Multiparty Computation, an Introduction

Ronald Cramer Ivan Damgård Jesper Buus Nielsen

May 25, 2008

Abstract

These lecture notes introduce the notion of secure multiparty computation. We introduce the universal composition framework for phrasing and proving security about protocols, and survey some known general results that describe when secure multiparty computation is possible. We then look at some general techniques for building secure multiparty protocols, including protocols for commitment and verifiable secret sharing, and we show how these techniques together imply general secure multiparty computation.

Our goal with these notes is to convey an understanding of some basic ideas and concepts from this field, rather than to give a fully formal account of all proofs and details. We hope the notes will be accessible to most graduate students in computer science and mathematics with an interest in cryptography.

Contents

1 What is Multiparty Computation?

In this section we give an overview of some of the central concepts in multiparty computation.

1.1 The MPC and VSS Problems

Secure *multiparty computation* (MPC) can be defined as the problem of n players to compute an agreed function of their inputs in a secure way, where security means guaranteeing the correctness of the output as well as the privacy of the players' inputs, even when some players cheat. Concretely, we assume we have inputs $x_1, ..., x_n$, where player i knows x_i, and we want to compute $f(x_1, ..., x_n) = (y_1, ..., y_n)$ such that player i is guaranteed to learn y_i, but can get nothing more than that. If all outputs are the same we often write $f(x_1, ..., x_n) = y$. Sometimes parties need to learn a randomized function of their inputs. In that case they evaluate a function $f(x_1, ..., x_n; r) = (y_1, ..., y_n)$, where r is a uniformly random value *unknown by all parties*.

As a toy example we may consider Yao's millionaire's problem: two millionaires meet in the street and want to find out who is richer. Can they do this without having to reveal how many millions they each own? The function computed in this case is a simple comparison between two integers: $f(x_1, x_2) = x_1 <^? x_2$ — here $x_1 <^? x_2$ is a function which is 1 if $x_1 < x_2$ and 0 otherwise. If the result is that the first millionaire is richer, then he knows that the other guy has fewer millions than him, but this should be all the information he learns about the other guy's fortune.

Another example is an electronic voting scheme: here all players have an integer as input, designating the candidate they vote for, and the goal is to compute how many votes each candidate has received. We want to make sure that the correct result of the vote, but *only* this result, is made public. If there are only two candidates and $x_i = 0$ is a vote on the first candidate and $x_i = 1$ is a vote on the second candidate, and we let the first candidate win if there is a draw, then the election should only leak the single bit $f(x_1, \ldots, x_n) = (\sum_{i=1}^{n} x_i) <^? n/2$ — here $a <^? b$ is a function which is 1 if $a < b$ and 0 otherwise. If we want to know how many votes each candidate got the function would be $f(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i$.

For a sealed-bid auction the function computed could be $f(x_1, \ldots, x_n) = (i, x_i)$, where $x_i = \max_{j=1}^{n} x_j$. If there are several parties P_j with $x_i = \max_{j=1}^{n} x_j$, then a random such party could be elected by evaluating a randomized function.

In the above examples all players learn the same result, i.e, $y_1 = \ldots = y_n$, but it can also be useful to have different results for different players. Consider for example the case of a blind signature scheme, which is useful in electronic cash systems. We can think of this as a two-party secure computation where the signer enters his private signing key sk as input, the user enters a message m to be signed, and the function $f(sk, m) = (y_1, y_2)$, where $y_1 = \epsilon$ is for the signer and is always the empty string, and where y_2 is for the user and is the signature on m. Again, security means exactly what we want: the user gets the signature and nothing else, while the signer learns nothing new. If the signature scheme is randomized, the parties would have to evaluate $f(sk, m; r)$, where r is used to compute the signature.

It is clear that if we can compute *any* function securely, we have a very powerful tool. However, some protocol problems require even more general ways of thinking. A secure payment system, for instance, cannot naturally be formulated as secure computation of a single function: what we want here is to continuously keep track of how much money each player has available and avoid cases where for instance people spend more money than they have. Such a system should behave like a secure general-purpose computer: it can receive inputs from the players at several points in time and each time it will produce results for each player computed in a specified way from the current inputs and from previously stored values. Therefore, the definition we give later for security of protocols, will be for this more general type, namely a variant of the *Universally Composable* security definition of Canetti. Another remark is that although the general protocol constructions we give are phrased as solutions to the basic MPC problem, they can in fact also handle the more general type of problem.

A key tool for secure MPC, interesting in its own right, is *verifiable secret sharing* (VSS): a dealer distributes a secret value s among the players, where the dealer and/or some of the players may be cheating. It is guaranteed that if the dealer is honest, then the cheaters obtain no information about s, and all honest players are later able to reconstruct s, even against the actions of cheating players. Even if the dealer cheats, a unique such value s will be determined already at distribution time, and again this value is reconstructable even against the actions of the cheaters.

1.2 Adversaries and Their Powers

It is common to model cheating by considering an adversary who may corrupt some subset of the players. For concreteness, one may think of the adversary as a hacker who attempts to break into the players' computers. When a player is corrupted, the adversary gets all the data held by this player, including complete information on all actions and messages the player has received in the protocol so far. This may seem to be rather generous to the adversary, for example one might claim that the adversary will not learn that much, if the protocol instructs players to delete sensitive information when it is no longer needed. However, first other players cannot check that such information really is deleted, and second even if a player has every intention of deleting for example a key that is outdated, it may be quite difficult to ensure that the information really is gone and cannot be retrieved if the adversary breaks into this player's computer. Hence the standard definition of corruption gives the entire history of a corrupted player to the adversary.

1.2.1 The Monolithic Adversary

The adversary cannot only model a hacker. It can also model that some of the parties in the protocol are trying to cheat by running alternative programs from those suggested by the protocol. In the case where an adversary e.g. models three separate parties which do not follow the suggested protocol it might seem overly pessimistic to model them by one adversary which controls these three parties, as this implicitly assumed that the three deviators are coordinating their cheating. Since, however, the corrupted parties coordinating their cheating is the worst-case seen from the point of view of the honest parties,

nothing is lost, security-wise, in considering one adversary which coordinates the actions of all corrupted parties. This is sometimes called a monolithic adversary. The model uses a monolithic adversary as it captures the worst case and makes the definition of security simpler.

1.2.2 Passive versus Active

One can distinguish between passive and active corruption. Passive corruption means that the adversary obtains the complete information held by the corrupted players, but the players still execute the protocol correctly. Active corruption means that the adversary takes full control of the corrupted players. One can think of passive corruption as a hacker which is able to inspect the execution of some parties but not control it, maybe because of an operating system with bad security. Alternatively it can be thought of as some of the parties in the protocol getting together with all the messages they saw during the execution of the protocol and trying to deduce more information about e.g. the inputs of the other parties.

1.2.3 The Adversary Structure

It is (at least initially) unknown to the honest players which subset of players is corrupted. However, no protocol can be secure if *any* subset can be corrupted. For instance, we cannot even define security in a meaningful way if all players are corrupt. We therefore need a way to specify some limitation on the subsets the adversary can corrupt. For this, we define an adversary structure $\mathcal{A}$, which is simply a family of subsets of the players. And we define an $\mathcal{A}$-adversary to be an adversary that can only corrupt a subset of the players if that subset is in $\mathcal{A}$. The adversary structure could for instance consist of all subsets with cardinality less than some threshold value t. In order for this to make sense, we must require for any adversary structure that if $A \in \mathcal{A}$ and $B \subset A$, then $B \in \mathcal{A}$. The intuition is that if the adversary is powerful enough to corrupt subset A, then it is reasonable to assume that he can also corrupt any subset of A. We say that an adversary structure must be monotone.

If we allow the adversary to corrupt all subsets of the parties of size at most t for some $t < n$, then we call it a threshold adversary and we call t the threshold.

1.2.4 Static versus Adaptive

Both passive and active adversaries may be static, meaning that the set of corrupted players is chosen once and for all before the protocol starts, or adaptive meaning that the adversary can at any time during the protocol choose to corrupt a new player based on all the information he has at the time, as long as the total corrupted set is in $\mathcal{A}$. A static adversary is a good model of a situation where some of the parties before the execution of the protocol get together and form a collusion against the other parties. Such a collusion could choose to either pool their view of the protocol after the execution (passive corruption) or could run some alternative coordinated programs (active corruption). Adaptive corruption is a better model of, e.g., a hacker which is trying to learn information by breaking into some

parties. Such a hacker could pick the next party to try to hack based on the information already collected.

1.3 Models of Communication

Two basic models of communication have been considered in the literature. In the cryptographic model, the adversary is assumed to have access to all messages sent, however, he cannot *modify* messages exchanged between honest players. This models a setting where all parties share an authenticated but otherwise insecure channel. This means that security can only be guaranteed in a cryptographic sense, i.e. assuming that the adversary cannot solve some computational problem.

In the information-theoretic model, it is assumed that the players can communicate over pairwise secure channels, in other words, the adversary gets no information at all about messages exchanged between honest players (except that something was sent). Security can then be guaranteed even when the adversary has unbounded computing power. In the information theoretic model is sometimes called the i.t. model and the secure-channels model.

For active adversaries, there is a further 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. In the distributed computing literature the term *broadcast* is sometimes used to refer to communication mechanisms which do not necessarily guarantee consistency if the sender is corrupted. We want to avoid this possible confusion, and therefore use the term consensus broadcast. 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*. 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.

We assume throughout that communication is synchronous, i.e., processors have clocks that are to some extent synchronized, and when a message is sent, it will arrive before some time bound. In more detail, we assume that a protocol proceeds in rounds: in each round, each player may send a message to each other player, and all messages are delivered before the next round begins. We assume that in each round, the adversary first sees all messages sent by honest players to corrupt players (or in the cryptographic scenario, all messages sent). If he is adaptive, he may decide to corrupt some honest players at this point. And only then does he have to decide which messages he will send on behalf of the corrupted players. This fact that the adversary gets to see what honest players say before having to act himself is sometimes referred to as a rushing adversary. Since communication in practice are not atomic events, but are implemented using sub-protocols, corrupted parties often has the ability to be last in practical networks — they can e.g. just fake that their TCP connection is hanging until they received messages from the honest parties. It is therefore important to assume a rushing adversary for the model to reflect reality.

In an asynchronous model of communication where message delivery or bounds on transit time is not guaranteed, it is still possible to solve most of the problems we consider

here. However, we stick to synchronous communication for simplicity, but also because problems can only be solved in a strictly weaker sense using asynchronous communication. As an example, assume that we want to tolerate that up to t of the n parties are corrupted. If an honest party waits for messages from more than $n-t$ parties, then it might potentially be waiting for a message from a corrupted party. This corrupted party might not have sent its message, and since no lower bound on message delivery is guaranteed, an unsent message cannot be distinguished from a slow message.[1] 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*, left alone having their inputs securely contribute to the result.

Exercise 1 *Consider a setting where two parties P_1 and P_2 want to find out whether they are both willing to cooperation in achieving some goal. This can be formalized as follows: Each P_i has an input $x_i \in \{0,1\}$, where $x_i = 1$ if and only if P_i wants to cooperate. They want to compute $y = f(x_1, x_2) = x_1 \wedge x_2$, where $x_1 \wedge x_2 = 1$ if $x_1 = 1$ and $x_2 = 1$, and $x_1 \wedge x_2 = 0$ otherwise. Note, in particular, that if P_1 does not want to cooperate and therefore inputs $x_1 = 0$, then $y = 0$ no matter the input of P_2. I.e., a party which does not want to cooperate does not learn whether the other party wanted to cooperate or not. This is sometimes called the* marriage problem*. Show that if the two parties already know how to securely solve Yao's millionaire's problem, then they can also securely solve the marriage problem. [Hint: From their inputs to the marriage problem they locally determine some inputs for the millionaire's problem and then solve that instance. From the result of the millionaire's problem they can determine the result of the marriage problem, and nothing else.]*

Exercise 2 *Consider a setting where three parties want to cooperate if they desire the same goal. They want to find out whether they desires the same goal or not, but in case they do not desire the same goal, they do not want to leak their preferred goals to the other parties. Let us assume there are g goals. Each P_i gives an input $x_i \in \{1, \ldots, g\}$ indexing which goal P_i wants to achieve. Then the parties want to learn $f(x_1, x_2, x_3) \in \{0,1\}$, where $f(x_1, x_2, x_3) = 1$ if and only if $x_1 = x_2 = x_3$.*

1. *Express $f(x_1, x_2, x_3)$ as a function of x_1, x_2 and x_3 using only the arithmetic operators $+$, $-$ and $\cdot$ and the special operator $=^?$, where $a =^? b$ is 1 if $a = b$ and 0 otherwise.*

2. *Replace the use of $=^?$ by a circuit using only the arithmetic operators. Assume that the computation is done in the finite field $\mathbb{F} = \mathbb{Z}_p$ for a prime $p > g$, and try to get a circuit of size at most $O(\log_2(p))$. [Hint: Fermat.]*

3. *Since $=^?$ is more expensive than the arithmetic operators, when it has to be implemented using these, it is desirable to use as few invocations of it as follows. Show how*

[1] In fact, it is easy to see that the ability to distinguish an unsent message from a sent-but-slow message is almost equivalent to knowing a bound on delivery time.

to do with just one invocation of $=^?$ and a constant number of arithmetic operators. You are allowed to assume that $\mathbb{F} = \mathbb{Z}_p$ for a prime p and that p is much larger than g (say, you decide the value of p after seeing g).

2 Defining Security, a First Look

In this section we give a first idea of how security of MPC is defined. We will later denote a separate section to fleshing out the details.

2.1 How to not do it

Defining security of MPC protocols is not easy, because the problem is so general. A good definition must automatically lead to a definition, for instance, of secure electronic voting because this is a special case of MPC. The classical approach to such definitions is to write down a list of requirements: the inputs must be kept secret (except for what can be learned from the output), the result must be correct, etc. However, apart from the fact that it may be hard enough technically to formalize such requirements, it can be very difficult to be sure that the list is complete. For instance, in a sealed-bid auction, we would clearly be unhappy about a solution that allowed a cheating bidder to bid in a way that relates in a particular way to an honest player's bid. We do, e.g., not want player P_1 to be able to behave such that his vote is always one euro higher than that of honest player P_2's vote. Yet a protocol with such a defect may well satisfy the demand that all inputs of honest players are kept private (except for what can be learned from the output), and that all submitted bids of the right form are indeed considered. Namely, it may be that a corrupt P_1 does not know how he bids, he just modifies P_2's "encrypted" bid in some clever way and submits it as his own.[2] So maybe we should demand that all players in a multiparty computation *know* which input values they contribute? Probably yes, but can we then be sure that there are no more requirements we should make in order to capture security properly?

2.2 The Ideal vs. Real World Approach

To get around this seemingly endless series of problems, we will take a completely different approach: in addition to the real world where the actual protocol and attacks on it take place, we will define an ideal world which is basically a specification of what we would like the protocol to do. The idea is then to say that a protocol is good if what it produces cannot be distinguished from what we could get in the ideal scenario.

To be a little more precise, we will in the ideal world assume that we have access to an incorruptible computer, a so-called ideal functionality $\mathcal{F}$. All players can privately send inputs to and receive outputs from $\mathcal{F}$. The ideal functionality $\mathcal{F}$ is programmed to execute a certain number of commands, and will, since it is incorruptible, always execute them correctly according its (public) specification, without leaking any information other than the outputs it is supposed to send to the players.

[2] Note that at least the seller should be unsatisfied with such a "feature". If there are only two bidders and P_1 knows that it values the sold good higher than P_2 and therefore surely is going to win, the goal of P_1 is to bid as low as possible while still winning. The above feature would allow P_1 to minimize the prize in this case. Without the feature P_1 would have to bid higher to be sure to win, which would give the seller a larger pay off.

As an example, the ideal functionality for secure function evaluation of f, let us call it $\mathcal{F}_{\text{SFE}}^{f}$, could be specified as follows: Wait for a message x_i from each P_i; Compute $(y_1, \ldots, y_n) = f(x_1, \ldots, x_n)$; Securely send each y_i to P_i. If each P_i is securely connected to $\mathcal{F}_{\text{SFE}}^{f}$ and $\mathcal{F}_{\text{SFE}}^{f}$ is truly incorruptible, then secure function evaluation is trivial. The parties just send their inputs to $\mathcal{F}_{\text{SFE}}^{f}$ and get back the results. Any other cryptographic task, such as commitment schemes, VSS or payments systems can as easily be phrased as ideal functionalities.

The goal of a protocol π_{SFE}^{f} for secure function evaluation of f is then to create, without help from trusted parties, and in presence of some adversary, a situation "equivalent" to the case where we have $\mathcal{F}_{\text{SFE}}^{f}$ available. If this is the case, we say that π_{SFE}^{f} *securely implements* $\mathcal{F}_{\text{SFE}}^{f}$.

By "equivalent" we mean that whatever information an adversary can collect in the real world it could also have collected in the ideal world, by corrupting the same parties. This shows that the protocol is no worse than the ideal setting.

As an example, let us consider an adversary which does a static, passive corruption of some parties indexed by $C \subset \{1, \ldots, n\}$. In the ideal world, where the parties send x_i to $\mathcal{F}_{\text{SFE}}^{f}$ and get back y_i, a corruption of P_i leaks only x_i and y_i. I.e., in the ideal world an adversary learns just

$$\text{IDEAL}_C(\mathcal{F}_{\text{SFE}}^{f}(x_1, \ldots, x_n)) = \{(i, x_i, y_i)\}_{i \in C} \ .$$

In the real world the adversary learns more, namely all the messages msg_i sent by each corrupted P_i plus the randomness r_i used by P_i in the computation. I.e., in the real world an adversary learns

$$\text{REAL}_C(\pi_{\text{SFE}}^{f}(x_1, \ldots, x_n)) = \{(i, x_i, y_i, r_i, \text{msg}_i)\}_{i \in C} \ .$$

We call the protocol secure against an adversary structure $\mathcal{A}$ if for all $C \in \mathcal{A}$ it holds that $\text{REAL}_C(\pi_{\text{SFE}}^{f}(x_1, \ldots, x_n))$ contains no more information than $\text{IDEAL}_C(\mathcal{F}_{\text{SFE}}^{f}(x_1, \ldots, x_n))$.

As with zero-knowledge we define this via simulation. In the case of zero-knowledge we requires that the entire transcript of the proof could be simulated. Here we require that the transcript $\text{REAL}_C(\pi_{\text{SFE}}^{f}(x_1, \ldots, x_n))$ can be simulated given $\text{IDEAL}_C(\mathcal{F}_{\text{SFE}}^{f}(x_1, \ldots, x_n))$. Technically, we require that there exists a probabilistic poly-time (PPT) simulator $\mathcal{S}$ which takes inputs of the form $\{(i, x_i, y_i)\}_{i \in C}$ and gives outputs of the form $\{(i, x_i, y_i, r_i, \text{msg}_i)\}_{i \in C} = \mathcal{S}(\{(i, x_i, y_i)\}_{i \in C})$, and we require that $\mathcal{S}(\text{IDEAL}_C(\mathcal{F}_{\text{SFE}}^{f}(x_1, \ldots, x_n)))$ and $\text{REAL}_C(\pi_{\text{SFE}}^{f}(x_1, \ldots, x_n))$ have the same distribution for all $C \in \mathcal{A}$. If this is the case we call π_{SFE}^{f} a perfectly secure implementation of $\mathcal{F}_{\text{SFE}}^{f}$. We also say that the protocol is secure in the sense of poly-time simulation.

Note that $\mathcal{S}$ essentially just constructively shows that $\text{REAL}_C(\pi_{\text{SFE}}^{f}(x_1, \ldots, x_n))$ contains no more information than $\text{IDEAL}_C(\mathcal{F}_{\text{SFE}}^{f}(x_1, \ldots, x_n))$ simply by showing how to compute $\text{REAL}_C(\pi_{\text{SFE}}^{f}(x_1, \ldots, x_n))$ from $\text{IDEAL}_C(\mathcal{F}_{\text{SFE}}^{f}(x_1, \ldots, x_n))$. In particular, any PPT ideal-world adversary which corrupts C and learns $\text{IDEAL}_C(\mathcal{F}_{\text{SFE}}^{f}(x_1, \ldots, x_n))$ could just run $\mathcal{S}$ after the execution of the protocol to sample a value $\mathcal{S}(\text{IDEAL}_C(\mathcal{F}_{\text{SFE}}^{f}(x_1, \ldots, x_n)))$ which has the exact same distribution as $\text{REAL}_C(\pi_{\text{SFE}}^{f}(x_1, \ldots, x_n))$, the values it would have learned by corrupting the parties C in the real-world protocol. So, for any PPT adversary there is

no advantage in attacking the real-world protocol over attacking the ideal world, and since the ideal world is as good as it gets, the protocol is as good as it gets!

If $\mathcal{S}(\text{IDEAL}_C(\mathcal{F}^f_{\text{SFE}}(x_1, \ldots, x_n)))$ and $\text{REAL}_C(\pi^f_{\text{SFE}}(x_1, \ldots, x_n))$ are only statistically close, then we call π^f_{SFE} a statistically secure implementation of $\mathcal{F}^f_{\text{SFE}}$. If $\mathcal{S}(\text{IDEAL}_C(\mathcal{F}^f_{\text{SFE}}(x_1, \ldots, x_n)))$ and $\text{REAL}_C(\pi^f_{\text{SFE}}(x_1, \ldots, x_n))$ are only computationally indistinguishable, then we call π^f_{SFE} a computationally secure implementation of $\mathcal{F}^f_{\text{SFE}}$.

3 Results on MPC

In this section we list some important known results on MPC.

3.1 Results for Threshold Adversaries

The classical results for the information-theoretic model due to Ben-Or, Goldwasser and Wigderson [5] and Chaum, Crépeau and Damgård [12] state that every function can be securely computed with perfect security in presence of an adaptive, passive (adaptive, active) adversary, if and only if the adversary corrupts less than $n/2$ ($n/3$) players. A lot of worked soon followed, improving on the efficiency of the first protocols, like Gennaro, Rabin and Rabin[26]. The currently fastest protocol is by Beerliova and Hirt[6].

When a broadcast channel is available, then every function can be securely computed with statistical security in presence of an adaptive, active adversary if and only if the adversary corrupts less than $n/2$ players. This was first shown by Rabin and Ben-Or[38]. Again the efficiency was soon improved, by e.g. Cramer, Damgård, Dziembowski, Hirt and Rabin[14]. The currently fastest protocol is by Damgård and Nielsen[23].

The most general results for the cryptographic model are by Goldreich, Micali and Wigderson [28] who showed that, assuming trapdoor one-way permutations exist, any function can be securely computed with computational security in presence of a static, active adversary corrupting less than $n/2$ players and by Canetti *et al.*[9] who showed that security against adaptive adversaries in the cryptographic model can also be obtained, although at the cost of a significant loss of efficiency. Under specific number theoretic assumptions, Damgård and Nielsen have shown that adaptive security can be obtained with a reasonable efficiency[22].

The following table summarizes which thresholds are obtainable for various qualities of security, where all results are for adaptive security.

	Passive	Active with broadcast	Active without broadcast
Perfect	$n/2$	$n/3$	$n/3$
Statistical	$n/2$	$n/2$	$n/3$
Computational	$n/2$	$n/2$	$n/2$

3.2 Results for General Adversaries

Hirt and Maurer [29] introduced the scenario where the adversary is restricted to corrupting any set in a general adversary structure.

