluating an arithmetic circuit using a general linear secret-sharing scheme $\mathcal{S}$, see Protocol GCEPS (General Circuit Evaluation with Passive Security). The only condition we need is that $\mathcal{S}$ is multiplicative.

We can then show the following result, where we let $\pi^f_{\mathtt{GCEPS}}$ denote the GCEPS-protocol instantiated for computing and arithmetic circuit for the function $f$, and $\text{Env}^{\mathtt{sync,A}}$ denote environments that corrupt subsets in adversary structure $\mathcal{A}$. We do not give the proof in detail here. It is very easy to derive from the corresponding proof for $\pi^f_{\mathtt{CEPS}}$, found in Example 4.10. As in that proof, the simulator runs the protocol with the environment, while running internally copies of the honest players using dummy inputs. Since the environment only sees shares from a set of players in $\mathcal{A}$, this makes no difference. In the final output step, the simulator patches the sets of shares of outputs known by the environment to match the real outputs. This is easy by adding an appropriate multiple of the vector $\mathbf{w}$ constructed in the proof of Theorem 6.3. Likewise, adaptive security follows because correct views for newly corrupted honest players can be constructed by patching the dummy views in the same way.

**Theorem 6.11** *$\pi^f_{GCEPS} \diamond F_{SC}$ implements $F^f_{SFE}$ in $\text{Env}^{sync,A}$ with perfect security, whenever the protocol is based on a multiplicative linear secret sharing scheme $\mathcal{S}$ with adversary structure $\mathcal{A}$.*

It is easy to see that to have security against $\text{Env}^{\mathtt{sync,A}}$ we do not actually need to have a linear scheme $\mathcal{S}$ whose adversary structure exactly equals $\mathcal{A}$, it is enough if it contains $\mathcal{A}$. We still have to demand, however, that the adversary structure of $\mathcal{S}$ be $Q2$, since otherwise we cannot compute all functions securely, by Theorem 6.1. This implies that $\mathcal{A}$ must be $Q2$ as well. Therefore by invoking also Theorem 6.9, we get:

**Corollary 6.12** *Any function can be computed with perfect passive security against an environment corrupting subsets in adversary structure $\mathcal{A}$, provided $\mathcal{A}$ is $Q2$. The complexity of the protocol can be made polynomial in the size of the smallest linear secret-sharing scheme whose adversary structure is $Q2$ and contains $\mathcal{A}$.*

## 6.5 Actively Secure Protocols

In this section we show how to get actively secure protocols from general linear secret-sharing. We do not give all details, as many of them are trivial to derive from the corresponding results and protocols in Chapter 5.

We will first assume that we are given the functionality $\mathsf{F_{COM}}$ for homomorphic commitment, show how to evaluate functions securely from this and then consider how to implement commitments based on linear secret sharing.

Protocol GCEAS (Generalized Circuit Evaluation, Active Security)

The protocol assumes a strongly multiplicative linear secret sharing scheme $\mathcal{S}$ with adversary structure $\mathcal{A}$ and proceeds in three phases: the input sharing, computation and output reconstruction phases.

**Input Sharing:** Each player $\mathsf{P}_i$ holding input $x_i \in \mathbb{F}$ distributes $[\![x_i; \mathbf{r}_{x_i}]\!]_{\mathcal{S}}$. If this fails, $\mathsf{P}_i$ is corrupt, so we will instead assign 0 as default input for $\mathsf{P}_i$, by distributing $[\![0; \mathbf{r}_0]\!]_{\mathcal{S}}$ for a fixed default value of $\mathbf{r}_0$, say the all-0 vector. This can be done by a number of public commit to 0's.

We then go through the circuit and process the gates one by one in the computational order. Just after the input sharing phase, we consider all input gates as being processed. We will maintain the following:

**Invariant:** Consider an input or an output wire for any gate, and let $a \in \mathbb{F}$ be the value assigned to this wire. Then, if the gate has been processed, the players hold $[\![a; \mathbf{r}_a]\!]_{\mathcal{S}}$.

We then continue with the last two phases of the protocol:

**Computation Phase:** Repeat the following until all gates have been processed (then go to the next phase): Consider the first gate in the computational order that has not been processed yet. According to the type of gate, do one of the following

**Addition gate:** The players hold $[\![a; \mathbf{r}_a]\!]_{\mathcal{S}}, [\![b; \mathbf{r}_b]\!]_{\mathcal{S}}$ for the two inputs $a, b$ to the gate. The players compute $[\![a + b; \mathbf{r}_a + f_b]\!]_{\mathcal{S}} = [\![a; \mathbf{r}_a]\!]_{\mathcal{S}} + [\![b; \mathbf{r}_b]\!]_{\mathcal{S}}$.

**Multiply-by-constant gate:** The players hold $[\![a; \mathbf{r}_a]\!]_{\mathcal{S}}$ for the input $a$ to the gate. The players compute $[\![\alpha a; \alpha f_a]\!]_{\mathcal{S}} = \alpha [\![a; \mathbf{r}_a]\!]_{\mathcal{S}}$.

**Multiplication gate:** The players hold $[\![a; \mathbf{r}_a]\!]_{\mathcal{S}}, [\![b; \mathbf{r}_b]\!]_{\mathcal{S}}$ for the two inputs $a, b$ to the gate.

1. The players compute $[\![ab; \mathbf{r}_a \otimes \mathbf{r}_b]\!]_{\hat{\mathcal{S}}} = [\![a; \mathbf{r}_a]\!]_{\mathcal{S}} \odot [\![b; \mathbf{r}_b]\!]_{\mathcal{S}}$. Note that this involves each $\mathsf{P}_j$ committing to his local products and proving that this was done correctly. So this may fail for some subset of players $A \in \mathcal{A}$.
2. Define $\mathbf{s} \overset{\text{def}}{=} \mathbf{r}_a \otimes \mathbf{r}_b$. Then the parties hold $[\![ab; \mathbf{s}]\!]_{\hat{\mathcal{S}}}$. Let $[\![ab; \mathbf{s}]\!]_{\hat{\mathcal{S}}} = (c_1, ...., c_{\hat{m}})$. For each $\mathsf{P}_i$ and each $c_j$ he owns, he distributes $[\![c_j; \mathbf{r}_{c_j}]\!]_{\mathcal{S}}$ from $\langle c_j \rangle_i$. Also this step may fail for a subset $A \in \mathcal{A}$.
3. Let $S$ be the indices of at the players for which the previous two steps did not fail. By strong multiplicativity of $\mathcal{S}$, there exists $\mathbf{r}_S$, a recombination vector for $S$, that is, a vector $\mathbf{r}_S = (r_1, \ldots, r_{\hat{m}})$ that players in $S$ can use to reconstruct secrets generated by $\hat{\mathcal{S}}$. In particular, we have $ab = \sum_{\phi(j) \in S} r_j c_j$. The players compute

$$\sum_{\phi(j)\in S} r_j [\![c_j; \mathbf{r}_{c_j}]\!]_{\mathcal{S}} = [\![ \sum_{\phi(j)\in S} r_j c_j; \sum_{\phi(j)\in S} r_j \mathbf{r}_{c_j}]\!]_{\mathcal{S}} = [\![ab; \sum_{\phi(j)\in S} r_j \mathbf{r}_{c_j}]\!]_{\mathcal{S}} \ .$$

**Output Reconstruction:** At this point all gates, including the output gates have been processed. So do the following for each output gate (labeled $i$): The players hold $[\![y; f_y]\!]_{\mathcal{S}}$ where $y$ is the value assigned to the output gate. Now we open $[\![y; f_y]\!]_{\mathcal{S}}$ towards $\mathsf{P}_i$.

We first define some notation. Let $\mathcal{S}$ be a linear secret sharing scheme, and recall that for an appropriately chosen vector $\mathbf{r}_a$ with $a$ the first coordinate, $g(\mathbf{r}_a)$ is a vector of shares of $a$. Also recall that $\phi$ is the labeling function that tells us which players receive the individual shares. Then we define

$$[\![a;\mathbf{r}_a]\!]_{\mathcal{S}} = (\langle g(\mathbf{r}_a)_1\rangle_{\phi(1)}, ..., \langle g(\mathbf{r}_a)_m\rangle_{\phi(m)}) \ .$$

This is just the natural generalization of the concept we defined earlier: the secret $a$ has been shared and players are committed to the shares they hold. We can then also define

$$[\![a;\mathbf{r}_a]\!]_{\mathcal{S}} + [\![b;\mathbf{r}_b]\!]_{\mathcal{S}},\ \ \alpha[\![a;\mathbf{r}_a]\!]_{\mathcal{S}},\ \ [\![a;\mathbf{r}_a]\!]_{\mathcal{S}} \odot [\![b;\mathbf{r}_b]\!]_{\mathcal{S}}$$

in the natural way, and it then immediately follows that we have

**Lemma 6.13** *The following holds for any $a, b, \alpha \in \mathbb{F}$ and any multiplicative linear secret-sharing scheme $\mathcal{S}$:*

$$[\![a;\mathbf{r}_a]\!]_{\mathcal{S}} + [\![b;\mathbf{r}_b]\!]_{\mathcal{S}} = [\![a+b;\mathbf{r}_a+\mathbf{r}_b]\!]_{\mathcal{S}}, \tag{6.5}$$

$$\alpha[\![a;\mathbf{r}_a]\!]_{\mathcal{S}} = [\![\alpha a;\alpha\mathbf{r}_a]\!]_{\mathcal{S}}, \tag{6.6}$$

$$[\![a;\mathbf{r}_a]\!]_{\mathcal{S}} \odot [\![b;\mathbf{r}_b]\!]_{\mathcal{S}} = [\![ab;\mathbf{r}_a\otimes\mathbf{r}_b]\!]_{\hat{\mathcal{S}}} \tag{6.7}$$

Similarly to what we saw in Chapter 5, when we say that $\mathsf{P}_i$ distributes $[\![a;\mathbf{r}_a]\!]_{\mathcal{S}}$, we mean that $\mathsf{P}_i$ commits to all entries in $\mathbf{r}_a$, we then use the `add`, `mult` commands to compute commitments to the shares in $a$ and finally use `transfer` commands to ensure that the party holding a share is committed to that share.

It is now straightforward to show

**Theorem 6.14** *$\pi^f_{\mathsf{GCEAS}} \diamond \mathsf{F}_{\mathsf{SC}} \diamond \mathsf{F}_{\mathsf{COM}}$ implements $\mathsf{F}^f_{\mathsf{SFE}}$ in $\mathrm{Env}^{\mathcal{A},sync}$ with perfect security for any $f$, whenever the protocol is based on a strongly multiplicative linear secret sharing scheme $\mathcal{S}$ with adversary structure $\mathcal{A}$.*

We note that it is possible to modify Protocol GCEAS to make it work also for a $Q2$ adversary structure and a multiplicative scheme $\mathcal{S}$. As in Chapter 5, the difficulty in this case is that if a player fails during processing of a multiplication gate, we cannot proceed since all players are needed to reconstruct in $\hat{\mathcal{S}}$ because do not assume strong multiplication. We can handle such failures by going back to the start of the computation and open the input values of the players that have just been disqualified. Now we restart the protocol, while the honest players simulate the disqualified players. Note that we can use default random choices for these players, so there is no communication needed for the honest players to agree on the actions of the simulated ones. This allows the adversary to slow down the protocol by a factor at most linear in $n$. In doing this, it is important that we do not have to restart the protocol after some party received its output — this is ensured by starting the Output Reconstruction phase only after all multiplication gates have been handled.

Let $\pi^f_{\mathtt{GCEAS\text{-}Q2}}$ be $\pi^f_{\mathtt{GCEAS}}$ modified to handle $Q2$-structures as we just discussed. We then have the following theorem, the proof of which we leave to the reader:

**Theorem 6.15** *$\pi^f_{\mathsf{GCEAS\text{-}Q2}} \diamond \mathsf{F}_{\mathsf{SC}} \diamond \mathsf{F}_{\mathsf{COM}}$ implements $\mathsf{F}^f_{\mathsf{SFE}}$ in $\mathrm{Env}^{\mathcal{A},sync}$ with perfect security for any $f$ when based on a multiplicative secret sharing scheme with adversary structure $\mathcal{A}$.*

### 6.5.1 Implementation of Homomorphic Commitment from Linear Secret Sharing

Let us first consider how we can implement $\mathsf{F}_{\mathtt{COM}}$ from $\mathsf{F}_{\mathtt{COM\text{-}SIMPLE}}$. Note first that from the results in Chapter 5, we already have

Protocol Generalized Perfect Commitment Multiplication

This protocol is based on a strongly multiplicative secret sharing scheme $\mathcal{S}$. If any command used during this protocol fails, all players output `fail`.

1. $\langle c \rangle_i \leftarrow ab$.

2. $\mathsf{P}_i$ chooses vectors $\mathbf{r}_a, \mathbf{r}_b$ with coordinates $a, \alpha_1, ..., \alpha_t$ and $b, \beta_1, ..., \beta_t$, for random $\alpha_j, \beta_j$. He computes $\mathbf{r}_c = \mathbf{r}_a \otimes \mathbf{r}_b$, let $c, \gamma_1, ..., \gamma_{(t+1)^2} - 1$ be the coordinates of $\mathbf{r}_c$. Then establish commitments as follows:

$$\begin{array}{rcl} \langle \alpha_j \rangle_i & \leftarrow & \alpha_j \text{ for } j = 1...t \\ \langle \beta_j \rangle_i & \leftarrow & \beta_j \text{ for } j = 1...t \\ \langle \gamma_j \rangle_i & \leftarrow & \gamma_j \text{ for } j = 1...(t+1)^2 - 1 \end{array}$$

3. Recall that the matrices of $\mathcal{S}, \hat{\mathcal{S}}$ are called $M, \hat{M}$. Then, for $k = 1...m$ the `add` and `mult` commands are invoked to compute:

$$\begin{array}{rcl} \langle a_k \rangle_i & = & M_{k,0} \langle a \rangle_i + M_{k,1} \langle \alpha_1 \rangle_i + ... + M_{k,t} \langle \alpha_t \rangle_i \\ \langle b_k \rangle_i & = & M_{k,0} \langle b \rangle_i + M_{k,1} \langle \beta_1 \rangle_i + ... + M_{k,t} \langle \beta_t \rangle_i \end{array}$$

   and for $k = 1...\hat{m}$ we compute:

$$\langle c_k \rangle_i = \hat{M}_{k,0} \langle b \rangle_i + \hat{M}_{k,1} \langle \gamma_1 \rangle_i + ... + \hat{M}_{k,(t+1)^2-1} \langle \gamma_{2t} \rangle_i$$

   Note that by this computation we have that $a_k$ is the $k$'th share in $a$, i.e., $a_k = \mathbf{m}_k \cdot \mathbf{r}_a$, similarly $b_k, c_k$ are the $k$'th shares in $b$ and $c$, respectively. Then for all $k$, we execute $(\mathsf{P}_{\phi(k)}) a_k \leftarrow \langle a_k \rangle_i$, $(\mathsf{P}_{\phi(k)}) b_k \leftarrow \langle b_k \rangle_i$ and $(\mathsf{P}_{\phi(k)}) c_k \leftarrow \langle c_k \rangle_i$.

4. For $k = 1...\hat{m}$, do: $\mathsf{P}_{\phi(k)}$ checks that $a_u b_v = c_k$ for the appropriate $u, v$ with $\phi(u) = \phi(v) = \phi(k)$ (see Lemma 6.4). He broadcasts `accept` if this is the case, else he broadcasts `reject` $k$.

5. For each $\mathsf{P}_{\phi(k)}$ who said `reject` $k$, do $a_u \leftarrow \langle a_u \rangle_i$, $b_v \leftarrow \langle b_v \rangle_i$ and $c_k \leftarrow \langle c_k \rangle_i$, all players check whether $a_u b_v = c_k$ holds. If this is not the case, all players output `fail`.

6. If we arrive here, no command failed during the protocol and all relations $a_u b_v = c_k$ that were checked, hold. All players output `success`.

