Multiparty Computation, an Introduction

Ronald Cramer and Ivan Damgård

CPT, Lecture Notes, DAIMI, 2002

1 What is Multiparty Computation?

1.1 The MPC and VSS Problems

Secure *multi-party 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.

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. If the result is that the first millionaire is richer, then he knows that the other guy has fewer millions then him, but this should be all the information he learns about the other guy's fortune. Another example is a 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. In these examples all players learn the same result, i.e, $y_1 = ... = y_n$, 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 is for the signer and is empty, and where y_2 is for the user and 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.

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.

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.

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.

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 $\mathcal{A}$.

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 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* (abbreviated i.t., sometimes also called secure channels) 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. Security can then be guaranteed even when the adversary has unbounded computing power.

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 (it is certainly not clear in the i.t. scenario). One therefore in general has to make a distinction between the case where a broadcast channel is given for free as a part of the model, or whether such a channel has to be simulated by a subprotocol. 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 sees what good guys say befoer acting is sometimes referred to as a *rushing* adversary.

In an asynchronous model of communication where message delivery is not guaranteed, it is still possible to solve most of the problems we consider here, but we stick to synchronous communication for simplicity, but also because problems can only be solved in a strictly weaker sense uisng asynchronous communication. Note, for instance, that if messages are not necessarily delivered, we cannot demand that a protocol generates any output.

1.4 Definitions of Security

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, 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 electronic voting, we would clearly be unhappy about a solution that allowed a cheating voter to vote in a way that relates in a particular way to an honest player's vote. Suppose, for instance, that the vote is a yes/no vote. Then we do not want player P_1 to be able to behave such that his vote is always the opposite 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, and that all submitted votes of the right form are indeed counted. Namely, it may be that a corrupt P_1 does not know how he votes, he just modifies P_2's vote in some clever way and submits it as his own. 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?

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 uncorruptible computer, a so called *Ideal Functionality* F. All players can privately send inputs to and receive outputs from F. F is programmed to execute a certain number of commands, and will, since it is uncorruptible, always execute them correctly according its (public) specification, without leaking any information other than the outputs it is supposed to send to the players. A bit more precisely, the interface of F is as follows: F has an input and an output port for every player. Furthermore, it has two special, so called *corrupt* input and output ports, used for communication with the adversary. In every round, F reads inputs from its input ports, and returns results on the output ports. The general rule is that whenever a player P_i is corrupted, F stops using the i'th input/output ports and the adversary then communicates on behalf of P_i over the corrupted inout/output ports.

The goal of a protocol π is to create, without help from trusted parties, and in presence of some adversary, a situation "equivalent" to the case where we have F available. If this is the case, we say that π *securely realizes* F. Forinstance, the goal of computing a function securely can be specified by an ideal functionality that receives inputs from the players, evaluates the function and returns results to the players. But in fact, any cryptographic task, such as commitment schemes or payments systems can be naturally modelled by an ideal functionality.

In order to give a precise definition, we need to say exactly what we mean by the protocol being "equivalent" to F. Let us reason a little about this. A couple of things are immediately clear: when F is used, corrupting some player P_i means you see the inputs and outputs of that player - but you will learn nothing else. An active attack can change the inputs that P_i uses, but can influence the results computed in no other way - F always returns results to players that are correctly computed based on the inputs it received. So clearly, a protocol that securely realizes F must satisfy something similar.

But more is true: we want that protocol and functionality are equivalent, *no matter* in which context the protocol is used. And we have to realize that this context contains more that just the adversary. It also consists, for instance, of human users or computer systems that supply inputs to the protocol. Or if the protocol is used as a subrutine in a bigger system, that system is certainly part of the environment. So in general, we can think of the environment as an entity that choses inputs that players will use in the protocol and receives the results they obtain. We will define equivalence to mean that *the entire environment* cannot tell eny essential difference between using the protocol and using the ideal functionality.

Towards formalizing this, an important observation is that the adversary is not really an entity separate from the environment, he is actually an integrated part of it. Consider for instance the case where the protocol is used as a subrutine in a higher level protocol. In such a case, the honest players may choose their inputs as a result of what they experience in the higher level protocol. But this higher level protocol may also be attacked by the adversary, and clearly this may give him some influence on the inputs that are chosen. In other words, the choice of inputs at some point in time may be a result of earlier adversarial activity. A second observation relates to the results that honest players compute. Again, if we think of the situation where our protocol is used as a subrutine in a bigger construction, it is clear that the result an honest player obtains may be used in the bigger contruction, and may affect his behavior later. As a result of this, the adversary may be able to deduce information about these results. In other words, adversarial activity now may be a function of results computed by the protocol earlier.

After this discussion, we go to the actual definition of the model: In the *real world*, we have the environment Z and the players $P_1, ..., P_n$ all of whom are modelled as interactive Turing machines (ITM's). The players communicate on a synchronous network using open channels or perfectly secure pairwise communication as specified earlier. In line with the discussion above, the environment Z should be thought of as a conglomerate of everything that is external to the the protocol execution. This includes the adversary, so therefore Z can do everything we described earlier for an adversary, i.e., it can corrupt players passively/actively and statically/adaptively, according to an adversary structure $\mathcal{A}$. This is called a $\mathcal{A}$-environment. The players follow their respective programs specified in protocol π, until they are corrupted and possibly taken over by Z. In addition to this, Z also communicates with the honest players, as follows: in every round Z sends a (possibly empty) input to every honest player, and at the end of every round each honest player computes a result that is then given to Z.

When the protocol is finished, Z outputs a single bit, the significance of which we will return to shortly. In addition to other inputs, all entities get as intitial input a security parameter value k, which is used to control the security level of the execution, e.g., the size of keys to use in the cryptographic scenario. To fully formalize the description, more details need to be specified, such as the exact order in which the different ITM's are activated. Details on this can be found in the appendix.