In the field of secret sharing we have a well-known generalization from threshold schemes to secret sharing over general access structures. Hirt and Maurer's generalization does the same for multiparty computation. One may think of the sets in their adversary structure as corresponding in secret sharing terminology to those subsets that cannot reconstruct the secret.

Let $Q2$ (and $Q3$) be the conditions on a structure that no two (no three) of the sets in the structure cover the full player set. The result of [29] can be stated as follows: In the information-theoretic scenario, every function can be securely computed with perfect security in presence of an adaptive, passive (adaptive, active) $\mathcal{A}$-adversary if and only if

$\mathcal{A}$ is $Q2$ ($Q3$). This is for the case where no broadcast channel is available. The threshold results of [5], [12], [28] are special cases, where the adversary structure contains all sets of size less than $n/2$ or $n/3$.

This general model leads to strictly stronger results. Consider, for instance, the following infinite family of examples: Suppose our player set is divided into two groups 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 group of active cheaters 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 an adversary structure satisfying $Q3$, and so multiparty computations are possible in this scenario. Nevertheless, no threshold solution exists, since the largest coalitions of corrupt players have size more than $n/3$.[3] The intuitive reason why threshold protocols fail here is that they will by definition have to attempt protecting against *any* coalition of size $9m/10$ — an impossible task. On the other hand this is overkill because not every coalition of this size actually occurs, and therefore multiparty computation is still possible using more general tools.

The protocols of [29] rely on quite specialized techniques. Cramer, Damgård and Maurer [15] show that any linear secret sharing scheme can be used to build MPC protocols. A linear secret sharing scheme is one in which each share is a fixed linear function (over some finite field) of the secret and some random field elements chosen by the dealer. Since all the most efficient general techniques for secret sharing are linear, this gives the fastest known protocols for general adversary structures. They also show that the $Q2$ condition is necessary and sufficient for MPC in the cryptographic scenario.

3.3 Unfair MPC

The research on MPC has considered a large number of different models, to try to find either more secure protocols or more efficient protocols. We will mention just one of these directions here. In [18] Canetti, Lindell Ostrovsky and Sahai, building on previous work, show that it is possible to get some security even if up to $t = n-1$ parties can be corrupted. A number of such protocols are known, but they are, however, all unfair in the sense that a single corrupted party can force the protocol to fail in such a way that the corrupted party itself learns the output of the computation, whereas the honest parties learns no information from the computation at all. This is in contrast to the fair protocols we mentioned above which guarantees that all parties learn the result. The bounds $t < n/2$ and $t < n/3$ mentioned above are known to be optimal for fair protocols.

[3] It can be shown that no weighted threshold solution exists either for this scenario, i.e., a solution using threshold secret sharing, but where some players are given several shares.

4 A Passive Secure Protocol

In this section we will sketch how to obtain a perfectly secure function evaluation protocol in the i.t.-model when $t < n/2$. More precisely, we will look at how to implement $\mathcal{F}_{\text{SFE}}^{f}$ with perfect security, where we assume a threshold adversary that can corrupt at most $t < n/2$ players.

4.1 Arithmetic Circuits

In the following we fix some 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 $p > n$.

We will present a protocol which can securely evaluate an arithmetic function over $\mathbb{F}$. For notational convenience we construct a protocol for the case where each party has exactly one input and one output from $\mathbb{F}$. I.e., $f : \mathbb{F}^n \to \mathbb{F}^n, (x_1, \ldots, x_n) \to (y_1, \ldots, y_n)$. The mapping $(y_1, \ldots, y_n) = f(x_1, \ldots, x_n)$ is described using an arithmetic circuit. There are n input gates, each labeled by the party P_i which is going to supply the secret input value x_i for that gate. Then there are a number of internal addition and multiplication gates. Finally there is for each P_i exactly one output gate labeled by i. The value of this gate is going to be y_i.

Considering arithmetic circuits is without loss of generality: Any function that is feasible to compute at all can be specified as a poly-sizes 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 *Assume that you are given an ideal functionality $\mathcal{F}_{\text{SFE}}^{f}$ which allows to compute any function $f : \mathbb{F}^n \to \mathbb{F}^n$ given by an arithmetic circuit. Show how it can be used to securely implement an ideal functionality $\mathcal{F}_{\text{SFE}}^{g}$ for any function $f : \{0,1\}^n \to \{0,1\}^n$, 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 for passive security. If parties can deviate from the protocol there is the problem that when the Boolean circuit is coded as an arithmetic circuit, it is important that all parties input 0 or 1. There is, however, no guarantee that this happens, as a corrupted party can input any $x_i \in \mathbb{F}$ to $\mathcal{F}_{\text{SFE}}^{f}$, and this can be a real problem: Consider a case with three parties, each with an input $x_i \in \{0,1\}$. Assume that $y_1 = (1 - x_1)x_2 + x_1x_3$. If $x_1 = 0$, then $y_1 = x_2$, and if $x_1 = 1$, then $y_1 = x_3$. I.e., 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 $(x_1', \ldots, x_n')$.]*

4.2 Secret Sharing

Our main tool to build the protocol will be secret sharing, in particular Shamir's scheme, which is based on polynomials over $\mathbb{F}$. 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 well known facts about this method are that any set of t or fewer shares contain no information on s, whereas it can be reconstructed from any $t+1$ or more shares. Both of these facts are proved using Lagrange interpolation.

4.2.1 Lagrange Interpolation

If $h(\mathtt{X})$ is a polynomial 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 $h(\mathtt{X}) = \sum_{i \in C} h(i)\delta_i(\mathtt{X})$.

Another consequence of Lagrange interpolation is that there exist easily computable values $r = (r_1, ..., r_n)$, such that

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

for all polynomials $h(X)$ of degree at most $n-1$. Note that the same r works for all $h(\mathtt{X})$. Namely, $r_i = \delta_i(0)$. We call $(r_1, ..., r_n)$ a recombination vector.

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 $\mathbb{F}^t$.

Exercise 4 *A useful first step to build MPC protocols is to design a secret sharing scheme with the property that a secret can be shared among the players such that no corruptible set has any information, whereas any non-corruptible set can reconstruct the secret. Shamir's scheme shows how to do this for a threshold adversary structure, i.e., where the corruptible sets are those of size t or less. In this exercise we will build a scheme for the non-threshold example we saw earlier. Here we have $2m$ players divided in subsets X, Y with m players in each, and the corruptible sets are those with at most $9m/10$ players from only X or only Y, and sets of less than $m/5$ players with players from both X and Y (we assume m is divisible by 10, for simplicity). More formally, $\mathcal{A}$ consists of all C with $C \subset X$ and $|C| \leq 9m/10$ plus all C with $C \subset Y$ and $|C| \leq 9m/10$ plus all C with $|C| < m/5$.*

1. *Suppose we shared secrets using Shamir's scheme, with $t = 9m/10$, or with $t = m/5 - 1$. What would be wrong with these two solutions in the given context?*

2. *Design a scheme that does work in the given context. [Hint: in addition to the secret s, create a random element $u \in \mathbb{F}$, and come up with a way to share it such that only subsets with players from* both *X and Y can compute u. Also use Shamir's scheme with both $t = 9m/10$ and $t = m/5 - 1$.]*

4.3 The Protocol

We give a protocol for the i.t. scenario, where there are secure channels between all parties. We assume a threshold adversary that can passively corrupt up to t players, where $t < n/2$. 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).

The protocol starts by:

Input Sharing: Each player P_i holding input $x_i \in \mathbb{F}$ secret shares x_i using Shamir's secret sharing scheme: it chooses at random a polynomial $x_i(\mathtt{X})$ of degree $\leq t$ with $x_i(0) = x_i$ and sends a share to each player, i.e., it sends $x_i(j)$ to P_j, for $j = 1, \ldots, n$.

We then work our way gate by gate through the given arithmetic circuit over $\mathbb{F}$, maintaining the following:

Invariant: All input values and all outputs from gates processed so far are secret shared, i.e., each such value $a \in \mathbb{F}$ is shared into shares $a_1, ..., a_n$, where P_i holds a_i, and where there exists a polynomial $a(\mathtt{X})$ of degree at most t such that $a(0) = a$ and $a_i = a(i)$. From the start, no gates are processed, and only the inputs are shared.

To determine which gate to process next, we simply take an arbitrary gate for which both of its inputs have been shared already.

Once a gate producing one of the final output values y has been processed, y can be reconstructed in the obvious way:

Output Reconstruction: The output y is shared by a polynomial $y(\mathtt{X})$ of degree $\leq t$. I.e., $y(0) = y$ and P_i holds $y_i = y(i)$. Each P_i securely sends y_i to the party that is supposed to learn y. That party uses Lagrange interpolation to compute $y = y(0)$ from $y(1), \ldots, y(t+1)$, or any other $t+1$ points.

Note that P_i actually only needs $t+1$ shares for this, and therefore has almost twice the number of shares needed. This is of course not a problem.

It is then sufficient to show how addition and multiplication gates are handled. Assume the input values to a gate are a and b. Assume, by invariant, that a is shared using a polynomial $a(\mathtt{X})$ and that b is shared using a polynomial $b(\mathtt{X})$, both with degree at most t. I.e., $a(0) = a$ and the parties hold shares $a_1 = a(1), ..., a_n = a(n)$, and $b(0) = b$ and the parties hold shares $b_1 = b(1), ..., b_n = b(n)$.

Addition: For $i = 1, \ldots, n$, P_i computes $c_i = a_i + b_i$. The shares $c_1, \ldots, c_n$ determine $c = a + b$ as required by the invariant.

Multiplication: Multiplication proceeds as follows:

Local multiplication step: For $i = 1, \ldots, n$, P_i computes $d_i = a_i \cdot b_i$.

Resharing step: P_i secret shares d_i, resulting in shares $d_{i1}, ..., d_{in}$, and sends d_{ij} to player P_j.

Recombination step: For $j = 1, \ldots, n$, player P_j computes $c_j = \sum_{i=1}^{n} r_i d_{ij}$, where $(r_1, \ldots, r_n)$ is the recombination vector. The shares $c_1, \ldots, c_n$ determine $c = ab$ as required by the invariant.

Note that we can handle addition and multiplication by a constant c by using a default sharing of c generated from, say, the constant polynomial $f(x) = c$.

4.4 Analysis

We will now prove that the protocol is a perfectly secure implementation.

4.4.1 Correctness

As for addition, note that if we let $c(\mathtt{X}) = a(\mathtt{X}) + b(\mathtt{X})$, then $c(\mathtt{X})$ is again a polynomial of degree at most t. Furthermore, $c(0) = a(0) + b(0) = a + b$ and P_i holds the share $c_i = a_i + b_i = a(i) + b(i) = c(i)$. Therefore $a + b$ is correctly shared using the polynomial $c(\mathtt{X})$.

As for multiplication, note that if we let $d(\mathtt{X}) = a(\mathtt{X})b(\mathtt{X})$, then $d(\mathtt{X})$ is a polynomial of degree at most $2t$. Furthermore, $d(0) = a(0)b(0) = ab$, and after the local multiplication step P_i holds the share $d_i = a_i b_i = a(i)b(i) = d(i)$. Therefore $d = ab$ is shared using the

polynomial $d(\mathtt{X})$, but $d(\mathtt{X})$ does not necessarily have degree t as required by the invariant. Handling this issue is the job of the next two steps, and is called secure degree reduction. Note that since $d(\mathtt{X})$ has degree at most $2t$ and $2t \leq n-1$, it follows from (1) that

$$d(0) = \sum_{i=1}^{n} r_i d(i) \ ,$$

which is that same as

$$ab = \sum_{i=1}^{n} r_i d_i \ .$$

Therefore the shares d_i determine ab using a known linear combination with $r = (r_1, \ldots, r_n)$. The parties essentially just compute this linear combination securely by acting on secret sharings of the values d_i: Let $d_i(\mathtt{X})$ denote the polynomial of degree at most t used to secret share d_i, such that $d_i(0) = d_i$ and $d_i(j) = d_{ij}$ and let

$$c(\mathtt{X}) = \sum_{i=1}^{n} r_i d_i(\mathtt{X}) \ .$$

Then clearly $c(\mathtt{X})$ has degree at most t, and

$$c(0) = \sum_{i=1}^{n} r_i d_i(0) = \sum_{i=1}^{n} r_i d_i = ab \ ,$$

$$c(j) = \sum_{i=1}^{n} r_i d_i(j) = \sum_{i=1}^{n} r_i d_{ij} = c_j \ .$$

Therefore ab is correctly shared using $c(\mathtt{X})$.

4.4.2 Privacy

The privacy of the overall protocol follows from the observation that all values are shared using random polynomials of degree at most t. Since any t corrupted parties have at most t shares of all polynomials and t shares leak no information, it follows that any subset of t corrupted parties only learn their own inputs and their own outputs (as they receive all shares of these). Below we formalize this argument by giving a simulator, as required by the definition in Section 2.

For simplicity we start by proving a weaker form of security called input indistinguishable computation. Here we require from all pairs of global inputs $x^{(0)} = (x_1^{(0)}, \ldots, x_n^{(0)})$ and $x^{(1)} = (x_1^{(1)}, \ldots, x_n^{(1)})$, and all $C \subset \{1, \ldots, n\}$ with $|C| \leq t$, that if

$$\text{IDEAL}_C(\mathcal{F}_{\text{SFE}}^f(x_1^{(0)}, \ldots, x_n^{(0)})) \sim^{\text{p}} \text{IDEAL}_C(\mathcal{F}_{\text{SFE}}^f(x_1^{(1)}, \ldots, x_n^{(1)})) \ ,$$

then

$$\text{REAL}_C(\pi_{\text{SFE}}^f(x_1^{(0)}, \ldots, x_n^{(0)})) \sim^{\text{p}} \text{REAL}_C(\pi_{\text{SFE}}^f(x_1^{(1)}, \ldots, x_n^{(1)})) \ .$$

I.e., if two sets of global inputs cannot be distinguished by the corrupted parties in the ideal world, then neither can they be distinguished in the real world.

Recall that

$$\text{IDEAL}_C(\mathcal{F}_{\text{SFE}}^f(x_1, \ldots, x_n)) = \{(i, x_i, y_i)\}_{i \in C}$$

and

$$\text{REAL}_C(\pi_{\text{SFE}}^f(x_1, \ldots, x_n)) = \{(i, x_i, y_i, r_i, \text{msg}_i)\}_{i \in C} \ .$$

So, the definition requires that if

$$\{(i, x_i^{(0)}, y_i^{(0)})\}_{i \in C} = \{(i, x_i^{(1)}, y_i^{(1)})\}_{i \in C} \ ,$$

then

$$\{(i, x_i^{(0)}, y_i^{(0)}, r_i^{(0)}, \text{msg}_i^{(0)})\}_{i \in C} = \{(i, x_i^{(1)}, y_i^{(1)}, r_i^{(1)}, \text{msg}_i^{(1)})\}_{i \in C} \ .$$

I.e., if the inputs and outputs of the corrupted parties are the same, then the distribution of the messages that they send and receive are also the same.

For the parts of the protocol which does not open values, it is easy to check that this is the case:

Input Sharing: Each corrupt $P_i \in C$ creates a random sharing of $x_i^{(b)}$. Since $x_i^{(0)} = x_i^{(1)}$, this clearly leads to the same distribution on the shares in the two cases.

Each honest $P_j \notin C$ creates a random sharing of $x_j^{(b)}$. It might be the case that $x_j^{(0)} \neq x_j^{(1)}$, but the corrupted parties only see the shares $\{x_j(i)^{(b)}\}_{i \in C}$. Since $|C| \leq t$, 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 corrupted parties have the same distribution, as all the values they received so far had the same distribution. In particular, a_i and b_i have the same distributions in the two cases. Therefore all values generated and sent by the corrupted parties have the same distribution. The honest parties only send shares of random sharings, and the corrupted parties only see t shares of each sharing. These are just uniformly random values.

Output Reconstruction: Here all shares of $y^{(b)}(\mathtt{X})$ are securely sent to some P_j. If P_j is honest, the corrupted parties do not see anything. But, if P_j is corrupted, then P_j, and thus the corrupted parties, sees *all* shares of $y^{(b)}(\mathtt{X})$. We therefore need to show that $(y^{(0)}(1), \ldots, y^{(0)}(n))$ and $(y^{(1)}(1), \ldots, y^{(1)}(n))$ are identically distributed.

By the condition $\{(i, x_i^{(0)}, y_i^{(0)})\}_{i \in C} = \{(i, x_i^{(1)}, y_i^{(1)})\}_{i \in C}$, we know that $y^{(0)}$ and $y^{(1)}$ are identical, and as we argued above, the shares $\{y^{(0)}(i)\}_{i \in C}$ and $\{y^{(1)}(i)\}_{i \in C}$ have the same distribution. Since $y^{(0)}(0) = y^{(0)}$ and $y^{(1)}(0) = y^{(1)}$, we can conclude that the shares $\{y^{(0)}(i)\}_{i \in C \cup \{0\}}$ and $\{y^{(1)}(i)\}_{i \in C \cup \{0\}}$ have the same distribution. From this it follows, using Lagrange interpolation, that $\{y^{(0)}(i)\}_{i=0}^n$ and $\{y^{(1)}(i)\}_{i=0}^n$

have the same distribution: From the $t+1$ points $\{y^{(b)}(i)\}_{i\in C\cup\{0\}}$ one can compute $\{y^{(b)}(i)\}_{i=0}^{n}$ using the Lagrange formulas for computing the missing points from the $t+1$ given one. I.e., $\{y^{(b)}(i)\}_{i=0}^{n} = F(\{y^{(b)}(i)\}_{i\in C\cup\{0\}})$ for some function F. When $\{y^{(0)}(i)\}_{i\in C\cup\{0\}}$ and $\{y^{(1)}(i)\}_{i\in C\cup\{0\}}$ have the same distribution, then of course $F(\{y^{(0)}(i)\}_{i\in C\cup\{0\}})$ and $F(\{y^{(1)}(i)\}_{i\in C\cup\{0\}})$ have the same distribution — this holds for any function F. In particular, $\{y^{(0)}(i)\}_{i=0}^{n} = F(\{y^{(0)}(i)\}_{i\in C\cup\{0\}})$ and $\{y^{(1)}(i)\}_{i=0}^{n} = F(\{y^{(1)}(i)\}_{i\in C\cup\{0\}})$ have the same distribution.

4.4.3 From Indistinguishability to Simulation

Above we only proved that the protocol is input indistinguishable. This is known sometimes to be weaker than poly-time simulation security, as defined in Section 2, but actually implies simulation security if the function f is invertible in poly-time (when restricted to some subset of the outputs), or if we let the simulator have unbounded computing power.

To see this, recall that the simulator gets input

$$\text{IDEAL}_C(\mathcal{F}_{\text{SFE}}^f(x_1,\ldots,x_n)) = \{(i,x_i,y_i)\}_{i\in C} \ .$$

Then it picks a set of inputs $x' = (x'_1,\ldots,x'_n)$ with the property that

$$\text{IDEAL}_C(\mathcal{F}_{\text{SFE}}^f(x'_1,\ldots,x'_n)) = \{(i,x_i,y_i)\}_{i\in C} \ ,$$

i.e., $\{(i,x_i,y_i)\}_{i\in C} = \{(i,x'_i,y'_i)\}_{i\in C}$ when $(y'_1,\ldots,y'_n) = f(x'_1,\ldots,x'_n)$. Since the original inputs, $(x_1,\ldots,x_n)$, fulfill this equation, a solution exists. So, a computationally unbounded simulator can find a solution by exhaustive search. Then $\mathcal{S}$ runs the protocol on $(x'_1,\ldots,x'_n)$ and outputs the internal state of the parties P_i, $i\in C$. I.e., it outputs

$$\mathcal{S}(\text{IDEAL}_C(\mathcal{F}_{\text{SFE}}^f(x_1,\ldots,x_n))) = \text{REAL}_C(\pi_{\text{SFE}}^f(x'_1,\ldots,x'_n)) \ .$$

Since

$$\text{IDEAL}_C(\mathcal{F}_{\text{SFE}}^f(x'_1,\ldots,x'_n)) = \text{IDEAL}_C(\mathcal{F}_{\text{SFE}}^f(x_1,\ldots,x_n)) \ ,$$

it follows from input indistinguishability that

$$\text{REAL}_C(\pi_{\text{SFE}}^f(x'_1,\ldots,x'_n)) \sim^{\text{p}} \text{REAL}_C(\pi_{\text{SFE}}^f(x_1,\ldots,x_n)) \ ,$$

which implies that

$$\mathcal{S}(\text{IDEAL}_C(\mathcal{F}_{\text{SFE}}^f(x_1,\ldots,x_n))) \sim^{\text{p}} \text{REAL}_C(\pi_{\text{SFE}}^f(x_1,\ldots,x_n)) \ ,$$

as desired.

This shows that simulation security with an unbounded simulator is implied by input indistinguishability. Furthermore, if $\mathcal{S}$ can compute x' from $\{(i,x_i,y_i)\}_{i\in C}$ in poly-time, then the entire simulation is poly-time, and we have a proof of poly-time simulation security as defined in Section 2.

4.4.4 Poly-Time Simulation Security

The protocol is, however, secure in the sense of Section 2 for *all* functions f. The simulation argument is just a little more involved, as sketched now.

The simulator $\mathcal{S}$ gets input

$$\text{IDEAL}_C(\mathcal{F}_{\text{SFE}}^f(x_1,\ldots,x_n)) = \{(i,x_i,y_i)\}_{i\in C} \ .$$

It then lets $x'_i = x_i$ for $i \in C$ and lets $x'_i = 0$ for $i \notin C$. Then $\mathcal{S}$ runs the protocol on $(x'_1,\ldots,x'_n)$ and records the internal state of the parties P_i, $i \in C$. I.e., it records

$$\text{REAL}_C(\pi_{\text{SFE}}^f(x'_1,\ldots,x'_n)) \ .$$

Since some of the corrupted parties P_i might get different outputs y'_i on input $(x'_1,\ldots,x'_n)$ than their output y_i on input $(x_1,\ldots,x_n)$ we cannot appeal to input indistinguishability. Note, however, that up to the point where an output y' is reconstructed, the view of the corrupted parties is the same in $\text{REAL}_C(\pi_{\text{SFE}}^f(x'_1,\ldots,x'_n))$ and $\text{REAL}_C(\pi_{\text{SFE}}^f(x_1,\ldots,x_n))$, as they have the same inputs and only see t shares of the values shared by the honest parties. So, the only point where the difference is spotted is when an incorrect output y' is reconstructed towards a corrupted party. Before the simulator outputs $\text{REAL}_C(\pi_{\text{SFE}}^f(x'_1,\ldots,x'_n))$ it simply corrects for this difference: It knows the true value y that the output should have — from its input $\{(i,x_i,y_i)\}_{i\in C}$. If $y' \neq y$, then it patches the execution $\text{REAL}_C(\pi_{\text{SFE}}^f(x'_1,\ldots,x'_n))$ as follows: Let $\{y'_i\}_{i\in C}$ be the shares of the corrupted parties. Define $y'(0) := y$ and then compute a polynomial $y'(\mathtt{X})$ of degree $\leq t$ with $y'(0) = y$ and $y'(i) = y'_i$ for $i \in C$. Then change the shares of the honest parties to be $y'_j := y'(j)$ for $j \notin C$, and send these shares instead. Do this patching for all outputs going to corrupted parties, and then output the patched version of $\text{REAL}_C(\pi_{\text{SFE}}^f(x'_1,\ldots,x'_n))$. The patching simply changes the shares of the honest parties to be consistent with the true output value y, as opposed to y'.