**Theorem 6.16** *There exists a protocol $\pi_{\texttt{TRANSFER,GMULT}}$ such that $\pi_{\texttt{TRANSFER,MULT}} \diamond \mathsf{F}_{\texttt{SC}} \diamond \mathsf{F}_{\texttt{COM-SIMPLE}}$ implements $\mathsf{F}_{\texttt{COM}}$ in* $\text{Env}^{t,sync}$ *with statistical security for all $t < n$.*

This is just a restatement of part of Theorem 5.1. The result can be used in this context because $\pi_{\texttt{TRANSFER,GMULT}}$ only uses commands from $\mathsf{F}_{\texttt{COM-SIMPLE}}$ and no secret-sharing, and because tolerating any number of corruptions less than $n$ is of course good enough to handle any non-trivial adversary structure. Furthermore, we also have

**Theorem 6.17** *There exists a protocol $\pi_{\texttt{GPTRANSFER,GPMULT}}$ such that $\pi_{\texttt{PTRANSFER,PMULT}} \diamond \mathsf{F}_{\texttt{SC}} \diamond \mathsf{F}_{\texttt{COM-SIMPLE}}$ implements $\mathsf{F}_{\texttt{COM}}$ in* $\text{Env}^{\mathcal{A},sync}$ *with perfect security when based on a strongly multiplicative secret sharing scheme $\mathcal{S}$ with adversary structure $\mathcal{A}$.*

The idea for constructing $\pi_{\texttt{GPTRANSFER,GPMULT}}$ is to start from $\pi_{\texttt{PTRANSFER,PMULT}}$, replace secret-sharing based on polynomials by $\mathcal{S}$ and otherwise do "the same". We show here how the multiplication proof can be done in this way. We argue informally why Protocol Generalized Perfect Commitment Multiplication is secure and leave a formal simulation proof to the reader:

If the prover $P_i$ is honest, the adversary gets no information on $a, b$: some values $a_k, b_k, c_k$ are revealed to corrupt players, but the corrupted set is in the adversary structure so the $a_k, b_k$ give no information and $c_k$ always equals $a_k b_k$. Furthermore, all honest players will say `accept` after the check, so only values the adversary already knows are broadcast. If $P_i$ is corrupt, note that the computation on commitments ensures that $\{a_k\} = [a, \mathbf{r}_a]_{\mathcal{S}}, \{b_k\} = [b, \mathbf{r}_b]_{\mathcal{S}}, \{c_k\} = [c, \mathbf{r}_c]_{\hat{\mathcal{S}}}$ are correct sets of shares computed under $\mathcal{S}, \mathcal{S}$ and $\hat{\mathcal{S}}$, respectively, and they therefore determine well defined secrets $a, b, c$. Furthermore the set of shares $[a, \mathbf{r}_a]_{\mathcal{S}} \odot [b, \mathbf{r}_b]_{\mathcal{S}}$ is a valid set of secrets under $\hat{\mathcal{S}}$ and determine the secret $ab$. By the checks done we know that if `success` is output the sets of shares $[a, \mathbf{r}_a]_{\mathcal{S}} \odot [b, \mathbf{r}_b]_{\mathcal{S}}$ and $[c, \mathbf{r}_c]_{\hat{\mathcal{S}}}$ agree in all entries owned by honest players. But the set of honest players is the complement of a set in the adversary structure of $\mathcal{S}$, it is therefore in the access structure of $\hat{\mathcal{S}}$ by the strong multiplication property, hence it is large enough to determine a secret in $\hat{\mathcal{S}}$ uniquely. It follows that $c = ab$ as desired.

**Exercise 6.6** Construct the rest of protocol $\pi_{\texttt{GPTRANSFER,GPMULT}}$, that is, construct a perfectly secure protocol for transferring a commitment from one player to another based on a linear secret sharing scheme $\mathcal{S}$ and use this to prove Theorem 6.17.

We now turn to the question of how to implement $\mathsf{F}_{\texttt{COM-SIMPLE}}$ from linear secret sharing. It is of course not surprising that the solution is similar to what we have seen in Chapter 5, but some issues need to be addressed.

Note first that if players hold $[s, \mathbf{r}_s]_{\mathcal{S}}$, the adversary structure $\mathcal{A}$ is $Q3$, and players broadcast their shares, then the correct value of $s$ is uniquely determined: we can in principle simply search for some $s', \mathbf{r}_{s'}$ such that $[s', \mathbf{r}_{s'}]_{\mathcal{S}}$ agrees with the set of shares broadcast except in some set of entries owned by some unqualified set $A' \in \mathcal{A}$. We claim that then $s'$ always equals the correct secret $s$. The set $A'$ may not be the set of players that are actually corrupted, but this does not matter: if we start from the set of all shares and then take away entries owned by corrupted players (say in set $A$) and also take away all entries in $A'$, then $[s', \mathbf{r}_{s'}]_{\mathcal{S}}$ agrees with $[s, \mathbf{r}_s]_{\mathcal{S}}$ in the remaining entries since these all correspond to honest players. This set of entries is in the access structure of $\mathcal{S}$ by the $Q3$ property so $s = s'$ as desired. On the other hand, the search will always stop because $(s', \mathbf{r}_{s'}) = (s, \mathbf{r}_s)$ is of course a good solution. Note that if this is used as a commitment scheme, players would not have to do the search (this would be inefficient in most cases), we could just ask the committer to point out the right solution.

To make a perfect homomorphic commitment scheme, it is therefore sufficient to force a committer to hand out consistent shares of the value he has in mind, i.e., we want something of form $[a; \mathbf{r}_a]_{\mathcal{S}}$. This ensures privacy against corruption of any unqualified set, and if we require the committer to reveal $\mathbf{r}_a$ at opening time it follows from what we just saw that he cannot change to a different value of $a$.

To enforce consistency, we reuse the approach from earlier and have the committer secret share his value but then also secret-share the randomness he used for this. This introduces the possibility of other players cross-checking what they have, ultimately enforcing consistency. We can phrase this as a redundant version of the original linear secret sharing scheme, similar to the bivariate polynomials we saw earlier.

Therefore, to commit to a value $a \in \mathbb{F}$, based on secret sharing scheme $\mathcal{S}$, $\mathsf{P}_i$ chooses a random symmetric $(t+1) \times (t+1)$ matrix $R_a$ with $a$ in the upper left corner. Then, for each index $k$, he sends $\mathbf{u}_k = R_a \cdot \mathbf{m}_k^{\mathsf{T}}$ to $\mathsf{P}_{\phi(k)}$ (recall that $\mathbf{m}_k$ is a row vector taken from the matrix of $\mathcal{S}$).

If we consider only the product of the first row of $R_a$ with $\mathbf{m}_k^{\mathsf{T}}$, this computes a normal share of $a$ according to $\mathcal{S}$, so the share $\beta_k$ of $a$ is defined to be the first entry in $\mathbf{u}_k$. Note that for any pair of indices $k, j$ we have (since $R_a$ is symmetric):

$$\mathbf{m}_j \cdot \mathbf{u}_k = \mathbf{m}_j \cdot R_a \cdot \mathbf{m}_k^\mathsf{T} = (R_a \cdot \mathbf{m}_j^\mathsf{T})^\mathsf{T} \cdot \mathbf{m}_k^\mathsf{T} = \mathbf{m}_k \cdot \mathbf{u}_j$$

so if we define $\beta_{k,j} = \mathbf{m}_j \cdot \mathbf{u}_k = \mathbf{m}_k \cdot \mathbf{u}_j$, this is a value that both $\mathsf{P}_{\phi(k)}$ and $\mathsf{P}_{\phi(j)}$ can compute, so this can be used as a consistency check on the information $\mathsf{P}_i$ distributes. We have

**Lemma 6.18** *If honest player $\mathsf{P}_i$ choses $R_a$ and sends $\mathbf{u}_k = R_a \cdot \mathbf{m}_k^\mathsf{T}$ to $\mathsf{P}_{\phi(k)}$ then the information given to any unqualified set $A$ has distribution independent of $a$.*

*Proof* The information given to the set $A$ can be written as $M_A \cdot R_a$. Since $A$ is unqualified, there exists a vector $\mathbf{w}$ that has first coordinate 1 and is orthogonal to all rows in $M_A$. Now, if we interpret the tensor product $\mathbf{w} \otimes \mathbf{w}$ as a matrix in the natural way, this matrix is symmetric and has 1 in the upper left corner. Therefore, if we define $R_0 = R_a - a \cdot (\mathbf{w} \otimes \mathbf{w})$, then $R_0$ is symmetric and has 0 in the upper left corner. Furthermore $M_A \cdot R_a = M_A \cdot R_0$. So for any set of shares that $A$ might receive, the number of random choices that could lead to these shares is the same for any value of $a$, namely the number of possible choices for $a = 0$. The lemma follows. □

Protocol Generalized Commit

1. On input $(\texttt{commit}, i, cid, a)$, $\mathsf{P}_i$ chooses $R_a$ and sends $\mathbf{u}_k = R_a \cdot \mathbf{m}_k^\mathsf{T}$ to $\mathsf{P}_{\phi(k)}$ as described in the text. $\mathsf{P}_{\phi(k)}$ sets $\beta_k$ to be the first entry in $\mathbf{u}_k$.
2. For each $j$, $\mathsf{P}_{\phi(j)}$ computes $\beta_{k,j} = \mathbf{m}_k \cdot \mathbf{u}_j$ and sends $\beta_{k,j}$ to $\mathsf{P}_{\phi(k)}$.
3. Each $\mathsf{P}_{\phi(k)}$ checks that $\beta_{k,j} = \mathbf{m}_j \cdot \mathbf{u}_k$. If so, it broadcasts `success`. Otherwise, it broadcasts $(\texttt{dispute}, k, j)$ for each inconsistency.
4. For each dispute reported in the previous step, $\mathsf{P}_i$ broadcasts the correct value of $\beta_{k,j}$.
5. If any $\mathsf{P}_{\phi(k)}$ finds a disagreement between what $\mathsf{P}_i$ has broadcast and what he received privately from $\mathsf{P}_i$, he knows $\mathsf{P}_i$ is corrupt and broadcasts $(\texttt{accuse}, k)$.
6. For any accusation from $\mathsf{P}_{\phi(k)}$ in the previous step, $\mathsf{P}_i$ broadcasts all vectors he sent to $\mathsf{P}_{\phi(k)}$.
7. If any $\mathsf{P}_k$ finds a new disagreement between what $\mathsf{P}_i$ has now broadcast and what he received privately from $\mathsf{P}_i$, he knows $\mathsf{P}_i$ is corrupt and broadcasts $(\texttt{accuse}, k)$.
8. If the information broadcast by $\mathsf{P}_i$ is not consistent, or if a set of players not in the adversary structure has accused $\mathsf{P}_i$, players output `fail`.

   Otherwise, players who accused $\mathsf{P}_i$ and had new vectors broadcast will accept these. All others keep the vectors they received in the first step. Now, for each $j$, $\mathsf{P}_{\phi(j)}$ stores $(cid, i, \beta_j)$. In addition $\mathsf{P}_i$ stores the first row of $R_a$. All players output `success`.

Now, to show the security of Protocol Generalized Commit, we need two main lemmas:

**Lemma 6.19** *If $\mathsf{P}_i$ remains honest throughout Protocol* Generalized Commit, *the view of any set of players $A \in \mathcal{A}$ players is independent of the committed value $a$, and all players who are honest at the end of the protocol will output the shares in $a$ that $\mathsf{P}_i$ distributed.*

This follows immediately from Lemma 6.18 and the fact that if $\mathsf{P}_i$ remains honest, then the values that are broadcast are only values that players in $A$ already know from the first step.

**Lemma 6.20** *If the adversary structure is $Q3$, then no matter how corrupt players behave in Protocol* Generalized Commit, *players who are honest at the end of the protocol will all output* `fail` *or will output a set of shares in some value $a'$ all consistent with a vector $\mathbf{r}_{a'}$.*

*Proof* The decision to output fail is based on public information, so it is clear that players who remain honest will either all output `fail` or all output some value, so assume the latter case occurs. If we take the set of all players and subtract the set $A'$ of players who accused $\mathsf{P}_i$ and also subtract the set of players who are corrupt at the end of the protocol, then since $A, A'$ are unqualified and the adversary structure is $Q3$, the remaining set $S$ must be qualified. Note that $S$ consists of honest players who did not accuse $\mathsf{P}_i$. Let $H$ be the set of all honest players.

Now, for any $j$ consider any player $\mathsf{P}_{\phi(j)} \in H$. We claim that for every $k$ with $\mathsf{P}_{\phi(k)} \in S$ holding $\mathbf{u}_k$ the two players agree on all $\beta_{k,j}$ values they have in common, that is, $\mathbf{m}_k \cdot \mathbf{u}_{\phi(j)} = \mathbf{m}_j \cdot \mathbf{u}_k$.

We argue this as follows: if $\mathsf{P}_{\phi(j)}$ had a new vector broadcast for him in Step 6, then the claim is true, since otherwise $\mathsf{P}_{\phi(k)}$ would have accused in Step 7. On the other hand, suppose no new values were broadcast for $\mathsf{P}_{\phi(j)}$. Then he keeps the values he was given in the first step, as does $\mathsf{P}_{\phi(k)}$. This also means that neither $\mathsf{P}_{\phi(j)}$ nor $\mathsf{P}_{\phi(k)}$ accused in Step 5. But our claim holds for the values sent in Step 1, for if not, a $(\mathtt{dispute}, k, j)$ would have been reported, and either $\mathsf{P}_{\phi(j)}$ or $\mathsf{P}_{\phi(k)}$ would have accused $\mathsf{P}_i$ in Step 5, and this did not happen.

Now form two matrices $U_H, U_S$ where the columns consist of all $\mathbf{u}_k$-vectors held by players in $H$, respectively $S$. If $M$ is the matrix of the secret sharing scheme $\mathcal{S}$, then $M_S \cdot U_S$ is a matrix containing all $\beta_{k,j}$-values that players in $S$ can compute. The claim above on agreement between players in $S$ and all players in $H$ can be compactly written as $M_H \cdot U_S = (M_S \cdot U_H)^{\mathsf{T}}$.

Let $\mathbf{v}$ be the reconstruction vector for $S$ that we know exists by Theorem 6.3. We claim that the vector $\mathbf{r}_{s'} = U_S \cdot \mathbf{v}$ defines the sharing that $\mathsf{P}_i$ has distributed, i.e., as $s'$ we can take the first entry in $\mathbf{r}_{s'}$ and the shares output by players in $H$ indeed form a sharing according to $\mathbf{r}_{s'}$, i.e., $M_H \cdot \mathbf{r}_{s'}$. To show this, we can use the above claim to compute as follows:

$$M_H \cdot \mathbf{r}_{s'} = M_H \cdot U_S \cdot \mathbf{v} = (M_S \cdot U_H)^{\mathsf{T}} \cdot \mathbf{v} = U_H^{\mathsf{T}} \cdot M_S^{\mathsf{T}} \cdot \mathbf{v} = U_H^{\mathsf{T}} \cdot \mathbf{e}$$

where the last step follows from the property the reconstruction vector has (as can be seen in the proof of Theorem 6.3), and $\mathbf{e}$ is the first canonical basis vector. Clearly the entries in the vector $U_H^{\mathsf{T}} \cdot \mathbf{e}$ are the first entries in all the $\mathbf{u}_j$-vectors held by players in $H$, and these entries are exactly what they output as their shares by specification of the protocol. □

Based on what we have seen now it is straightforward to construct a complete protocol $\pi_{\texttt{GEN-PERFECT-COM-SIMPLE}}$ for implementing $\mathsf{F}_{\texttt{COM-SIMPLE}}$ based on a secret sharing scheme $\mathcal{S}$. This, as well as the simulation proof of security, follows the same recipe as we used for the proof of Theorem 5.6. We therefore have

**Theorem 6.21** *$\pi_{\texttt{GEN-PERFECT-COM-SIMPLE}} \diamond \mathsf{F}_{\texttt{SC}}$ implements $\mathsf{F}_{\texttt{COM-SIMPLE}}$ in $\mathrm{Env}^{\mathcal{A},sync}$ with perfect security when based on a linear secret sharing scheme $\mathcal{S}$ with $Q3$ adversary structure $\mathcal{A}$.*

**Exercise 6.7** Construct the rest of protocol $\pi_{\texttt{GEN-PERFECT-COM-SIMPLE}}$ and show Theorem 6.21.

If we only go for statistical security in the end, and assume we have a broadcast functionality given, then assuming access to the $\mathsf{F}_{\texttt{ICSIG}}$ functionality from Section 5.4.1, we can implement $\mathsf{F}_{\texttt{COM-SIMPLE}}$ even for $Q2$ adversary structures. The resulting protocol, $\pi_{\texttt{GEN-COM-SIMPLE}}$ can be constructed in exactly the same way as we did $\pi_{\texttt{COM-SIMPLE}}$ for the threshold case earlier: to commit we do the same as in *Protocol* Generalized Commit, except that the committer must sign everything he sends using $\mathsf{F}_{\texttt{ICSIG}}$. All accusations against the committer can now be proved because the accuser can demonstrate that he indeed received a particular message from the committer. This leads to:

**Theorem 6.22** *$\pi_{\texttt{GEN-COM-SIMPLE}} \diamond \pi_{\texttt{ICSIG}} \diamond \mathsf{F}_{\texttt{SC}}$ implements $\mathsf{F}_{\texttt{COM-SIMPLE}}$ in $\mathrm{Env}^{\mathcal{A},sync}$ with statistical security when based on a linear secret sharing scheme $\mathcal{S}$ with $Q2$ adversary structure $\mathcal{A}$.*

### 6.5.2 General Results on MPC from Linear Secret Sharing

If we combine everything we have seen so far, we obtain a number of results for general adversary structures that are natural generalizations of the results for threshold adversaries.