The ideal world contains the same environment we have in the ideal world, but there are no players. Instead, we have an *ideal functionality* F, and a *simulator* S. As mentioned above, F cannot be corrupted, and it will be programmed to carry out whatever task we want to execute securely, such as computing a function. Recall that we described the interface of F: F has an input and an output port for every player in the real protocol, and *corrupt* input/output ports, for communication with the environment/adversary.

The whole idea is that the environment Z we looked at in the real world should be able to act in the same way in the ideal world. Now, Z has two kinds of activities. First, it is allowed to send inputs to the honest players and see their outputs. We handle this by relaying these data directly to the relevant input/output ports of F. Second, Z expects to be able to attack the protocol by corrupting players, seeing all data they send/receive and possibly control their actions. For this purpose, we have the simulator S. Towards Z, S attempts to provide all the data Z would see in a real attack, namely internal data of newly corrupted players and protocol messages that corrupted players receive. We want Z to work exactly like it does in the real world, so therefore S must go through the protocol in the right time ordering and in every round show data to Z that look like what it would see in the real world. S is *not* allowed to rewind Z. The only help S gets to complete this job is that it gets to use the corrupt input/output ports of F, i.e., towards F, it gets to provide inputs and see outputs on behalf of corrupted players. Concretely, as soon as Z issues a request to corrupt player P_i, both S and F are notified about this. Then the following happens: S is given all input/outputs exchanged on the i'th input/output ports of F until now. F then stops using input/output port number i. Instead it expects S to provide inputs "on behalf of P_i" on the corrupt input port and sends output meant for P_i to S on the corrupt output port. One way of stating this is: we give to S exactly the data that the protocol *is supoosed to release* to corrupt players, and based on this, it should be possible to simulate towards Z all the rest that corrupted players would see in a real protocol execution.

It is quite obvious that whatever functionality we could possibly wish for, could be securely realized simply by programming F appropriately. *However*, do not forget that the ideal world does not exist in real life, it only provides a specification of a functionality we would like to have. The point is that we can have confidence that any reasonable security requirement we could come up with will be automatically satisfied in the ideal world, precisely because everything is done by an uncorruptible party - and so, if we can design a protocol that is in a strong sense equivalent to the ideal functionality, we know that usage of the protocol will guarantee the same security properties – even those we did not explicitly specify beforehand!

We can now start talking about what it means that a given protocol π securely realizes ideal functionality F. Note that the activities of Z have the same form in real as in ideal world. So Z will output one bit in both cases. This bit is a random variable, whose distribution in the real world may depend on the programs of π, Z and also on the security parameter k and Z's input z. We call this variable $REAL_{\pi,Z}(k,z)$. Its distribution is taken over the random choices of all ITM's that take part. Similarly, in the ideal world, the bit output by Z is a random variable called $IDEAL_{F,S,Z}(k,z)$. We then have:

Definition 1. *We say that π $\mathcal{A}$-securely realizes F, if for any $\mathcal{A}$-environment Z, there exists a simulator S with complexity polynomial in that of Z such that for any input z, we have that*

$$|Pr(REAL_{\pi,Adv}(k,z)=0) - Pr(IDEAL_{F,S,Adv}(k,z)=0)|$$

is negligible in k

Some remarks on how to intepret this definition: The output bit of Z can be thought of as its guess at which world it is in. So the definition basically demands that for every context in which the protocol is used, there is a simulator S using no more computing power than the environment, allowing the same environment to use the ideal functionality instead. Moreover, the environment cannot tell which world it is acting in. So in this sense, the definition says that using the protocol "equivalent" to using the ideal functionality.

For instance, the definition implies that the protocol does not release more information to corrupt players than it is "allowed to": in the ideal world, the simulator S gets results for corrupted players directly from F, and based on only this, S can produce a view of the protocol that looks exactly like what corrupt players would in the real world. The definition also implies that honest players get correct results: this is automatically ensured in the ideal world, and any mismatch in the real world could be detected by Z so that the definition could not be satisfied.

There are several possible variants of this definition. The one we gave requires so-called *statistical security*, but can be made stronger by requiring that the two involved probabilities are equal for all k, and not just close. This is called *perfect security.* In both cases we consider all (potentially unbounded) adversaries and environments. This fits with the i.t. scenario. For the cryptographic scenario, we need to restrict adversaries and environments to polynomial time, and we will only be able to prove protocols relative to some complexity assumption - we then speak of *computational security.*

The definition we gave here is a variant of the *universally composable* (UC) security definition given in [8]. This definition builds on several earlier works (see e.g. [1, 23, 6]).

The most useful feature of UC security is exactly the composability: Let us define a *G-hybrid model*, as follows: G is assumed to be an ideal functionality, just like we described above. A protocol π in the G-hybrid model is a real-world protocol that is also allowed to make calls to G through the usual interface, that is, honest player P_i may privately specify inputs to G by sending data directly to the i'th input port, and G returns results to P_i on the i'th output port. If the environment corrupts a player, it uses the corrupt input/output ports of G to exchange data on behalf of the corrupted player. The model allows the protocol to run several independent instances of G, and there is no assumption on the timing of different calls, in particular, they may take place simultaneously.

Of course, π may itself be a secure realization of some ideal functionality F, or put another way: π describes how to implement F securely, assuming functionality G is available. This is defined formally in exactly the same way as above, except that we replace in the definition the real world with the G-hybrid model. Now suppose we have a protocol ρ that securely realizes G

in the real world. We let π^ρ denote the real-world protocol that is obtained by replacing each call in π to G by a call to ρ. Note that this may cause several instances of ρ to be running concurrently. We make no assumption on any synchronization between these instances. Then we have the following, which is proved in the appendix:

Theorem 1. *If protocol π in the G-hybrid model securely realizes F, and protocol ρ in the real world securely realizes G, then protocol π^ρ securely realizes F in the real world.*

As we shall see, this result is incredibly useful when constructing and proving protocols: when building π, we can assume that ideal functionality G is "magically" available, and not worry about how to implement it. When we build ρ, we only have to worry about realizing G, and not about how the protocol will be used later.