Note that the patched version is a function $G(\text{REAL}_C(\pi_{\text{SFE}}^f(x'_1,\ldots,x'_n)),\{y_i\}_{i\in C})$ of the prefix of the execution $\text{REAL}_C(\pi_{\text{SFE}}^f(x'_1,\ldots,x'_n))$ which does not include the output reconstruction and therefore is identically distributed to the same prefix in $\text{REAL}_C(\pi_{\text{SFE}}^f(x_1,\ldots,x_n))$. In particular,

$$G(\text{REAL}_C(\pi_{\text{SFE}}^f(x'_1,\ldots,x'_n)),\{y_i\}_{i\in C}) \sim^{\text{p}} G(\text{REAL}_C(\pi_{\text{SFE}}^f(x_1,\ldots,x_n)),\{y_i\}_{i\in C}) \ .$$

By definition the output of $\mathcal{S}$ is

$$\mathcal{S}(\text{IDEAL}_C(\mathcal{F}_{\text{SFE}}^f(x_1,\ldots,x_n))) = G(\text{REAL}_C(\pi_{\text{SFE}}^f(x'_1,\ldots,x'_n)),\{y_i\}_{i\in C}) \ .$$

So,

$$\mathcal{S}(\text{IDEAL}_C(\mathcal{F}_{\text{SFE}}^f(x_1,\ldots,x_n))) \sim^{\text{p}} G(\text{REAL}_C(\pi_{\text{SFE}}^f(x_1,\ldots,x_n)),\{y_i\}_{i\in C}) \ .$$

Since the outputs computed in $\text{REAL}_C(\pi_{\text{SFE}}^f(x_1,\ldots,x_n))$ are actually $\{y_i\}_{i\in C}$, by the correctness of the protocol, patching with $\{y_i\}_{i\in C}$ has no effect. I.e.,

$$G(\text{REAL}_C(\pi_{\text{SFE}}^f(x_1,\ldots,x_n)),\{y_i\}_{i\in C}) = \text{REAL}_C(\pi_{\text{SFE}}^f(x_1,\ldots,x_n)) \ ,$$

which proves that $\mathcal{S}(\text{IDEAL}_C(\mathcal{F}_{\text{SFE}}^f(x_1,\ldots,x_n))) \sim^{\text{p}} \text{REAL}_C(\pi_{\text{SFE}}^f(x_1,\ldots,x_n))$, as desired.

4.5 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 exercise 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 P_1. Assume that $n = 3$ and $t = 1$. I.e., we have three parties P_1, P_2, P_3 and one can be corrupted. For sake of example, say it is 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	$\mathbf{P_1}$	P_2	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 6, 12 respectively 15. In the table all shares that P_1 would see in this case are put in bold.

We want that 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 sake of example. We let the party P_1 make the hypothesis that $a = 1$ and $b = 5$. Hopefully it cannot exclude this hypothesis. As a starter, P_1 imagines that the network is configured as follows, not knowing the shares of P_1 and P_2:

Variable	Value	$\mathbf{P_1}$	P_2	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 P_1 fills these values into the table it concludes that the network must be configuration as follows for its hypothesis to hold:

Variable	Value	$\mathbf{P_1}$	P_2	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 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 5 *In the above example, P_1 could also have made the hypothesis that $a = 0$ and $b = 6$. Show that P_1 cannot exclude this example, by filling in the below table and noting that it is consistent.*

Variable	*Value*	$\mathbf{P_1}$	P_2	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	$\mathbf{P_1}$	P_2	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 α_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 α_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 P_1 used the polynomial $d_1(\mathtt{X}) = 6 - 2\mathtt{X}$, P_2 used the polynomial $d_2(\mathtt{X}) = 4 + 2\mathtt{X}$ and 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.

Again, an example will reveal that any other hypothesis, like $a = 1$ and $b = 6$ would have given the exact same view to P_1. The reader is encourage to do that, by solving the following exercise.

Exercise 6 *Show that the values seen by 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*	$\mathbf{P_1}$	P_2	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.]*

4.6 Optimality of the Corruption Bound

What if $t \geq n/2$? We will argue that then there are functions that cannot be computed securely.

Towards a contradiction, suppose there is a protocol π, with *perfect privacy* and *perfect correctness* for two players P_1, P_2 to securely evaluate the logical AND of their respective private input bits b_1, b_2, i.e., $b_1 \wedge b_2$.

Assume that the players communicate using a perfect *error-free communication channel*. One of the players may be corrupted by an *infinitely powerful*, *passive* adversary.

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

1. Each player 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. P_1 sends the first message m_{11}, followed by P_2's message m_{21}.

   This continues until P_2 has sent sufficient information for P_1 to compute $r = b_1 \wedge b_2$. Finally, P_1 sends r (and some halting symbol) to P_2.

   The *transcript* of the conversation is

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

   For $i = 1, 2$, the *view* of P_i is

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

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 always the correct result is computed.

Perfect privacy means that given their respective views, each of the players learns nothing more about the other player's input b' than what can be inferred from the own input b and from the resulting function output $r = b \wedge b'$.

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 has no information 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 $\mathcal{T} = (m_{11}, m_{m2}, \ldots, m_{1t}, m_{mt})$ we say that it is consistent with input $b_1 = 1$ if there exists r_1 such that if P_1 is run with input $b_1 = 1$ and randomness r_1 and receiving the message m_{2r} in rounds $r = 1, \ldots, t$ would make it send exactly the messages m_{1r} in rounds $r = 1, \ldots, t$. Let $C_{b_1=1}$ denote the transcripts consistent with $b_1 = 1$.

It follows from the perfect security that

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

Assume namely that P_1 has input $b_1 = 0$ and P_2 has input $b_2 = 0$, and that P_2 sees a transcript $\mathcal{T}$ which is not from $C_{b_1=1}$. Then P_2 can conclude that $b_1 = 0$, contradicting the perfect security.

From the perfect correctness we can conclude that

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

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

From the two above observations we see that $\mathcal{T}(0,0) \cap \mathcal{T}(0,1) = \emptyset$. Note, however, that when P_1 has $b_1 = 0$, then it sees a transcript $\mathcal{T}$ from either $\mathcal{T}(0,0)$ or $\mathcal{T}(0,1)$. Since they are disjoint, P_1 can determine the input of P_2 simply by checking whether $\mathcal{T} \in \mathcal{T}(0,0)$ or $\mathcal{T} \in \mathcal{T}(0,1)$. This contradicts the perfect security.

The argument can be generalized to show that impossibility for statistical security and statistical correctness, and even weaker security notions allowing small constant insecurity and incorrectness.

Exercise 7 *Show that there is no perfectly secure and perfectly correct protocol for the OR function* $(b_1, b_2) \mapsto b_1 \vee b_2$.

Exercise 8 *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* P_1 *sends* b_1 *to* P_2 *and* P_2 *sends* b_2 *to* P_1. *Then they both output* $b_1 \oplus b_2$.

Exercise 9 *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* $(1,0,0,0)$, *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.*

4.6.1 Computational Security

The assumptions about the players' 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 decryption ciphertexts under ek. We assume that these two key generators give encryption keys with the same distribution

If $b_1 = 0$, then P_1 samples ek uniformly at random without learning the decryption key. If $b_1 = 1$, then P_1 samples ek in such a way that it learns the decryption key dk. It sends ek to P_2. Then P_2 sends $C = E_{pk}(b_2)$ to P_1. If $b_1 = 0$, then P_1 outputs 0 and sends $r = 0$ to P_2 which outputs r. If $b_1 = 1$, then P_1 decrypts C to learn b_2, outputs b_2 and sends $r = b_2$ to P_2 which outputs r.

Since the two ways to sample the public key gives the same distribution, the protocol is perfectly secure for P_1. The security of P_2 depends on the encryption hiding b' when $b = 0$ and is therefore computational. In particular, a computationally unbounded 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 made robust by letting the parties use generic zero-knowledge proofs to show that they followed the protocol, and in principle leads to secure two-party protocols for any function. For more information, see for instance [16].

Exercise 10

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 inputs 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.]*

5 Definition of Security

It is easy to see that the above protocol is not actively secure. If e.g. some of the corrupted parties send incorrect shares in the opening phase and the recipient uses these to reconstruct its output, the output could become incorrect. In Section 6 we will show how to make the protocol secure against such deviations, but before doing this we need a model of active security.

Until now we have only looked at passive security, which allowed us to use a relatively simple definition of security: The view of the corrupted parties in the protocol must be simulatable given the view of the same parties in the ideal protocol. We say that the real leakage can be simulated given the ideal leakage.

We now turn our attention to the case of active corruptions, where the corrupted parties might deviate from the protocol. We have to require that this does not render the protocol insecure. Specifically we want that whatever the corrupted parties can obtain by deviating in the real world (where they run the protocol), they could have obtained by deviating in the ideal world (where an ideal functionality does the computation). Thereby we require that the protocol is still as secure as conceivable. Since the deviations done by the corrupted parties are done to influence the execution of the protocol, possibly to make it leak more values, we call the possible deviations in the protocol the real influence and we call the possible deviations in the ideal world the ideal influence. The requirement is then that the real influence can be simulated given the ideal influence.

When we consider function evaluation of a function f, then the ideal world is that it is done by a fully trusted third party, which we called $\mathcal{F}_{\text{SFE}}^{f}$. Here the only influence of a corrupted party P_i is that it can replace its input x_i to $\mathcal{F}_{\text{SFE}}^{f}$ by some alternative input x_i'. After this the function is evaluated on the supplied inputs and the parties get back their own outputs. Since we consider a monolithic adversary which controls all corrupted parties, the ideal influence and leakage on an input $(x_1, \ldots, x_n)$ is the following:

1. First the adversary gets the inputs $\{x_i\}_{i \in C}$, where C is the set of corrupted parties.
2. Then the adversary specifies alternative inputs $\{x_i'\}_{i \in C}$ for the corrupted parties.
3. Then $(y_1, \ldots, y_n) = f(x_1', \ldots, x_n')$ is computed, where $x_i' = x_i$ for $i \notin C$ — i.e., all honest parties use their correct inputs.
4. Then the adversary is given the outputs $\{y_i\}_{i \in C}$ of the corrupted parties.

In the real world an adversary controlling the corrupted parties can of course also choose alternative inputs x_i' for the corrupted parties and then honestly run the protocol on these inputs. This is indistinguishable from the case where the parties were honest and just happened to have these inputs, and therefore not really to be considered as corrupted behavior. In the real world the corrupted parties, however, have many more ways to influence the computation, like changing the way the messages are computed or not sending messages at all. By requiring that all such deviations can be simulated in the ideal world (by choosing an alternative input) we essentially require that all possible deviations correspond to the corrupted parties choosing alternative inputs.

As a simple example we could build into a protocol that if some party P_i does not send any messages at all, then the other parties themselves run a copy of the program that P_i should have run when it has input $x_i = 0$. This allows them to complete the protocol even when P_i sends no messages, and ensures that the influence P_i has by not sending messages just corresponds to choosing an alternative input $x_i' = 0$.

When formally specifying what it means for the real leakage to be simulatable given the ideal leakage we had a fairly simple job, as both of them were values — bit strings. Now we have the problem that both the real influence and the ideal influence are not simple bit strings, but behavior. Therefore the simulator will not just translate one bit string into another bit string, but will translate behavior into behavior — for each possible way to deviate in the real world it must specify an equivalent way to deviate in the ideal world. It turns out that the right way to formalize this is to let the simulator be an poly-time interactive machine. On one side it sees inputs corresponding to corrupted behavior in the real world (like which corrupted parties send which messages) and on the other side it then gives outputs corresponding to corrupted behavior in the ideal world (like which corrupted parties use which alternative inputs). Its job is to demonstrate that the two world are equivalent under this mapping of real influence to ideal influence. We now proceed to formalize this.

The security model that we use is the UC model developed by Ran Canetti[10]. Here UC stands for universally composable, which denotes that if a protocol is UC secure according to the formal definition, then it is secure to use in any context (where it would have been secure to use the ideal functionality). We later return to what this exactly means.

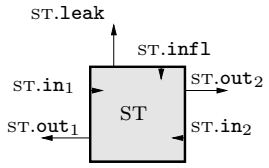
5.1 Specifying the Ideal World

We already introduces the idea of modeling the ideal world by an ideal functionality $\mathcal{F}$. Formally an ideal functionality will be an interactive machine.[4] The interface of $\mathcal{F}$ is as follows: $\mathcal{F}$ has an input port and an output port for every player. These $2n$ ports are called the protocol ports. The input port for P_i is named $\texttt{in}_i$ and the output port for P_i is named $\texttt{out}_i$. Party P_i uses $\texttt{in}_i$ to deliver its inputs to $\mathcal{F}$ and P_i gets back its outputs on $\texttt{out}_i$.

In addition to the protocol ports, $\mathcal{F}$ has two special ports, an input port called the influence port, and named `infl`, and an output port called the leakage port, and named `leak`. The leakage port is used to let $\mathcal{F}$ leak information about the inputs, as a way to specify that leaking this information is also allowed by a secure implementation of $\mathcal{F}$. The influence port is used to specify which influence the adversary is allowed to have on $\mathcal{F}$, i.e. the ideal influence.

As an example, we specify an ideal functionality $\mathcal{F}_{\text{ST}}$ for secure transfer in Fig. 1. Only the port structure for two parties is shown. The code is general enough to handle any number of parties. The influence port is used to determine in which order the messages are delivered, except that messages from one party to another are not reordered. Since more

[4] Formally, we model an interactive machine by an interactive Turing machine[10]. We could in principle have chosen any other notion of computation device which is well defined and which can receive and send messages and keep state, like a Java object.



- On receiving a value (mid, P_j, m) on $\text{ST.}\mathtt{in}_i$, output $(mid, P_i, P_j, |m|)$ on ST.`leak` and store (mid, P_i, P_j, m); Here mid is a message identifier used to distinguish stored messages.
- On a later input $(\mathtt{deliver}, P_i, P_j, mid)$ on ST.`infl`, where mid is the message identifier of the currently oldest message not deliver from P_i to P_j, delete (mid, P_i, P_j, m) and output (mid, P_i, m) on $\text{ST.}\mathtt{out}_j$.

Figure 1: The ideal functionality $\mathcal{F}_{\text{ST}}$ for secure transfer

than one message can be in transit[5] at a time, we use message identifiers to distinguish the messages. The leakage of $(mid, P_i, P_j, |m|)$ specifies that also an implementation of $\mathcal{F}_{\text{ST}}$ is allowed to leak the message identifier and the length of m. This is important, as no cryptosystem can fully hide the size of the information being encrypted. Without the leakage of $|m|$, there would not exist secure implementations of $\mathcal{F}_{\text{ST}}$.

Exercise 11 *Specify an ideal functionality $\mathcal{F}_{\text{MILL}}$ for the millionaire's problem, with the twist that if $x_1 = x_2$, then we allow either of the parties to be announced as winner, by a non-deterministic choice.*

The special ports are also used to model which information is allowed to leak when a party is corrupted and which control a hacker 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 `infl`, output $(\mathtt{state}, i, \sigma)$ on `leak`, where σ is called the ideal internal state of P_i and is defined to be all previous inputs on $\mathtt{in}_i$ along with all previous outputs on $\mathtt{out}_i$. This models that if a party is corrupted, the hacker only learns the inputs and outputs of that party.[6]

The above only models passive corruption. We model active corruption as follows. On input $(\mathtt{active\ corrupt}, i)$ on `infl`, the ideal functionality $\mathcal{F}$ starts ignoring all inputs on $\mathtt{in}_i$ and stops giving outputs on $\mathtt{out}_i$. Instead, whenever it gets an input $(\mathtt{alternative\ input}, i, x)$ on `infl`, it behaves exactly as if x had arrived on $\mathtt{in}_i$, and whenever $\mathcal{F}$ is about to output some value y on $\mathtt{out}_i$, it instead outputs $(\mathtt{output}, i, y)$ on `leak`. The way to think about

[5] The value (mid, P_i, P_j, m) is stored, but (mid, P_i, m) has not been output to P_j yet.

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

this is that after a party has become actively corrupted, all its inputs are chosen by the adversary via `infl` and all its outputs are leaked, as they are seen by the hacker. Since it is impossible to protect against the giving of alternative inputs (even an otherwise honest party could do this) and it is inevitable that outputs intended for some party is seen by a hacker controlling that party, the standard corruption behavior models an ideal situation where a hacker gets only these inevitable powers.

5.2 Specifying the Real World

We will later specify an ideal functionality for secure function evaluation. First we will, however, show how to specify a real-world protocol.

We continue with the secure transfer example. We want to implement $\mathcal{F}_{\text{ST}}$ using an authenticated channel and a public-key encryption scheme. In the real world there will be two parties, which we call P_1 and P_2. These will communicate using an authenticated channel. We describe how a secure transfer from P_1 to P_2 is implemented. All pairs of parties use the same implementation. We use the following protocol:

1. First P_2 samples a key pair (ek, dk) and sends the encryption key ek to P_1 over the authenticated channel.
2. Then P_1 encrypts the message, $C \leftarrow E_{ek}(m)$, and returns C over the authenticated channel.
3. Then P_2 outputs $m = D_{dk}(C)$.

We want to formally model this real world scenario.

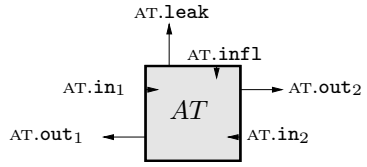
In the UC model communication devices are also ideal functionalities. I.e., an ideal functionality can play two roles, as a protocol specification and as a communication device.

To describe the real-world protocol for secure transfer we therefore need an ideal functionality $\mathcal{F}_{\text{AT}}$ for authenticated transfer, as that is the communication device used by the protocol. See Fig. 2.

The only difference from $\mathcal{F}_{\text{ST}}$ is that m is leaked, and not just $|m|$. This models that m is not necessarily kept secret by an authenticated channel.

In general a protocol will consist of a communication device and n parties. Each party is simply a machine connected to the protocol ports of the communication device on one hand, and which has open ports named as the ideal functionality that the protocol is trying to implement on the other hand. As a consequence the protocol has the same open protocol ports as the ideal functionality it is trying to implement. See the top row of Fig. 3 for an illustration.

Note that in addition to the ports mentioned above, party P_i has a leakage port $\text{AT.}\texttt{leak}_i$ and an influence port $\text{AT.}\texttt{infl}_i$. These are used to model corruption: If P_i receives a special symbol (`passive corrupt`) on $\text{AT.}\texttt{infl}_i$, then it returns its internal state σ on $\text{AT.}\texttt{leak}_i$. The internal state σ consists of all randomness used by the party so far along with all inputs sent and received on its ports. On input (`active corrupt`) on $\text{AT.}\texttt{infl}_i$, P_i starts



- On receiving a value (mid, P_j, m) on AT.$\texttt{in}_i$, output (mid, P_i, P_j, m) on AT.`leak` and store (mid, P_i, P_j, m); Here mid is a message identifier used to distinguish stored messages.
- On a later input $(\texttt{deliver}, P_i, P_j, mid)$ on AT.`infl`, where mid is the message identifier of the currently oldest message not deliver from P_i to P_j, delete (mid, P_i, P_j, m) and output (mid, P_i, m) on AT.$\texttt{out}_j$.

Figure 2: The ideal functionality $\mathcal{F}_{\text{AT}}$ for authenticated transfer

ignoring all inputs and never again produces an output — think of the machine running the code of P_i as having been shut down.[7]

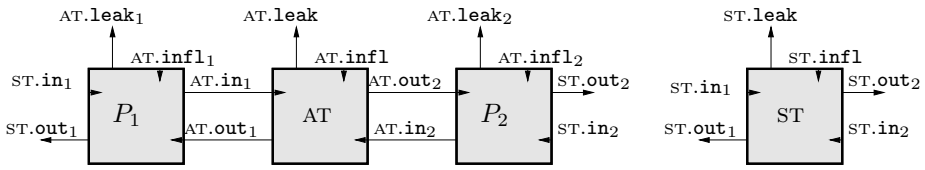
By passive corrupting a party in a protocol we mean that first (`passive corrupt`) is input on AT.$\texttt{infl}_i$ and then (`passive corrupt`$, i$) is input on AT.`infl` on the communication device. By active corrupting a party in a protocol we mean that first (`active corrupt`) is input on AT.$\texttt{infl}_i$ and then (`active corrupt`$, i$) is input on AT.`infl` on the communication device.

It might be puzzling that a corrupted party is just "shut down", as opposed to letting the adversary control what the party sends in the future. To see why we do not need an explicit modeling of active corruption of the party, recall that we assume that all ideal functionalities, and thereby all communication devices, have the standard corruption behavior. So, if P_1 is active corrupted in our example (by inputting (`active corrupt`) on AT.$\texttt{infl}_1$ and inputting (`active corrupt`$, 1$) on AT.`infl`), then arbitrary messages can now be sent on behalf of P_1 via AT.`infl`:

1. On input $(\texttt{alternative input}, (mid, P_2, m'))$ on AT.`infl`, $\mathcal{F}_{\text{AT}}$ will treat (mid, P_2, m') as an input on AT.$\texttt{in}_1$ and will in particular store (mid, P_1, P_2, m').
2. On a later input $(\texttt{deliver}, mid, P_1, P_2)$ on AT.`infl` it then outputs (mid, P_1, m') on AT.$\texttt{out}_2$.

So, indeed, after P_1 is active corrupted, arbitrary messages can be sent from P_1 to P_2. There is no need for explicitly controlling P_1. Therefore we halt the machine for simplicity.

[7] A note on naming. It is convenient that all port names have some prefix, such as AT or ST when more ports with the same name, like $\texttt{in}_i$, are present. We have chosen to prefix the special ports of a party by the name of the communication device that it is using and not the one it is implementing (AT in Fig. 3 instead of ST). We could have made any other choice. The current choice was made simply because it ensures that all special ports in the real world have the same prefix.



The protocol π_{ST} is given by the following two parties.

PARTY P_1:

1. On input (mid, P_2, m) on ST.$\texttt{in}_1$, output $(mid, P_2, \texttt{hello})$ on AT.$\texttt{in}_1$.
2. On a later input (mid, P_2, ek) on AT.$\texttt{out}_1$, sample a random encryption $C \leftarrow E_{ek}(m)$ and output (mid, P_2, C) on AT.$\texttt{in}_1$.

PARTY P_2:

1. On input $(mid, P_1, \texttt{hello})$ on AT.$\texttt{out}_2$, sample a random key pair (ek, dk) and output (mid, P_1, ek) on AT.$\texttt{in}_2$.
2. On a later input (mid, P_1, C) on AT.$\texttt{out}_2$, output $(mid, P_1, D_{dk}(C))$ on ST.$\texttt{out}_2$.