We first note that broadcast can be implemented securely from secure point-to-point channels, if the actively corrupted set of players is from a $Q3$ adversary structure [FM98]:

**Theorem 6.23** *There exists a protocol $\pi_{\texttt{GEN-BROADCAST}}$ such that $\pi_{\texttt{GEN-BROADCAST}} \diamond \mathsf{F}_{\texttt{PPC}}$ implements $\mathsf{F}_{\texttt{SC}}$ in $\mathrm{Env}^{\mathcal{A},sync}$ with perfect security when $\mathcal{A}$ is $Q3$.*

Then, Theorems 6.17, 6.14, 6.21 and 6.23 imply that any function can be computed with perfect, active security against $Q3$ adversary structures, given access to secure point to point channels. More formally, we have:

**Theorem 6.24** *$\pi^{f}_{\texttt{GCEAS}} \diamond \pi_{\texttt{GPTRANSFER,GPMULT}} \diamond \pi_{\texttt{GEN-PERFECT-COM-SIMPLE}} \diamond \pi_{\texttt{GEN-BROADCAST}} \diamond \mathsf{F}_{\texttt{PPC}}$ implements $\mathsf{F}^{f}_{\texttt{SFE}}$ in $\mathrm{Env}^{\mathcal{A},sync}$ with perfect security when based on a strongly multiplicative linear secret sharing scheme $\mathcal{S}$ with $Q3$ adversary structure $\mathcal{A}$.*

The final result in this chapter concerns the case where we assume access to $\mathsf{F}_{\texttt{SC}}$, i.e., we assume secure channels and broadcast and go for statistical security. In this case, putting previous results together, we get that Theorems 6.17, 6.15 and 6.22 imply that any function can be computed with statistical, active security if less than $n/2$ players are corrupt, given access to secure point to point channels and broadcast. More formally, we have:

**Theorem 6.25** *$\pi^{f}_{\texttt{GCEAS-Q2}} \diamond \pi_{\texttt{TRANSFER,MULT}} \diamond \pi_{\texttt{COM-SIMPLE}} \diamond \mathsf{F}_{\texttt{SC}}$ implements $\mathsf{F}^{f}_{\texttt{SFE}}$ in $\mathrm{Env}^{\mathcal{A},sync}$ with statistical security when based on a multiplicative linear secret sharing scheme $\mathcal{S}$ with $Q2$ adversary structure $\mathcal{A}$.*

These results show what can be done for active security with a given secret sharing scheme. We might also ask: what is the best protocol we can have for a given adversary structure $\mathcal{A}$?

One can first note that in general, it is clearly sufficient for security to come up with a secret sharing scheme $\mathcal{S}$ whose adversary structure *contains* $\mathcal{A}$. The protocol built from $\mathcal{S}$ would then protect against an even stronger adversary.

Now, for perfect security, we know that $\mathcal{A}$ must be $Q3$ to allow any function to be computed with perfect active security, and finally any $Q3$ structure admits a strongly multiplicative secret sharing scheme. Therefore we have:

**Corollary 6.26** *Any function can be computed with perfect active security against an environment corrupting subsets in adversary structure $\mathcal{A}$, provided $\mathcal{A}$ is $Q3$, and we have access to $\mathsf{F}_{\texttt{PPC}}$. The complexity of the protocol can be made polynomial in the size of the smallest strongly multiplicative linear secret-sharing scheme whose adversary structure contains $\mathcal{A}$.*

For the case of statistical security, it is sufficient to have any linear scheme with a $Q2$-adversary structure containing $\mathcal{A}$, by Theorem 6.9. So for this case we get:

**Corollary 6.27** *Any function can be computed with statistical active security against an environment corrupting subsets in adversary structure $\mathcal{A}$, provided $\mathcal{A}$ is $Q2$, and we have access to $\mathsf{F}_{\texttt{SC}}$. The complexity of the protocol can be made polynomial in the size of the smallest linear secret-sharing scheme whose adversary structure contains $\mathcal{A}$.*

One can also note in this last corollary, that if $\mathcal{A}$ happens to be $Q3$ then we can replace access to $\mathsf{F}_{\texttt{SC}}$ by access to $\mathsf{F}_{\texttt{PPC}}$ and simulate broadcast using Theorem 6.23. This gives a statistical version of the first corollary, and this can be significant for efficiency because the smallest linear scheme for an adversary structure may (for all we know) be much smaller than the best strongly multiplicative one.

# Chapter 7

# Cryptographic MPC Protocols

**Contents**

## 7.1 Introduction

The main focus of this book is information theoretic solutions, and therefore we only cover the case of computational security very superficially. Basically, we survey some of the main results and give pointers for further reading.

## 7.2 The Case of Honest Majority

A main completeness result for computationally secure multiparty computation is that any function can be computed with computational security in a model where authenticated point to point channels are given. The adversary may corrupt $t < n/2$ players actively. Recall that computational security is defined just like statistical security, except that the simulator is only required to work for polynomial time bounded environments. More formally, the result is as follows:

**Theorem 7.1** *If non-committing encryption schemes exist, then there exists a protocol* $\pi^f_{\texttt{COMPSEC}}$ *such that* $\pi^f_{\texttt{COMPSEC}} \diamond \mathsf{F}_{\texttt{AT}}$ *implements* $\mathsf{F}^f_{\texttt{SFE}}$ *in* $\mathrm{Env}^{t,\texttt{sync},\ \texttt{poly}}$ *with computational security for all* $t < n/2$.

We will explain what non-committing encryption is in a moment. For now we note that the assumption that $t < n/2$ is necessary to guarantee that the protocol terminates and gives output to honest players. In the 2-party case, for instance, it is clear that we cannot terminate if one of the players simply stops playing. We discuss below what can be done if we do not assume honest majority.

Note that we have already seen *statistically* secure protocols for the case $t < n/2$, but there we had to assume access to secure channels and broadcast ($\mathsf{F}_{\texttt{SC}}$), so the above result says that we can make do with $\mathsf{F}_{\texttt{AT}}$ if we settle for computational security.

The reader might complain about the fact that we assume authenticated channels – if we are willing to assume computationally secure cryptography, we can public-key encrypt and sign, so can we not get a protocol that starts from nothing? This, however, is not possible, at least in many practical scenarios. The issue is that even in a two-player protocol, an honest player would need to make sure he is talking to the other player and not an external adversary, otherwise the protocol would not be secure against an external observer in the case where both players are honest, and this seems to be a property one would often need. We can, however, minimize the use of authenticated channels, namely by using signatures. This will mean that we only need authenticated channels in an initial phase to communicate public keys.

To show Theorem 7.1, one notes that from Theorem 5.13 we already know that we can implement $\mathsf{F}_{\mathsf{SFE}}^{f}$ with statistical security if we are given access to $\mathsf{F}_{\mathsf{SC}}$. It will therefore be sufficient to implement $\mathsf{F}_{\mathsf{SC}}$ given access to $\mathsf{F}_{\mathsf{AT}}$. i.e., implement secure communication based on authenticated communication. This can be done using encryption (because signatures are enough to get the broadcast we also need). But to get adaptive security one needs a solution that is secure even if both players are corrupted at any time. One can do this using so-called non-committing encryption, where the simulator is able to construct "ciphertexts" that are not the result of properly encrypting something, but can later be claimed to contain any plaintext the simulator desires. In this way, the simulator can handle the case where it must simulate a conversation between two honest players, and where one or both of these players is later corrupted. Non-committing encryption from one-way trapdoor permutations with certain extra properties was constructed in [CFGN96], with efficiency improvements under specific assumptions in [DN00]. Informally speaking, what one needs is that one can generate a permutation (a public key) obliviously, that is, while learning nothing about the corresponding trapdoor (secret key). This is also sometimes known as a Simulatable Trapdoor Permutations (STP).

Even the best non-committing encryption schemes known are very inefficient compared to regular public-key encryption, the size of a ciphertext is a factor $\kappa$ larger than the plaintext where $\kappa$ is the security parameter. However, if we are only after static security, any chosen-ciphertext secure (CCA-secure) cryptosystem can be used instead, and this will lead to a much more efficient protocol with current state of the art.

Note also that this approach gives a completely different proof of Theorem 7.1 than the first one given in [GMW87], which only showed static security, assuming existence of one-way trapdoor permutations. The approach used there was to build a protocol for oblivious transfer (OT) from trapdoor permutations, and then base the rest of the protocol on OT and zero-knowledge proofs.

OT is a primitive that one may think of as an ideal functionality that gets two input bits $b_0, b_1$ from a sender and one input bit $c$ from a receiver, and then returns $b_c$ to the receiver. In fact, given only OT as a black box, one can evaluate any function securely for any number of players, this was shown in [Kil88].

It is instructive to note the difference in efficiency between the two approaches we have outlined and where it comes from. The first approach combines a statistically secure protocol $\pi_{stat}$ based on secret sharing with a computationally secure implementation of $\mathsf{F}_{\mathsf{SC}}$. Since $\pi_{stat}$ only requires that we do linear algebra in relatively a small field, it can be implemented very efficienctly, in fact, the complexity can be close to linear in the security parameter. The implementation of $\mathsf{F}_{\mathsf{SC}}$ does require public-key cryptography which is much less efficient, but since we only need the cryptography as a secure transport mechanism, we can combine the public-key systems with much more efficient symmetric cryptosystems, at least if we settle for static security. Such an implementation of $\mathsf{F}_{\mathsf{SC}}$ corresponds closely to what the well known (and very efficient) SSL and TLS protocols offer in practice on the Internet. In contrast, the approach from [GMW87] uses public-key techniques throughout to do the actual computation and is therefore less efficient when used directly. The reader should note, however, that with a more recent technique for implementing the OTs needed in [GMW87], the difference in efficiency becomes much smaller. This technique is known as OT-extension [IKNP03], and the idea is do a small number of OTs using expensive public-key cryptography and then extend these to a much larger number of OTs by cheaper techniques based on special types of hash functions.

However, regardless of the approach used, if we want adaptive security and resort to non-committing encryption, we will get an impractical protocol. It is therefore interesting to note that we can have adaptively secure solutions that are more efficient that if we are given public-key cryptosystems with stronger properties. For instance, some cryptosystems are additively homomorphic, that is, the plaintexts come from some ring, and given ciphertexts $c_1, c_2$ that are encryptions of $m_1, m_2$, and the public key (but not the secret key) one can easily compute $c_3$ that looks exactly like a fresh encryption of $m_1 + m_2$. It was shown in [CDN01] that we can compute arithmetic circuits over the plaintext ring securely if the secret key is secret-shared among the players. This only gives static security, but in [DN03] it was shown that if we base the protocol on Paillier's public-key cryptosystem [Pai99], adaptive security can be obtained at cost only a constant factor over the static solution.

An even stronger property is fully homomorphic encryption, where we can also compute a new ciphertext that contains the product of the plaintexts. The first such schemes were constructed in [Gen09]. With such a scheme multiparty computation is easy in principle: players publish ciphertext (under a single public key) containing their input, everyone can now evaluate a ciphertext containing the desired output, and if we assume that the secret key has been secret shared among the players, then they can collaborate to decrypt the result. This solution is extremely efficient in terms of communication, in particular, we need only a constant number of rounds, and the amount of communication is independent of the circuit we compute. However, with current state of the art, the computational overhead is very large. At the time of writing, it is still unclear whether fully homomorphic encryption will be of real practical importance.

## 7.3 The Case of Dishonest Majority

When $t$ is not assumed to be less than $n/2$, we have already seen that we cannot have information theoretic security at all, and we also cannot guarantee that the protocol terminates (unless we only go for passive security). This means that the best we can hope for is a computationally secure solution of a weaker version of $\mathsf{F}^f_{\mathtt{SFE}}$ where the environment gets the output first and can then tell the functionality whether the honest players should get the result or not. Let $\mathsf{F}^{f,abort}_{\mathtt{SFE}}$ be this weaker variant.

Even if we only go for a UC-secure implementation of $\mathsf{F}^{f,abort}_{\mathtt{SFE}}$, this turns out to be impossible is most cases, as shown in [CKL06]. Basically, to do anything non-trivial in the UC model for dishonest majority, one needs a set-up assumption, some ideal functionality that is assumed available but is not implemented. A popular such assumption is the Common Reference String model (CRS), where we assume a functionality that outputs a string to all players with a certain distribution. Of course, one can argue that this makes the model less realistic, but on the other hand, all results up to now have assumed at least access to $\mathsf{F}_{\mathtt{AT}}$, and this is also a type of set-up assumption, even if it is often not considered as such in the literature.

The basic reason why simulation arguments do not always go through in the plain UC model is that the simulator needs to be able to do "something" that an adversary could not do in a real attack. For example suppose a protocol instructs a player $\mathsf{P}_i$ to send a ciphertext and prove to another player $\mathsf{P}_j$ that he knows the plaintext, of course without revealing it. A simulator for this procedure is typically required to simulate what a corrupt $\mathsf{P}_j$ would see in this game. The simulation must be done without knowing the plaintext, this is exactly how we argue that the plaintext stays hidden from $\mathsf{P}_j$. But on the other hand, if instead $\mathsf{P}_i$ corrupt and does not know the paintext, why can he not do what the simulator does, and in this way cheat $\mathsf{P}_j$? In the so called stand-alone model where we do not consider composition of protocols, the answer is often that the simulator is allowed to rewind its "opponent" (the adversary) to a previous state, and rewinding the other player is of course not something one could do in a real protocol execution.

However, rewinding does not work in the UC model: to have security under arbitrary concurrent composition, we must forbid the simulator to rewind the environment. Therefore, we need to be able to give the simulator an edge in some other way. In the CRS model, the edge is that the simulator gets to choose the reference string – in fact it must do so since it has to simulate everything the environment sees in the protocol. The point is that it may be able to generate the CRS together with some trapdoor relating to the CRS that a real life adversary does not get to see, and this is why the simulator can fake a convincing view of the protocol, whereas an adversary cannot do so.

Using such a set-up assumption, we can indeed do general multiparty computation also with dishonest majority. In [CLOS02] the following was shown:

**Theorem 7.2** *Assuming STPs exist, there exists a protocol $\pi^f_{\texttt{COMPSEC-DM}}$ in the CRS model, such that $\pi^f_{\texttt{COMPSEC-DM}} \diamond \mathsf{F}_{\mathsf{AT}}$ implements $\mathsf{F}^{f,abort}_{\mathsf{SFE}}$ in $\mathrm{Env}^{t,sync,\ poly}$ with computational security for all $t \leq n$.*

Based on the fact that information theoretic security is not possible for dishonest majority, one might think that information theoretic methods for protocol design are not useful in this setting. This is very far from true, however, and we will have much more to say about this in the following chapters, for instance in Chapter 8.

# Chapter 8

# Some Techniques for Efficiency Improvements

**Contents**

## 8.1 Introduction

In this chapter we cover some techniques for improving the efficiency of the protocols we have seen earlier in the book, but some of the techniques also apply to secure computing protocols in general.

## 8.2 Circuit Randomization

Recall the way we represented secret data when we constructed the first MPC protocol for passive security: for $a \in \mathbb{F}$, we defined the object $[a; f_a]_t$ to be the set of shares $f_a(1), ..., f_a(n)$ where $f_a(0) = a$ and the degree of $f_a$ is at most $t$. At the same time it was understood that player $\mathsf{P}_i$ holds $f_a(i)$.

One of the most important properties of this way to represent data is that it is linear, that is, given representations of values $a$ and $b$ players can compute a representation of $a + b$ by only local computation, this is what we denoted by $[a; f_a]_t + [b; f_b]_t = [a + b; f_a + f_b]_t$ which for this particular representation means that each $\mathsf{P}_i$ locally computes $f_a(i) + f_b(i)$.

Of course, this linearity property is not only satisfied by this representation. The representation $[\![a; f_a]\!]_t$ we defined, based on homomorphic commitments to shares, is also linear in this sense.

A final example can be derived from additively homomorphic encryption: if we represent $a \in \mathbb{F}$ by $E_{pk}(a)$ where $pk$ is a public key, and the corresponding secret key is shared among the players, then the additive homomorphic property exactly ensures that players can add ciphertexts and obtain an encryption of the sum, i.e., it holds that $E_{pk}(a) + E_{pk}(b) = E_{pk}(a + b)$, where the addition of ciphertexts is an operation that can be computed efficiently given only the public key.

In the following, for simplicity we will use the notation $[a]$ to denote any representation that is linear in the sense we just discussed. In doing so, we suppress the randomness that is usually used to form the parts held by the players (such as the polynomial $f_a$ in $[a; f_a]_t$).