2 Results on MPC

We now list some important known results on MPC. A remark on terminology: the security definition works with an environment Z, that includes the adversary as an integrated part that may potentially influence everything the environment does. It is therefore really a matter of taste whether one wants to speak of Z as "the environment" or "the adversary". In the following, we will use both terms, but the formal interpretation will always be the entity Z as defined above. Furthermore, when we speak below of "securely computing" a function, this formally means securely realizing a functionality F_{MPC} that is defined in more detail later.

2.1 Results for Threshold Adversaries

The classical results for the information-theoretic model due to Ben-Or, Goldwasser and Wigderson [4] and Chaum, Crépeau and Damgård [10] state that every function can be securely computed with perfect security in presence of an adaptive, passive adversary, resp. an adaptive, active adversary if and only if the adversary corrupts less than $n/2$, resp. less than $n/3$ players. The fastest known protocols can be found in Gennaro, Rabin and Rabin[18].

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[27]. The most efficient protocols in this scenario are by Cramer, Damgård, Dziembowski, Hirt and Rabin [12].

The most general results for the cryptographic model are by Goldreich, Micali and Wigderson [19] 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. who show [7] that security against adaptive adversaries in the cryptographic model can also be obtained, although at the cost of a significant loss of efficiency.

2.2 Results for General Adversaries

Hirt and Maurer [20] 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 [20] 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 $\mathcal{A}$-adversary, resp. an adaptive, active $\mathcal{A}$-adversary if and only if $\mathcal{A}$ is $Q2$, resp. $\mathcal{A}$ is $Q3$. This is for the case where no broadcast channel is available. The threshold results of [4], [10], [19] 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 $n/3$. It can be shown that no weighted threshold solution exists either for this scenario. 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 [20] rely on quite specialized techniques. Cramer, Damgård and Maurer [13] 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 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 MPC Protocols

In this section we will sketch how to show some of the general results we listed above. More precisely, we wil look at ways to securely realize the following functionality, where we assume a threshold adversary that can corrupt at most t players, and the function to be computed is a function $f : (\{0,1\}^*)^n \rightarrow (\{0,1\}^*)^n$

Some notation: when we say that a functionality receives a message of form $(P_i : mes)$, this means that if P_i is honest at this point, mes was received on the i'th input port, and if P_i has been corrupted, $P_i : mes$ was received on the corrupt input port, i.e., it was sent by environment or simulator as a message on behalf of a corrupted player.

Functionality F_{MPC}
The behavior of the functionality depends on two integer parameters $InputDelay, ComputeDelay$, that are expliained in more detail below.

1. Initially, set $x_i = \perp$ (the empty string) for $i = 1..n$.
2. In the first round, collect all messages received of form $(P_i : Input, v)$, and let I be the set of P_i's occurring as senders. If I includes all honest players, set $x_i = v$, for each $P_i \in I$ and send "Inputs received" on the corrupt output port. If I does not include the set of honest players, send all internal data to the corrupt output port and stop.
   If in a round before round number $InputDelay$, $(P_i : change, v')$ for corrupt player P_i is received, set $x_i = v'$ (note that we may have $v' = \perp$.)
3. If any non-empty message is received from an honest player after Step 2, send all internal data to the corrupt output port and stop. Wait $ComputeDelay$ rounds, then set $(y_1, ..., y_n) = f(x_1, ..., x_n)$, send y_i to P_i (on the i'th output port if P_i is honest, and otherwise on the corrupt output port).

Two remarks on this functionality: The intended way to use the functionality is that all honest players should send their inputs in the first round (along with those corrupt players that want to contribute input), and after this point no honest player should send input. The functionality is defined such that security is only required if it is used as intended. If anything else happens, all internal data are revealed (to environment or simulator) and it becomes trivial to simulate. The reason for the peculiar way to define the input step is to model that honest players must know from the start what input they contribute, but a corrupt player need not be bound to its inout until after $InputDelay$ rounds, and may for instance start the protocol honestly and then stop. The functionality waits for $ComputeDelay$ rounds before it sends the results out. This is to model the fact that the protocol implementing the actual computation takes some number of rounds to finish.

To build a concrete protocol for this problem, we assume that a fixed finite field K is given, and that the function we want to compute is specified as an arithmetic circuit over K. That is, all input values are elements in K and the desired computation is specified as a number of additions and multiplications in K of the input values (or intermediate results). This is without loss of generality: Any function that is feasible to compute at all can be specified as a polynomial size Boolean circuit using, for instance, and, or and not-operations. But any such circuit can be simulated by operations in K: Boolean values true or false can be encoded as 1 resp. 0. Then the negation of bit b is $1-b$, the and of bits b, b' is $b \cdot b'$ and the or becomes $1-(1-b)(1-b')$.

The only necessary restriction on K is that $|K| > n$, but we will assume for concreteness and simplicity that $K = Z_p$ for some prime $p > n$.

Our main tool to build the potocol will be *Secret Sharing*, in particular Shamir's scheme, which is based on polynomials over K. A value $s \in K$ is shared by choosing a random polynomial $f_s()$ of degree at most t such that $f_s(0) = s$. And then sending privately to player P_j the value $f_s(j)$. The well known facts about this methods 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*:

If $h(X)$ is a polynomial of degree at most l and if C is a subset of K with $|C| = l+1$, then

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

where $\delta_i(X)$ 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,

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

We briefly recall why this sholds. The right hand side $\sum_{i \in C} h(i)\delta_i(X)$ is clearly a polynomial of degree at most l that on input i evaluates to $h(i)$ for $i = 1, \ldots, n$. Therefore, it it were not equal to $h(X)$, the difference of the two polynomials would be a non-zero polynomial whose number of zeroes exceeds its degree. This is nonsense.