Similar code is included for the direction from P_2 to P_1, and besides this behavior, both parties of course have the standard corruption behavior.

Figure 3: The protocol π_{ST} for secure transfer and the ideal functionality $\mathcal{F}_{\text{ST}}$ for secure transfer

Pictorially one can think of the hacker as turning off and disconnecting the machine that has been taken over and then connecting his own machine to the communication interface previous used by the machine that was taken over.

5.3 Comparing the Real World to the Ideal World

We now want to argue that π_{ST} is equivalent to $\mathcal{F}_{\text{ST}}$. We would like to define this by requiring that a distinguisher which gets to play with either π_{ST} or $\mathcal{F}_{\text{ST}}$ by sending and receiving messages over the open ports cannot tell the difference. There is one problem with this approach: $\mathcal{F}_{\text{ST}}$ leaks $(mid, P_1, P_2, |m|)$ on ST.$\texttt{leak}$ and π_{ST} leaks $(mid, P_1, P_2, \texttt{hello})$, (mid, P_2, P_1, ek) and $(mid, P_1, P_2, E_{ek}(m))$. This makes it trivial to distinguish the systems, as even the structures of the leaked values are difference. Even the port names of the open special ports are different (cf. Fig. 3).

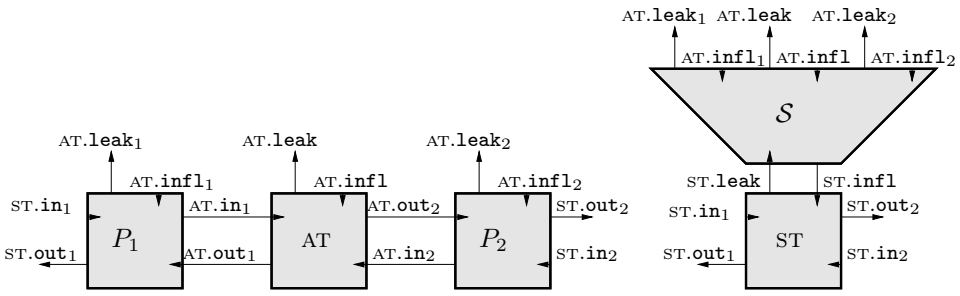


Figure 4: Port structure of the protocol π_{ST} for secure transfer and the ideal functionality $\mathcal{F}_{\text{ST}}$ for secure transfer, along with a simulator $\mathcal{S}$

5.3.1 Introducing the Simulator

Intuitively, however, leakage of $(mid, P_1, P_2, \texttt{hello})$, (mid, P_2, P_1, ek) and $(mid, P_1, P_2, E_{ek}(m))$ should be as benign as leakage of $(mid, P_1, P_2, |m|)$ if the cryptosystem is semantic secure, as this exactly means that an encryption leaks at most the length of a message. We are going to formalize this using a simulation argument, as we did with zero-knowledge. There we wanted to formalize that a protocol did not leak anything but the truth value of the claim, even though a lot of other communication went on. Here we want to formalize that π_{ST} does not leak anything but $(mid, P_1, P_2, |m|)$, though more actual communication is going on. As for zero-knowledge we argue this by giving a simulator, which simulates the actual communication given the allowed information (there the truth value, and here the value $(mid, P_1, P_2, |m|)$).

In the UC framework the simulator is a poly-time machine which connects to the leakage port and influence port of the ideal functionality and has leakage ports and influence ports named like the open leakage ports and influence ports of the protocol trying to implement the ideal functionality. See Fig. 4 for an example.

The only restriction on the simulator, besides being poly-time, is that it does not corrupt a party on the ideal functionality until that party has been corrupted in the simulation. If we consider our example in Fig. 4, this just means that $\mathcal{S}$ e.g. only inputs $(\texttt{active corrupt}, i)$ on ST.infl if it received the input (active corrupt) on $\text{AT.}\texttt{infl}_1$.

Now at least the systems $\pi_{\text{ST}} = P_1 \cup \mathcal{F}_{\text{AT}} \cup P_2$ and $\mathcal{F}_{\text{ST}} \cup \mathcal{S}$ have the same open ports.[8] The job of the simulator is then to make the systems look the same to any distinguisher.

At first, let us ignore the special ports of the parties and consider only AT.leak and AT.infl. Intuitively, the simulator sees the leakage in the ideal world (called the allowed leakage) and produce some outputs on its open leakage port (called the simulated leakage). The values output by the communication device in the protocol we call the real leakage. The job of the simulator is to translate the allowed leakage into simulated leakage looking

[8] We use the notation $\cup$ for joining two systems. Formally a system just consists of a set of machines, where we think of identically named ports being connected and the rest open. Therefore a joining of two system is formally just a set union.

like the real leakage.

In addition to this "translation" of ideal leakage to simulated real leakage, the simulator must take the inputs on its open leakage port, AT.`infl` in the example, and "translate" it into input values on the influence port of the ideal functionality, ST.`infl` in the example. We say that it translates real influence into allowed influence

Specifying a simulator $\mathcal{S}$ for our example will make the job of the simulator clearer. At first we ignore the leakage ports and influence ports of the parties. One simulator could then work as follows:

1. On input (mid, P_1, P_2, l) on ST.`leak` (from $\mathcal{F}_{\text{ST}}$, and with $l = |m|$) it outputs $(mid, P_1, P_2, \texttt{hello})$ on AT.`leak`.

2. On a later input $(\texttt{deliver}, mid, P_1, P_2)$ on AT.`infl` it samples a key pair (ek, dk) and outputs (mid, P_2, P_1, ek) on AT.`leak`.

3. On a later input $(\texttt{deliver}, mid, P_2, P_1)$ on AT.`infl` it samples a random encryption $C \leftarrow E_{ek}(m')$ and outputs (mid, P_1, P_2, C) on AT.`leak`. Here $m' = 0^l$ is an all-zero message of length l.

4. On a later input $(\texttt{deliver}, mid, P_1, P_2)$ on AT.`infl` it outputs $(\texttt{deliver}, mid, P_1, P_2)$ on ST.`infl`, which makes $\mathcal{F}_{\text{ST}}$ output (mid, P_1, m) on ST.$\texttt{out}_2$.

The reason why $\mathcal{S}$ uses m' and not the real m is that $\mathcal{F}_{\text{ST}}$ 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.

Consider now some distinguisher $\mathcal{Z}$ which gets to play with one of the two systems in Fig. 4. 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 P_1 to P_2. Then $\mathcal{Z}$ works as follows:

1. It picks some message m and inputs (mid, P_2, m) on ST.$\texttt{in}_1$.

2. Then it sees $(mid, P_1, P_2, \texttt{hello})$ on AT.`leak` and inputs $(\texttt{deliver}, mid, P_1, P_2)$ on AT.`infl`.

3. Then it sees some (mid, P_2, P_1, ek) on AT.`leak` and inputs $(\texttt{deliver}, mid, P_2, P_1)$ on AT.`infl`.

4. Then it sees some (mid, P_1, P_2, C'') on AT.`leak` and inputs $(\texttt{deliver}, mid, P_1, P_2)$ on AT.`infl`.

5. In response to this it sees some (mid, P_1, m'') output on ST.$\texttt{out}_2$.

It could of course refuse some of the deliveries, which would only have it see less messages and thus make the distinguishing of the systems harder.

Note that by design of $\mathcal{S}$, $\mathcal{Z}$ will see both system behave as specified above. The only difference between the two systems is that when playing with $\mathcal{F}_{\text{ST}} \cup \mathcal{S}$, then $C'' \leftarrow E_{ek}(0^{|m|})$

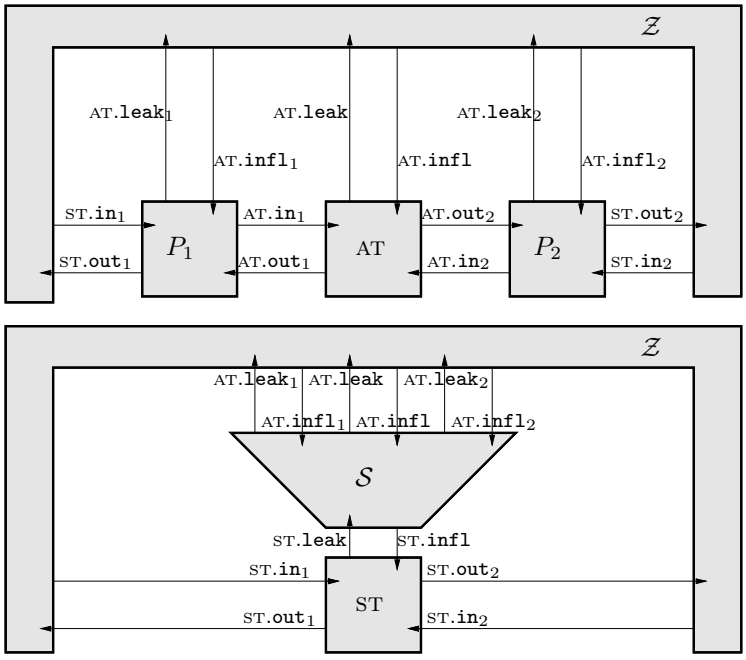


Figure 5: Example of two systems closed using the same distinguisher $\mathcal{Z}$

and $m'' = m$ and when playing with π_{ST}, then $C'' \leftarrow E_{ek}(m)$ and $m'' = D_{dk}(C'')$. If the encryption scheme has perfect correctness, then $m'' = D_{dk}(E_{ek}(m)) = m$, making the only difference that $C'' \leftarrow E_{ek}(m)$ or $C'' \leftarrow E_{ek}(0^{|m|})$. So, a distinguisher 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. Intuitively, the distinguisher should not be able to do this if we require the distinguisher to be poly-time and we use a semantic secure encryption scheme. The formal definition of what it means to distinguish two systems is carefully chosen to make that the case.

5.3.2 Comparing Systems with the Same Open Ports

Given two networks $\mathcal{N}_0$ and $\mathcal{N}_1$ with the same open ports (think of $\mathcal{N}_0 = \mathcal{F}_{\text{ST}} \cup \mathcal{S}$ and $\mathcal{N}_1 = \pi_{\text{ST}}$) a distinguisher is a poly-time machine $\mathcal{Z}$ with the "dual" ports of the system, so that $\mathcal{N}_0 \cup \mathcal{Z}$ and $\mathcal{N}_1 \cup \mathcal{Z}$ are closed systems. See Fig. 5 for an illustration. This notion of a distinguisher enforces the important restriction that a distinguisher only is allowed to play with the systems over the open ports — if it could inspect the structure of the systems, it would be trivial to distinguish. A distinguisher is often also call an environment, as it represents everything outside the systems to be compared. The environment is the ultimate monolithic adversary.

To measure how well the environment $\mathcal{Z}$ distinguishes, we pick a bit $b \in_R \{0,1\}$ uniformly at random and run the system $\mathcal{N}_b \cup \mathcal{Z}$. The execution runs as follows:

1. First all machines are given the security parameter k and fresh randomness.
2. Then $\mathcal{Z}$ iteratively inputs messages on some input ports of $\mathcal{N}_b$ and receives back messages on some output ports of the system.
3. In the end $\mathcal{Z}$ outputs a bit $c \in \{0,1\}$, which we think of as its guess at b.

Since a random guess is correct with probability $\frac{1}{2}$, we call

$$\mathrm{adv}_{\mathcal{Z}} \stackrel{\text{DEF}}{=} \left| \Pr[c = b] - \frac{1}{2} \right|$$

the advantage of $\mathcal{Z}$, and require that it is negligible. It is easy to see that this is the same as requiring that

$$\|\mathcal{N}_0 \cup \mathcal{Z}, \mathcal{N}_1 \cup \mathcal{Z}\| \stackrel{\text{DEF}}{=} \Pr[c = 1 | b = 1] - \Pr[c = 1 | b = 0]$$

is negligible. This gives us the following definition.

Definition 1 *We say that $\mathcal{N}_0$ and $\mathcal{N}_1$ are computationally indistinguishable if $\|\mathcal{N}_0 \cup \mathcal{Z}, \mathcal{N}_1 \cup \mathcal{Z}\|$ is negligible in the security parameter for all poly-time $\mathcal{Z}$. In that case we write $\mathcal{N}_0 \sim^c \mathcal{N}_1$.*

Essentially we just extended the notion of computational indistinguishability to interactive systems by using an interactive distinguisher.

The notion of indistinguishable systems have an important property, which we use later. Consider two systems $\mathcal{N}_0$ and $\mathcal{N}_1$ with the same open ports and let $\mathcal{N}$ be some third system which can connect to some of the ports of those systems. If $\mathcal{N}_0$ and $\mathcal{N}_1$ are indistinguishable and $\mathcal{N}$ is poly-time, then $\mathcal{N}_0 \cup \mathcal{N}$ and $\mathcal{N}_1 \cup \mathcal{N}$ are also indistinguishable. Let namely $\mathcal{Z}$ be any distinguisher for $(\mathcal{N}_0 \cup \mathcal{N})$ and $(\mathcal{N}_1 \cup \mathcal{N})$, and let $\mathcal{Z}' = (\mathcal{N} \cup \mathcal{Z})$. Then

$$\begin{aligned} \|(\mathcal{N}_0 \cup \mathcal{N}) \cup \mathcal{Z}, (\mathcal{N}_1 \cup \mathcal{N}) \cup \mathcal{Z}\| &= \|\mathcal{N}_0 \cup (\mathcal{N} \cup \mathcal{Z}), \mathcal{N}_1 \cup (\mathcal{N} \cup \mathcal{Z})\| \\ &= \|\mathcal{N}_0 \cup \mathcal{Z}', \mathcal{N}_1 \cup \mathcal{Z}'\| \ , \end{aligned}$$

and the right-hand-side is negligible by the assumption that $\mathcal{N}_0 \sim^c \mathcal{N}_1$ and the fact that $\mathcal{Z}'$ is poly-time.

In words, this property says that if we extend two indistinguishable systems by the same poly-time sub-system, the results are again indistinguishable. Not very surprising, but a nice property to have.

In addition, the notion is transitive. I.e., if $\mathcal{N}_0$ and $\mathcal{N}_1$ are indistinguishable, and $\mathcal{N}_1$ and $\mathcal{N}_2$ are indistinguishable, then $\mathcal{N}_0$ and $\mathcal{N}_2$ are indistinguishable.

5.4 The Security Definition

Putting the above together we get the following definition.

Definition 2 *Let $\mathcal{F}_{\text{NAME}}$ be an ideal functionality and let π_{NAME} be a protocol with the same open protocol ports. We say that π_{NAME} securely implements $\mathcal{F}_{\text{NAME}}$ if there exists a poly-time simulator $\mathcal{S}$ such that $\pi_{\text{NAME}} \sim^c \mathcal{F}_{\text{NAME}} \cup \mathcal{S}$. If $\pi_{\text{NAME}} \sim^c \mathcal{F}_{\text{NAME}} \cup \mathcal{S}$ only holds for distinguishers which make passive corruptions, we say that the protocol is a passive secure implementation. If $\pi_{\text{NAME}} \sim^c \mathcal{F}_{\text{NAME}} \cup \mathcal{S}$ only holds for distinguishers which make at most t corruptions, we say that the protocol is t-secure. If $\pi_{\text{NAME}} \sim^c \mathcal{F}_{\text{NAME}} \cup \mathcal{S}$ only holds for distinguishers which makes corruption from some adversary structure $\mathcal{A}$, we say that the protocol is $\mathcal{A}$-secure. If $\pi_{\text{NAME}} \sim^c \mathcal{F}_{\text{NAME}} \cup \mathcal{S}$ only holds for distinguishers which makes all corruptions before having any other interactions with the system, we say that the protocol is static secure.*

Writing the definition out a bit, it say that there should exist a poly-time simulator $\mathcal{S}$ such that all poly-time distinguishers $\mathcal{Z}$ output negligibly close guesses in $\pi_{\text{NAME}} \cup \mathcal{Z}$ and $(\mathcal{F}_{\text{NAME}} \cup \mathcal{S}) \cup \mathcal{Z}$. These two systems are illustrated for our secure transfer example in Fig. 5.

Note how we do not have an explicit notion of an adversary. This is because we consider the adversary to be part of the environment $\mathcal{Z}$. When distinguishing the systems, the environment $\mathcal{Z}$ can give inputs to parties and schedule the execution (which constitutes normal use of the system) but can also see all values leaked by the communication device used by the protocol, corrupt parties and send arbitrary messages on behalf of actively corrupted parties. Therefore the environment can launch an attack, where even the choice of inputs to the protocol are a coordinated part of the attack. This is the ultimate monolithic adversary. For the same reason we will sometimes call $\mathcal{Z}$ the adversary and e.g. say that $\mathcal{F}_{\text{AT}}$ leaks (mid, P_1, P_2, m) to the adversary.

5.5 Modular Composition

An important property of the UC framework is that when e.g. π_{ST} implements $\mathcal{F}_{\text{ST}}$, then $\mathcal{F}_{\text{ST}}$ can securely be replaced by π_{ST} in any protocol. Consider some third ideal functionality $\mathcal{F}_{\text{N}}$ doing some interesting task ideally secure. Assume that we can design a protocol π_{N}, which uses $\mathcal{F}_{\text{ST}}$ as communication device and which securely implements $\mathcal{F}_{\text{N}}$. Designing a secure implementation of $\mathcal{F}_{\text{N}}$ using secure transfer as communication device is potentially much easier than designing a protocol using only authenticated transfer. The structure of such a protocol is shown in the top row of Fig. 6, along with $\mathcal{F}_{\text{N}}$.

To get an implementation using only authenticated transfer, we can replace the use of $\mathcal{F}_{\text{ST}}$ by the use of the protocol π_{ST} — we write $\pi_{\text{N}}[\pi_{\text{ST}}/\mathcal{F}_{\text{ST}}]$. This is possible as $\mathcal{F}_{\text{ST}}$ and π_{ST} have the same protocol ports. The result is shown in the bottom row in Fig. 6. We consider $Q_1 \cup P_1$ as one party and consider $P_2 \cup Q_2$ as one party. As an example, $P_1' = Q_1 \cup P_1$ is just a machine with open ports AT.in_1 and AT.out_1 connecting it to the protocol's communication device $\mathcal{F}_{\text{AT}}$ and with open protocol ports N.in_1 and N.out_1 named as the ideal functionality $\mathcal{F}_{\text{N}}$ that the protocol is trying to implement. The ports inside $Q_1 \cup P_1$ are just particularities of how the machine is implemented.[9] In addition $Q_1 \cup P_1$ has some special ports which allow to corrupt it. We could insist on somehow

[9] If we use interactive Turing machines as the underlying machine model, then these internal ports will just correspond to work tapes of the machine.

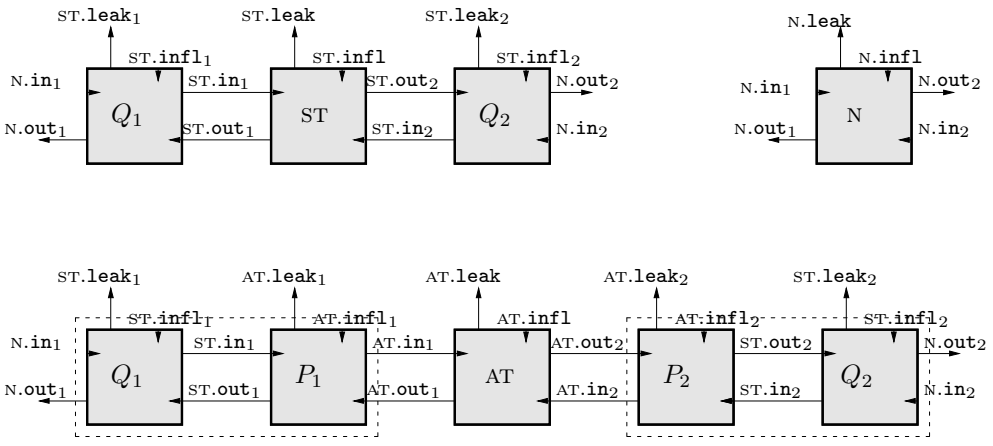


Figure 6: The protocol π_{N}, the ideal functionality $\mathcal{F}_{\mathrm{N}}$, and the protocol $\pi_{\mathrm{N}}[\pi_{\mathrm{N}}/\mathcal{F}_{\mathrm{ST}}]$

joining these ports, but allowing to corrupt the components of P_1' separately just gives more power to en entity corrupting P_1'. A passive corruption of P_1' is done by inputting (`passive corrupt`) on both ST.$\mathtt{infl}_1$ and AT.$\mathtt{infl}_1$, 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 P_1' is done by inputting (`active corrupt`) on both ST.$\mathtt{infl}_1$ and AT.$\mathtt{infl}_1$, and inputting (`active corrupt`, 1) on AT.`infl`. After this one can then send messages on behalf of P_1' using AT.`infl`.

The question is whether $\pi_{\mathrm{N}}[\pi_{\mathrm{ST}}/\mathcal{F}_{\mathrm{ST}}]$ securely implements $\mathcal{F}_{\mathrm{N}}$? For this, there must exist a poly-time simulator $\mathcal{U}$ which simulates all 10 special ports of the protocol $\pi_{\mathrm{N}}[\pi_{\mathrm{ST}}/\mathcal{F}_{\mathrm{ST}}]$ given just the two special ports of $\mathcal{F}_{\mathrm{N}}$. The answer is in the affirmative. This is a rather powerful result to have as it allows us to design protocols e.g. for a setting with ideally secure channels and then later plug in protocols using more realistic communication devices without losing security, and without having to reprove security each time. This is called secure modular composition.

Theorem 1 *Let π_{ST} be a secure implementation of some $\mathcal{F}_{\mathrm{ST}}$ and let π_{N} be a secure implementation of some $\mathcal{F}_{\mathrm{N}}$. If the parties of π_{N} are poly-time and π_{N} uses $\mathcal{F}_{\mathrm{ST}}$ as communication device, then $\pi_{\mathrm{N}}[\pi_{\mathrm{ST}}/\mathcal{F}_{\mathrm{ST}}]$ securely implements $\mathcal{F}_{\mathrm{N}}$. If both protocols are passive secure, then the composition is passive secure. If both protocols are t-secure, then the composition is t-secure. If both protocols are $\mathcal{A}$-secure, then the composition is $\mathcal{A}$-secure. If both protocols are static secure, then the composition is static secure.*

In the theorem we maintained the names from our secure transfer example, but the protocols and ideal functionalities can be arbitrary. We sketch the proof of the theorem.