We assume some main properties of a linear representation, that we only define informally here. For any of the examples we mentioned, it is easy to see what they concretely mean is each of the cases.

**Definition 8.1** *A linear representation $[\cdot]$ over a finite field $\mathbb{F}$ satisfies the following properties:*

1. *Any player $\mathsf{P}_i$ can collaborate with the other players to create $[r]$, where $r \in \mathbb{F}$ is chosen by $\mathsf{P}_i$. If $\mathsf{P}_i$ is honest, the process reveals no information on $r$. $[r]$ will be correctly formed no matter whether $\mathsf{P}_i$ is honest or not.*
2. *If players hold representations $[a], [b]$ and a public constant $\alpha$, they can locally compute a new representations $[a] + [b] = [a + b]$ and $\alpha[a] = [\alpha a]$ of the same form as $[a]$ and $[b]$.*
3. *Given $[a], [b]$, the players can interact to compute a new representation $[a][b] = [ab]$ of the same form as $[a]$ and $[b]$ while revealing nothing new about $a, b$.*
4. *The players can collaborate to open any representation, in particular also representations of form $[a + b]$ or $[ab]$. This will reveal $a + b$, respectively $ab$ but will reveal no other information on $a, b$.*

Following the pattern of several protocols we have already seen, it is clear how one could do secure function evaluation with a linear representation, namely players first create representations of their inputs, then they work their way through a circuit representing the desired function using the procedures for addition and multiplication, and finally we open the representations of the outputs.

There is, however, a way to push a lot of the work needed into a preprocessing phase, and this technique is known as circuit randomization. The idea is to preprocess a number of secure multiplications on random values and then later use these to multiply more efficiently the actual values occurring in the desired computation.

More concretely, we first create a number of so-called multiplication triples. Such a triple has form $[a], [b], [c]$ where $a, b$ are uniformly random, unknown to all players and $c = ab$. One way to create such triples is to have each player $\mathsf{P}_i$ create $[a_i], [b_i]$ for random $a_i, b_i$, and the use the assumed operations to compute

$$[a] = [a_1] + ... + [a_n], \quad [b] = [b_1] + ... + [b_n] \text{ and finally } [c] = [ab].$$

We note already now that this is not the most efficient method, we look at better solutions below. But in any case, it is clear that whatever protocol we use should create many triples in parallel. This opens the possibility of using techniques specialized for this purpose where the amortized cost per triple can be minimized, moreover this can usually be done in a constant number of rounds and hence be quite efficient.

We can then revisit the MPC protocol we sketched before: In a preprocessing phase, we would create a large number of multiplication triples. Once we know the function to compute and the inputs are fixed, we execute the on-line phase. Here we have players create representations of their inputs and then any linear computation on represented values can be done by only local operations. But for multiplication, we would make use of the preprocessing:

Assume we are given $[a], [b]$ and want to compute $[ab]$ securely. To do this, we take the next unused triple $[x], [y], [z]$ from the preprocessing. We then compute

$$[a] - [x] = [a - x], \ [b] - [y] = [b - y]$$

and open $e = a - x, d = b - y$. Note that because we assume $x, y$ to be uniformly random in the view of the environment, this is secure to do: $e, d$ will be uniformly random and hence easy to simulate. Moreover, it is easy to see that we have

$$ab = xy + eb + da - ed = z + eb + da - ed,$$

and hence players can compute locally

$$[ab] = [z] + e[b] + d[a] - ed.$$

This will reduce the work we need in the on-line phase to opening two representations per secure multiplication plus some local computation, whereas doing $[x][y]$ directly would typically require more interaction. Since communication is usually more costly in practice than local computation, this gives a significant performance improvement for many concrete representations. Moreover, for the case of dishonest majority, where we know that we need to use public-key cryptography in some form, it turns out to be possible to push the use of such expensive cryptography into the preprocessing phase and use a more efficient representation in the on-line phase. We return to this below.

Protocol On-Line Phase summarizes the protocol we have just sketched. It is straightforward to formalize it, by first specifying the representation as an ideal functionality with operations for creation, arithmetic and opening. One can then show that the protocol implements $\mathsf{F}_{\mathsf{SFE}}^{f}$ when the circuit used computes $f$. We do not so here, however, since describing the exact "efficiency milage" one gets out of a concrete representation usually requires that one considers the concrete protocols we use for computing with representations, instead of abstracting them away. This will then require a special purpose proof. The basic idea always remains the same, however.

## 8.3 Hyper-invertible Matrices

A hyper-invertible martrix $M$ over a finite field $\mathbb{F}$ is a matrix with the property that any square submatrix is invertible. More precisely,

**Definition 8.2** *A matrix $M$ is hyper-invertible if the following holds: Let $R$ be a subset of the rows, and let $M_R$ denote the sub-matrix of $M$ consisting of rows in $R$. Likewise, let $C$ be a subset of columns and let $M^C$ denote the sub-matrix consisting of columns in $C$. Then we require that $M_R^C$ is invertible whenever $|R| = |C| > 0$.*

Hyper-invertible matrices have nice properties that make them very useful in MPC. We will look at this shortly, first we show that they actually exist, at least if the underlying field is large enough. Some notation that will be useful below: when $C$ is a set of indices designating some of the entries in $\mathbf{x}$, we let $\mathbf{x}_C$ denote the vector containing only the coordinates designated by $C$.

**Theorem 8.1** *Let $\alpha_1, ..., \alpha_n, \beta_1, ..., \beta_m \in \mathbb{F}$ be arbitrary but distinct (so we assume $|\mathbb{F}| \geq n+m$). Consider the function that maps $(x_1, ...., x_n)$ to $(y_1, ..., y_m)$ as follows: first compute the unique polynomial $f$ of degree at most $n-1$ such that $f(\alpha_i) = x_i, i = 1...n$. Then output $f(\beta_1), ..., f(\beta_m)$. This is clearly a linear mapping. The matrix $M$ of this mapping is hyper-invertible.*

*Proof* First consider any input $\mathbf{x} = (x_1, ...., x_n)$ and corresponding output $\mathbf{y} = (y_1, ..., y_m)$. Then, given a total of $n$ of the $x_i$'s and $y_i$'s, we can compute all the other $x_i$'s and $y_i$'s. This just follows from the fact that $n$ values are enough to compute $f$ by Lagrange interpolation, and then we evaluate $f$ in the other points to get the remaining $x_i$'s and $y_i$'s.

Now, assume we are given sets $R, C$ designating rows and columns with $|C| = |R|$ and let $\bar{C}$ be the complement of $C$. Then clearly $|\bar{C}| + |R| = n$. Therefore it follows from what we just observed that given $\mathbf{x}_{\bar{C}}, \mathbf{y}_R$, we can compute $\mathbf{x}_C$.

Protocol On-Line Phase

This protocol computes an arithmetic circuit securely over a field $\mathbb{F}$. It assumes a linear representation of values in the field, written $[a]$, with properties as described in the text. We assume also a preprocessing phase that creates a number of random multiplication triples of form $[a], [b], [c]$ with $ab = c$. We assume for simplicity that all outputs are public.

**Input Sharing:** Each player $\mathsf{P}_i$ holding input $x_i \in \mathbb{F}$ creates a representation $[x_i]$.

We then go through the circuit and process the gates one by one in the computational order. Just after the input sharing phase, we consider all input gates as being processed. We will maintain the following:

**Invariant:** Computing with the circuit on inputs $x_1, ..., x_n$ assigns a unique value to every wire. Consider an input or an output wire for any gate, and let $a \in \mathbb{F}$ be the value assigned to this wire. Then, if the gate has been processed, the players hold $[a]$.

**Computation Phase:** Repeat the following until all gates have been processed (then go to the next phase): Consider the first gate in the computational order that has not been processed yet. According to the type of gate, do one of the following

**Addition gate:** The players hold $[a], [b]$ for the two inputs $a, b$ to the gate. The players compute $[a] + [b] = [a + b]$.

**Multiply-by-constant gate:** The players hold $[a]$ for the input $a$ to the gate, and public constant $\alpha$. The players compute $\alpha[a] = [\alpha a]$.

**Multiplication gate:** The players hold $[a], [b]$ for the two inputs $a, b$ to the gate. Take the next unused triple $[x], [y], [z]$ from the preprocessing phase.

1. Players compute $[a] - [x] = [a - x]$, $[b] - [y] = [b - y]$, and open $e = a - x, d = b - y$.
2. Players can compute locally $[ab] = [z] + e[b] + d[a] - ed$.

**Output Reconstruction:** At this point all gates, including the output gates have been processed. So for each output gate, the players hold $[y]$ where $y$ is the value assigned to the output gate. We open $[y]$ to reveal $y$ to all players.

Finally, suppose we are given an input $\mathbf{y}_R$. Then we set $\mathbf{x}_{\bar{C}}$ to be the all-0 vector. The above observation says we can now compute full vectors $\mathbf{x}, \mathbf{y}$ with $\mathbf{y} = M\mathbf{x}$. But this and $\mathbf{x}_{\bar{C}} = 0$ implies that in particular we have found $\mathbf{x}_R$ such that $\mathbf{y}_R = M_R^C \mathbf{x}_C$. This holds for any $\mathbf{y}_R$, whence $M_R^C$ must be invertible. □

We now state two useful properties of hyper-invertible matrices:

**Lemma 8.2** *Let $M$ be hyper-invertible, and let $C$ be a subset of the columns, $R$ a subset of the rows, with $|\bar{C}| = |R|$, that is, $|C| + |R| = n$. Consider any $\mathbf{x}, \mathbf{y}$ with $\mathbf{y} = M\mathbf{x}$. Then given $\mathbf{x}_C, \mathbf{y}_C$, one can easily compute $\mathbf{x}_{\bar{C}}, \mathbf{y}_{\bar{R}}$.*

*Proof* It is easy to see that

$$\mathbf{y}_R = M_R \mathbf{x} = M_R^C \mathbf{x}_C + M_R^{\bar{C}} \mathbf{x}_{\bar{C}}.$$

SInce $M$ is hyperinvertible, $M_R^{\bar{C}}$ is invertible, so rearranging the above, we have

$$\mathbf{x}_{\bar{C}} = (M_R^{\bar{C}})^{-1}(\mathbf{y}_R - M_R^C \mathbf{x}_C)$$

Now we have all of $\mathbf{x}$, so $\mathbf{y}$ is trivial to compute. □

**Lemma 8.3** *Let $M$ be hyper-invertible, and consider the linear mapping $\phi_M : \mathbb{F}^n \to \mathbb{F}^m$ induced by $M$. Suppose we fix $k$ of the input coordinates to arbitrary values. Then consider the linear mapping $\phi'$ naturally induced by $\phi_M$ from the remaining $n-k$ coordinates to any $n-k$ coordinates of the output (so we assume that $m \geq n-k$). Then $\phi'$ is a bijection.*

*Proof* The $n-k$ input coordinates correspond naturally to a subset $C$ of the columns of $M$, and the selection of output coordinates naturally to a subset $R$ of the rows. Given an input vector $\mathbf{z} \in \mathbb{F}^{n-k}$ for $\phi'$, we can construct a vector $\mathbf{x} \in \mathbb{F}^n$ such that $\mathbf{x}_C = \mathbf{z}$ and $\mathbf{x}_{\bar{C}}$ contains the $k$ fixed coordinates. Then

$$\phi'(\mathbf{z}) = M_R \mathbf{x} = M_R^C \mathbf{x}_C + M_R^{\bar{C}} \mathbf{x}_{\bar{C}}$$

Since $M_R^{\bar{C}} \mathbf{x}_{\bar{C}}$ is a fixed vector and $M_R^C$ is invertible, this is clearly a bijection. □

**Randomness Extraction and Efficient Secure Multiplication**

We now look at how the above properties of hyper-invertible matrices can be used in multiparty computation. In particular we have seen the concept of a linear representation of secret data, and we have seen that, for instance, to create multiplication triples it is very useful to be able to create a representation of random field element that is unknown to the environment. We also saw a trivial way to do this where each player creates a representation of his own random element and then we add all contributions. This works, but seems wasteful because each of the $n-t$ honest player contributes a random element but we end up with only 1 random representation.

Hyper-invertible matrices offer a better solution as shown in Protocol Extract Random Representations. The protocol uses an $n$ by $n$ hyper-invertible matrix $M$ and creates first a set of representations $[x_i],\ i = 1...n$. Now note that since we can do linear computation on representations, we can consider these as a vector with entries $[x_i],\ i = 1...n$, multiply $M$ on this vector, and obtain in a natural way a vector of representations $[y_i],\ i = 1...n$. More precisely $[y_i] = M_{i,1}[x_1] + ... + M_{i,n}[x_n]$. Note that, by our assumptions on linear representations, the $[y_i]$ can be computed by only local computation.

Protocol Extract Random Representations

This protocol produces a number of random representations based on contributions from each player.

1. Each player $\mathsf{P}_i$ creates $[x_i]$ where $x_i$ is uniformly random.
2. The players apply matrix $M$ to this vector $[x_1], ..., [x_n]$ of representations (as explained in the text), to get via local computation representations $[y_1], ..., [y_n]$.
3. Output the representations $[y_1], ...., [y_{n-t}]$.

We claim that the $y_i$'s contained in the output $[y_1], ...., [y_{n-t}]$ are uniformly random in the view of the adversary/environment.

To see this, note that first if we let $\mathbf{x} \in \mathbb{F}^n$ be the vector with the $x_i$'s as entries, and define a vector $\mathbf{y}$ from the $y_i$'s in a similar way, then $\mathbf{y} = M\mathbf{x}$. Now, let $C$ be the set of indices designating the $n-t$ honest players, and let $R$ be the set of indices $1, ..., n-t$. Now, by Lemma 8.3, for any set of values in $\mathbf{x}_{\bar{C}}$ chosen by the corrupt players, the values taken by $y_1, ..., y_{n-t}$ are in 1-1 correspondence with the entries in $\mathbf{x}_C$. But these are uniformly random since they were chosen by the honest players, and hence so are $y_1, ..., y_{n-t}$.

In this way, the amortized cost of outputting a representation of a random value is, up to a constant factor, only that of having a single player create a representation, and hence a factor $n$ more efficient than the naive method.

It is instructive to consider that this idea can also be seen as follows: we are given $x_1, ..., x_n$, and we know that $n - t$ of these values are random, but not which ones. Now, using a hyper-invertible matrix, we can compute $n - t$ new values that are guaranteed to be random, so in other words, we can extract all the randomness that is available.

**Exercise 8.1** In this exercise we assume passive corruption and we use representations of form $[a; f_a]_t$, where $t < n/2$.

Using Protocol Extract Random Representations as a starting point, construct a protocol that will produce as output a number of pairs $[r; f_r]_t, [r, g_r]_{2t}$, where $r$ is random and unknown to all players and $f_r, g_r$ are independent and randomly chosen polynomials.

Show that the total communication done in your protocol is $O(n)$ field elements per pair that is output.

The result of the exercise can be used to construct an efficient protocol that starts from $[a; f_a]_t, [b; f_b]_t$ and outputs $[ab, f_{ab}]_t$, still assuming passive (adaptive) corruption.

To see this, recall that players can compute using local computation $[ab; f_a f_b]_{2t}$. Now, if we assume we have a pair $[r; f_r]_t, [r, g_r]_{2t}$ available, then by further local computation, players can compute $[ab - r; g_r - f_a f_b]_{2t}$. This representation we can safely open because $r$ and $g_r$ are random, and so all players now know $ab - r$. We can then make a default representation of $ab - r$ using the polynomial that is constantly $ab - r$ : $[ab - r; ab - r]_t$. Adding this to $[r, f_r]_t$ will produce $[ab, f_{ab}]_t$, where $f_{ab} = f_r + ab - r$.

An important detail about this protocol is that, to open $[ab - r; g_r - f_a f_b]_{2t}$, there is no reason to have all players send their share to all other players. It is sufficient to send to just one player, say $\mathsf{P}_1$, he can reconstruct $ab - r$ and send this value to all players. Note that this works since we assume passive corruption so $\mathsf{P}_1$ will follow the protocol, and that this only requires sending $O(n)$ field elements.

Together with the exercise, we see that we now have a protocol for doing many secure multiplications that costs communication $O(n)$ field elements per multiplication. In comparison, the first protocol we saw in Chapter 3 required $O(n^2)$ field elements.

### Checking consistency of secret sharings and other linear properties

In the first part of this subsection, we concentrate on linear representations based on secret sharing. The simplest form we have seen is $[a; f_a]_t$ where player $\mathsf{P}_j$ holds $f_a(j)$ and $f_a(0) = a$. Having player $\mathsf{P}_i$ create such a representation is simple if we settle for passive security: $\mathsf{P}_i$ just selects $f_a$ and sends $f_a(j)$ to each $\mathsf{P}_j$.