Another consequence of Lagrange interpolation is that if $h(X)$ is a polynomial of degree at most $n-1$, then there exist easily computable values $r_1, ..., r_n$, such that

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

Namely, $r_i = \delta_i(0)$. We call $(r_1, ..., r_n)$ a *recombination vector.*

We are going to need the following simple fact about recombination vectors:

Lemma 1. *Let* $(r_1, ..., r_n)$ *be any recombination vector, and let* I *be any subset of* $\{1, 2, ..., n\}$ *of size less than* $n/2$. *Then there always exists an* $i \notin I$ *with* $r_i \neq 0$.

Proof. Suppose we share values a, b resulting shares $a_1, ..., a_n, b_1, ..., b_n$, using polynomials f, g of degree $\leq t$, where t is maximal such that $t < n/2$. Then $a_1b_1, a_2b_2, ..., a_nb_n$ is a sharing of ab based on fg which is of degree at most $2t \leq n-1$. If the Lemma was false, there would exist a set I of size at most t which could use r and their shares in a, b to compute ab, but this contradicts the fact that any t or fewer shares contain no information on a, b.

Since the function we are to compute is specified as an arithmetic circuit over K, our task is, loosely speaking to compute a number of additions and multiplications in K of the input values (or intermediate results), while revealing nothing except for the final result(s).

Exercise. 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 simpicity).

- 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?
- Design a scheme that does work in the given context. Hint: in addition to the secret s, create a random element $u \in K$, 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 $t = m/5 - 1$.

3.1 The Passive Case

This section covers the i.t. scenario with a passive adversary. We assume a threshold adversary that can corrupt up to t players, where $t < n/2$. The protocol starts by

Input Sharing Each player P_i holding input $x_i \in K$ secret shares x_i using Shamir's secret sharing scheme: he chooses at random a polynomial f of degree $\leq t$ and send a share to each player, i.e., he sends $f(j)$ to P_j, for $j = 1, ..., n$.

We then work our way gate by gate through the given arithmetic circuit over K, maintaining the following

Invariant All input values and all outputs from gates processed so far are secret shared, i.e. each such value $a \in K$ is shared into shares $a_1, ..., a_n$, where P_i holds a_i. Remark: if a depends on an input from an honest player, this must be a *random* set of shares with the only constraint that it determines a. 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 input 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 by broadcasting the shares $y_1, ..., y_n$, or if y is a value that should go to only player P_j, the shares are sent privately to P_j.

It is therefore sufficient to show how addition and multiplication gates are handled. Assume the input values to a gate are a and b, determined by shares $a_1, ..., a_n$ and $b_1, ..., b_n$, respectively.

Addition For $i = 1, \ldots, n$, P_i computes $a_i + b_i$. The shares $a_1 + b_1, \ldots, a_n + b_n$ determine $a + b$ as required by the invariant.

Multiplication For $i = 1, \ldots, n$, P_i computes $a_i \cdot b_i = \tilde{c}_i$.
Resharing step: P_i secret shares $\tilde{c}_i$, resulting in shares $c_{i1}, ..., c_{in}$, and sends c_{ij} to player P_j.
Recombination step: For $j = 1, \ldots, n$, player P_j computes $c_j = \sum_{i=1}^{n} r_i c_{ij}$, 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 $f(x) = c$. We are going to assume that every output from the circuit comes out of a multiplication gate. This is without loss of generality since we can always introduce a multiplication by 1 on the output without changing the result. This is not strictly necessary, but makes life easier in the proof of security below.

Proof of Security for the Passive Case In this section, we will argue the following result:

Theorem 2. *The protocol described in the previous section realizes F_{MPC} in the i.t. scenario with perfect security against an unbounded, adaptive and passive environment corrupting at most $t < n/2$ players, and with $InputDelay = 1$ and $ComputeDelay$ equal to the depth of the circuit used to implement the function computed.*

For simplicity, we show here a proof of security assuming that each player P_i gets as input a single value $x_i \in K$, and is to receive a single value $y_i \in K$. This generalizes trivially to the case where inputs and outputs can be several values in K.

Recall that to prove security, our task is to build a simulator S which interacts with the environment Z and the ideal functionality.

Since corruptions are passive, we may assume that Z specifies messages for corrupt players to send by following the protocol, by definition of the model these messages are given to S, and S must generate messages on behalf of honest players and show these to Z.

As a result of this, the algorithm of S is as follows, where throughout, A denotes the currently corrupted set, specified as a set of indices chosen from $\{1, 2, ..., n\}$:

1. Whenever Z requests to corrupt a new player P_i, S will as a result see the inputs (if any) specified so far for P_i by Z and results received from F_{MPC} (and will from now on learn future inputs and outputs). Now, S will use this information to reconstruct a complete view of P_i taking part in the protocol up the point of corruption, and will show this view to Z. The view must, of course, be consistent with what Z has seen so far. We describe this reconstruction procedure in more detail below. Finally, we set $A := A \cup \{i\}$. Note that these corruptions may take place at any point during the simulation below of input sharing, computation and output generation.