Since π_{ST} securely implements $\mathcal{F}_{\mathrm{ST}}$, there exists a poly-time simulator $\mathcal{S}$ such that the two systems in the top row of Fig. 7 are indistinguishable. Here $\mathcal{F}_{\mathrm{AT}}$ denotes any ideal

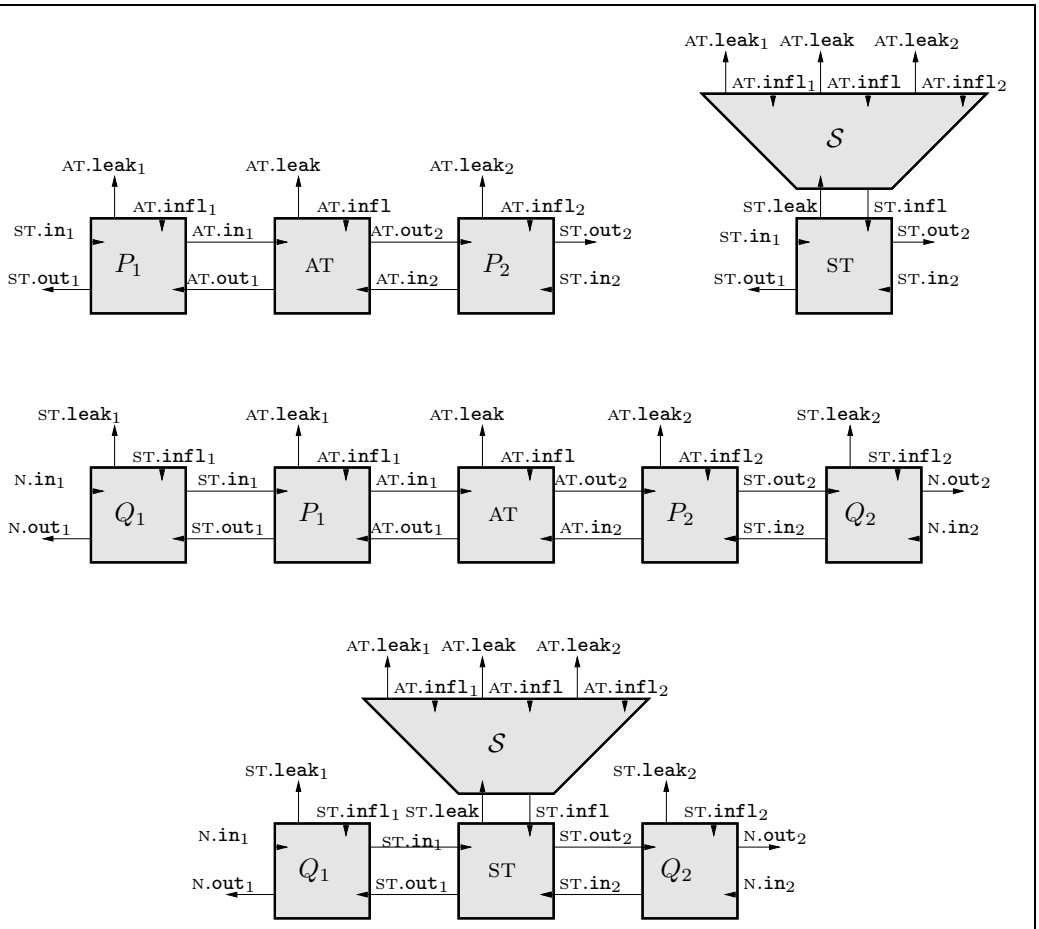


Figure 7: Extending π_{ST} and $\mathcal{F}_{\text{ST}} \cup \mathcal{S}$ with Q_1 and Q_2

functionality that π_{ST} might use. Again we use the naming from our example for clarity. In the second row, we extended π_{ST} by connecting Q_1 and Q_2 to the system. In the last row, we extended $\mathcal{F}_{\text{ST}} \cup \mathcal{S}$ by connecting Q_1 and Q_2 to the same ports as in the extension of π_{ST}. Since Q_1 and Q_2 are poly-time, the systems in the last two rows are indistinguishable by the fact that indistinguishability of systems is closed under poly-time extensions.

Since π_{N} securely implements $\mathcal{F}_{\text{N}}$, there exists a poly-time simulator $\mathcal{T}$ such that the two systems in the top row of Fig. 8 are indistinguishable. In the bottom row of Fig. 8 we extended both systems by connecting $\mathcal{S}$ to the same ports. Since $\mathcal{S}$ is poly-time, the systems in the bottom row are indistinguishable by the fact that indistinguishability of systems is closed under poly-time extensions.

Now notice that the system in the bottom row of Fig. 7 is identical to the system to the left in the bottom row of Fig. 8. By transitivity it follows that the system in the second row in Fig. 7 is indistinguishable from the system to the right in the bottom row of Fig. 8. The system in the second row in Fig. 7 is $\pi_{\text{N}}[\pi_{\text{ST}}/\mathcal{F}_{\text{ST}}]$. If we let $\mathcal{U} = \mathcal{S} \cup \mathcal{T}$, then the system to

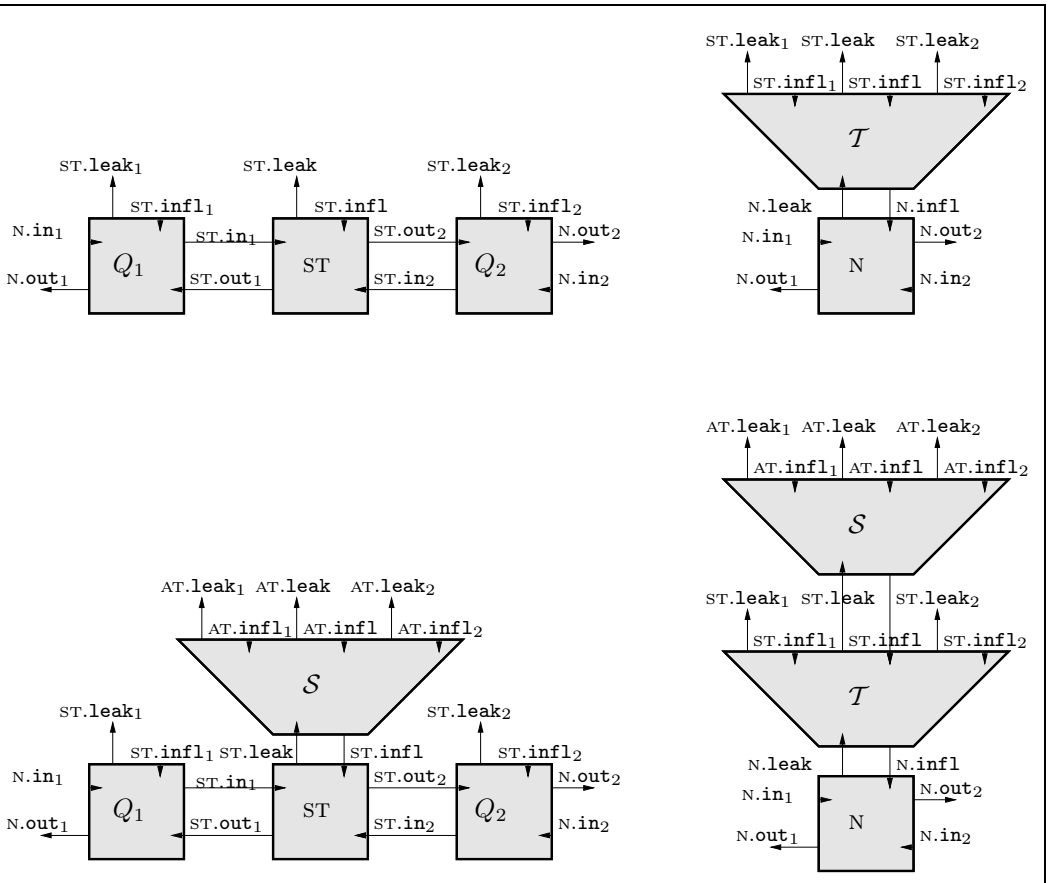


Figure 8: Extending π_{N} and $\mathcal{F}_{\mathrm{N}} \cup \mathcal{T}$ with $\mathcal{S}$

the right in the bottom row of Fig. 8 is $\mathcal{F}_{\mathrm{N}} \cup \mathcal{U}$. So, it follows that $\mathcal{F}_{\mathrm{N}} \cup \mathcal{U}$ and $\pi_{\mathrm{N}}[\pi_{\mathrm{ST}}/\mathcal{F}_{\mathrm{ST}}]$ are indistinguishable. But then $\mathcal{U}$ is exactly a simulator showing that $\pi_{\mathrm{N}}[\pi_{\mathrm{ST}}/\mathcal{F}_{\mathrm{ST}}]$ securely implements $\mathcal{F}_{\mathrm{N}}$. Not very surprisingly, the simulator for the composition of two protocols is a composition of the simulators for the individual protocols.

Exercise 12 *Above we only proved that if π_{ST} securely implements $\mathcal{F}_{\mathrm{ST}}$ and π_{N} securely implements $\mathcal{F}_{\mathrm{N}}$, then $\pi_{\mathrm{N}}[\pi_{\mathrm{ST}}/\mathcal{F}_{\mathrm{ST}}]$ securely implements $\mathcal{F}_{\mathrm{N}}$. Argue that the same holds for t-security. I.e., argue that if π_{ST} t-securely implements $\mathcal{F}_{\mathrm{ST}}$ and π_{N} t-securely implements $\mathcal{F}_{\mathrm{N}}$, then $\pi_{\mathrm{N}}[\pi_{\mathrm{ST}}/\mathcal{F}_{\mathrm{ST}}]$ t-securely implements $\mathcal{F}_{\mathrm{N}}$.*

5.6 Example Proofs

We conclude our secure transfer example by formally proving that the protocol is a secure implementation of the ideal functionality.

5.6.1 No Corruptions

We start by a proof for the case where there are no corruptions. I.e., for the case where the distinguisher does not use the leakage ports and influence ports of the parties.

Theorem 2 *If the encryption scheme has perfect correctness and is semantic secure, then* π_{ST} *securely implements* $\mathcal{F}_{\text{ST}}$ *as long as there are no corruption (formally it is* 0*-secure).*

Proof: The semantic security of an encryption scheme is defined via a game with an adversary A: First we sample a random key pair (ek, dk) and give ek to A. Then A outputs two messages m_0 and m_1 with $|m_0| = |m_1|$. Then we flip a random bit b and input $C \leftarrow E_{ek}(m_b)$ to A. Then A outputs a bit c, which we think of as its guess at b. We call $\text{adv}_A = |\Pr[c = b] - \frac{1}{2}|$ the advantage of A, and require that adv_A is negligible for all poly-time A.

Consider now some poly-time distinguisher $\mathcal{Z}$, which expects to play with either π_{ST} or $\mathcal{F}_{\text{ST}} \cup \mathcal{S}$. We have to argue that $\text{adv}_{\mathcal{Z}}$ is negligible. We do that by translating $\mathcal{Z}$ into an equivalent poly-time adversary against the encryption scheme. The adversary A runs $\mathcal{Z}$ as a sub-routine, as follows:

1. First receive ek.
2. Then run $\mathcal{Z}$ until it outputs some (mid, P_2, m) on ST.$\texttt{in}_1$.
3. Then output $m_0 = m$ and $m_1 = 0^{|m|}$ to the semantic security game and get back $C \leftarrow E_{ek}(m_b)$ for some unknown b.
4. Then input $(mid, P_1, P_2, \texttt{hello})$ on $\mathcal{Z}$.AT.$\texttt{leak}$ and run $\mathcal{Z}$ until it outputs $(\texttt{deliver}, mid, P_1, P_2)$ on AT.$\texttt{infl}$.
5. Then input (mid, P_2, P_1, ek) on $\mathcal{Z}$.AT.$\texttt{leak}$ and run $\mathcal{Z}$ until it outputs $(\texttt{deliver}, mid, P_2, P_1)$ on AT.$\texttt{infl}$.
6. Then input (mid, P_1, P_2, C) on $\mathcal{Z}$.AT.$\texttt{leak}$ and run $\mathcal{Z}$ until it outputs $(\texttt{deliver}, mid, P_1, P_2)$ on AT.$\texttt{infl}$.
7. Then input (mid, P_1, m) on $\mathcal{Z}$.ST.$\texttt{out}_2$ and run $\mathcal{Z}$ until it outputs some guess c.
8. Then output c.

It is clear that if $b = 1$, then $C = E_{ek}(0^{|m|})$ and the messages shown to $\mathcal{Z}$ have exactly the same distribution as in $\mathcal{F}_{\text{ST}} \cup \mathcal{S}$, and therefore c has the same distribution as in $(\mathcal{F}_{\text{ST}} \cup \mathcal{S}) \cup \mathcal{Z}$. If $b = 0$, then $C = E_{ek}(m)$ and the messages shown to $\mathcal{Z}$ have exactly the same distribution as in π_{ST}, and therefore c has the same distribution as in $\pi_{\text{ST}} \cup \mathcal{Z}$. This implies that $\text{adv}_A = \text{adv}_{\mathcal{Z}}$. Since adv_A is negligible, we conclude that $\text{adv}_{\mathcal{Z}}$ is negligible, as desired. QED

The important point to remember from this proof is how the simulator takes the allowed leakage and uses it to produce the more elaborate actual leakage, and how actual influence (deciding the delivery time of the three messages of the protocol) is mapped into allowed

influence (deciding the delivery time of the one message in the ideal setting). All our proofs will have this structure.

In the above proof, the simulation produced by $\mathcal{S}$ is only computationally indistinguishable. A computationally unbounded $\mathcal{Z}$ could easily distinguish $E_{ek}(m)$ and $E_{ek}(0^{|m|})$. If the simulated values have the exact same distribution as the real values, we say that the implementation is perfectly secure. This is the same as the advantage of any computationally unbounded $\mathcal{Z}$ being 0. If the advantage of any computationally unbounded $\mathcal{Z}$ is negligible, then we say that the implementation is statistically secure.

5.6.2 One Active Corruption.

We then extend the analysis to consider one static corruption. I.e., we now allow the distinguisher to corrupt one party, and it must do so before any other interactions.

Theorem 3 *If the encryption scheme has perfect correctness and is semantic secure, then π_{ST} securely implements $\mathcal{F}_{\text{ST}}$ as long as there is at most one static corruption (formally it is static, 1-secure).*

Proof: If there are no corruptions, the simulator behaves as in the proof of 0-security. If there is one corruption, the simulator behaves as described below.

We first consider the case where the sender is active corrupted. I.e., the environment inputs (`active corrupt`) on AT.$\texttt{infl}_1$ and (`active corrupt`, 1) on AT.`infl` before the execution (cf. Fig. 5). The simulator starts by also corrupting P_1. I.e., it outputs (`active corrupt`, 1) on ST.`infl`. Now $\mathcal{S}$ provides inputs to $\mathcal{F}_{\text{ST}}$ on behalf of P_1.

In the real world, $P_1 \cup \mathcal{F}_{\text{AT}} \cup P_2$, $\mathcal{Z}$ can send arbitrary messages to P_2 via AT.`infl`. If $\mathcal{Z}$ sends a message of the form (mid, P_1, C) to P_2 [10] after P_2 sent ek, then P_2 outputs $(mid, P_1, D_{dk}(C))$ on ST.$\texttt{out}_2$. This is easily simulated. The simulator $\mathcal{S}$ continues as follows:

1. If $\mathcal{Z}$ inputs (`alternative input`, 1, $(mid, P_2,$ `hello`)) and then (`deliver`, mid, P_1, P_2) on AT.`infl`, then sample (ek, dk) and output (mid, P_2, P_1, ek) on AT.`leak`.

2. If $\mathcal{Z}$ later inputs (`alternative input`, 1, (mid, P_2, C)) and then (`deliver`, mid, P_1, P_2) on AT.`infl`, then let $m' = D_{dk}(C)$ and output (`alternative input`, 1, (mid, P_2, m')) on ST.`infl`, and then output (`deliver`, mid, P_1, P_2) on ST.`infl`.

This makes $\mathcal{F}_{\text{ST}}$ output $(mid, P_1, m'' = D_{dk}(C))$ on ST.$\texttt{out}_2$. Therefore $\mathcal{Z}$ sees the two systems respond in exactly the same way. Note how the power of P_1 to send an arbitrary C was mapped into the ideal power of sending an arbitrary message m', by using $m' = D_{dk}(C)$.

We then consider the case where the receiver P_2 is active corrupted. I.e., the environment inputs (`active corrupt`) on AT.$\texttt{infl}_2$ and (`active corrupt`, 2) on AT.`infl` before the execution. The simulator starts by outputting (`active corrupt`, 2) on ST.`infl`. Now $\mathcal{F}_{\text{ST}}$ no longer outputs on ST.$\texttt{out}_2$, but sends the output to $\mathcal{S}$ on ST.`leak` instead. The simulator proceeds as follows.

[10] Formally $\mathcal{Z}$ inputs (`alternative input`, 1, (mid, P_2, C)) and then (`deliver`, mid, P_1, P_2) on AT.`infl`

1. On input (mid, P_1, P_2, l) on ST.`leak`, the simulator knows that some (mid, P_2, m) was input to $\mathcal{F}_{\text{ST}}$ on ST.$\texttt{in}_1$. The simulator inputs $(\texttt{deliver}, mid, P_1, P_2)$. Since P_2 is corrupted this makes $\mathcal{F}_{\text{ST}}$ output $(\texttt{output}, 2, (mid, P_1, m))$ to $\mathcal{S}$. The simulator stores m.

2. The simulator then outputs $(mid, P_1, P_2, \texttt{hello})$ on AT.`leak`.

3. On a later input $(\texttt{deliver}, mid, P_1, P_2)$ on AT.`infl`, output $(\texttt{output}, 2, (mid, P_1, \texttt{hello}))$ on AT.`leak`. *This is what would have happened in* $P_1 \cup \mathcal{F}_{\text{AT}} \cup P_2$.

4. If $\mathcal{Z}$ inputs $(\texttt{alternative input}, 2, (mid, P_1, ek))$ and then $(\texttt{deliver}, mid, P_2, P_1)$ on AT.`infl`, then compute $C \leftarrow E_{ek}(m)$ and output (mid, P_1, P_2, C) on AT.`leak`.

5. On a later input $(\texttt{deliver}, mid, P_1, P_2)$ on AT.`infl`, output $(\texttt{output}, 2, (mid, P_1, C))$ on AT.`leak`. *Again, this is what would have happened in* $P_1 \cup \mathcal{F}_{\text{AT}} \cup P_2$.

It can again be seen that the values that $\mathcal{Z}$ sees have the exact same distribution in the protocol $P_1 \cup \mathcal{F}_{\text{AT}} \cup P_2$ and in the simulation $\mathcal{F}_{\text{ST}} \cup \mathcal{S}$. QED

Exercise 13 *Verify that the simulation of the case of a corrupted receiver is perfect, by tracking the message that* $\mathcal{Z}$ *would see in* $P_1 \cup \mathcal{F}_{\text{AT}} \cup P_2$ *if it behaves as in the interaction with the simulator above.*

5.7 Modeling Synchronous n-Party Protocols

In the following all our protocols will consist of n-parties running a synchronous protocol. It is therefore convenient to once and for all fix some conventions for how we model, and talk about, that setting. It is easier to start by describing how we model a synchronous ideal functionality.

For our purposes here, a synchronous ideal functionality will be one which proceeds through a number of rounds, which all proceed as follows: First the ideal functionality waits for an input x_i from all parties (or at least all honest parties). Then it computes outputs y_i for all parties and returns y_i to each party P_i. Between getting the inputs x_i and giving the outputs y_i, the ideal functionality might output `running` to the parties a number of times. This models that it takes several rounds for the ideal functionality to compute the output (which will be the case when it is replaced by a protocol). For simplicity we let the adversary decide how many rounds it takes before the outputs are delivered — this is the worst case. We do, however, require that all parties get outputs in the same round. I.e., they output `running` the same number of times before they give the real output y_i. We also let the adversary decide in which order the outputs are delivered — again this is the worst case. The details are given in Fig. 9.

In the rest of the note we assume that all ideal functionalities have this generic synchronous behavior. In particular, when we specify ideal functionalities below, all we need to specify is the behavior of the ideal functionality in **compute outputs**, i.e., how the outputs $(y_1, \ldots, y_n)$ are computed from the input $(x_1, \ldots, x_n)$.[11] Since **compute outputs** is executed only when all honest parties gave an input, we can assume that all honest

[11] We sometimes also specify some additional values to be leaked in `get input`.

A generic synchronous ideal functionalities, $\mathcal{F}_{\text{SYNC}}$, proceed as:

init: Initially, let $\text{output}_i = \texttt{true}$ and $\text{running}_i = \texttt{false}$ for $i = 1, \ldots, n$.[a]

get inputs: On an input x_i on $\text{SYNC}.\texttt{in}_i$ while $\text{output}_i = \texttt{true}$ and $\text{running}_i = \texttt{false}$, let $\text{running}_i = \texttt{true}$ and store (i, x_i), and output $(\texttt{running}, i)$ on $\text{SYNC}.\texttt{leak}$ (possibly along with other messages to be leaked).

running: On an input $(\texttt{running}, i)$ on $\text{SYNC}.\texttt{infl}$ while $\text{running}_i = \texttt{true}$, output $\texttt{running}$ on $\text{SYNC}.\texttt{out}_i$.

compute outputs: On input $(\texttt{compute outputs})$ on $\text{SYNC}.\texttt{infl}$ while $\text{output}_i = \texttt{true}$ and $\text{running}_i = \texttt{true}$ for all honest P_i, compute an output y_i for each P_i, from the stored inputs (j, x_j).[b] Let $\text{output}_i = \texttt{false}$ for all P_i.

deliver outputs: On an input $(\texttt{deliver}, i)$ on $\text{SYNC}.\texttt{infl}$ while $\text{output}_i = \texttt{false}$, output y_i on $\text{SYNC}.\texttt{out}_i$, and let $\text{running}_i := \texttt{false}$ and $\text{output}_i := \texttt{true}$. It is enforced that all honest parties output $\texttt{running}$ the same number of times between getting the input x_i and giving the output y_i.[c]

Note that while $\text{running}_i = \texttt{true}$, all inputs on $\text{SYNC}.\texttt{in}_i$ are ignored by the ideal functionality.

[a] The Boolean output_i tells whether P_i delivered the output from the last round already. The Boolean running_i tells whether P_i is computing the next output.

[b] If $\text{running}_i = \texttt{false}$ for some P_i, then simply ignore the input $(\texttt{compute outputs})$.

[c] This is done as follows: If an input $(\texttt{running}, i)$ is given on $\text{SYNC}.\texttt{infl}$, which would violate the restriction, because P_i would end up outputting $\texttt{running}$ more times than an honest party P_j which already provided its output y_j, then the input is simply ignored. Furthermore, if an input $(\texttt{deliver}, i)$ is given, which would violate the restriction, because some honest party P_j already output $\texttt{running}$ more than P_i, then the input is simply ignored.

Figure 9: A generic synchronous ideal functionality

parties give inputs in all rounds when specifying the input-output behavior of the ideal functionality. We will never mention the rounds where just $\texttt{running}$ is output: we think of the whole process from the x_i are input till the y_i are output as one round on the ideal functionality.

A synchronous protocol is a protocol which uses a synchronous ideal functionality as communication device, and where each party proceeds as in Fig. 5.7. We require from all protocols that all parties provide an output y_i on $\text{SYNC}.\texttt{out}_i$ after receiving the same number of outputs Y_i from the ideal functionality. That way the parties will stay synchronized.

A party P_i for a generic synchronous protocol π_{SYNC} proceeds as follows:

init: Initially, let $\text{running}_i = \texttt{false}$.

get input: On an input x_i on SYNC.$\texttt{in}_i$, let $\text{running}_i = \texttt{true}$, and go to **running**.

running:

1. Input some message X_i to $\mathcal{F}$.
2. Wait for a message Y_i from $\mathcal{F}$. If $Y_i \neq \texttt{running}$, then go to Step 3. Otherwise, output `running` on SYNC.$\texttt{out}_i$ and go to Step 2 [*sic*].
3. When $Y_i \neq \texttt{running}$, do one of the following:
   - Output `running` on SYNC.$\texttt{out}_i$ and to Step 1.
   - Output some output y_i on SYNC.$\texttt{out}_i$, let running$:=\texttt{false}$, and go to **get input**.