This does not work for active security, as we have no guarantee that what $\mathsf{P}_i$ sends is consistent with a polynomial of degree at most $t$. We have seen in Chapter 5 a protocol that can be used to force even a corrupt $\mathsf{P}_i$ to select consistent shares (Protocol Commit). This protocol has perfect security if $t < n/3$, but requires a lot of interaction and communicates much more data than the simple solution where $\mathsf{P}_i$ just sends one value to all players.

We now show in Protocol Check Consistent Secret-Shares that hyper-invertible matrices can be used to verify whether $\mathsf{P}_i$ has behaved correctly and this method will be much more efficient if we want to create many random shared values. We will assume for simplicity that $n = 3t + 1$.

In the following, we will say that a representation $[a; f_a]_t$ is well formed if the values held by honest players are consistent with a polynomial of degree at most $t$. Note that we abuse notation slightly by writing $[a; f_a]_t$ with subscript $t$ even if there is no guarantee that the degree of $f_a$ is at most $t$ until this has been checked.

Protocol Check Consistent Secret-Shares

This protocol checks that a set of shares of values distributed by player $\mathsf{P}_i$ are well formed.

1. $\mathsf{P}_i$ distributes $[x_1; f_{x_1}]_t, ..., [x_n; f_{x_n}]_t$ for random $x_1, ..., x_n$.
2. The players apply matrix $M$ to this vector of representations (in the same way as in the previous subsection), to get via local computation representations $[y_1; f_{y_1}]_t, ..., [y_n; f_{y_n}]_t$.
3. For $j = 1...2t$, all players send their share in $[x_j; f_{x_j}]_t$ and $[y_j; f_{y_j}]_t$ to $\mathsf{P}_j$. Furthermore, for $j = 2t + 1..n$, all players send their share in $[x_j; f_{x_j}]_t$ to $\mathsf{P}_j$. Player $\mathsf{P}_j$ checks that all sets of shares he receives are consistent with a polynomial of degree at most $t$, and broadcasts `accept` or `reject` accordingly.
4. If all players said `accept`, then output $[y_{2t+1}; f_{y_{2t+1}}]_t, ..., [y_n; f_{y_n}]_t$.

The claim is now that first, if all players say `accept` then $[y_{2t+1}; f_{y_{2t+1}}]_t, ..., [y_n; f_{y_n}]_t$ are well formed even if $\mathsf{P}_i$ is corrupt, i.e., the degree of $f_{x_j}$ for $j = 2t+1...n$ is at most $t$. And second, that if $\mathsf{P}_i$ is honest, then $y_{2t+1}, ..., y_n$ are uniformly random in the view of the adversary/environment.

To show the first claim, notice that if all players say `accept`, then of the $2n$ representations $[x_1; f_{x_1}]_t, ..., [x_n; f_{x_n}]_t, [y_1; f_{y_1}]_t, ..., [y_n; f_{y_n}]_t$, we know that at least $n$ of them are well formed, namely the $2t + 1$ $[x_j; f_{x_j}]_t$'s that were checked by honest players and the $t$ $[y_j; f_{y_j}]_t$'s that were checked by honest players. ones that were checked by honest players. Then it follows from Lemma 8.2 that all the other representations can be computed form these $n$ well formed ones. In particular $[y_{2t+1}; f_{y_{2t+1}}]_t, ..., [y_n; f_{y_n}]_t$ can be computed as a linear combination of well formed representations, and so it immediately follows that they are well formed as well. This is because a linear combination of polynomials of degree at most $t$ is again of degree at most $t$.

To show the second claim, note that the adversary knows $t$ of the values $x_1, ..., x_n$,, namely those for which the corresponding representations were "checked" by a corrupt player. It now follows from Lemma 8.3 that the remaining $n - t$ $x_j$'s (which are unknown to the adversary) are in 1-1 correspondence with any subset of $n - t$ of the $y_j$'s. We form such a subset as the union of $y_{2t+1}, ..., y_n$ and $t$ $y_j$'s that were checked by honest players (at least $t$ such $y_j$'s must be available). All these $y_j$'s are uniformly random in the view of the adversary, so in particular this is true for $y_{2t+1}, ..., y_n$.

Note that if Protocol Check Consistent Secret-Shares has success, then it outputs a constant fraction of the representations that were created initially. Hence, up to a constant fraction, the cost of the protocol is the same as simply sending one share to each player per representation [1].

It is of course a natural question what we do if Protocol Check Consistent Secret-Shares is not successful? The protocol as it stands would just not generate any output. However, there are more productive ways of dealing with such a situation. The basic observation is that if some $\mathsf{P}_j$ says `reject`, then either $\mathsf{P}_i$ or $\mathsf{P}_j$ must be corrupt. One option is now to simply eliminate both players from the global protocol. Note that if we had less than one third corrupted players before the elimination, this is also the case after, so we can still hope to finish the computation eventually. Also note that such a problem can at most occur $t$ times, so this upper bounds the extra work we do as a result of eliminations. See the notes below for pointers to literature on this issue.

[1] Here, we conveniently ignore the broadcasts towards the end of the protocol, and these will usually not come for free. However, if we created enough representations in parallel and only do one set of broadcasts, then the amortized cost of these will become insignificant.

It is not hard to see that the principle from Protocol CHECK CONSISTENT SECRET-SHARES can be used to check other properties of values inside representations, as long as these properties are preserved under linear transformations. As an example, suppose we use representations of form $[a; f_a]_t$ and we want to check that a player has created a number of representations $[0; f_1]_t, ..., [0, f_n]_t$. This can be done with a simple variation of Protocol CHECK CONSISTENT SECRET-SHARES, where players who verify representations should also verify that the value in the checked representation is 0. This works, exactly because the property of containing a 0 is preserved under linear mappings on representations.

**Exercise 8.2** Specify the protocol sketched above and prove that if $n = 3t + 1$, then your protocol can output $t$ random sharings of 0. Specify and prove a protocol for producing a number of pairs of representations of form $([a; f_a]_t, [a, g_a]_t)$, i.e., the same element is contained in both representations.

## 8.4 Packed Secret-Sharing.

In this section we present a technique that allows us to secret-share several field elements in one go, in such a way that each player only receives one field element as his share. We must pay for this by being able to tolerate a smaller number of corrupted players, but the idea can nevertheless be used in combination with other techniques to get better efficiency in many cases.

The idea is a simple twist on Shamir secret sharing. We will assume for simplicity of notation that the underlying field is $\mathbb{F} = \mathbb{Z}_p$ for a prime $p$, where $p > \ell + n$ and $n$ is the number of players. This means that the numbers $-\ell + 1, ...., 0, 1, ..., n$ can be interpreted as distinct field elements in a natural way. To share a vector of elements $\mathbf{s} = (s_1, ..., s_\ell) \in \mathbb{F}^\ell$, tolerating up to $t$ corrupted players, one selects a random polynomial $f_{\mathbf{s}}(\mathtt{X})$ of degree at most $d = \ell - 1 + t$ with the property that $f_{\mathbf{s}}(-j + 1) = s_j$ for $j = 1...\ell$. Then the share given to $\mathsf{P}_i$ is $f_{\mathbf{s}}(i)$. We can now easily prove:

**Lemma 8.4** *Let the secret vector* $\mathbf{s}$ *and random polynomial* $f_{\mathbf{s}}$ *be defined as above. Then, any subset of at most* $t$ *shares have distribution independent of* $\mathbf{s}$ *and from any set of at least* $\ell + t$ *shares, one can reconstruct* $\mathbf{s}$*.*

*Proof* The last statement on reconstruction follows trivially from Lagrange interpolation. As for the first statement, assume without loss of generality that we consider the shares $f_{\mathbf{s}}(1), ..., f_{\mathbf{s}}(t)$. Note that by Lagrange interpolation, we can construct a polynomial $h$ of degree at most $d = \ell - 1 + t$ such that $h(1) = ... = h(t) = 0$ and $h(-j + 1) = -s_j$ for $j = 1...\ell$. Hence for each polynomial $f_{\mathbf{s}}(\mathtt{X})$ for sharing $\mathbf{s}$, there exists exactly one polynomial, namely $f_{\mathbf{s}}(\mathtt{X}) + h(\mathtt{X})$ for sharing the vector $(0, .., 0)$, generating the same first $t$ shares. Since polynomials are chosen uniformly, it follows that for every secret vector, the distribution of the $t$ shares is the same, namely the one resulting from sharing the all-zero vector. □

We now show how to use packed secret-sharing to get passively secure computation on vectors, while only having players operate on single field elements. We will need to assume that the degree $d$ of the polynomials is such that $2d < n$. We will also assume that $t$ and $\ell$ are both in $\Theta(n)$, this is clearly possible while still ensuring the demand on $d$ is satisfied.

We can first define a linear representation of vectors with $\ell$ coordinates, denoted

$$[\mathbf{a}; f_{\mathbf{a}}]_d = (f_{\mathbf{a}}(1), ..., f_{\mathbf{a}}(n)).$$

And if we define addition of representations by coordinate-wise addition as usual, one easily sees that we have

$$[\mathbf{a}; f_{\mathbf{a}}]_d + [\mathbf{b}; f_{\mathbf{b}}]_d = [\mathbf{a} + \mathbf{b}; f_{\mathbf{a}} + f_{\mathbf{b}}]_d.$$

In other words, if the $n$ players hold representations of two vectors, we can add them securely by each player doing just one local addition.

In the same way, we define $[\mathbf{a}; f_{\mathbf{a}}]_d * [\mathbf{b}; f_{\mathbf{b}}]_d$ as the coordinate-wise product of the two sets of shares, and we have

$$[\mathbf{a}; f_{\mathbf{a}}]_d * [\mathbf{b}; f_{\mathbf{b}}]_d = [\mathbf{a} * \mathbf{b}; f_{\mathbf{a}} f_{\mathbf{b}}]_{2d}.$$

This means that if $2d < n$, we can do a multiplication protocol in much the same way as we have seen before, this can be seen in Protocol Packed Secure Multiplication.

The protocol assumes that auxiliary representations $[\mathbf{r}; f_{\mathbf{r}}]_{2d}, [\mathbf{r}; g_{\mathbf{r}}]_d$ for a random vector $\mathbf{r}$ are available. Such pairs can be constructed using the techniques we have seen in the previous section (see exercise below). Note that it is secure to send all shares of $\mathbf{a} * \mathbf{b} - \mathbf{r}$ to $\mathsf{P}_1$ because $\mathbf{r}$ and $f_{\mathbf{r}}$ are random.

Protocol Packed Secure Multiplication

This protocol takes as input two representations $[\mathbf{a}; f_{\mathbf{a}}]_d$, $[\mathbf{b}; f_{\mathbf{b}}]_d$.
It is assumed that we have available a pair of auxiliary representations $[\mathbf{r}; f_{\mathbf{r}}]_{2d}, [\mathbf{r}; g_{\mathbf{r}}]_d$ for a random vector $\mathbf{r}$,

1. $[\mathbf{a}; f_{\mathbf{a}}]_d * [\mathbf{b}; f_{\mathbf{b}}]_d = [\mathbf{a} * \mathbf{b}; f_{\mathbf{a}} f_{\mathbf{b}}]_{2d}$ by local multiplication.
2. The difference $[\mathbf{a} * \mathbf{b}; f_{\mathbf{a}} f_{\mathbf{b}}]_{2d} - [\mathbf{r}; f_{\mathbf{r}}]_{2d}$ is computed by local computation and all shares are sent to player $\mathsf{P}_1$.
3. $\mathsf{P}_1$ reconstructs $\mathbf{a} * \mathbf{b} - \mathbf{r}$ from the shares received, then forms $[\mathbf{a} * \mathbf{b} - \mathbf{r}; h]_d$ and sends a share to each player.
4. Players compute by local operations $[\mathbf{a} * \mathbf{b} - \mathbf{r}; h]_d + [\mathbf{r}; g_{\mathbf{r}}]_d = [\mathbf{a} * \mathbf{b}; h + g_{\mathbf{r}}]$.

**Exercise 8.3** Using the randomness extraction techniques from the previous section, build a protocol that outputs a set of pairs of form $[\mathbf{r}; f_{\mathbf{r}}]_{2d}, [\mathbf{r}; g_{\mathbf{r}}]_d$ for random $\mathbf{r}$ as required in the multiplication above. Assuming $t$ and $\ell$ are $\Theta(n)$, show that your protocol uses communication of $O(n)$ field elements per pair produced.

Let us consider what these ideas actually achieve: we have seen that if $2d = 2(t + \ell - 1) < n$ then we can do secure addition and multiplication of $\ell$-vectors using the same communication (and computational work) that we used in the previous subsection to do addition and multiplication on *single* field elements, namely we send $O(n)$ field elements per operation. So we get $\ell$ operations "for the price of one". Of course, this only works in a so-called "same instruction, multiple data" fashion (SIMD), i.e., we always have to do the *same* operation on all $\ell$ entries in a block. Moreover, the SIMD operations do not come for free, we have to accept a smaller threshold value for the number of corruptions we can tolerate, but we can still tolerate corruption of a constant fraction of the players.

An obvious application of this is to compute securely $\ell$ instances in parallel of the same arithmetic circuit $C$. Since we chose $\ell$ to be $\Theta(n)$, the communication we need per gate computed is a constant number of field elements.

It is natural to ask if we can use this approach to save on computation and work also when computing a *single* circuit securely, and whether we can get active security as well? The answer turns out to be yes in both cases, and this leads to protocols that are close to optimal, at least in some respects. The notes section below has more information on this.

## 8.5 Information Theoretic Protocols in the Preprocessing Model

In this section we show how some of the techniques we have seen can be used for the case of *dishonest majority*, i.e. we assume up to $t$ active corruptions and $t \leq n-1$. We have already seen in Chapter 7 that for this case, we have to use public-key machinery which is typically quite expensive. In comparison, the information theoretically secure primitives we have seen are computationally simpler and more efficient.

It may therefore seem that we cannot have really efficient solutions for dishonest majority. This is not entirely true, however: what we can do is to assume a preprocessing phase (using public-key cryptography) where the function to compute and the inputs are not yet known. This phase will produce some "raw material" to be used in an on-line phase where the actual computation is done and where we can use the more efficient primitives. More concretely, we will use the pattern we saw in the section on circuit randomization, so the idea will be to define an appropriate linear representation and assume a preprocessing that outputs multiplication triples in this representation.

To define our linear representation, we will use again a technique we saw in the implementation of $\mathsf{F}_{\mathtt{ICSIG}}$, namely to use information theoretic message authentication codes (MACs). This will lead to statistical security, and in general the protocols we will see have error probability $1/|\mathbb{F}|$. We assume for simplicity that the field is so large that this is negligible, but there are also solutions that work for small fields (see the Notes section below for details on this).

A key $K$ for our MACs is a random pair $K = (\alpha, \beta) \in \mathbb{F}^2$, and the authentication code for a value $a \in \mathbb{F}$ is $\mathtt{MAC}_K(a) = \alpha a + \beta$.

We recall briefly how these MAC's are to be used: One party $\mathsf{P}_i$ will hold $a, \mathtt{MAC}_K(a)$ and another party $\mathsf{P}_j$ holds $K$. The idea is to use the MAC to prevent $\mathsf{P}_i$ from lying about $a$ when he is supposed to reveal it to $\mathsf{P}_j$. It will be very important in the following that if we keep $\alpha$ constant over several different MAC keys, then one can add two MACs and get a valid authentication code for the sum of the two corresponding messages. More concretely, two keys $K = (\alpha, \beta), K' = (\alpha', \beta')$ are said to be *consistent* if $\alpha = \alpha'$. For consistent keys, we define $K + K' = (\alpha, \beta + \beta')$ so that it holds that $\mathtt{MAC}_K(a) + \mathtt{MAC}_{K'}(a') = \mathtt{MAC}_{K+K'}(a + a')$.

In the protocol below, we will give to $P_i$ several different values $m_1, m_2, \ldots$ with corresponding MACs $\gamma_1, \gamma_2, \ldots$ computed using keys $K_i = (\alpha, \beta_i)$ that are random but consistent. It is then easy to see that if $P_i$ claims a false value for any of the $m_i$'s (or a linear combination of them) he can guess an acceptable MAC for such a value with probability at most $1/|\mathbb{F}|$. This follows by an argument similar to what we saw in the proof of Theorem 5.7.