2. In the first round, S will learn, by definition of F_{MPC}, whether Z has used the functionality correctly, i.e., whether it has specified inputs for all honest players or not. If not, all inputs are revealed, and it becomes trivial to simulate. So we continue, assuming inputs were specified as expected. S specifies arbitrary input values for corrupt players and send them to F_{MPC} (this is no problem, we will learn thr correct values soon).
   In the next round, S does the following for each player P_i: if $i \in A$, record the shares Z has generated on behalf of corrupt players, and reconstruct x_i (which is easy by the assumption that Z follows the protocol). Send $(P_i : \mathit{change}, x_i)$ to F_{MPC}.
   If $i \notin A$, choose t random independent elements in K send these to Z and record them for later use. These elements play the role of the shares of x_i held by corrupt players.
3. S must now simulate towards Adv the computation and reconstruction of the outputs. To simulate the computation, S goes through the circuit with the same order of gates as in the real protocol.
   For each addition gate, where we add intermediate results a, b, each corrupt P_i holds shares a_i, b_i (whihc are known to S). S now simply records the fact that P_i now should add the shares to get $c_i = a_i + b_i$, and also records c_i as the share of $a + b$ known by P_i.
   For each multiplication gate, where we multiply intermediate results a, b, each corrupt P_i holds shares a_i, b_i of a, b. S sets $\tilde{c}_i = a_i b_i$, watches perform Z a normal secret sharing of $\tilde{c}_i$ and record for later use the shares c_{ij} generated. For each honest P_i, S chooses random values $\{c_{ij} | \ P_j \in A\}$ to simulate the resharing done by honest players and sends the values to Adv. Finally, S records the fact that each corrupt P_j now computes $c_j = \sum_{i=1}^{n} r_i c_{ij}$, and also records c_j as the share of ab known by P_j.
4. To simulate the computation of the final result, S uses the fact that it knows from F_{MPC} the result y_i for each corrupt P_i. From the simulation of the circuit in the previous step, S has created a value s_j for each $P_j \in A$, and this value plays the role as P_j's share of y_i.
   S now computes a polynomial $f_{y_i}()$ of degree at most t such that $f_{y_i}(0) = y_i$ and $f_{y_i}(j) = s_j$ for all $P_j \in A$. Then S sets $s_j = f_{y_i}(j)$ for all $P_j \notin A$, and sends these values to Adv, pretending that these are the shares in y_i sent to P_i by honest players.
5. Finally, we describe how S can reconstruct the view of a player P_i taking part in the protocol up to a given point, such that this is consistent with the data generated by S so far. This can be thought of as a list of polynomials chosen by P_i in order to secret share various values and a list of shares received from other players. We describe how to do the reconstruction when the entire computation has already taken place. This is without loss of generality: if P_i is corrupted earlier, we just truncate the reconstruction procedure in the natural way.
   Input sharing We now know x_i, the input of P_i, and S has already specified random shares r_j for $P_j \in A$. Now choose a random polynomial $f_{x_i}()$ of degree at most t subject to $f_{x_i}(0) = x_i, f_{x_i}(j) = r_j$. List $f_{x_i}()$ as the polynomial used by P_i to share x_i. As for inputs shared by another player P_k, do as follows: if $P_k \in A$, a polynomial $f_{x_k}()$ for x_k has already been chosen, so just list $f_{x_k}(i)$ as the share received by P_i. If $P_k \notin A$, choose a random value as the share in x_k received by P_i.

Additions We may assume that we already listed a_i, b_i as P_i's shares in the summands, so we just list $a_i + b_i$ as his share in the sum.

Multiplications The following method will work for all multiplication operations except those leading to output values of already corrupted players, which are handled in the next item. We may assume that we already listed a_i, b_i as P_i's share in the factors, so we compute $\tilde{c}_i = a_i b_i$. We now reconstruct P_i's sharing of $\tilde{c}_i$ in exactly the same way as we reconstructed his sharing of x_i above. We also follow the method from input sharing to reconstruct the shares P_i receives of $\tilde{c}_j$'s of other players. Finally we can compute c_i, P_i's share in the product following the normal interpolation algorithm from the protocol.

Output generation As for y_i, the output of P_i, this is now known from F_{MPC}, and the shares in y_i held by corrupt players have been fixed earlier, so we follow the same method for simulating the shares P_i receives in output reconstruction stage that we already described above.

For an output value y_j of an already corupted player P_j, we have the problem that we already showed to the adversary what was supposed to be P_i's share s_i in y_j. Recall we assumed that any output y_j comes from a multiplication gate. So we have to specify the values involved in P_i's handling of this multiplication such that they will be consistent with s_i, but also consistent with the view of P_i we generated so far. This is done as follows: Let the multiplication in the circuit leading to y_j be $y_j = ab$, let a_i, b_i be the shares in a, b we already specified for P_i, and let $\tilde{c}_i = a_i b_i$. The multiplication protocol involves sharing $\tilde{c}_i$, and this has already taken place, in the sense that S has sent random values c_{ij} to players in A pretending they came from P_i. So we now choose a random polynomial $f_{\tilde{c}_i}()$ of degree at most t such that $f_{\tilde{c}_i}(0) = \tilde{c}_i, f_{\tilde{c}_i}(j) = c_{ij}, j \in A$, list this as the polynomial chosen by P_i for the multiplication. Finally, P_i receives in the real protocol shares c_{ji}, for every j, and is supposed to compute his share in the product as $s_i = \sum_j r_j c_{ji}$. Of the c_{ji}'s, we have already fixed the ones coming from corrput players, $\{c_{ji} | j \in A\}$ and $c_{ii} = f_{\tilde{c}_i}(i)$, altogether at most t values (P_i has just been corrupted, so there could be at most $t-1$ corruptions earlier). We now choose the remaining values c_{ji} as random independent values, subject only to $s_i = \sum_j r_j c_{ji}$. So actually, we select a random solution to a linear equation. By Lemma 1, there always exists a solution.

This concludes the description of S. To show that S works as required, we begin by fixing, in both the real and ideal world arbitrary values for the input and random tape of Z. This means that only source of randomness is the random choices of the players in the real world and those of S in the ideal world. We claim that, for every set of fixed values, Z sees exactly the same distribution when interacting with the ideal as with the real world, if we use S in the ideal world as described above. This of course implies that the protocol realizes F_{MPC} with perfect security since Z will then output 1 with the same probability in the two cases.

What Z can observe is the outputs generated by the players, plus it sees the view of the corrupt players as they execute the protocol. It will clearly be sufficient to prove the following

Claim: In every round j, for $j = 0$ up to the final round, the view of Z has the same distribution in ideal as in real world, given the fixed input and random tape for Z.

We argue this by induction on j. The basis $j = 0$ is trivial as nothing has happened in the protocol before the first round. So assume we have completed round j having produced some correctly distributed view for Z so far. We need to argue that given this, what S shows to Z is round $j + 1$ is corectly distributed.