When running $=$ `true`, party P_i ignores all inputs on SYNC.$\texttt{in}_i$.

Figure 10: A generic synchronous party.

The ideal functionality $\mathcal{F}_{\text{SB}}$ proceeds as follows in each round:

get input: Parse the input x_i from P_i as $x_i = (m_{i,0}, m_{i,1}, \ldots, m_{i,n})$. Output the value $(i, m_{i,0}, \{m_{i,j}\}_{j \in C}, \{|m_{i,j}|\}_{j \in H})$ on SB.`leak`, where C denotes the set of corrupted parties and $H = \{1, \ldots, n\} \setminus C$ denotes the honest parties.[a] For the corrupted parties P_i which did not give an input, let $x_i = (\perp, \perp, \ldots, \perp)$.

compute outputs: The output to P_i is $y_i = (m_{1,i}, \ldots, m_{n,i}, m)$, where $m = (m_{1,0}, \ldots, m_{n,0})$.

[a] Leaking these values already now models rushing communication.

Figure 11: The ideal functionality for Secure transfer and consensus Broadcast. In the input $x_i = (m_{i,0}, m_{i,1}, \ldots, m_{i,n})$, $m_{i,0}$ is the message P_i wants to send to all parties and $m_{i,j}$ is the message P_i wants to send securely to P_j.

6 An Active Secure Protocol

In this section, we show how to modify the protocol from Section 4 to make it secure also against active cheating. We will postulate in the following that we have a certain ideal functionality $\mathcal{F}_{\text{COM}}$ available. This functionality can then be implemented both in the i.t. and the cryptographic scenario. We consider such implementations later.

We note already now, however, that in the cryptographic scenario, $\mathcal{F}_{\text{COM}}$ can be implemented if $t < n/2$ (or in general, the adversary is $Q2$) and we make an appropriate computational assumption. In the i.t. scenario we need to require $t < n/3$ in case of protocols with zero error and no broadcast given. If we assume a broadcast channel and allow a non-zero error, then $t < n/2$ will be sufficient. All these bounds are tight.

Before we start, a word on broadcast: with passive corruption, broadcast is by definition not a problem, we simply ask a player to send the same message to everyone. But with active adversaries where no broadcast is given for free, a corrupt player may say different things to different players, and so broadcast is not immediate. Fortunately, in this case, we will always have that $t < n/3$ for the i.t. scenario and $t < n/2$ for the cryptographic scenario, as mentioned. And in these cases there are in fact protocols for solving the consensus broadcast problem efficiently. So we can assume that broadcast is given as an ideal functionality. In the following, when we say that a player broadcasts a message, this means that we call this functionality.

6.1 Some Ideal Functionalities

In the following we will design protocols which use secure channels and broadcast. Formally, we use a synchronous ideal functionality $\mathcal{F}_{\text{SB}}$ to model synchronous communication with secure transfer and consensus broadcast, and the protocols will then use this ideal functionality as communication device. The details are given in Fig. 11.

The ideal functionality $\mathcal{F}_{\text{SFE}}^{f}$ proceeds as follows in each round:

get inputs: For the corrupted P_i which did not give an input, or which gave an input which is not from $\mathbb{F}$, let $x_i = 0$.

compute outputs: The outputs are computed as $(y_1, \ldots, y_n) = f(x_1, \ldots, x_n)$.

Figure 12: The ideal functionality for secure function evaluation

We will use $\mathcal{F}_{\text{SB}}$ as communication device to implement secure function evaluation. The ideal functionality for secure function evaluation (SFE) of a function $f : \mathbb{F}^n \to \mathbb{F}^n, (x_1, \ldots, x_n) \mapsto (y_1, \ldots, y_n)$ proceeds as detailed in Fig. 12.

We assume that all honest parties always input a value x_i from $\mathbb{F}$. Formally we only prove security for environments that behave like this. This will guarantee that it is secure to use our implementation of $\mathcal{F}_{\text{SFE}}^{f}$ in any protocol where the honest parties always input values from $\mathbb{F}$.[12]

6.2 Model for Homomorphic Commitments and Auxiliary Protocols

We will assume that each player P_i can commit to a value $a \in \mathbb{F}$. This will later be implemented by distributing and/or broadcasting some information to other players. We model it here by assuming that we have an ideal functionality $\mathcal{F}_{\text{COM}}$. To commit, one simply sends a to $\mathcal{F}_{\text{COM}}$, who will then keep it until P_i asks to have it revealed. Formally, we assume $\mathcal{F}_{\text{COM}}$ is equipped with commands `commit` and `open` described in Fig. 13 (more will be defined later).

Some general remarks on the definition of $\mathcal{F}_{\text{COM}}$: 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 command to $\mathcal{F}_{\text{COM}}$ 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 $\mathcal{F}_{\text{COM}}$ is not used as intended, e.g., the honest players do not agree on the command to execute, $\mathcal{F}_{\text{COM}}$ will output all its private data on COM.`leak` and let the adversary fully determine all outputs via COM.`infl`.

Below we will use the following short-hand for describing interactions with $\mathcal{F}_{\text{COM}}$. The symbol $[\cdot]_i$ denotes a variable in which $\mathcal{F}_{\text{COM}}$ keeps a committed value received from player P_i. Thus when we write $[a]_i$, this means that player P_i has committed to a. By $P_i : [a]_i \Leftarrow a$ we mean that the parties use the `commit` command of $\mathcal{F}_{\text{COM}}$ to let P_i commit to a. By $[a]_i \Leftarrow a$ we mean that the parties use the public `commit` command to force P_i to commit to a. By $a \Leftarrow [a]_i$ we mean that the parties use the `open` command to let all parties learn the

[12] In general, if an implementation is proven secure for a restricted set of environments (restricted on the values they can input on the protocol ports) then it is secure to use the implementation inside any protocol which gives inputs according to the restriction.

The ideal functionality $\mathcal{F}_{\text{COM}}$ has the following basic commands:

commit: This command is executed if in some round player P_i sends $(\texttt{commit}, i, cid, a)$ and in addition all honest players send $(\texttt{commit}, i, cid, ?)$. In this case $\mathcal{F}_{\text{COM}}$ 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 $(\texttt{commit}, i, cid, \texttt{success})$.

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

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

If 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 $(\texttt{open}, i, cid, j)$, in which case we require that some (i, cid, a) is stored. The output to 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 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.

[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 P_i is forced to accept the commitment. In an implementation the other parties can in principle just remember that P_i is committed to the known a, but it is convenient to have an explicit command for this.

Figure 13: The basic commands of the ideal commitment functionality.

value of a. By $P_j : a \leftarrow [a]_i$ we mean that the parties use the designated `open` command to let 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 $\mathcal{F}_{\text{COM}}$ to manipulate committed values, namely to add committed values and multiply them by public constants, as long as the variables belong to the same party. The details are given in Fig. 14. We write $[a_3]_i \leftarrow [a_1]_i + [a_2]_i$ and $[a_3]_i \leftarrow \alpha[a_2]_i$, where it is understood that $a_3 = a_1 + a_2$ respectively $a_3 = \alpha a_2$.

Later we show how to implement the basic commands and the simple manipulation commands. For now we just assume that we have them. Once we have these commands it

The ideal functionality $\mathcal{F}_{\text{COM}}$ has the following simple manipulation commands:

add: This command is executed if in some round all honest players send $(\texttt{add}, i, cid_1, cid_2, cid_3)$, in which case we require that some (i, cid_1, a_1) is stored and some (i, cid_2, a_2) is stored. As a result $\mathcal{F}_{\text{COM}}$ 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 $(\texttt{mult}, i, \alpha, cid_2, cid_3)$, in which case we require that $\alpha \in \mathbb{F}$ and that some (i, cid_2, a_2) is stored. As a result $\mathcal{F}_{\text{COM}}$ stores $(i, cid_3, \alpha a_2)$. The output to all parties is $(\texttt{mult}, i, \alpha, cid_2, cid_3, \texttt{success})$.

Figure 14: The simple manipulation commands of the ideal commitment functionality.

The ideal functionality $\mathcal{F}_{\text{COM}}$ has the following advanced manipulation commands:

transfer: This command is executed if in some round all honest players send $(\texttt{transfer}, i, cid, j)$, in which case we require that some (i, cid, a) is stored. As a result $\mathcal{F}_{\text{COM}}$ stores (j, cid, a). The output to all parties is $(\texttt{transfer}, i, cid, j, \texttt{success})$, except P_j which gets $(\texttt{transfer}, i, cid, j, a)$.

If 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 $(\texttt{transfer}, i, cid, j, \texttt{fail})$.

mult: This command is executed if in some round all honest players send $(\texttt{mult}, i, cid_1, cid_2, cid_3)$, in which case we require that some (i, cid_1, a_1) is stored and that some (i, cid_2, a_2) is stored. As a result $\mathcal{F}_{\text{COM}}$ stores $(i, cid_3, a_1 a_2)$. The output to all parties is $(\texttt{mult}, i, cid_1, cid_2, cid_3, \texttt{success})$.

If 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 $(\texttt{transfer}, i, cid, j, \texttt{fail})$.

Figure 15: The advanced manipulation commands of the ideal commitment functionality.

is possible to use $\mathcal{F}_{\text{COM}}$ to implement $\mathcal{F}_{\text{COM}}$. The first step is to extend $\mathcal{F}_{\text{COM}}$ with the more advanced manipulation commands in Fig. 15. We write $[a]_j \leftarrow [a]_i$ and $[a_3]_i \leftarrow [a_1]_i[a_2]_i$ for these commands.

The advanced commands are special in that they can be implemented given the basic commands and the simple manipulation commands.

6.2.1 The Transfer Protocol

The implementation of the command $[a]_j \leftarrow [a]_i$ starts as follows:

1. $P_j : a \Leftarrow [a]_i$.
2. $P_j : [a]_j \Leftarrow a$.
3. If the first command fails, then all parties output `fail`.[13] If the second command fails, then P_j is corrupted. In that case the following is executed:
   (a) $a \Leftarrow [a]_i$.
   (b) $[a]_j \Leftarrow a$.

   If the first command fails, then all parties output `fail`. The second command cannot fail.

Note that if P_j is corrupted and does not do a commitment, then all parties learn a and simply forces a commitment to a for P_j. This is secure, as when P_j is corrupted, then a becomes know to the adversary in the ideal world also (as a is output to P_j). So, the simulator learns a and can simulate the protocol simply by running it on the right a.

This protocol is, however, not sufficient. If P_j is corrupted it could run $P_j : [a']_j \Leftarrow a'$ for $a' \neq a$. And, it is not sufficient to do $P_i : a' \Leftarrow [a']_j$ and let P_i check that $a' = a$, as also P_i could be corrupted. The ideal implementation asks that $a' = a$ even if P_i and P_j are corrupted, therefore this should hold also for the protocol.

The next step is therefore that P_i and P_j prove to the other parties that $a' = a$. We describe this protocol for a single verifier. To convince all the players, the protocol is simply repeated independently (for instance in parallel), each other player P_k taking his turn as the verifier. The outcome of all the proofs are visible by all parties, and the parties accept the proof if and only if all over individual proofs are accepting. Each individual proof proceeds as follows:

1. First P_i picks a uniformly random $r \in_R \mathbb{F}$ and sends it securely to P_j.
2. Then the parties execute $P_i : [r]_i \leftarrow r$ and $P_j : [r]_j \leftarrow r$.
3. Then the verifier P_k broadcasts a challenge $e \in \mathbb{F}$.
4. Then the parties execute $[s]_i \Leftarrow e[a]_i + [r]_i$ and $[s]_j \Leftarrow e[a]_j + [r]_j$.
5. Then $s \Leftarrow [s]_i$ and $s' \Leftarrow [s]_j$.
6. The parties accept the proof only if $s = s'$.

[13] In this case P_i is clearly corrupted, so the command is allowed to fail, as a corrupted P_i in the ideal command also has the influence that it can make the command fail.

It is clear that if P_i and P_j are honest, then all proofs will be accepting. Second, if P_i and 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 P_i or P_j is corrupted, there is nothing to simulate as the simulator learns a in the ideal world. What remains is to argue soundness. I.e., if $a' \neq a$, then the proof will fail with high probability.

So, assume that $a' \neq a$. I.e., P_j made a commitment $[a + \Delta_a]$ for $\Delta_a \neq 0$. Then P_i and P_j makes commitments $[r]_i$ respectively $[r + \Delta_r]_i$. Again 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 P_k is honest and picked after Δ_a and Δ_r are fixed, the probability that $e = \Delta_a^{-1}(-\Delta_r)$ is exactly $1/|\mathbb{F}|$. There are $n - t > n/2$ honest parties, so the probability that all proofs with honest verifiers are accepting is at most $1/|\mathbb{F}|^{n/2}$. If that is not negligible the whole process can be repeated a number of times in parallel to make the error e.g. $2^{-\kappa}$, where κ is the security parameter.

If the proof fails, it could be due to P_j being corrupted, so we run Steps (a) and (b) as described above to give P_i a chance to reveal a and let the other parties do a forced commitment of P_j to a.

6.2.2 The Multiplication Protocol

The implementation of the command $[c]_j \leftarrow [a]_i[b]_i$ starts as follows:

1. $P_i : [c]_i \Leftarrow ab$.

Of course P_i can cheat and commit to $c \neq ab$. So, again a proof is run to check that the commitments are to consistent values. Again we describe the proof only for a single prover P_k.

1. P_i chooses a uniformly random $\beta \in \mathbb{F}$.
2. $P_i : [\gamma]_i \Leftarrow \alpha b$.
3. P_j broadcasts a uniformly random challenge $e \in \mathbb{F}$.
4. $[A]_i \leftarrow e[a]_i + [\alpha]_i$; $A \Leftarrow [A]_i$.
5. $[D]_i \leftarrow A[b]_i - e[c]_i - [\gamma]_i$; $D \Leftarrow [D]_i$.
6. The parties accept the proof only if $D = 0$.

It is easy to show that if P_i remains honest, then the proof succeeds and all values opened are random (or fixed to 0) and so reveal no extra information to the adversary. Using an analysis similar to that for the transfer protocol it can be shown that if $c = ab+\Delta$ for $\Delta \neq 0$, then the proof fails except with probability $1/|\mathbb{F}|$.

6.3 An MPC Protocol for Active Adversaries

The active secure protocol runs be emulating the passive secure protocol, and just in addition makes sure that all parties are committed to all their shares, and that all shares are computed correctly.

Input sharing: Each player P_i holding input $x_i \in \mathbb{F}$ commits to x_i, secret shares x_i using Shamir's secret sharing scheme and makes sure the other parties are committed to their shares:

1. $P_i : [x_i]_i \Leftarrow x_i$.
2. P_i chooses at random a polynomial $x_i(\mathtt{X}) = x_i + \sum_{j=1}^{t} \alpha_j \mathtt{X}^j$, of degree $\leq t$ with $x_i(0) = x_i$.
3. For $j = 1, \ldots, t$: $P_i : [\alpha_j]_i \Leftarrow \alpha_j$.[14]
4. For $x = 1, \ldots, n$: $[x_i(x)]_i \leftarrow \sum_{j=1}^{t} x^j [\alpha_j]_i$.[15][16]
5. For $j = 1, \ldots, n$: $[x_i(j)]_j \leftarrow [x_i(j)]_i$.[17]

Addition: The operand a is shared by a polynomial $a(\mathtt{X})$ of degree $\leq t$, and the parties committed to their shares. I.e., $a(0) = a$ and P_i holds $a_i = a(i)$, and $\mathcal{F}_{\text{COM}}$ holds $[a_i]_i$, for $i = 1, \ldots, n$. The same for b.

For $i = 1, \ldots, n$, the parties execute $[c]_i = [a]_i + [b]_i$.

Multiplication Each operand is shared by a polynomial, as describe in **Addition**. Multiplication proceeds as follows:

Local multiplication step: For $i = 1, \ldots, n$, the parties execute $[d_i]_i = [a_i]_i [b_i]_i$.

Resharing step: P_i secret shares $[d_i]_i$ the same way $[x_i]_i$ was shared in **Input Sharing**, resulting in commitments $[d_{i1}]_1, \ldots, [d_{in}]_n$.

Recombination step: For $j = 1, \ldots, n$, the parties execute $[c_j]_j = \sum_{i=1}^{n} r_i [d_{ij}]_j$, where $(r_1, \ldots, r_n)$ is the recombination vector.

Output Reconstruction: The output y is shared by a polynomial $y(\mathtt{X})$ of degree $\leq t$, and the parties committed to their shares. I.e., $y(0) = y$ and P_i holds $y_i = y(i)$, and $\mathcal{F}_{\text{COM}}$ holds $[y_i]_i$, for $i = 1, \ldots, n$. Let P_j be the party to learn y.

1. For $i = 1, \ldots, n$: $P_j : y_i \Leftarrow [y_i]_i$.
2. Some openings might fail, but since there are at most t corrupted parties, P_j can collect at least $n - t$ shares y_i. Since $n - t > t$, this allows to compute y using Lagrange interpolation

[14] The party commits to the coefficients of the polynomial.

[15] The party evaluates the polynomial inside the commitments.

[16] This linear combination is computed by first computing the multiplications by a constant $x^j [\alpha_j]_i$ and then using the `add` command.

[17] The shares are transferred.

The security follows almost directly from the security of the passive secure protocol. In particular, since the manipulation commands of $\mathcal{F}_{\text{COM}}$ are used, all parties compute all shares exactly as in the passive secure protocol, and $\mathcal{F}_{\text{COM}}$ keeps the shares as secret. The worst that can happen is that some party refuses to carry out one of the commands. We describe how to handle this now.

First, if P_i refuses any of the commands in **Input sharing**, each party P_j will simply get share $[x_j]_j \Leftarrow 0$. This ensure that the shares are shares of the 0-polynomial and that all parties are committed. This corresponds to P_i having chosen input 0.

In **Addition** there are no commands which can fail, and we described how to handle the failures in **Output Reconstruction**.

It remains to be described what should be done if a player P_i fails in **Multiplication**. In general, the simplest way to handle such failures is to go back to the start of the computation, open the input values of the players that have just been disqualified, and restart the computation, simulating openly the disqualified players. This allows the adversary to slow down the protocol by a factor at most linear in n. This solution works in all cases. However, in the i.t. case when $t < n/3$, we can do better: after multiplying shares locally, we have points on a polynomial of degree $2t$, which in this case is less than the number of honest players, $n - t$. In other words, reconstruction of a polynomial of degree $2t$ can be done by the honest players on their own. So, the recombination step can always be carried out: we just tailor the recombination vector to the set of at least $n-t$ players that actually completed the local multiplication step correctly.

6.4 Realization of $\mathcal{F}_{\text{COM}}$: Information Theoretic Scenario

We assume throughout this subsection that we are in the i.t. scenario and that $t < n/3$. 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 14 *A player P sends a value a_i to each player P_i (also to himself). P is supposed to choose these such that $a_i = f(i)$ for all i, for some polynomial $f(X)$ of degree at most t where $t < n/3$ is the maximal number of corrupted players. At some later time, P is supposed to reveal the polynomial $f(X)$ he used, and each P_i reveals a_i, The polynomial is accepted if values of at most t players disagree with $f(X)$ (we cannot demand fewer disagreements, since we may get t of them even if P was honest).*

1. *We assume here (for simplicity) that $n = 3t + 1$. Suppose the adversary corrupts P. Show how to choose two different polynomials $f(X), f'(X)$ of degree at most t and values $\tilde{a}_i$ for P to send, such that P can later reveal and have accepted both $f(X)$ and $f'(X)$.*

2. *Suppose for a moment that we would settle for computational security, and that P must send to P_i, not only a_i, but also his digital signature s_i on a_i. We assume that we can force P 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 P 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 P 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 outline 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.

6.4.1 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 correction.

Concretely, define the shares

$$\mathbf{s}_f = (f(1), \ldots, f(n)),$$

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

$$\tilde{\mathbf{s}} = \mathbf{s} + \mathbf{e}.$$

Then a is uniquely defined by $\tilde{\mathbf{s}}$.

In fact, more is true, since the entire polynomial f 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 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 $w_H(\mathbf{e}), w_H(\mathbf{u}) \leq t$ and $t < n/3$, there are at $\geq n - 2t > t$ positions in which the coordinates of both are simultaneously zero. Thus, for more than t values of i we have

$$f(i) = g(i).$$

Since both polynomials have degree at most t, this means that

$$f(\mathtt{X}) = g(\mathtt{X}) \ .$$

6.4.2 The Protocol

Based on the above observation, we see that we have a commitment scheme if we can find a protocol which ensures that all honest parties hold consistent shares. We describe this mechanism later, but first described the implementation assuming that we have such a protocol.

commit: The party to commit to a value x generates a Shamir sharing of x using the sub-protocol assumed above. As a result each P_i gets a share x_i, and the shares of the honest parties are on a polynomial of degree at most t.

public commit: Each P_i takes his share to be $a_i = a$. This is a share on the polynomial $a(\mathtt{X}) = a$.

open: The value a is shared as follows: If P_i is honest, then P_i knows a polynomial $a(\mathtt{X})$ of degree $\leq t$, and P_j holds $a_j = a(j)$. Even if P_i is corrupted will the shares of the honest parties be on some polynomial $a(\mathtt{X})$ of degree $\leq t$.

1. P_i broadcasts $a(\mathtt{X})$.
2. For $j = 1, \ldots, n$ each P_j broadcasts a_j.
3. The opening is accepted if and only if $a(\mathtt{X})$ has degree $\leq t$ and $a_j = a(j)$ for at least $n - t$ parties. In that case, the value of the opening is taken to be $a = a(0)$.

designated open: As above, but the shares are only sent to P_j. If P_j rejects the opening, it broadcasts a public complaint, and P_i must then do a normal opening. If that one fails, all parties output `fail`.

add: To add two commitments $[a]_i$ and $[b]_i$, shared by $a(\mathtt{X})$ and $b(\mathtt{X})$, the party P_i computes $c(\mathtt{X}) = a(\mathtt{X}) + b(\mathtt{X})$, and each P_j computes $c_i = a_i + b_i$.

multiplication by constant: To multiply a commitment $[b]_i$, shared by $b(\mathtt{X})$, by a constant α the party P_i computes $c(\mathtt{X}) = \alpha b(\mathtt{X})$, and each P_j computes $c_i = \alpha b_i$.

6.4.3 Forcing Consistent Shares

The mechanism we will use to force consistent shares is called dispute control. It has the advantage, over some other techniques, that it is very efficient as long as there are no corruptions, which is the typical case in practice.

The basic idea is the following. To check that P_i creates a consistent sharing, we could simply let P_i first do a sharing, where each P_j learns $s_j = s(j)$. Then P_i broadcasts $s(\mathtt{X})$ and each P_j broadcasts s_j. Then it is checked that $s(\mathtt{X})$ has degree $\leq t$ and that $s(j) = s_j$ for all P_j. There are two obvious problems with this approach:

1. It is insecure to broadcast $s(\mathtt{X})$ and the shares, as it reveals the secret s.

2. Even if P_i is honest could some corrupt P_j broadcast $s_j^* \neq s_j$, which would result in $s_j^* \neq s(j)$ and the sharing being rejected.