To represent a value $a \in \mathbb{F}$, we will give a share $a_i$ to each party $\mathsf{P}_i$, where the $a_i$ are randomly chosen, subject to $a = \sum_i a_i$. In addition, $P_i$ will hold MAC keys $K^i_{a_1}, \ldots, K^i_{a_n}$. He will use key $K^i_{a_j}$ to check the share of $P_j$, if we decide to make $a$ public. Finally, $P_i$ also holds a set of authentication codes $\mathtt{MAC}_{K^j_{a_i}}(a_i)$. We will denote $\mathtt{MAC}_{K^j_{a_i}}(a_i)$ by $m_j(a_i)$ from now on. Party $P_i$ will use $m_j(a_i)$ to convince $P_j$ that $a_i$ is correct, if we decide to make $a$ public. Summing up, we have the following way of representing $a$:

$$[a] = [\{a_i, \{K^i_{a_j}, m_j(a_i)\}^n_{j=1}\}^n_{i=1}]$$

where $\{a_i, \{K^i_{a_j}, m_j(a_i)\}^n_{j=1}\}$ is the information held privately by $\mathsf{P}_i$.

We say that $[a] = [\{a_i, \{K^i_{a_j}, m_j(a_i)\}^n_{j=1}\}^n_{i=1}]$ is *consistent*, with $a = \sum_i a_i$, if $m_j(a_i) = \mathtt{MAC}_{K^j_{a_i}}(a_i)$ for all $i, j$. Two representations

$$[a] = [\{a_i, \{K^i_{a_j}, m_j(a_i)\}^n_{j=1}\}^n_{i=1}], \quad [a'] = [\{a'_i, \{K^i_{a'_j}, m_j(a'_i)\}^n_{j=1}\}^n_{i=1}]$$

are said to be *key-consistent* if they are both consistent, and if for all $i, j$ the keys $K^i_{a_j}, K^i_{a'_j}$ are consistent. We will want *all* representations in the following to be key-consistent: this is ensured by letting $P_i$ use the same $\alpha_j$-value in keys towards $P_j$ throughout. Therefore the notation $K^i_{a_j} = (\alpha^i_j, \beta^i_{a_j})$ makes sense and we can do linear computations with these representations, as detailed in Protocol Operations on $[\cdot]$-representations.

Protocol Operations on $[\cdot]$-representations

**Opening:** We can reliably open a consistent representation to $\mathsf{P}_j$: each $\mathsf{P}_i$ sends $a_i, m_j(a_i)$ to $\mathsf{P}_j$. $\mathsf{P}_j$ checks that $m_j(a_i) = \mathtt{MAC}_{K^j_{a_i}}(a_i)$ and broadcasts `accept` or `fail` accordingly. If all is OK, $\mathsf{P}_j$ computes $a = \sum_i a_i$, else we abort. We can modify this to opening a value $[a]$ to all parties, by opening as above to every $\mathsf{P}_j$.

**Addition:** Given two key-consistent representations as above we get that

$$[a + a'] = [\{a_i + a'_i, \{K^i_{a_j} + K^i_{a'_j}, m_j(a_i) + m_j(a'_i)\}^n_{j=1}\}^n_{i=1}]$$

is a consistent representation of $a + a'$. This new representation can be computed only by local operations.

**Multiplication by constants:** In a similar way, we can multiply a public constant $\delta$ "into" a representation. This is written $\delta[a]$ and is taken to mean that all parties multiply their shares, keys and MACs by $\delta$. This gives a consistent representation $[\delta a]$.

**Addition of constants:** We can add a public constant $\delta$ into a representation. This is written $\delta + [a]$ and is taken to mean that $\mathsf{P}_1$ will add $\delta$ to his share $a_1$. Also, each $\mathsf{P}_j$ will replace his key $K^j_{a_1} = (\alpha^j_1, \beta^j_{a_1})$ by $K^j_{a_1+\delta} = (\alpha^j_1, \beta^j_{a_1} - \delta\alpha^j_1)$. This will ensure that the MACs held by $\mathsf{P}_1$ will now be valid for the new share $a_1 + \delta$, so we now have a consistent representation $[a + \delta]$.

We now specify what exactly we need the preprocessing phase to do for us, namely it must implement the ideal functionality $\mathsf{F}_{\mathtt{TRIP}}$. For multiplication and input sharing later, we will need it to output both random single values $[a]$ and triples $[a], [b], [c]$ where $a, b$ are random and $c = ab \mod p$. Also, all singles and triples produced must be key consistent, so that we can freely add them together.

The specification of the functionality follows an important general principle that is very useful whenever a functionality is supposed to output shares of secret data to the players: The environment is allowed to specify to the functionality all the data that the corrupted parties should hold, including all shares of secrets, keys and MACs. Then, the functionality chooses the secrets to be shared and constructs the data for honest parties so it is consistent with the secrets and the data specified by the environment.

This may at first sight seem overly complicated: why don't we just let the functionality choose all shares and output them to the players? However, it is important to understand that if we had specified the functionality this way, it could not be implemented! The point is that the simulator we need to construct to prove an implementation secure needs to show to the environment a simulated execution of the protocol we use to create the triples. The shares, keys and MACs for corrupt parties created here must be consistent with the data that $\mathsf{F}_{\mathtt{TRIP}}$ creates for honest parties, since this will always be the case for a real execution. This consistency is only ensured if the functionality is willing to be consistent with what the environment wants.

Note also that $\mathsf{F}_{\mathtt{TRIP}}$ may get input "stop" from the environment and in this case will not generate any output. This is because we work with dishonest majority where termination cannot be guaranteed.

Since $\mathsf{F}_{\mathtt{TRIP}}$ outputs random multiplication triples, we can securely compute multiplications

Agent $\mathrm{F}_{\mathtt{TRIP}}$

Ideal functionality specifying the preprocessing needed for the $[\cdot]$-representation.

**Initialize:** On input $(init, \mathbb{F})$ from all parties the functionality stores a description of the field $\mathbb{F}$. For each corrupted party $\mathsf{P}_i$ the environment specifies values $\alpha^i_j, j = 1, \ldots, n$, except those $\alpha^i_j$ where both $\mathsf{P}_i$ and $\mathsf{P}_j$ are corrupt. For each honest $\mathsf{P}_i$, the functionality chooses $\alpha^i_j, j = 1, \ldots, n$ at random.

**Singles:** On input $(singles, u)$ from all parties $\mathsf{P}_i$, the functionality does the following, for $v = 1, \ldots, u$:

1. It waits to get from the environment either "stop", or some data as specified below. In the first case it sends "fail" to all honest parties and stops. In the second case, the environment specifies for each corrupt party $\mathsf{P}_i$, a share $a_i$ and $n$ pairs of values $(m_j(a_i), \beta^i_{a_j}), j = 1, \ldots, n$, except those $(m_j(a_i), \beta^i_{a_j})$ where both $\mathsf{P}_i$ and $\mathsf{P}_j$ are corrupt.
2. The functionality chooses $a \in \mathbb{F}$ at random and creates the representation $[a]$ as follows:
   (a) First it chooses random shares for the honest parties such that the sum of these and those specified by the environment is correct: Let $A$ be the set of corrupt parties, then $a_i$ is chosen at random for $\mathsf{P}_i \notin A$, subject to $a = \sum_i a_i$.
   (b) For each honest $\mathsf{P}_i$, and $j = 1, \ldots, n$, $\beta^i_{a_j}$ is chosen as follows: if $\mathsf{P}_j$ is honest, $\beta^i_{a_j}$ is chosen at random, otherwise it sets $\beta^i_{a_j} = m_i(a_j) - \alpha^i_j a_j$. Note that the environment already specified $m_i(a_j), a_j$, so what is done here is to construct the key to be held by $\mathsf{P}_i$ to be consistent with the share and MAC chosen by the environment.
   (c) For all $i = 1, \ldots, n, j = 1, \ldots, n$ it sets $K^i_{a_j} = (\alpha^i_j, \beta^i_{a_j})$, and computes $m_j(a_i) = \mathtt{MAC}_{K^j_{a_i}}(a_i)$.
   (d) Now all data for $[a]$ is created. The functionality sends $\{a_i, \{K^i_{a_j}, m_j(a_i)\}_{j=1,\ldots,n}\}$ to each honest $\mathsf{P}_i$ (no need to send anything to corrupt parties, the environment already has the data).

**Triples:** On input $(triples, u)$ from all parties $\mathsf{P}_i$, the functionality does the following, for $v = 1, \ldots, u$:

1. Step 1 is done as in "Singles".
2. For each triple to create it chooses $a, b$ at random and sets $c = ab$. Now it creates representations $[a], [b], [c]$, each as in Step 2 in "Singles".

exactly as we saw in the section on circuit randomization. This leads to Protocol On-Line Phase, MAC-based, which is essentially Protocol On-Line Phase, specialized for the MAC-based representation we use here.

Let $\pi_{\mathtt{ON\text{-}LINE}}$ denote Protocol On-Line Phase, MAC-based. Then we have

**Theorem 8.5** *$\pi_{\mathtt{ON\text{-}LINE}} \diamond \mathsf{F}_{\mathtt{TRIP}}$ implements $\mathsf{F}^{f,abort}_{\mathtt{SFE}}$ in* $\mathrm{Env}^{t,sync,\ static}$ *with statistical security for all $t < n$.*

*Proof* We construct a simulator $\mathcal{S}_{\mathtt{ON\text{-}LINE}}$ such that a poly-time environment $\mathcal{Z}$ cannot distinguish between $\pi_{\mathtt{ON\text{-}LINE}} \diamond \mathsf{F}_{\mathtt{TRIP}}$ and $\mathsf{F}^{f,abort}_{\mathtt{SFE}} \diamond \mathcal{S}_{\mathtt{ON\text{-}LINE}}$. We assume here static, active corruption. The

Protocol On-Line Phase, MAC-based

The protocol assumes access to $\mathsf{F}_{\mathtt{TRIP}}$, and we assume that a field $\mathbb{F}$ and a circuit $C$ to compute securely are given at some point after $\mathsf{F}_{\mathtt{TRIP}}$ has been invoked. Protocol Operations on $[\cdot]$-representations is used for linear operations on representations.

**Initialize:** The parties first invoke $\mathsf{F}_{\mathtt{TRIP}}(init, \mathbb{F})$. Then, they invoke $\mathsf{F}_{\mathtt{TRIP}}(triples, u)$ and $\mathsf{F}_{\mathtt{TRIP}}(singles, u)$ a sufficient number of times to create enough singles and triples. Here, only an upper bound on the size of circuit to compute needs to be known.

At some later stage when $C$ and the inputs are known, do the following:

**Input Sharing:** To share $\mathsf{P}_i$'s input $[x_i]$, $\mathsf{P}_i$ takes a single $[a]$ from the set of available ones. Then, the following is performed:

1. $[a]$ is opened to $\mathsf{P}_i$.
2. $\mathsf{P}_i$ broadcasts $\delta = x_i - a$.
3. The parties compute $[x_i] = [a] + \delta$.

We then go through the circuit and process the gates one by one in the computational order. Just after the input sharing phase, we consider all input gates as being processed. We will maintain the following:

**Invariant:** Computing with the circuit on inputs $x_1, ..., x_n$ assigns a unique value to every wire. Consider an input or an output wire for any gate, and let $a \in \mathbb{F}$ be the value assigned to this wire. Then, if the gate has been processed, the players hold $[a]$.

**Computation Phase:** Repeat the following until all gates have been processed (then go to the next phase): Consider the first gate in the computational order that has not been processed yet. According to the type of gate, do one of the following

**Addition gate:** The players hold $[a], [b]$ for the two inputs $a, b$ to the gate. The players compute $[a] + [b] = [a + b]$.

**Multiply-by-constant gate:** The players hold $[a]$ for the input $a$ to the gate, and public constant $\alpha$. The players compute $\alpha[a] = [\alpha a]$.

**Multiplication gate:** The players hold $[a], [b]$ for the two inputs $a, b$ to the gate. Take the next unused triple $[x], [y], [z]$ from the preprocessing phase.

1. Players compute $[a] - [x] = [a - x]$, $[b] - [y] = [b - y]$, and open $e = a - x, d = b - y$.
2. Players can compute locally $[ab] = [z] + e[b] + d[a] - ed$.

**Output Reconstruction:** At this point all gates, including the output gates have been processed. So for each output gate, the players hold $[y]$ where $y$ is the value assigned to the output gate. If $\mathsf{P}_i$ is to learn this value, we open $[y]$ to $\mathsf{P}_i$.

simulator will internally run a copy of $\mathsf{F}_{\mathtt{TRIP}}$ composed with $\pi_{\mathtt{ON\text{-}LINE}}$ where it corrupts the parties specified by $\mathcal{Z}$. The simulator relays messages between parties/$\mathsf{F}_{\mathtt{TRIP}}$ and $\mathcal{Z}$, such that $\mathcal{Z}$ will see the same interface as when interacting with a real protocol. During the run of the internal protocol the simulator will keep copies of the shares, MACs and keys of both honest and corrupted parties and update them according to the execution.

The idea is now, that the simulator runs the protocol with the environment, where it plays the role of the honest parties. Since the inputs of the honest parties are not known to the simulator these will be random values (or zero). However, if the protocol is secure, the environment will not be able to tell the difference.

Agent SIMULATOR $\mathcal{S}_{\texttt{ON-LINE}}$

In the following, $H$ and $C$ represent the set of corrupted and honest parties, respectively.

**Initialize:** The simulator initializes the copy with $\mathbb{F}$ and creates the desired number of triples. Here the simulator will read all data of the corrupted parties specified to the copy of $\mathsf{F}_{\texttt{TRIP}}$.

**Input:** If $\mathsf{P}_i \in H$ the copy is run honestly with dummy input, for example 0. If in Step 1 during input, the MACs are not correct, the protocol is aborted.

If $\mathsf{P}_i \in C$ the input step is done honestly and then the simulator waits for $\mathsf{P}_i$ to broadcast $\delta$. Given this, the simulator can compute $x'_i = a + \delta$ since it knows (all the shares of) $a$. This is the supposed input of $\mathsf{P}_i$, which the simulator now gives to the ideal functionality $\mathsf{F}_{\texttt{SFE}}^{f,abort}$.

**Add:** The simulator runs the protocol honestly and calls *add* on the ideal functionality $\mathsf{F}_{\texttt{SFE}}^{f,abort}$.

**Multiply:** The simulator runs the protocol honestly and, as before, aborts if some share from a corrupted party is not correct. Otherwise it calls *multiply* on the ideal functionality $\mathsf{F}_{\texttt{SFE}}^{f,abort}$.

**Output:** If $\mathsf{P}_i \in H$ the output step is run and the protocol is aborted if some share from a corrupted party is not correct. Otherwise the simulator calls *output* on $\mathsf{F}_{\texttt{SFE}}^{f,abort}$.

If $\mathsf{P}_i \in C$ the simulator calls *output* on $\mathsf{F}_{\texttt{SFE}}^{f,abort}$. Since $\mathsf{P}_i$ is corrupted the ideal functionality will provide the simulator with $y$, which is the output to $\mathsf{P}_i$. Now it has to simulate shares $y_j$ of honest parties such that they are consistent with $y$. This is done by changing one of the internal shares of an honest party. Let $\mathsf{P}_k$ be that party. The new share is now computed as $y'_k = y - \sum_{i \neq k} y_i$. Next, a valid MAC for $y'_k$ is needed. This, the simulator can compute from scratch as $\texttt{MAC}_{K^i_{y_k}}(y'_k)$ since it knows from the beginning the keys of $\mathsf{P}_i$. This enables it to compute $K^i_{y_k}$ by the computations on representations done during the protocol. Now the simulator sends the internal shares and corresponding MACs to $\mathsf{P}_i$.

In general, the openings in the protocol do not reveal any information about the values that the parties have. Whenever we open a value during Input or Multiply we mask it by subtracting with a new random value. Therefore, the distribution of the view of the corrupted parties is exactly the same in the simulation as in the real case. Then, the only method there is left for the environment to distinguish between the two cases, is to compare the protocol execution with the inputs and outputs of the honest parties and check for inconsistency.

If the simulated protocol fails at some point because of a wrong MAC, the simulator aborts which is consistent with the internal state of the ideal functionality since, in this case, the simulator also makes the ideal functionality fail.

If the simulated protocol succeeds, the ideal functionality is always told to output the result of the function evaluation. This result is of course the correct evaluation of the input matching the shares that were read from the corrupted parties in the beginning. Therefore, if the corrupted parties during the protocol successfully cheat with their shares, this would not be consistent.

However, as we have seen earlier, the probability of a party being able to claim a wrong value for a given MAC is $1/|\mathbb{F}|$. Therefore, since we abort as soon as a check fails, we conclude that if the protocol succeeds, the computation is correct except with probability $1/|\mathbb{F}|$. □

**Exercise 8.4** Show that $\pi_{\texttt{ON-LINE}} \diamond \mathsf{F}_{\texttt{TRIP}}$ implements $\mathsf{F}_{\texttt{SFE}}^{f,abort}$ in $\text{Env}^{t,\texttt{sync}}$ with statistical security for all $t < n$, i.e. $\pi_{\texttt{ON-LINE}}$ is in fact adaptively secure.