Assume first that $j+1$ is not the final round. Then the only messages Z will see from honest players are sharings of values they hold. This is simulated perfectly: both in simulation and in real protocol, the adversary sees $\leq t$ independent random values in K as a result of every such sharing. Indeed, it is straightforward to show, using interpolation, that any vector of $\leq t$ shares of a random threshold-t Shamir sharing consists of independent random values. The only other source of information for Z is what it will see as a result of corrupting a player P_i in round $j+1$. Since round $j + 1$ is not the final round, the view reconstruction procedure will only execute the input sharing, addition and multiplication steps. By definition of the model, we start with the correct value of x_i, and also with correctly distributed shares of inputs of other players. It is then straightforward to see that the rest of the values in the view follow in a correct way from the starting values.

Then assume that round $j + 1$ is the final round. This means that Z will see results for all players. In the ideal world, these results are computed according to the given function by F_{MPC} from the inputs specified by Z. But in the real world, one can check by straightforward inspection of the protocol that all players will compute the same function of the inputs specified by Z. In addition, Z will see the corrupted players' view of the output reconstruction. Note that by induction hypothesis, the shares in a final result y_i held by corrupted players just before the output reconstruction stage has the same distribution in simulation as in real life. If y_i goes to an honest player, nothing further is revealed. If y_i goes to a corrupt player, observe that in the real protocol, the polynomial that determines y_i is random of degree at most t with the only constraint that it determines y_i and is consistent with the shares held by corrupt players - since by Lemma 1, at least one random polynomial chosen by an honest player is added into the polynomial determining y_i. It is now clear that the procedure used by S to construct a corresponding polynomial leads to the same distribution. Finally, one can check by inspection and arguments similar to the above, that also the output generation step of the procedure for reconstructing the view of a newly corrupted player P_i chooses data with the correct distribution, again conditioned on inputs and random tapes we fixed for Z and everything Z has seen earlier.

Optimality of 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 $\rho_i \in \{0,1\}^*$ 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
   $$\mathcal{T} = (m_{11}, m_{21}, \dots, m_{1t}, m_{2t}, r).$$
   For $i = 1, 2$, the *view* of P_i is
   $$\text{view}_i = (b_i, \rho_i, \mathcal{T}).$$

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.

We now argue that there is a strategy for a corrupted P_1 to always correctly determine the input bit b_2 of P_2, even if his input b_1 equals 0, thereby contradicting privacy.

Let P_1 have input bit $b_1 = 0$, and let the players execute the protocol, resulting in some particular transcript $\mathcal{T}$.

If P_2 has input bit $b_2 = 0$, he doesn't learn anything about b_1 by privacy. Hence, the transcript is also consistent with $b_1 = 1$.

But if $b_2 = 1$, then by correctness, the transcript cannot also be consistent with $b_1 = 1$: in that case its final message r is not equal to the AND of the input bits.

This gives rise to the following strategy for P_1.

1. P_1 sets $b_1 = 0$.
2. P_1 and P_2 execute the assumed protocol Π. This results in a fixed transcript $\mathcal{T}$.

3. P_1 verifies whether the transcript $\mathcal{T} = (m_{11}, m_{21}, \ldots, m_{1t}, m_{2t}, r)$ is also consistent with $b_1 = 1$.
   The consistency check can be performed as follows. P_1 checks whether there exists a random string σ_1 such that the same transcript $\mathcal{T}$ results, given that P_1 starts with $b_1 = 1$ and σ_1. P_1 can do this with an exhaustive search over all σ_1 and "simulating" P_2 by having him "send" the same messages as in the execution.
   More precisely, he first checks whether $(b_1 = 1, \sigma_1)$ leads to m_{11}. If so, he "receives" P_2's message m_{21}, and checks whether his own next message would equal m_{22}, and so forth, until perhaps exactly the same transcript $\mathcal{T}$ results.
   This process may take a long time, but that doesn't hurt since we have assumed an all powerful adversary.
4. If so, he decides that $b_2 = 0$. Otherwise he decides that $b_2 = 1$.

Similar arguments can be given if we relax the assumptions on privacy and correctness.

The assumptions about the players' computational resources and the communication channel are essential.

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.

In principle, this leads to secure two-party protocols for any function. For more information, see for instance [14]

3.2 The Active Case

In this section, we show how to modify the protocol secure against a passive adversary to make it secure also against active cheating. We will postulate in the following that we have a certain ideal functionality F_{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, F_{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 this so called Byzantine agreement 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. Although real broadcast protocols take several rounds to finish, we will assume here for simplicity that broadcast happens in one round.

Model for Homomorphic Commitments and Auxiliary Protocols We will assume that each player P_i can commit to a value $a \in K$. 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 F_{Com}. To commit, one simply sends a to F_{Com}, who will then keep it until P_i asks to have it reveled. Formally, we assume F_{Com} is equipped with the two commands described below (more will be defined later).

Some general remarks on the definiiton of F_{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 F_{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 F_{Com} is not used as intended, e.g., the honest players do not agree on the command to execute, F_{Com} will send all it's private data to all players and stop working. As with F_{MPC}, this is just a way to specify that no security is required if the functionality is not used as intended.

Notation: $CurrentRound$ always denotes the index of the current round. Some commands take some hmber of rounds to finish. This number for command Xxx is called $XxxDelay$.

Commit This command is executed if in some round player P_i sends $(commit, i, cid, a)$ and in addition all honest players send $(commit, i, cid, ?)$. In this case F_{Com} records the triple (i, cid, a). Here, cid is just an identifier, and a is the value committed to. We require that all honest players agree to the fact that a commitment schould be made because an implementation will require the active participation of all honest players. If P_i is corrupted and in a round before $CurrentRound + CommitDelay$ sends $(commit, i, cid, a')$, then (i, cid, a) is replaced by (i, cid, a'). A corrupt player may choose to have a be $\perp$ and not a value in K. This is taken to mean that the player refuses to commit.
In round $CurrentRound{+}CommitDelay$, if i, cid, a, $a \in K$ is stored, send $(commit, i, success)$ to all players. If $a = \perp$ send $(Commit, i, fail)$.