Before we describe how to solve these problems we introduce some notation. When P_i shared a secret s by giving share s_j to P_j, we write

$$[s]_i = (s_1, \ldots, s_n) \ .$$

I.e., $[s]_i$ is the vector of shares. We call $[s]_i$ consistent if there exists a polynomial $s(\mathtt{X})$ of degree $\leq t$ such that $s(j) = s_j$ for all *honest* parties.

After having dealt

$$[s]_i = (s_1, \ldots, s_n) \ ,$$

P_i will pick a uniformly random $r \in \mathbb{F}$ and deal a sharing

$$[r]_i = (r_1, \ldots, r_n) \ .$$

Then a challenge $e \in \mathbb{F}$ is given[18] and the parties compute

$$[a]_i = e[s]_i + [r]_i = (s_1 + er_1, \ldots, s_n + er_n) \ .$$

It can be see that the set of consistent sharings constitutes a linear vector space. So, if both $[s]_i$ and $[r]_i$ are consistent, then so is $[a]_i$. Equally important, if $[s]_i$ is not consistent, then the probability that $[a]_i$ is consistent is at most $1/|\mathbb{F}|$: If $[s]_i$ is not in the linear vector space of consistent sharings, then the probability that $e[s]_i + [r]_i$ happens to be in the space is at most $1/|\mathbb{F}|$. Therefore it is sufficient to test that $[a]_i$ is consistent. Finally, since r is random, it is secure to reveal $a = es + r$, and we can do the test by broadcasting the shares of $[a]_i$ as described above.

The second problem is that even though $[a]_i = (a_1, \ldots, a_n)$ is consistent, a corrupted P_j can broadcast $a_j^* \neq a_j$. To handle this each P_i has an associated dispute set $D_i \subseteq \{1, \ldots, n\}$ known by all parties. Initially all $D_i = \emptyset$. If some P_j broadcasts $a_j^* \neq a_j$ in the test, then P_j is added to D_i and the test is repeated. When j is added to D_i we say that a dispute arises.

To avoid that the same dispute arises twice, P_i will give P_j the share 0 in all future sharings. We call this a corrected sharing. I.e., when the test is run again it will be the case that $s_j = 0$ and $r_j = 0$, and therefore $a_j = es_j + r_j = 0$.[19] This way it is known by all parties what are the shares of P_j. Therefore P_j does not have to broadcast a_j — the parties simply use $a_j \stackrel{\text{DEF}}{=} 0$ for $j \in D_i$. If P_i broadcasts $a(\mathtt{X})$ with $a(j) \neq 0$ for $j \in D_i$ it is now clear that P_i is corrupted. In fact, we can just enforce that P_i always broadcast $a(\mathtt{X})$ with $a(j) = 0$ for $j \in D$ simply by letting all parties define the message broadcast by P_i to be $a(\mathtt{X}) \stackrel{\text{DEF}}{=} 0$ if that is not the case. After this convention, it is clear that if it again happens that $a_j \neq a(j)$, then this is for some $j \notin D_i$. We can therefore add j to D_i and repeat again. This will eventually terminate, as D_i can get size at most n.

[18] Again the other parties can all act as verifiers in one of n proofs.

[19] Fixing the share of $P_j \in D_i$ is secure, as P_j is corrupted when P_i is honest. Therefore the corrupted parties know the share of P_j anyway. So, fixing it to a known value does not leak new information.

In fact, the process terminates already if $|D_i| > t$: It is clear that if P_i is honest, then $j \in D_i$ implies that P_j is corrupted. So, for all honest P_i it will always be the case that $|D_i| \leq t$. In particular, if it happens that $|D_i| > t$ for some P_i, then all parties consider P_i corrupted and start ignoring all messages from P_i.

This means that whenever some P_i has to share a value s, it will be the case that $|D_i| \leq t$. Therefore it is possible to pick a polynomial $s(\mathtt{X})$ of degree $\leq t$ for which $s(0) = s$ and $s(j) = 0$ for $j \in D_i$ (as at most $t+1$ points are fixed). Furthermore, a random such polynomial can be picked efficiently. Here is one way to do it:

1. First pick a uniformly random polynomial $s'(\mathtt{X})$ of degree $\leq t$ and with $s'(0) = s$. Then compute $s'_j = s'(j)$ for $j = 1, \ldots, n$.

2. Compute a correction polynomial $c(\mathtt{X})$ of degree $\leq t$ with $c(0) = 0$ and $c(j) = s'_j$ for $j \in D_i$. If $|D_i| < t$, then add the restrictions $c(j) = 0$ for $t - |D_i|$ parties P_j with $j \notin D_i$, to get $t+1$ restrictions. From these $t+1$ restriction $c(\mathtt{X})$ can be computed using Lagrange interpolation.

3. Let $s(\mathtt{X}) = s'(\mathtt{X}) - c(\mathtt{X})$.

Clearly, $s(0) = s$ and $s(j) = 0$ for $j \in D_i$. Furthermore, since all parties could compute $c(\mathtt{X})$ themselves, dealing a sharing using $s(\mathtt{X})$ is as secure as dealing with $s'(\mathtt{X})$ and then all parties doing the correction $s_j = s'_j - c(j)$ themselves. Since dealing with $s'(\mathtt{X})$ is perfectly secure (it is a standard sharing) it follows that dealing with the corrected $s(\mathtt{X})$ is also perfectly secure. This shows how to do a corrected sharing and that it is secure to use corrected sharings.

This shows how to make consistent sharings, which in turn gives a secure implementation of $\mathcal{F}_{\text{COM}}$ and then $\mathcal{F}_{\text{SFE}}$. Note that we assumed that $t < n/3$. Below we show how to get $t < n/2$ in the cryptographic model. It is possible to get $t < n/2$ even in the i.t. model, but we will not look at this.

6.4.4 Formal Proof for the $\mathcal{F}_{\text{COM}}$ realization

We have not given a full formal proof that the $\mathcal{F}_{\text{COM}}$ realization we presented really implements $\mathcal{F}_{\text{COM}}$ securely according to the definition. For this, one needs to present a simulator and prove that it acts as it should according to the definition. We will not do this in detail here, but we will give the main ideas one needs to build such a simulator — basically, one needs the following two observations:

- If player P_i is honest and commits to some value x_i, then since the commitment is based on secret sharing, this only results in the adversary seeing an unqualified set of shares, insufficient to determine x_i (anything else the adversary sees follows from these shares). The set of shares is easy to simulate even if x_i is not known, e.g., by secret sharing an arbitrary value and extracting shares for the currently corrupted players. This simulation is perfect because our analysis above shows that an unqualified set of shares have the same distribution regardless of the value of the secret.

If the (adaptive) adversary corrupts P_i later, it expects to see all values related to the commitment. But then the simulator can corrupt P_i in the ideal process and learn the value x_i that was committed to. It can then easily make a full set of shares that are consistent with x_i and show to the adversary. This can be done by solving a set of linear equations, since each share is a linear function of x_i and randomness chosen by the committer.

- If P_i is corrupt already when it is supposed to commit to x_i, the adversary decides all messages that P_i should send, and the simulator sees all these messages. As we discussed, either the commitment is rejected by the honest players and P_i is disqualified, or the messages sent by P_i determine uniquely a value x_i'. So, the simulator can compute x_i' and commit to x_i' on $\mathcal{F}_{\text{COM}}$ on behalf of the corrupted P_i.

6.5 Realization of $\mathcal{F}_{\text{COM}}$: Cryptographic Scenario

We have now seen how to implement $\mathcal{F}_{\text{COM}}$ in the i.t. scenario. Handling the cryptographic case can be done in various ways, each of which can be thought of as different ways of adapting the information theoretic solution to the cryptographic scenario.

6.5.1 Using Encryption to Implement the Channels

A very natural way to adapt the information theoretic solution is the following: since the i.t. protocol works assuming perfect channels connecting every pair of players, we could simply run the information theoretically secure protocol, but implement the channels using encryption, say by encrypting each message under the public key of the receiver. Intuitively, if the adversary is bounded and cannot break the encryption, he is in a situation no better than in the i.t. scenario, and security should follow from security of the information theoretic protocol.

In fact, we showed in Section 5 that this is when the adversary is static. So, for a static adversary, standard semantically secure encryption provides a secure realization of this communication functionality. It turns out that for an adaptive adversary, one needs a strong property known as non-committing encryption [11]. The reason is as follows: suppose player P_i has not yet been corrupted. Then the adversary of course does not know his input values, but it has seen encryptions of them. The simulator doesn't know the inputs either, so it must make fake encryptions with some arbitrary content to simulate the actions of P_i. This is all fine for the time being, but if the adversary corrupts P_i later, then the simulator gets an input for P_i, and must produce a good simulation of P_i's entire history to show to the adversary, and this must be consistent with this input and what the adversary already knows. Now the simulator is stuck: it cannot open its simulated encryptions the right way. Non-committing encryption solves exactly this problem by allowing the simulator to create "fake" encryptions that can later be convincingly claimed to contain any desired value.

Both semantically secure encryption and non-committing encryption can be implemented based on any family of trapdoor one-way permutations, so this shows that these general complexity assumptions are sufficient for general cryptographic MPC. More effi-

cient encryption schemes exist based on specific assumptions such as hardness of factoring. However, known implementations of non-committing encryption are significantly slower, typically by a factor of κ, where κ is the security parameter.

6.5.2 Cryptographic implementations of higher-level functionalities

The above approach only gives the threshold $t < n/3$, as the i.t. protocol had threshold $t < n/3$. Recall, however, that we only needed the assumption $t < n/3$ when implementing $\mathcal{F}_{\text{COM}}$. The implementation of $\mathcal{F}_{\text{SFE}}$ from $\mathcal{F}_{\text{COM}}$ was secure for $t < n/2$. So, if we could get an implementation of $\mathcal{F}_{\text{COM}}$ for $t < n/2$, we would have an implementation of $\mathcal{F}_{\text{SFE}}$ for $t < n/2$. In the cryptographic model, getting such an implementation can be done in several ways. We sketch one of them.

If the adversary is static, we can use, e.g., the commitments from [13] based on q-one-way homomorphisms, which exists, e.g., if RSA is hard to invert or if the decisional Diffie-Hellman problem in some prime order group is hard. We then require that the field over which we compute is $\mathbb{Z}_q$. A simple example is if we have primes p, q, where $q|p-1$ and g, h, y are elements in $\mathbb{Z}_p^*$ of order q chosen as public key by player P_i. Then $[a]_i$ is of the form $(g^r, y^a h^r)$, i.e., a Diffie-Hellman (El Gamal) encryption of y^a under public key g, h. It is clear how to implements the simple manipulation commands, as the commitment is linear.

One could then implement the advanced manipulation commands in the way we did above. One can, however, do better. In [13], protocols are shown for proving efficiently in zero-knowledge that you know the contents of a commitment, and that two commitments contains the same value, even if they were done with respect to different public keys. It is trivial to derive a `transfer` protocol from this: P_i privately reveals the contents and random bits for $[a]_i$ to P_j (by sending them encrypted under P_j's public key). If this is not correct, P_j complains, otherwise he makes $[a]_j$ and proves it contains the same value as $[a]_i$. In [13] there is also given a protocol showing that a commitment contains the product of two other commitments. This gives an efficient `mult` protocol. We note that, in order to be able to do a simulation-based proof of security of this $\mathcal{F}_{\text{COM}}$ implementation, each player must give zero-knowledge, proof of knowledge of his secret key initially, as well as prove that he knows the contents of each commitment he makes.

If the adversary is adaptive, the above technique will not work, for the same reasons as explained in the previous subsection. It may seem natural to then go to commitments and encryption with full adaptive security, but this means we need to use non-committing encryption and so we will loose efficiency. However, under specific number theoretic assumptions, it is possible to build adaptively secure protocols using a completely different approach based on homomorphic public key encryption, without loosing efficiency compared to the static security case[22].

7 Protocols Secure for General Adversary Structures

It is relatively straightforward to use the techniques we have seen to construct protocols secure against general adversaries, i.e., where the adversary's corruption capabilities are not described only by a threshold t on the number of players that can be corrupt, but by a general adversary structure, as defined earlier.

What we have seen so far can be thought of as a way to build secure MPC protocols from Shamir's secret sharing scheme. The idea is now to replace Shamir's scheme by something more general, but otherwise use essentially the same high-level protocol.

To see how such a more general scheme could work, observe that the evaluation of shares in Shamir's scheme can be described in an alternative way. If the polynomial used is $f(\mathtt{X}) = s + a_1\mathtt{X} + ... + a_t\mathtt{X}^t$, we can think of the coefficients $(s, a_1, ..., a_t)$ as being arranged in a column vector $\mathbf{a}$. Evaluating $f(\mathtt{X})$ in points $1, 2, .., n$ is now equivalent to multiplying the vector by a Van der Monde matrix M, with rows of form $(i^0, i^1, ..., i^t)$. We may think of the scheme as being defined by this fixed matrix, and by the rule that each player is assigned 1 row of the matrix, and gets as his share the coordinate of $M\mathbf{a}$ corresponding to his row.

It is now immediate to think of generalizations of this: to other matrices than Van der Monde, and to cases where players can have more then one row assigned to them. This leads to general linear secret sharing schemes, also known as Monotone Span Programs (MSP). The term "linear" is motivated by the fact any such scheme has the same property as Shamir's scheme, that sharing two secrets s, s' and adding corresponding shares of s and s', we obtain shares of $s + s'$. The protocol constructions we have seen have primarily used this linearity property, so this is why it makes sense to try to plug in MSP's instead of Shamir's scheme. There are, however, several technical difficulties to sort out along the way, primarily because the method we used to do secure multiplication only generalizes to MSP's with a certain special property, so called multiplicative MSP's. Not all MSP's are multiplicative, but it turns that any MSP can be used to construct a new one that is indeed multiplicative.

Furthermore, it turns out that for *any* adversary structure, there exists an MSP-based secret sharing scheme for which the unqualified sets are exactly those in the adversary structure. Therefore, these ideas lead to MPC protocols for any adversary structure where MPC is possible at all.

For details on how to use MPS's to do MPC, see [15].

8 A Double Action

8.1 Introduction

In this section we describe a practical application of secure MPC to implement a secure double auction, which was used to clear the Danish 2008 market for contracts on sugar beets. This was the first large scale application of MPC.

In fact, despite the obvious potential that MPC has in solving a wide range of problems, we have seen virtually no practical applications of MPC in the past. This is probably in part due to the fact that direct implementation of the first general protocols, as those described in above sections, would lead to very inefficient solutions. Another factor has been a general lack of understanding in the general public of the potential of the technology.

A lot of research has gone into solving the efficiency problem, both for general protocols (cf. [22, 26, 14]) and for special types of computations such as voting. The sugar beets double auction was developed as part of two research projects SCET (Secure Computing, Economy and Trust) and SIMAP (Secure Information Management and Processing) carried out at Aarhus University (cf. `http://sikkerhed.alexandra.dk/uk/projects/scet/` and `http://sikkerhed.alexandra.dk/uk/projects/simap/`). These projects aimed at improving the efficiency of MPC, this time with an explicit focus on a range of economic applications, which were believe to have particularly interesting for practical use.

In the economic field of mechanism design the concept of a trusted third party has been a central assumption since the 70's [27, 33, 21]. Ever since the field was initiated it has grown in momentum and turned into a truly cross disciplinary field. Today, many practical mechanisms require a trusted third party. In particular, the SCET and SIMAP projects considered:

- Various types of auctions. This is not limited to only standard highest bid auctions with sealed bids but also includes, for instance, variants with many sellers and buyers, s-called double auctions: essentially scenarios where one wants to find a fair market price for a commodity given the existing supply and demand in the market.

- Benchmarking, where several companies want to combine information on how their businesses are running, in order to compare themselves to best practice in the area. The benchmarking process is either used for learning, planning or motivation purposes. This of course has to be done while preserving confidentiality of companies' private data.

When looking at such applications, it was found that the computation needed is basically elementary arithmetic on integers of moderate size, typically around 32 bits. More concretely, quite a wide range of the cases require only addition, multiplication and comparison of integers. The known generic MPC protocols can usually handle addition and multiplication very efficiently, by using the field $\mathbb{F} = \mathbb{Z}_p$ for a prime p chosen large enough compared to the input numbers to avoid modular reductions. This gives integer addition and multiplication by doing addition and multiplication in $\mathbb{F}$.

This is efficient because each number is shared "in one piece"[20] using a linear secret

[20] As opposed to a bit-wise sharing.

sharing scheme, so that secure addition, for instance, requires only one local addition by each player. Unfortunately, this also implies that comparison is much harder. A generic solution would express the comparison operation as an arithmetic circuit over $\mathbb{Z}_p$, but this would be far too large to give a practical solution, because the circuit would not have access to the binary representation of the inputs. Instead, special purpose techniques for comparison have been developed. We will have a closer look at these in Section 8.5.

8.2 The Application Scenario

In this section we describe the practical case in which the secure auction was deployed. In Denmark, several thousand farmers produce sugar beets, which are sold to the company Danisco, which is the only sugar producing company on the Danish market. Farmers have contracts that give them production rights, that is, a contract entitles a farmer to produce a certain amount of beets per year and deliver them to Danisco for a fixed price. These contracts can be traded between farmers, but trading has historically been very limited and has been done only via bilateral negotiations. In the years up to 2008, however, the EU drastically reduced the support for sugar beet production. This and other factors meant that there was now an urgent need to reallocate contracts to farmers where productions payed off best. It was realized that this was best done via a nation-wide exchange, a double auction. The details of this mechanism can be found in [2].

Briefly, the goal is to find the so-called market clearing price, which is a price per unit of the commodity that is traded (here the contracts for growing sugar beets for Danisco). What happens is that each buyer specifies, for each potential price, how much he is willing to buy at that price, similarly all sellers say how much they are willing to sell at each price. All bids go to an auctioneer, who computes, for each price, the total supply and demand in the market. Since we can assume that supply grows and demand decreases with increasing price, there is a price where total supply equals total demand, and this is the price we are looking for. Finally, all bidders who specified a non-zero amount to trade at the market clearing price get to sell/buy the amount at this price.

This could in principle be implemented with a single trusted party as the auctioneer. However, in the given scenario, there are some additional security concerns implying that this is not a satisfactory solution: Bids clearly reveal information on a farmer's economic position and her productivity, and therefore farmers would be reluctant to accept Danisco acting as auctioneer, given its position in the market. Even if Danisco would never misuse its knowledge of the bids in future price negotiations, the mere fear of this happening could affect the way farmers bid and lead to a suboptimal result of the auction. On the other hand, contracts in some cases act as security for debt that farmers have to Danisco, and hence the farmers' organization DKS running the auction independently would not be acceptable for Danisco. Finally, the common solution of delegating the legal and practical responsibility by paying e.g. a consultancy house to be the trusted auctioneer would be a very expensive solution. It was therefore decided to implement an electronic double auction, where the role of the auctioneer would be played by a multiparty computation done by representatives for Danisco, DKS and the SIMAP project.

A three party solution was selected, partly because it was natural in the given scenario, but also because it allowed using efficient information theoretic tools such as secret sharing,

rather than more expensive cryptographic methods needed when there are only two parties.

8.3 The Auction System

In the system that was deployed, a web server was set up for receiving bids, and three servers were set up for doing the secure computation. Before the auction started, a public/private key pair was generated for each computation server, and a representative for each involved organization stored the private key on a USB stick, protected under a password.

Encrypt and Share Curve. Each bidder logged into the webserver and an applet was downloaded to her PC together with the public keys of the computation servers. After the user typed in her bid, the applet secret shared the bids, creating one share for each server, and encrypted the shares under the respective server's public key. Finally the entire set of ciphertexts were stored in a database by the webserver.

As for security precautions on the client side, the system did not explicitly implement any security against cheating bidders, other than verifying their identity. The reason being that the method used for encrypting bids implicitly gives some protection: it is a variant of a technique called non-interactive VSS based on pseudorandom secret sharing presented in [24]. We will not look at the details of this method, but using it, an encrypted bid is either obviously malformed, or is guaranteed to produce consistently shared values. This means that the only cheating that is possible, is to submit bids that are not monotone, i.e., bids where, for instance, the amount you want to buy does not decrease with increasing price, as it should. It is easy to see that this cannot be to a bidders advantage.

Secure Computation. After the deadline for the auction had passed, the servers were connected to the database and each other, and the market clearing price was securely computed, as well as the quantity each bidder would buy/sell at that price. The representative for each of the involved parties triggered the computation by inserting her USB stick and entering her password on her own machine.

The computation was based on standard Shamir secret sharing over $\mathbb{F} = \mathbb{Z}_p$, where p was a 64-bit prime. Standard protocols with passive security were used for addition and multiplication, while a variant of a protocol from [20] was used for secure comparison. The SIMAP protocol settled for passive security because the most important goal was to avoid that any party would need access to bids in cleartext at any point, and passive security already achieves this.

The system worked with a set of 4000 possible values for the price, meaning that after the total supply and demand had been computed for all prices, the market clearing price could be found using binary search over 4000 values, which means about 12 secure comparisons.

8.4 Practical Evaluation and Potential

The bidding phase ran smoothly, with very few technical questions asked by users. The only issue was that the applet on some PC's took up to a minute to complete the encryption of the bids. It is not surprising that the applet needed a non-trivial amount of time, since

each bid consisted of 4000 numbers that had to be handled individually. A total of 1200 bidders participated in the auction, each of these had the option of submitting a bid for selling, for buying, or both.

The actual computation was done January 14, 2008 and lasted about 30 minutes. Most of this time was spent on decrypting shares of the individual bids, which is not surprising, as the input to the computation consisted of about 9 million individual numbers. As a result of the auction, about 25.000 tons of production rights changed owner.

Other than the fact that the system worked and produced correct results, it is of course important what users think. In this connection, one can note the results of an on-line survey that was conducted simultaneously with the bidding phase. Here, about 80% of the respondents said that it was important to them that the bids were kept confidential, and also that they were happy about the confidentiality that the system offered. Also Danisco and DKS were satisfied with the system, and said that they may well run the auction again in following years.

In judging the further potential of multiparty computation, it is important to ask what motivated, at the end of the day, DKS and Danisco to try using such a new and untested technology? One important factor was simply the obvious need for a nation-wide exchange for production rights, which had not existed before, so the opportunity to have a cheap electronic solution —secure or not— was certainly a major reason.

It does seem, however, that security also played a role. If Danisco and DKS would have tried to run the auction using conventional methods, one or more people would have had to have access to the bids, or control over the system holding the bids in cleartext. As a result, some security policy would have had to be agreed, answering questions such as: who should have access to the system and when? who has responsibility if data leaks, and what are the consequences? Since the parties have conflicting interests, this would have lead to very lengthy discussions, possibly bringing the whole project to a halt. Alternatively, the parties might have found a solution in collaboration with a consultancy house as mediator, but this would have been a more expensive solution, and the parties would still have had to agree on whether the mediator's security policy was satisfactory.