By inspection of the protocol, one sees that the work invested by each player in $\pi_{\texttt{ON-LINE}}$ amounts to $O(n|C|)$ elementary field operations, where $|C|$ is the size of the circuit computed. This basically follows from the fact that the cost of doing the multiplications dominate, and for each of these, a player needs to check $O(n)$ MACs. Thus, for a constant number of players, the work one needs to invest to do the protocol is comparable to the work one would need to just compute the circuit in the clear with no security properties ensured. Using a more complicated protocol, one can improve this to $O(|C|)$ operations (see the Notes section for more on this).

Also note that $\pi_{\texttt{ON-LINE}}$ is only really efficient if the circuit $C$ is defined over a large field $\mathbb{F}$, where large means that $1/|\mathbb{F}|$ is an acceptable error probability. If this is not the case, we could have more than one MAC on each value. This would make the error smaller, but the protocol would be less efficient. There are, however, solutions that are as efficient as $\pi_{\texttt{ON-LINE}}$ even for the field with 2 elements, i.e., for Boolean circuits (see the Notes section for more details).

Of course, in order to actually use a protocol like $\pi_{\texttt{ON-LINE}}$ in practice, one needs to also implement the preprocessing. This will require use of computationally secure cryptography (since otherwise we would contradict the fact that information theoretic security is not possible with dishonest majority), the details of this are not the main focus of this book. We just mention that it can be done quite efficiently using public-key cryptosystems with homomorphic properties. It is not hard to see why an additively homomorphic scheme would help here: suppose player $\mathsf{P}_i$ has a key pair $(sk, pk)$ and holds the secret key $sk$ while other players have the public key $pk$. Suppose further that the encryption function is linearly homomorphic over $\mathbb{F}$, i.e., we have $E_{pk}(a) + \alpha \cdot E_{pk}(b) = E_{pk}(a + \alpha b)$.

Now, if $\mathsf{P}_i$ has a message $a$ and $\mathsf{P}_j$ holds a MAC key $(\alpha, \beta)$, then $\mathsf{P}_i$ can send $E_{pk}(a)$ to $\mathsf{P}_j$ who can compute and return to $\mathsf{P}_i$ a ciphertext $E_{pk}(\alpha a + \beta)$. Decrypting this, $\mathsf{P}_i$ will get a MAC on $a$ under $\mathsf{P}_j$'s key. If $\mathsf{P}_j$ is able to do his operations such that the ciphertext he returns is a random ciphertext containing $\alpha a + \beta$, we are sure that $\mathsf{P}_i$ learns nothing except for the MAC and hence cannot beak the MAC scheme later. However, to have active security, we still need to ensure that players send well formed ciphertexts computed according to the protocol. The notes section has pointers to literature with details on how this can be done.

## 8.6 Open Problems in Complexity of MPC

In this section we give a short and very informal overview of some things we know about the complexity of multiparty computation with unconditional security and mention some open problems.

We begin with the model that we consider in most of this book, namely we assume secure point-to-point channels (and broadcast in case more than $n/3$ players are corrupt) and less than $n/2$ corruptions. For this case, the ideas of circuit randomization, hyper-invertible matrices, packed secret sharing and more lead to protocols where the total work players need to invest to compute circuit $C$ securely is essentially $|C|$ field operations (where we ignore some factors that depend logarithmically on $n$ and $|C|$, the notes section has more details). Since computing $C$ securely in particular means you compute it, we cannot hope to do better. It should be noted, however, that this result only holds in an asymptotic sense for a large number of players and large circuits, and the underlying protocols are not practical even if they are asymptotically efficient. The upper bound on the computational work also implies a similar bound on the communication complexity of the protocol. However, it is completely open whether we actually need to send $\Omega(|C|)$ field elements to compute $C$. For computational security, we know that fully homomorphic encryption (FHE) allows us to have protocols where the communication does not depend on $|C|$, but we do not know if something like "FHE for unconditional security" exists.

Another issue is the number of messages we need to send to compute $C$. Or, in case of synchronous networks, the number of rounds we need. All the general protocols we have seen require a number of rounds that is linear in the depth of $C$. We do not know if this is inherent, in fact, we do not even know which functions can be computed with unconditional security and a constant number of rounds. We do know large classes of functions that can indeed be computed with unconditional security and a constant number of rounds, but there is nothing to suggest that these functions are the only ones with this property.

If we go to the preprocessing model, we know protocols that require total computational work essentially $\Omega(n|C|)$ field operations. These protocols also require communication of $\Omega(n|C|)$ field elements and the players also need to store this many fields elements from the preprocessing. It can be shown that storage required is optimal, and the computational work per player is optimal, in the following sense: for any player $\mathsf{P}_i$, there exists a circuit $C$ for which secure computing of it in the preprocessing model would require $\mathsf{P}_i$ to do $\Omega(|C|)$ operations, but it is not guaranteed that other players would also have to work this hard when computing $C$. It is intuitively quite reasonable that each player needs to invest $\Omega(|C|)$ operations: the circuit needs to be computed, so it is clear that the *total* number of field operations done by the players should be $\Omega(|C|)$. However, it might be the case that only one player is honest and everybody else works against him, so it would be quite surprising if this player would not have to do work at least equivalent to computing the circuit on his own. We emphasize, however, that such a result is not known in full generality at the time of writing.

As for communication and rounds, also in the preprocessing model, it is completely open what the optimal complexities are.

## 8.7 Notes

The circuit randomization idea from the first section of the chapter was suggested by Beaver [Bea91a].

The hyper-invertible matrices and applications of these was suggested by Beerliová-Trubíniová and Hirt [BTH08], and packed secret sharing was suggested by Franklin and Yung [FY92].

The work in [BTH08] is one of many results in a line of research known as scalable MPC, where one tries to build protocols that scale well as the number of players increase. In [BTH08] it is shown how circuit randomization and hyper-invertible matrices lead to perfectly secure evaluation of a circuit $C$ for less than $n/3$ corrupted players and secure point-to-point channels with total communication complexity $O(n|C|)$ field elements plus an additive overhead that depends polynomially on the number of players and linearly on the depth of $C$, but not on the size of $C$. To reach this goal, [BTH08] also does an in-depth treatment of how one deals with the case where Protocol CHECK CONSISTENT SECRET-SHARES is not successful.

Ben-Sasson et al.[BSFO11] show a similar result for less than $n/2$ corruptions when broadcast is also given for free. They show a protocol with statistical security (we cannot do better in this model) and communication complexity $O((n \log n + k/n^c)|C|)$ field elements where $k$ is the security parameter and $c$ is an arbitrary constant. Here, there is again an additive term that depends polynomially on $n$ and $k$ and the depth of $C$ but not $|C|$.

A closely related line of research tries to minimize also the computational work as a function of the number of players. The latest work in this line is by Damgård et al. [DIK10] who show a perfectly secure protocol for less than $(1/3-\epsilon)n$ corruptions, where $\epsilon > 0$ is an arbitrary constant. In this protocol the total number of field operations players need to do is $|C| \log |C| \log^2 n$ plus an additive term that depends polynomially on $n$ and the depth of $C$ but only logarithmically on $|C|$. On one hand this is stronger than [BTH08] because the dependency on $n$ is reduced and it is the computational work that is optimized. On the other hand, the corruption tolerance is not quite optimal (because of the $\epsilon$).

The protocol in [DIK10] is based on packed secret sharing and therefore needs to work with only block-wise operations where the same operation is done on all positions in a block, but still one needs to compute an arbitrary Boolean circuit. This is handled using a technique by which arbitrary circuit can be restructured so it computes the same function but the new circuit only uses block-wise operations and a small number of intra-block permutation, and is a logarithmic factor larger. We have already seen passively secure protocols for packed secret sharing, active security is obtained using error correction which works if the number of honest players is large enough. The use of error correction and packed secret sharing means that the basic protocol $\pi$ is only secure against a small (but constant) fraction of corrupted players. This is then boosted to the almost optimal threshold of $(1/3 - \epsilon)n$ using the *committees technique*. Here, we form a number of committees each consisting of a constant number of players. Each committee emulates a player in $\pi$. The emulation is done using an expensive protocol with optimal threshold (but this does not hurt overall asymptotic efficiency as committees have constant size). If at most $(1/3 - \epsilon)n$ players are corrupt, we can choose the committees in such a way that only a small constant fraction of them contain more than a third corrupted players. This means that the (emulated) execution of $\pi$ will be secure.

The committees approach was suggested by Bracha [Bra87] for the purpose of broadcast and was introduced in multiparty computation in [DIK$^+$08].

We emphasize that all the results on scalable MPC that we mentioned hold asymptotically for a large number of players and even larger circuits. This is because the protocols typically use the approach we saw in Protocol CHECK CONSISTENT SECRET-SHARES, where one runs a protocol that very efficiently handles a large number of secret-sharings and works correctly as long as players follow the protocol. On the other hand, honest players can verify that the protocol has been followed and complain if this is not the case. If complaints occur we need to resolve the conflicts and this usually requires doing something very expensive. This approach makes sense if the computation we do is large enough that it makes sense to do many secret-sharings in parallel, *and* if we can make sure that complaints will not occur very often, in which case the cost of handling them will be insignificant compared to the rest of the work.

The material in the section on preprocessing is from Bendlin et al.[BDOZ11]. This paper also introduces the notion of semi-homomorphic public-key encryption and shows it is sufficient for implementing the preprocessing specified in $\mathsf{F}_{\mathtt{TRIP}}$. In a subsequent paper, Damgård et al. [DPSZ11] show that the total computational complexity can be improved from quadratic in $n$ to $O(n|C| + n^3)$ field operations, where the error probability is $1/|\mathbb{F}|$. The amount of data players have to store from the preprocessing as well as the communication complexity is also $O(n|C| + n^3)$ field elements. It is also shown in [DPSZ11] that the required storage from the preprocessing is optimal and the computational work is optimal is the sense explained earlier: for any player $\mathsf{P}_i$, there exists a circuit $C$ for which secure computing of it in the preprocessing model would require $\mathsf{P}_i$ to do $\Omega(|C|)$ operations, but it is not guaranteed that other players would also have to work this hard. Finally, [DPSZ11] shows how to implement the preprocessing more efficiently based on somewhat homomorphic encryption. Informally, this is a weaker form of fully homomorphic encryption where many additions but only a small bounded number of multiplications can be performed.

# Chapter 9

# Asynchronous MPC Protocols

In this chapter we cover:

- (Im)possibility results for asynchronous MPC

# Chapter 10

# Secret Sharing

In this chapter we cover:

- Formal definition of secret sharing schemes and related concepts
- General (im)possibility results on secret sharing
- Asymptotics of strongly multiplicative linear Secret Sharing

# Chapter 11

# Applications of MPC

In this chapter we cover:

- Secure Integer Comparison
- Secure Auctions
- MPC used for two-party protocols: MPC in the head etc.

# Chapter 12

# Historical Notes

**Contents**

## 12.1 Introduction

This chapter gives a short historical overview of the early research in Multiparty Computation and explains where the material in the first chapters of the book comes from. We do not intend to give a complete account of even the early research in the field, indeed this would border on impossible. Instead we focus on the research that has formed the background for this book. We give a few pointers for additional reading, but note that more such pointers can be found in the chapters on Cryptographic Protocols, Efficiency Improvements and Applications, these have their own notes on background literature.

## 12.2 The Basic Completeness Theorems

The history of Multiparty Computation starts in 1986 with the work of Yao [Yao86] who suggested the famous Yao-garbling technique. This technique can actually not be found in [Yao86], but was apparently mentioned in an oral presentation of that work. The garbling technique shows that any function can be securely evaluated with computational security by two players.

In 1987 Goldreich, Micali and Wigderson [GMW87] showed the same result for any number of players and honest majority, based on existence of one-way trapdoor permutations. This result was also obtained independently, but later, by Chaum, Damgaard and van de Graaf [CDG87], based on the quadratic residuosity assumption. It should be mentioned that Yao-garbling leads to constant-round protocols, which is not the case for [GMW87, CDG87]. Constant round for an arbitrary number of players was obtained in 1990 by Beaver, Micali and Rogaway [BMR90].

While all the work mentioned so far was based on computational assumptions the work that large parts of this book is based on was started in 1988 by Ben-Or, Goldwasser and Wigderson [BGW88] and independently by Chaum, Crépeau and Damgård [CCD88]. They showed that in the model with secure point-to-point channels, any function can be securely evaluated with unconditional security against $t$ corrupted players, where $t < n/2$ for a passive adversary and $t < n/3$ for an active adversary. In 1989, Ben-Or and Rabin [RB89] showed that if broadcast is given for free and an small error probability is allowed, $t < n/2$ can be obtained even for an active adversary.

One can also consider so-called mixed adversaries that may corrupt some players passively and others actively. Also for this case one can show optimal bounds on the number of corruptions that can be tolerated, this was done in 1998 by Fitzi, Hirt and Maurer [FHM98]. If $t_p, t_a$ are the number of passively and actively corrupted players, respectively, then perfectly secure general multiparty computation is possible if and only if $2t_p + 3t_a < n$ [1].

Security against general adversary structures was first considered by Hirt and Maurer [HM00], who showed that the basic completeness theorems for unconditional security generalize from $t < n/2, t < n/3$ to $Q2$ and $Q3$ adversary structures, respectively.

In 2000, Cramer, Damgård and Maurer [CDM00] showed that passive and active multiparty computation follows from multiplicative and strongly multiplicative linear secret sharing schemes, respectively, where the complexity of the protocol is polynomial in the size of the secret sharing scheme.

One may wonder if this result is the most general one possible, i.e., can one base unconditionally secure multiparty computation on *any* secret scheme? An answer to this was given by Cramer, Damgård and Dziembowski [CDD00] who showed that, although verifiable secret sharing does follow by an efficient black-box reduction from any secret sharing scheme, no such reduction exists for multiparty computation, at least not one that would be efficient for any access structure. Therefore some extra property (such as linearity) must be assumed for a general reduction to exist.

Concerning the material in this book, the passively secure protocol we give in Chapter 3 is essentially the protocol from [BGW88], except that the way we do multiplications is different. The approach we use, where players multiply shares locally and re-share the product, was suggested by Michael Rabin, shortly after [BGW88] was published.

The actively secure protocol from Chapter 5 is again similar to [BGW88] in spirit, but with some major differences in how multiplications are handled. We do verifiable secret sharing by having players commitment to the shares they hold in the basic secret sharing scheme, and then we implement the commitments using secret sharing again. This approach is from [CDM00]. It makes the presentation more modular and it also allows a simple generalization to linear secret sharing. The resulting protocol is, however, less efficient compared to [BGW88] who do verifiable secret sharing directly, without passing through a construction of commitments. This is possible because the protocol for commitment is actually already a verifiable secret sharing when the underlying secret sharing scheme is Shamir's. This is not true for linear secret sharing schemes in general, so in the later chapter on linear secret sharing, the modular approach is actually necessary.

The results on linear secret sharing in Chapter 6 are from [CDM00], except the efficient construction of secret sharing for the dual access structure which is by Serge Fehr (manuscript, available from the author's web page). The model of linear secret sharing that we use in this chapter is essentially that of monotone span programs which were introduced by Karchmer and Wigderson [KW93]. We do not, however, introduce the full formalism of span programs, this is more relevant if one wants to see span programs as a model of computation, which is not our focus here.

## 12.3 Models and Definitions of Security

In hindsight, it can seem quite surprising that at the time when the first completeness results were established, we did not have generally accepted definitions of security for multiparty computation protocols. These crystallized only much later after a lot of research work. There are many reasons for this, but a main complication was the issue of composition: It is a natural requirement that protocols should remain secure even when composed with other protocols and for a long time it was not clear how a definition should be put together to ensure this.

[1]The reader should note that the proceedings version of [FHM98] states a different incorrect result, the authors proved the correct result later.

In 1991, Micali and Rogaway [MR91] as well as Beaver[Bea91b] put forward definitions, but it was not until the work around 2000 of Canetti [Can01] and independently Pfitzmann, Schunter and Waidner [PSW00] that security under arbitrary concurrent composition could be expressed.

It is natural to ask if the early protocols actually turned out to be secure under these later definition? It turns out that all the protocols with unconditional security underlying the completeness results from the late 80-ties are in fact secure in the strongest sense: they are UC secure against adaptive adversaries (see below for more on the history of the security definitions). In contrast, the computationally secure protocols, for instance [GMW87] are only known to be statically secure. This is mainly due to a technical problem related to the fact that when simulating encrypted communication, the simulator will implicitly commit itself to some plaintext that it would have to change if the sender or receiver are corrupted. Adaptive security can be obtained using so called non-commiting encryption [CFGN96], but only at the cost of significant technical complications.

It may superficially seem that all the technical problems that prevent us from proving adaptive security in the computational case go away in the information theoretic setting. This is not true, however, there are natural examples of information theoretically secure protocols that are statically secure but not adaptively secure [CDD$^+$99].