Open This command is executed if in some round all honest players send $(open, i, cid)$. In addition P_i should send x, where x may be $accept$ or $refuse$, and where $x = accept$ if P_i is honest. In this case F_{Com} looks up the triple (i, cid, a), and if $x = accept$, it sends in the next round $(open, cid, a)$ to all players, else it sends $(open, cid, fail)$.

As a minor variation, we also consider *private* opening of a commitment. This command is executed if in some round all honest players send $(open, i, cid, j)$. The only difference in its execution is that F_{Com} sends its output to player P_j only, rather than to all players. The effect is of course that only P_j learns the committed value.

The symbol $[\cdot]_i$ denotes a variable in which F_{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. It is clear from the above that all players know at any point which committed values have been defined. Of course, such a value is not known the players (except the committer), but nevertheless, they can ask F_{Com} to manipulate committed values, namely to add committed values, multiply them by public constants, or transfer a committed value to another player (the final operation is called a Commitment Transfer Protocol (CTP)):

CommitAdd This command is executed if all honest players send $(commitadd, cid1, cid2, cid3)$ (in the same round), and if triples $(i, cid1, a), (i, cid2, b)$ have been stored previously. Then F_{Com} stores the triple $(i, cid3, a + b)$.

ConstantMult This command is executed if all honest players send $(constantmult, cid1, cid2, u)$ (in the same round) where $u \in K$, and if a triple $(i, cid1, a)$ has been stored previously. Then F_{Com} stores the triple $(i, cid2, u \cdot a)$.

CTP This command is executed if all honest players send $(ctp, i, cid1, j, cid2)$ (in the same round), and if a triple $(i, cid1, a)$ has been stored earlier. If P_i is corrupt, he may send $(cid1, refuse)$ in some round before $CurrentRound + CTPDelay$. If this happens, then F_{Com} sends $(cid1, cid2, fail)$ to all players. Otherwise, F_{Com} stores $(j, cid2, a)$, sends a to P_j, and $(cid1, cid2, success)$ to everyone.

In our abbreviated language, writing $[a]_i + [b]_i = [a+b]_i$ means that the CommitAdd command is executed, creating $[a+b]_i$, and $u \cdot [a]_i = [ua]_i$ refers to executing the ConstantMult command. The CTP command can be thought of as creating $[a]_j$ from $[a]_i$. Note that we only require that the addition can be applied to two commitments made by *the same* player. Note also that there is no delay involved in the CommitAdd and ConstantMult commands, so an implementation cannot use any interaction between players.

A last basic command we assume is that F_{Com} can be asked to confirm that three commitments $[a]_i, [b]_i, [c]_i$ satisfy that $ab = c$. This is known as a Commitment Multiplication Protocol (CMP).

CMP This command is executed if all honest players send $(cmp, cid1, cid2, cid3)$ (in the same round), and if triples $(i, cid1, a), (i, cid2, b), (i, cid3, c)$ have been stored earlier. If P_i is corrupt, he may send $(cid1, cid2, cid3, refuse)$ in some round before $CurrentRound + CMPDelay$. If this happens, or if $ab \neq c$, then in round $CurrentRound + CMPDelay$, F_{Com} sends $(cid1, cid2, cid3, fail)$ to all players. Otherwise, F_{Com} sends $(cid1, cid2, cid3, success)$ to everyone.

The final command we need from F_{Com} is called a Commitment Sharing Protocol (CSP). It starts from $[a]_i$ and produces a set of commitments to shares of a: $[a_1]_1, ..., [a_n]_n$, where $(a_1, ..., a_n)$ is a correct threshold-t Shamir-sharing of a, generated by P_i. More formally:

CSP This command is executed if all honest players send $(csp, cid0, cid1, ..., cidn)$ (in the same round), and if a triple $(i, cid0, a)$ has been stored earlier. If P_i is honest, he should also send (coefficients of) a polynomial $f_a()$ of degree at most t, such that $f_a(0) = a$. If P_i is corrupt, he may send a correct polynomial in some round before number $CurrentRound + CSPDelay$, or he may send $(cid0, cid1, ..., cidn, refuse)$. When we reach round number $CurrentRound + CSPDelay$, if a correct polynomial has been received, store triples $(j, cidj, f_a(j))$ for $j = 1..n$, and send $(cid0, cid1, ..., cidn, success)$ to everyone, else send $cid0, cid1, ..., cidn, fail)$.

The CTP, CMP, and CSP commands are special: although they can be implemented "from scratch" like the other commands, they can also be implemented using the commands we already defined.

For *CTP*, we have the following.

1. Given a commitment $[a]_i$, P_i sends privately to P_j his total view of the protocol execution in which $[a]_i$ was created [1]. If this information is in any way inconsistent, P_j broadcasts a complaint, and we go to Step 4.
   Otherwise (if P_i was honest) P_j is a situation equivalent to having made $[a]_i$ himself.
2. P_j commits himself to a, resulting in $[a]_j$.
3. We use the ConstantMult command to get $[-a]_j$ and the CommitAdd command to get $[a]_i + [-a]_j$ Note that, assuming that the information P_j got in step 1 was correct, this makes sense since then the situation is equivalent to the case where P_j had been the committer when $[a]_i$ was created. Then $[a]_i + [-a]_j$ is opened, and we of course expect this to succeed with output 0. If this happens, the protocol ends. Otherwise do Step 4.
4. If we arrive at this step, it is clear that at least one of P_i, P_j are corrupt, so P_i must then open $[a]_i$ in public, and we either end with fail (if the opening fails) or a becomes public. We then continue with a default commitment to a assigned to P_j.