As it happened, there was no need for this kind of negotiations at all, since the multiparty computation ensured that no one needed to have access to bids at any point. The conclusion is that the ability of multiparty computation to keep secret everything that is not intended to be public, really is useful in practice, because it short-circuits discussions and concerns about which parts of the data are sensitive and what common security policy one should have for handling such data. In contrast, if some part of the system —even a secure hardware device— has access to the private data in cleartext, one is forced to administrate that part via a security policy that all parties can agree on. It may be time-consuming, expensive or even impossible to reach such an agreement if parties have conflicting interests. One might expect that multiparty computation will turn out to be useful in many similar practical scenarios in the future

8.5 Implementation Details

In this section we describe the technical detail of how the secure computation of the market clearing price was performed.

We assume that there are P different price, $p_1, \ldots, p_P$, B buyers, S sellers, and n servers. We start at a point where each buyer $j = 1, \ldots, B$ for each price p_i shared an integer $d_{i,j}$ among the servers. We denote the sharing by $[d_{i,j}]$. The integer $d_{i,j}$ specifies how much the buyer is willing to buy if the price turns out to be p_i. Similarly, we assume that each seller $j = 1, \ldots, S$ for each price p_i shared an integer $s_{i,j}$ among the servers. We denote the sharing by $[s_{i,j}]$. The integer $s_{i,j}$ specifies how much the seller is willing to sell if the price turns out to be p_i.

8.5.1 Discrete Market Clearing Price

The goal, given by economic mechanism design, is now to find the price where most goods are moved. For this, the servers first run the following code:

1. For all prices p_i, compute
$$[d_i] = \sum_{j=1}^{P} [d_{i,j}]$$
and
$$[s_i] = \sum_{j=1}^{P} [s_{i,j}] \ ,$$
by locally adding shares.

2. The output is a demand "curve" $[d_1], \ldots, [d_P]$ and a supply "curve" $[s_1], \ldots, [s_P]$.

If the demand d and supply s were continuous functions of the price p and were monotonously decreasing respectively monotonously increasing, then the price p_{MCP} where most goods would be traded is given by $d(p_{\text{MCP}}) = s(p_{\text{MCP}})$. Below this price the supply is smaller (or at least not larger) and above this price the demand is smaller (or at least not larger), which leads to a lower amount being traded (or at least not more).

We assume that the curves computed in shared form are monotone as specified above. They are, however, not continuous. We will therefore first compute

$$i_{\text{MCP}} = \max(i | d_{i_{\text{MCP}}} > s_{i_{\text{MCP}}}) \ .$$

We can assume that $d_1 > s_1$ (by e.g. setting $p_1 = 0$), which ensures that i_{MCP} is well-defined.[21] It is then easy to see that the price trading most goods is $p_{i_{\text{MCP}}}$ or $p_{i_{\text{MCP}}+1}$. For our purpose here we simply define the discrete market clearing price to be $p_{i_{\text{MCP}}}$. If the price grid is fine enough we expect the amount of good being traded at $p_{i_{\text{MCP}}}$ and $p_{i_{\text{MCP}}+1}$ to be the same for all practical purposes.

[21] Alternatively we can introduce some dummy price p_0 and define $s_0 = 0$ and d_0 to be the maximal amount and create some dummy sharings $[s_0]$ and $[d_0]$.

Protocol COMPARE:

1. Compute $([a_\ell], \ldots, [a_0]) = \text{BITS}([a])$ and $([b_\ell], \ldots, [b_0]) = \text{BITS}([b])$.
2. For $i = 0, \ldots, \ell$ compute $[c_i] = [a_i] + [b_i] - 2[a_i][b_i]$.
3. Compute $([d_\ell], \ldots, [d_0]) = \text{MS1}([c_\ell], \ldots, [c_0])$.
4. For $i = 0, \ldots, \ell$ compute $[e_i] = [a_i][d_i]$.
5. Output $[c] = \sum_{i=0}^{\ell} [e_i]$.

Figure 16: Secure Comparison

8.5.2 Binary Search

To find i_{MCP} the servers simply do a binary search on $i \in \{1, \ldots, P\}$. They start with $i = \lceil P/2 \rceil$ and securely test whether $d_i > s_i$. If so, they go to a higher price; if not, they go to a lower price. This way they arrive at i_{MCP} using $\log_2(P)$ secure comparisons.

Note that doing a binary search does not violate the security goal of leaking only i_{MCP}. Given i_{MCP} one knows that $d_i > s_i$ for all $i \leq i_{\text{MCP}}$ and $d_i \leq s_i$ for all $i > i_{\text{MCP}}$. Therefore the outcome of all the comparisons done during the computation can be simulated given just the result i_{MCP}.

The binary search is important for the feasibility of the system, as each secure comparison is relatively expensive, meaning that the difference between e.g. 4000 comparison and $\lceil \log_2(4000) \rceil = 12$ comparisons would be a difference between running for minutes and running for days. If we, in a misunderstood attempt to get better security, tried to phrased the entire computation as a binary circuit which computes i_{MCP} while keeping all intermediary results secret shared, the computation would run for weeks.

After i_{MCP} is found, the individual values $d_{i_{\text{MCP}},j}$ and $s_{i_{\text{MCP}},j}$ are reconstructed, and the goods are traded at price $p_{i_{\text{MCP}}}$ and in the revealed amounts.

8.5.3 Secure Comparison

The above approach leaves us only with the problem of taking two secret shared integers $[a]$ and $[b]$ and securely computing a bit $c \in \{0, 1\}$, where $c = 1$ if and only if $a > b$. We write

$$c = [a] \overset{?}{>} [b] \ .$$

Unfortunately there is no efficient algorithm for computing c from a and b using only addition and multiplication modulo p. Instead we will take an approach which involves first securely computing sharings of the individual bits of a and b, and then performing the comparison on the bitwise representations.

Assume first that we have a protocol BITS which given a sharing $[a]$ securely computes sharings $[a_\ell], \ldots, [a_0]$, where $\ell = \lfloor \log_2(p) \rfloor$ and $a_\ell \cdots a_0$ is the binary representation

Protocol MS1:

1. The input is $([c_L], \ldots, [c_1])$.
2. If $L = 1$, then let $[d_1] = [c_1]$ and return $[d_1]$.
3. If $L = 2$, then let $[d_2] = [c_2]$ and $[d_1] = (1 - [c_2])[c_1]$ and return $([d_2], [d_1])$
4. Let $L' = L/2$ and for $i = 1, \ldots, L'$, let $([e_{2i}], [e_{2i-1}]) = \text{MS1}([c_{2i}], [c_{2i-1}])$ and let $[f_i] = [e_{2i}] + [e_{2i-1}]$.[a]
5. Let $([g_{L'}], \ldots, [g_1]) = \text{MS1}([f_{L'}], \ldots, [f_1])$.[b]
6. For $i = 1, \ldots, L'$, let $([d_{2i}], [d_{2i-1}]) = ([g_i][e_{2i}], [g_i][e_{2i-1}])$.
7. Return $([d_L], \ldots, [d_1])$.

[a] $f_i \in \{0, 1\}$ and $f_i = 1$ if $c_{2i} = 1$ or $c_{2i-1} = 1$.
[b] A unary representation of the index of the first pair (c_{2i}, c_{2i-1}) containing a 1.

Figure 17: Most Significant 1

of a, with a_0 being the least significant bit. I.e., $a = \sum_{i=0}^{\ell} 2^i a_i$. Assume furthermore that we have a protocol MS1 which given sharings $[c_\ell], \ldots, [c_0]$ of bits computes sharings $[d_\ell], \ldots, [d_0]$ of bits, where $d_i = 1$ for the largest i for which $c_i = 1$ and where $d_i = 0$ for all other i — if all $c_i = 0$, then let all $d_i = 0$. I.e., $[d_\ell], \ldots, [d_0]$ can be seen as a unary representation of the index i of the most significant 1 in c.

We assume that it never happens that $a = b$. I.e., $a > b$ or $b > a$. This can be guaranteed by introducing some dummy, different least significant bits. The comparison is then performed as in Fig. 16. It is easy to see that $c_i \in \{0, 1\}$ and that $c_i = 1$ if and only if $a_i \neq b_i$. So, the i for which $d_i = 1$ is the most significant bit position in which a and b differ. Therefore $a > b$ if and only if $a_i > b_i$, and as $a_i \neq b_i$ we have that $a_i > b_i$ if and only if $a_i = 1$. So, the result is $c = a_i$, which can be computed as the inner product between a and d. So, c is the correct value.

Clearly the protocol is private, as no values are opened during the computation. Note that we computed a sharing of c. When c is needed one can simply reconstruct. If the comparison is done as a part of a larger secure computation it is, however, in some cases necessary to not leak c. We will see an example of that below.

We also use a version BitCompare which starts with bitwise sharings $([a_L], \ldots, [a_0])$ and $([b_L], \ldots, [b_0])$ and compute $[c]$.[22] Using similar techniques we can develop a secure protocol BitAdd, which takes bitwise sharings $([a_L], \ldots, [a_0])$ and $([b_L], \ldots, [b_0])$ and produces a bitwise sharing $([c_{L+1}], [c_L], \ldots, [c_0])$ of $c = a + b$ (without any modular reduction). And we can produce a secure protocol BitSub, which takes bitwise sharings $([a_L], \ldots, [a_0])$

[22] We use L as bound as it might be the case that $L \neq \ell$.

Protocol BITS:

1. The input is $[a]$.
2. Run $([r], [r_\ell], \ldots, [r_0]) \leftarrow$ RANDOMSOLVEDBITS.
3. Let $[c] = [a] - [r]$ and open this sharing to learn $c = a - r \bmod p$.
4. Let $c_\ell, \ldots, c_0$ be the bitwise representation of c and use BITADD to compute from $([r_\ell], \ldots, [r_0])$ and $(c_\ell, \ldots, c_0)$ sharings $([d_{\ell+1}], \ldots, [d_0])$ of the bits in $d = c + r$.
5. Note that either $d = a$ or $d = a + p$. We can find out which (without leaking it) by computing $[e] = ([d_{\ell+1}], \ldots, [d_0]) \stackrel{?}{>} (p_{\ell+1}, \ldots, p_0)$ using BITCOMPARE. Here $(p_{\ell+1}, p_\ell, \ldots, p_0)$ is a bitwise representation of p; in particular, $p_{\ell+1} = 0$.
6. Without leaking $[e]$ we then subtract p from d if and only if $e = 1$. This can be written as $f = d - ep = a$. This is done by creating dummy sharings $([p_{\ell+1}], [p_\ell], \ldots, [p_0])$ of a bitwise representation of p and computing $[ep_i] = [e][p_i]$ for $i = 0, \ldots, \ell + 1$. This gives a bitwise sharing of ep. Then we use BITSUB on $([d_{\ell+1}], \ldots, [d_0])$ and $([ep_{\ell+1}], \ldots, [ep_0])$ to compute a bitwise sharing $([f_{\ell+1}], \ldots, [f_0])$. Since $f = d - ep = a$ we know that $f_{\ell+1} = 0$ and can throw that one away.
7. Return $([f_\ell], \ldots, [f_0])$.

Figure 18: Splitting a Number into Bits

and $([b_L], \ldots, [b_0])$ of $a \geq b$ and produces a bitwise sharing $([c_L], \ldots, [c_0])$ of $c = a - b$.

Exercise 15 *Implement* BITADD *and* BITSUB *using* $\mathcal{O}(\ell)$ *multiplications.*

We are now left with computing MS1 and BITS. We start with the easier one, which is MS1. For notational convenience we index from 1 and assume that L is a power of two. If there are only two bits c_2c_1, the solution is $d_2 = c_2$ and $d_1 = (1 - c_2)c_1$. Namely, if $c_2 = 1$ then this yields $(d_2, d_1) = (1, 0)$, as it should; if $c_2 = 0$, then it yields $(d_2, d_1) = (0, c_1)$, as it should. The general algorithm then uses a standard recursion technique. See Fig. 17. Security follows from the fact that no sharings are opened. The correctness is straightforward.

We then turn our attention to BITS. At first this seems as a hard challenge, but as often is the case in MPC, a random self reduction will do the job. I.e., we generate a random solved instance of the problem, and then use it to solve to original problem. In doing that we use the randomness of the solved instance to mask the original instance, which allows to reveal the masked values and do most of the hard operations on plaintext. This is a

Protocol RandomSolvedBits:

1. For $i = 0, \ldots, \ell$, run $[r_i] \leftarrow$ RandomBit.
2. Run BitCompare to securely test if $r = \sum_{i=0}^{\ell} 2^i r_i$ is less than p, and leak only the result. If $r \geq p$, then go to Step 1.
3. Let $[r] = \sum_{i=0}^{\ell} 2^i [r_i]$.
4. Return $([r], [r_\ell], \ldots, [r_0])$.

Figure 19: Generating a Random Number and its Bits

very common technique.

Assume that we have a secure protocol RandomSolvedBits which outputs a random sharing $[r]$ along with sharings of its bits $([r_\ell], \ldots, [r_0])$. As we show later, producing such a random solved instance is fairly easy. The protocol then proceeds as described in Fig. 18. The security follows from c being a uniformly random number, which does not leak information about a. The correctness is straight-forward.

We are then stuck with RandomSolvedBits. For a last time we push some of the burden into the future, by assuming that we have a protocol RandomBit which generates a sharing of a random bit. Then a random solved instance is generated as in Fig. 19. Since $r < p$ with probability at least $\frac{1}{2}$, by the definition of ℓ, this protocol terminates after an expected two iterations, and on termination r is clearly a uniformly random integer from $[0..p)$, as desired.

Note that the reason why generating a random solved instance is easier than solving a given instance is that we do not solve the random instance — we generate the solution (the bits) and then generate the instance from the solution.

All that is left is then to implement RandomBit. One inefficient way of doing it is to let all parties generate a sharing of a random bit and then e.g. take the secure XOR of these bits. The reason why this is inefficient is that it would require $n - 1$ multiplications for each of the generated bits. With three servers that is fine, but in general it can be rather inefficient. Furthermore, it is not actively secure as the servers might not all contribute bits, and as can be seen, all the above protocols are indeed active secure, so we should try to also make the generation of random bits active secure.

Our active secure protocol is based on the fact that squaring a non-zero element modulo an odd prime is a 2-to-1 mapping, and given $b = a^2$ one has no idea if the pre-image was a or $-a$. We will let the "sign" of such an a be our random bit. We use a sub-protocol RandomFieldElement which generates a sharing of a random field element. It is implemented by each server sharing a random field elements and then adding them. Here there is no room to cheat: as long as just one server contributes a random field element, the result is random.

The protocol is given in Fig. 20. Note that $c = 1$ if $b = a$ and $c = -1$ if $b = -a$. Since

Protocol RandomBit:

1. Generate $[a] \leftarrow$ RandomFieldElement
2. Compute $[a^2] = [a][a]$ and open it to learn $a^2 \bmod p$. If $a^2 = 0$, the go to Step 1.
3. Let $b \in \{1, \ldots, (p-1)/2\}$ be the smallest square root of a^2.
4. Let $[c] = (b^{-1} \bmod p)[a]$.
5. Output $[r] = 2^{-1}([c] + 1)$.

Figure 20: Generating a Random Bit

b is always taken to be the smallest square root and a is picked at random, it follows that $c = 1$ with probability $\frac{1}{2}$. It follows that $r \in \{0, 1\}$ and that $r = 0$ with probability $\frac{1}{2}$, as desired.

8.5.4 Conclusion

By inspection of the above protocols, and given a solution to Exercise 15, it can be seen that all protocols use a number of multiplications in the order of $\log_2(p)$, and the unmentioned constant is fairly small. This allows to perform comparisons of shred integers relatively efficiently. The ability to split a shared number has many other applications. One is to move a number x shared modulo one prime p into a sharing modulo another prime q. If q is smaller than p, this computes a modulo reduction of x modulo q. Many similar tricks exists, and are constantly being produced, to allow efficient secure computation of specific operations.

Exercise 16 *Assume that you are given a sharing $[a]$ of an element $a \neq 0$, and assume that you can generate a sharing $[r]$ of a uniformly random element $r \neq 0$. Argue that it is secure to compute $[ar] = [a][r]$ and reveal ar. Show how to securely compute $[a^{-1}]$ from $[a]$.*

Exercise 17 *In a Vickrey auction there is a single item, and a number of buyers B_i. Each buyer B_i bids a price p_i. The winner is the buyer with the highest bid, and the winner gets to buy the item, but at the second highest bid. Assume that all prices are different and describe a protocol which computes the winner and the price, and which leaks noting else. You are allowed to assume that you have access to some servers of which more than half are honest. Try to make the system as efficient as possible.*

References

[1] D. Beaver: *Foundations of Secure Interactive Computing*, Proc. of Crypto 91.

[2] P. Bogetoft, K. Boye, H. Neergaard-Petersen, K. Nielsen: *Reallocating sugar beet contracts: Can sugar production survive in Denmark?*, European Review of Agricultural Economics 2007 (34):, pp. 1–q20.

[3] L. Babai, A. Gál, J. Kollár, L. Rónyai, T. Szabó, A. Wigderson: *Extremal Bipartite Graphs and Superpolynomial Lowerbounds for Monotone Span Programs*, Proc. ACM STOC '96, pp. 603–611.

[4] J. Benaloh, J. Leichter: *Generalized Secret Sharing and Monotone Functions*, Proc. of Crypto '88, Springer Verlag LNCS series, pp. 25–35.

[5] M. Ben-Or, S. Goldwasser, A. Wigderson: *Completeness theorems for Non-Cryptographic Fault-Tolerant Distributed Computation*, Proc. ACM STOC '88, pp. 1–10.

[6] Z. Beerliova, M. Hirt: *Perfectly-Secure MPC with Linear Communication Complexity.* TCC 2008, pp. 213–230.

[7] E. F. Brickell: *Some Ideal Secret Sharing Schemes*, J. Combin. Maths. & Combin. Comp. 9 (1989), pp. 105–113.

[8] R. Canetti: *Studies in Secure Multiparty Computation and Applications*, Ph. D. thesis, Weizmann Institute of Science, 1995. (Better version available from Theory of Cryptography Library).

[9] R.Canetti, U.Fiege, O.Goldreich and M.Naor: *Adaptively Secure Computation*, Proceedings of STOC 1996.

[10] R.Canetti: *Universally Composable Security*, The Eprint archive, www.iacr.org.

[11] R. Canetti, U. Feige, O. Goldreich, M. Naor: *Adaptively Secure Multi-Party Computation*, Proc. ACM STOC '96, pp. 639–648.

[12] D. Chaum, C. Crépeau, I. Damgård: *Multi-Party Unconditionally Secure Protocols*, Proc. of ACM STOC '88, pp. 11–19.

[13] R. Cramer, I. Damgård: *Zero Knowledge for Finite Field Arithmetic or: Can Zero Knowledge be for Free?*, Proc. of CRYPTO'98, Springer Verlag LNCS series.

[14] R. Cramer, I. Damgård, S. Dziembowski, M: Hirt and T. Rabin: *Efficient Multiparty Computations With Dishonest Minority*, Proceedings of EuroCrypt 99, Springer Verlag LNCS series.

[15] R. Cramer, I. Damgård and U. Maurer: *Multiparty Computations from Any Linear Secret Sharing Scheme.* In: Proc. EUROCRYPT '00.

[16] R. Cramer. Introduction to Secure Computation. Available from `http://www.brics.dk/~cramer`

[17] C. Crepeau, J.vd.Graaf and A. Tapp: *Committed Oblivious Transfer and Private Multiparty Computation*, proc. of Crypto 95, Springer Verlag LNCS series.

[18] R. Canetti, Y. Lindell, R. Ostrovsky, A. Sahai: *Universally composable two-party and multi-party secure computation.* STOC 2002. pp. 494-503.

[19] D. Dolev, C. Dwork, and M. Naor, *Non-malleable cryptography*, Proc. ACM STOC '91, pp. 542–552.

[20] I. Damgård, M. Fitzi, E. Kiltz, J.B. Nielsen, T. Toft: *Unconditionally Secure Constant-Rounds Multi-party Computation for Equality, Comparison, Bits and Exponentiation.* Proc. of TCC 2006, pp. 285–304, Springer Verlag LNCS.

[21] P. Dasgupta, P. Hammond, E. Maskin: *The Implementation of Social Choice Rules: Some General Results on Incentive Compatibility*, Review of Economic Studies 1979 (46):, pp. 27–42.

[22] I. Damgård and J.B. Nielsen: *Universally Composable Efficient Multiparty Computation from Threshold Homomorphic Encryption*, Proc. of Crypto 2003, Springer Verlag LNCS.

[23] I. Damgård, J.B. Nielsen: *Scalable and Unconditionally Secure Multiparty Computation.* CRYPTO 2007. pp. 572–590.

[24] I. Damgåard and R. Thorbek: *Non-Interactive Proofs for Integer Multiplication*, proc. of EuroCrypt 2007.

[25] M. Fitzi, U. Maurer: *Efficient Byzantine agreement secure against general adversaries*, Proc. Distributed Computing DISC '98.

[26] R. Gennaro, M. Rabin, T. Rabin, *Simplified VSS and Fast-Track Multiparty Computations with Applications to Threshold Cryptography*, to appear in Proc of ACM PODC'98.

[27] A. Gibbard: *Manipulation of Voting Schemes: A General Result*, Econometrica 1973 (41):, pp. 587–601.

[28] O. Goldreich, S. Micali and A. Wigderson: *How to Play Any Mental Game or a Completeness Theorem for Protocols with Honest Majority*, Proc. of ACM STOC '87, pp. 218–229.

[29] M. Hirt, U. Maurer: *Complete Characterization of Adversaries Tolerable in General Multiparty Computations*, Proc. ACM PODC'97, pp. 25–34.

[30] M. Karchmer, A. Wigderson: *On Span Programs*, Proc. of Structure in Complexity, 1993.

[31] J.Kilian: *Founding Cryptography on Oblivious Transfer*, Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing, pages 20-31, Chicago, Illinois, 2-4 May 1988.

[32] S. Micali and P. Rogaway:*Secure Computation*, Manuscript, Preliminary version in Proceedings of Crypto 91.

[33] R.B. Myerson: *Incentives Compatibility and the Bargaining Problem*, Econometrica 1979 (47):, pp. 61–73.

[34] J.B. Nielsen: *Protocol Security in the Cryptographic Model*, PhD thesis, Dept. of Comp. Science, Aarhus University, 2003.

[35] T. P. Pedersen: *Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing*, Proc. CRYPTO '91, Springer Verlag LNCS, vol. 576, pp. 129–140.

[36] P. Pudlák, J. Sgall: *Algebraic Models of Computation and Interpolation for Algebraic Proof Systems* Proc. Feasible Arithmetic and Proof Complexity, Springer Verlag LNCS series.

[37] T. Rabin: *Robust Sharing of Secrets when the Dealer is Honest or Cheating*, J. ACM, 41(6):1089-1109, November 1994.

[38] T. Rabin, M. Ben-Or: *Verifiable Secret Sharing and Multiparty Protocols with Honest majority*, Proc. ACM STOC '89, pp. 73–85.

[39] A. Shamir: *How to Share a Secret*, Communications of the ACM 22 (1979) 612–613.

[40] M. van Dijk: *Secret Key Sharing and Secret Key Generation*, Ph.D. Thesis, Eindhoven University of Technology, 1997.

Index