The model and security definitions in this book are based on Canetti's work on Universal Composition, the UC model. We depart somewhat, however, from his original model. Basically, we sacrifice some generality to have a simpler model that is easier to explain and use. For instance, we demand that agents pass control in a certain nicely structured way, in order to preserve polynomial running time when we compose agents. We also define our agents as the more abstract IO automata rather than Interactive Turing machines.

It should also be mentioned that several variants of the UC model have been proposed more recently in the literature, see for instance the GUC framework by Canetti, Dodis, Pass and Walfish [CDPW07] and the GNUC framework by Hofheintz and Shoup [HS11].

# Bibliography

[ACM88] *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, Chicago, Illinois, 2–4 May 1988.

[BDOZ11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In Kenneth G. Paterson, editor, *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 169–188. Springer, 2011.

[Bea91a] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *Advances in Cryptology - Crypto '91*, pages 420–432, Berlin, 1991. Springer-Verlag. Lecture Notes in Computer Science Volume 576.

[Bea91b] Donald Beaver. Foundations of secure interactive computing. In Joan Feigenbaum, editor, *Advances in Cryptology - Crypto '91*, pages 377–391, Berlin, 1991. Springer-Verlag. Lecture Notes in Computer Science Volume 576.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In ACM [ACM88], pages 1–10.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *Proceedings of the Twenty Second Annual ACM Symposium on Theory of Computing*, pages 503–513, Baltimore, Maryland, 14–16 May 1990.

[Bra87] Gabriel Bracha. An o(log n) expected rounds randomized byzantine generals protocol. *J. ACM*, 34(4):910–920, 1987.

[BSFO11] Eli Ben-Sasson, Serge Fehr, and Rafail Ostrovsky. Near-linear unconditionally-secure multiparty computation with a dishonest minority. *IACR Cryptology ePrint Archive*, 2011:629, 2011.

[BTH08] Zuzana Beerliová-Trubíniová and Martin Hirt. Perfectly-secure mpc with linear communication complexity. In Ran Canetti, editor, *TCC*, volume 4948 of *Lecture Notes in Computer Science*, pages 213–230. Springer, 2008.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science*, pages 136–145, Las Vegas, Nevada, 14–17 October 2001. IEEE. Full version in [?].

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In ACM [ACM88], pages 11–19.

[CDD$^+$99] Ronald Cramer, Ivan Damgård, Stefand Dziembowski, Martin Hirt, and Tal Rabin. Efficient multiparty computations secure against an adaptive adversary. In Jacques Stern, editor, *Advances in Cryptology - EuroCrypt '99*, pages 311–326, Berlin, 1999. Springer-Verlag. Lecture Notes in Computer Science Volume 1592.

[CDD00] Ronald Cramer, Ivan Damgård, and Stefan Dziembowski. On the complexity of verifiable secret sharing and multiparty computation. In *Proceedings of the Thirty-Second Annual ACM Symposium on the Theory of Computing*, pages 325–334, Portland, OR, USA, 21–23 May 2000.

[CDG87] David Chaum, Ivan Damgård, and Jeroen van de Graaf. Multiparty computations ensuring privacy of each party's input and correctness of the result. In Carl Pomerance, editor, *Advances in Cryptology - Crypto '87*, pages 87–119, Berlin, 1987. Springer-Verlag. Lecture Notes in Computer Science Volume 293.

[CDM00] Ronald Cramer, Ivan Damgård, and Ueli Maurer. General secure multi-party computation from any linear secret-sharing scheme. In Bart Preneel, editor, *Advances in Cryptology - EuroCrypt 2000*, pages 316–334, Berlin, 2000. Springer-Verlag. Lecture Notes in Computer Science Volume 1807.

[CDN01] Ronald Cramer, Ivan Damgaard, and Jesper Buus Nielsen. Multiparty computation from threshold homomorphic encryption. In *Advances in Cryptology - EuroCrypt 2001*, pages 280–300, Berlin, 2001. Springer-Verlag. Lecture Notes in Computer Science Volume 2045.

[CDPW07] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In Salil P. Vadhan, editor, *TCC*, volume 4392 of *Lecture Notes in Computer Science*, pages 61–85. Springer, 2007.

[CFGN96] Ran Canetti, Uri Feige, Oded Goldreich, and Moni Naor. Adaptively secure multi-party computation. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing*, pages 639–648, Philadelphia, Pennsylvania, 22–24 May 1996.

[CKL06] Ran Canetti, Eyal Kushilevitz, and Yehuda Lindell. On the limitations of universally composable two-party computation without set-up assumptions. *J. Cryptology*, 19(2):135–167, 2006.

[CLOS02] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *Proceedings of the Thirty-Fourth Annual ACM Symposium on the Theory of Computing*, pages 494–503, Montreal, Quebec, Canada, 2002.

[DIK$^+$08] Ivan Damgård, Yuval Ishai, Mikkel Krøigaard, Jesper Buus Nielsen, and Adam Smith. Scalable multiparty computation with nearly optimal work and resilience. In David Wagner, editor, *CRYPTO*, volume 5157 of *Lecture Notes in Computer Science*, pages 241–261. Springer, 2008.

[DIK10] Ivan Damgård, Yuval Ishai, and Mikkel Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In Henri Gilbert, editor, *EUROCRYPT*, volume 6110 of *Lecture Notes in Computer Science*, pages 445–465. Springer, 2010.

[DN00] Ivan Damgård and Jesper Buus Nielsen. Improved non-committing encryption schemes based on a general complexity assumption. In Mihir Bellare, editor, *Advances in Cryptology - Crypto 2000*, pages 432–450, Berlin, 2000. Springer-Verlag. Lecture Notes in Computer Science Volume 1880.

[DN03] Ivan Damgård and Jesper Buus Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In D. Boneh, editor, *Advances in Cryptology - Crypto 2003*, pages 247–264, Berlin, 2003. Springer-Verlag. Lecture Notes in Computer Science Volume 2729.

[DPSZ11] I. Damgard, V. Pastro, N.P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. Cryptology ePrint Archive, Report 2011/535, 2011.

[FHM98] Matthias Fitzi, Martin Hirt, and Ueli M. Maurer. Trading correctness for privacy in unconditional multi-party computation (extended abstract). In Hugo Krawczyk, editor, *CRYPTO*, volume 1462 of *Lecture Notes in Computer Science*, pages 121–136. Springer, 1998.

[FM98] Matthias Fitzi and Ueli M. Maurer. Efficient byzantine agreement secure against general adversaries. In Shay Kutten, editor, *DISC*, volume 1499 of *Lecture Notes in Computer Science*, pages 134–148. Springer, 1998.

[FY92] Matthew K. Franklin and Moti Yung. Communication complexity of secure computation (extended abstract). In *STOC*, pages 699–710. ACM, 1992.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *STOC*, pages 169–178. ACM, 2009.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pages 218–229, New York City, 25–27 May 1987.

[HM00] Martin Hirt and Ueli Maurer. Player simulation and general adversary structures in perfect multiparty computation. *Journal of Cryptology*, 13(1):31–60, winter 2000.

[HS11] Dennis Hofheinz and Victor Shoup. Gnuc: A new universal composability framework. *IACR Cryptology ePrint Archive*, 2011:303, 2011.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boheh, editor, *Advances in Cryptology - Crypto 2003*, pages 145–161, Berlin, 2003. Springer-Verlag. Lecture Notes in Computer Science Volume 2729.

[Kil88] Joe Kilian. Founding cryptography on oblivious transfer. In ACM [ACM88], pages 20–31.

[KW93] Mauricio Karchmer and Avi Wigderson. On span programs. In *Structure in Complexity Theory Conference*, pages 102–111, 1993.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):381–401, July 1982.

[MR91] Silvio Micali and Phillip Rogaway. Secure computation. In Joan Feigenbaum, editor, *Advances in Cryptology - Crypto '91*, pages 392–404, Berlin, 1991. Springer-Verlag. Lecture Notes in Computer Science Volume 576.

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residue classes. In Jacques Stern, editor, *Advances in Cryptology - EuroCrypt '99*, pages 223–238, Berlin, 1999. Springer-Verlag. Lecture Notes in Computer Science Volume 1592.

[PSW00] Birgit Pfitzmann, Matthias Schunter, and Michael Waidner. Secure reactive systems. Technical Report RZ 3206, IBM Research, Zürich, May 2000.

[RB89] Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, pages 73–85, Seattle, Washington, 15–17 May 1989.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science*, pages 162–167, Toronto, Ontario, Canada, 27–29 October 1986. IEEE.

# Notation

Not all in the below list should survive to the final version, but it is nice to have for our own use. This todo note was produced by writing `\todo{Not all ... }`

- $x \overset{\text{def}}{=} y$, states that $x$ is defined by $y$, e.g., $\delta(x,y) \overset{\text{def}}{=} |x-y|$. (`\Def`)
- $x = y$, the statement that $x$ is equal to $y$, e.g., $2+2=4$.
- $x \leftarrow y$, the action that $x$ is assigned $y$, e.g., $i \leftarrow 1$. (`\leftarrow`)
- $x \leftarrow D$, where $D$ is a distribution, means that $x$ is sampled according to $D$.
- $x \in_{\mathrm{R}} S$, where $S$ is a finite set, means that $x$ is sampled uniformly at random from $S$. (`\inR`)
- $a \leftarrow A(x;r)$, running randomized algorithm on input $x$ and randomness $r$ and assigning the output to $a$.
- $\mathbb{N} \overset{\text{def}}{=} \{0,1,2,\ldots\}$. (`\N`)
- $\mathbb{Z}_n \overset{\text{def}}{=} \{0,\ldots,n-1\}$. (`\Z_n`)
- $\mathbb{F}$, generic finite field. (`\F`)
- $\mathrm{GF}(q^e)$, the Galois field with $q^e$ elements. (`\GF(q^e)`)
- $[a,b]$, closed interval from $a$ to $b$. (`[a,b]`)
- $\kappa \in \mathbb{N}$, the security parameter. (`\kappa`)
- $\epsilon$, the empty string.
- $\Pr[E|C]$, probability of event $E$ given event $C$. (`\Pr{E \vert C}`)
- $\Pr[a \leftarrow A; b \in_{\mathrm{R}} B : E]$, probability of event $E$ given $a$ and $b$ are generated as specified. (`\Pr{... : E}`)
- $\delta(X,Y)$, statistical distance between random variable $X$ and $Y$. (`\delta(X,Y)`)
- $\textsc{Adv}_A(X,Y)$, (distinguishing) advantage of $A$ for random variables $X$ and $Y$. (`\Adv_A(X,Y)`)
- $\overset{\text{perf}}{\equiv}, \overset{\text{stat}}{\equiv}, \overset{\text{comp}}{\equiv}$, perfectly indistinguiahable, statistically indistinguiahable respectively computationally indistinguiahable. (`\perfind,\statind,\compind`)
- $\overset{\text{perf}}{\not\equiv}, \overset{\text{stat}}{\not\equiv}, \overset{\text{comp}}{\not\equiv}$, *not* perfectly indistinguiahable, statistically indistinguiahable respectively computationally indistinguiahable. (`\nperfind,\nstatind,\ncompind`)
- $\mathtt{X}$, formal variable in polynomials, $f(\mathtt{X})$. (`\X`)
- $1,2,\ldots,n \in \mathbb{F}$, naming of $n$ distinct, non-zero elements from $\mathbb{F}$, for secret sharing.

- $[a; f] = (f(1), \ldots, f(n))$, secret sharing of $a$ using polynomial $f$. States that $f(0) = a$. (`[a;f]`)
- $[a]$, secret sharing of $a$. States that there exists $f(\mathtt{X})$ such that $[a] = [a; f]$. (`[a]`)
- $\deg(f(\mathtt{X}))$, degree of polynomial. (`\deg`)
- $[a]_d$, degree-$d$ secret sharing of $a$. States that there exists $f(\mathtt{X})$ such that $\deg(f(\mathtt{X})) \leq d$ and $[a] = [a; f]$. (`[a]_d`)
- $[a; \rho]_M = M(a, \rho)$, secret sharing of $a$ using matrix $M$.
- $U, V, M$, vector spaces and matrices are in capital, normal font.
- $\mathsf{A}, \mathsf{B}, \mathsf{P}_i$, agents and parties in interactive systems. (`\ant{A},\ant{B},\ant{P}_i`)
- $\mathcal{IS}$, interactive system. (`\IS`)
- $\mathcal{IS}_1 \diamond \mathcal{IS}_2$, composing interactive systems (`\IS_1\Comp\IS_2`)
- $A^{\mathsf{T}}$, matrix transpose. (`A^\tr`)
- $AB$, matrix multiplication.
- $\mathbf{r} = (r_1, \ldots, r_\ell)$, column vector. (`\vec{r}`)
- $\mathbf{a} \cdot \mathbf{b} = \mathbf{a}^{\mathsf{T}}\mathbf{b}$, inner product. (`\cdot`)
- For $\mathbf{a} = (a_1, \ldots, a_n)$ and $\mathbf{b} = (b_1, \ldots, b_n)$, $\mathbf{a} * \mathbf{b} = (a_1 b_1, \ldots, a_n b_n)$, Schur product. (`\schur`)
- $a_{ij}$, entry in row $i$, column $j$ in matrix $A$.
- $\mathbf{a}_i$ and $\mathbf{a}_{i*}$, row $i$ of matrix $A$. (`\vec{a}_i,\vec{a}_{i*}`)
- $\mathbf{a}_{*j}$, column $j$ of $A$. (`\vec{a}_{*j}`)
- ker, kernel. (`\ker`)

The following LaTeX-code produces the below figures and the first two lines in the **List of Algorithms** section.

```
\begin{algorithm}{Simple}
  \begin{enumerate}
  \item output $42$.
  \end{enumerate}
\end{algorithm}

\begin{protocol}{Trivial}
  \begin{enumerate}
  \item \ant{S}: send $42$ to \ant{R}.
  \end{enumerate}
\end{protocol}
\noindent
\algoref{Simple} is a simple algorithm, and \protref{Trivial} is a
trivial protocol.
```

Algorithm SIMPLE

1. output 42.

Protocol TRIVIAL

1. S: send 42 to R.

Algorithm SIMPLE is a simple algorithm, and Protocol TRIVIAL is a trivial protocol.

The following LaTeX-code produces the below exercise:

```
\begin{exercise}\exelab{hard}
  Solve this exercise.
\end{exercise}
\exeref{hard} can be hard.
```

**Exercise 12.1** Solve this exercise.

Exercise 12.1 can be hard.

The following LaTeX-code produces the below example:

```
\begin{example}\exalab{exex}
  Example of an example.
\end{example}
\exaref{exex} is an example of an example of an example.
```

**Example 12.1** Example of an example. △

Example 12.1 is an example of an example of an example.

The following LaTeX-code produces the below theorem:

```
\begin{theorem}\thmlab{tau}
  Theorems are true.
\end{theorem}
\thmref{tau} is true.
```

**Theorem 12.1** *Theorems are true.*

Theorem 12.1 is true.

There are similar environments/commands for lemmas, propositions and corollaries: `lemma`, `\lemmlab`, `\lemmref`; `proposition`, `\proplab`, `\propref`; `corollary`, `\corlab`, `\corref`;

There is also a `proof` environment, producing something like:
*Proof* Follows by construction. □

Chapters can be refed using `\chpref`{Name}, where Name is the caption of the chapter without whitespace. As an example, the LaTeX-code `\chpref`{`MultipartyComputation`} will produce Chapter 3.

There is a similar command `\secref`{Name}, and sections are labeled using `\seclab`{Name}.

Equations are labeled and refed as follows:

```
\begin{equation}\eqlab{i}
i
\end{equation}
In \eqref{i} we simply show an $i$.
```

$$i \tag{12.1}$$

In Eq. 12.1 we simply show an $i$.

One can make and ref a boxed figure as follows:

```
\begin{boxfig}{This shows text in a box.}{text}
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam
blandit, ante quis auctor fringilla, ante quam pretium quam, ut
aliquam est nunc quis nibh. Maecenas vitae diam ante. Mauris sed
sapien vitae purus tristique luctus eget id ante. Sed augue eros,
dapibus vel sagittis in, bibendum ac ipsum.
\end{boxfig}
See \figref{text}.
```

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam blandit, ante quis auctor fringilla, ante quam pretium quam, ut aliquam est nunc quis nibh. Maecenas vitae diam ante. Mauris sed sapien vitae purus tristique luctus eget id ante. Sed augue eros, dapibus vel sagittis in, bibendum ac ipsum.

Figure 12.1: This shows text in a box.

See Fig. 12.1.

The first time some important notion `name` is defined, write `\defterm{name}`, which will produce something like `name`. Use `\idefterm{name}` to put `name` in the index at the same time.

# List of Algorithms

Add agents to this list.

# List of Todos

## Todo list