For *CMP*, we describe this protocol for a prover and a single verifier. To convince all the players, the protocol is simply repeated independently (for instance in parallel), each other player P_j taking his turn as the verifier. In the end, all verifying players broadcast their decision, and the prover is accepted by everyone if there are more than t accepting verifiers. This guarantees that at least one honest verifier has accepted the proof.

1. Inputs are commitments $[a]_i, [b]_i, [c]_i$ where P_i claims that $ab = c$. P_i chooses a random β and makes commitments $[\beta]_i, [\beta b]_i$.

[1] As is standard, the view of a protocol consists of all inputs and random coins used, plus all messages received during the protocol execution

2. P_j generates a random challenge $r \in K$, and sends it to P_j.
3. P_i opens the commitments $r[a]_i + [\beta]_i$ to reveal a value r_1. P_i opens the commitment $r_1[b]_i - [\beta b]_i - r[c]_i$ to reveal 0.
4. If any of these opening fail, P_j rejects the proof, else he accepts it.

It is easy to show that if P_i remains honest, then all values opened are random (or fixed to 0) and so reveal no extra information to the adversary. If P_i is corrupt, then it is also straightforward to show that if, after committing in step 2, P_i can answer correctly two different challenges, then $ab = c$. Thus the error probability is at most $1/|K|$.

Finally, for *CSP*, assuming $[a]_i$ has been defined, P_i chooses a random polynomial f_a of degree at most t such that $f_a(0) = a$. He makes commitments to the coefficients of f: $[v_1]_i, ..., [v_t]_i$ (the degree-0 coefficient of f_a is a and has already been comitted). Let $(a_1, ..., a_n) = (f_a(1), ..., f_a(n))$ be the shares resulting from sharing a using the polynomial f_a. Then the a_i's are a linear function of the committed values, and commitments to the shares $([a_1]_i, ..., [a_n]_i)$ can be created by calling the CommitAdd and ContstantMult commands, e.g.,

$$[a_j]_i = [a]_i + [v_1]_i \cdot j + [v_2]_i \cdot j^2 + ... + [v_t]_i \cdot j^t$$

Finally, we call CTP to create $[a_j]_j$ from $[a_j]_i$, for $j = 1, \ldots, n$.

Committing to a and then performing CSP is equivalent to what is known as *verifiably secret sharing a* (VSS): the value a is uniquely defined when the CSP is executed, and it is guaranteed that the honest players can reconstruct it: the commitments to shares prevent corrupted players from contributing false shares when the secret is reconstructed. All we need is that at least $t+1$ good shares are in fact revealed.

An MPC Protocol for Active Adversaries The protocol starts by asking each player to VSS each of his input values as described above: he commits to the value and then performs CSP. If this fails, the player is disqualified and we take default values for his inputs.

We then work our way through the given arithmetic circuit, maintaining as invariant that all inputs and intermediate results computed so far are VSS'ed as described above, i.e. each such value a is shared by committed shares $[a_1]_1, ..., [a_n]_n$ where *all* these shares are correct, also those held by corrupted players. Moreover, if a depends on an input from an honest player, this must be a *random* set of shares determining a. From the start, only the input values are classified as having been computed.

Once an output value y has been computed, it can be reconstructed in the obvious way by opening the commitments to the shares $y_1, ..., y_n$. This will succeed, as the honest players will contribute enough correct shares, and a corrupted player can only choose between contributing a correct share, or have the opening fail.

It is therefore sufficient to show how addition and multiplication gates are handled. Assume the input values to a gate are a and b, determined by committed shares $[a_1]_1, ..., [a_n]_n$ and $[b_1]_1, ..., [b_n]_n$.

Addition For $i = 1..n$, P_i computes $a_i + b_i$ and CommitAdd is called to create $[a_i + b_i]_i$. By linearity of the secret sharing, $[a_1 + b_1]_1, ..., [a_n + b_n]_n$ determine $a + b$ as required by the invariant.

Multiplication For $i = 1..n$, P_i computes $a_i \cdot b_i = \tilde{c}_i$, commits to it, and performs CMP on inputs $[a_i]_i, [b_i]_i, [\tilde{c}_i]_i$.

Resharing step: P_i performs CSP on $[\tilde{c}_i]_i$, resulting in commitments $[c_{i1}]_1, ..., [c_{in}]_n$.

We describe below how to recover if any of this fails.

Recombination step: For $j = 1..n$, player P_j computes $c_j = \sum_{i=1}^{n} r_i c_{ij}$, where $(r_1, ..., r_n)$ is the recombination vector. Also all players compute (non-interactively) $[c_j]_j = \sum_{i=1}^{n} r_i [c_{ij}]_j = [\sum_{i=1}^{n} r_i c_{ij}]_j$. By definition of the recombination vector and linearity of commitments, the commitments $[c_1]_1, ..., [c_n]_n$ determine $c = ab$ as required by the invariant.

It remains to be described what should be done if a player P_i fails in the multiplication and resharing step above. 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 players that actually completed the multiplication step correctly.

3.3 Realization of F_{Com}: Information Theoretic Scenario

We assume here that we are in the i.t. scenario and that $t < n/3$

We first look at the commitment scheme: 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 false shares, and can then easily “open” a commitment in several ways, as detailed in the exercise below.

Exercise 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 $f()$ of degree at most t where $t < n/3$ is maximal number of corrupted players. At some later time, P is supposed to reveal the polynomial $f()$ he used, and each P_i reveals a_i, The polynomial is accepted if values of at most t players disagree with $f()$ (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 $f(), f'()$ of degree at most t and values $\tilde{a}_i$ for P to send, such that P can later reveal and have accepted both $f()$ and $f'()$.
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 corupted 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 $f_1()$ 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 $f_2()$, in which case C_2 is the corrupted set. Note that since 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 $f_1() = f_2()$.
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 of degree at most t, without harming privacy of course.

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 we don't know which t of them are.

Concretely, define the shares

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

and let $\mathbf{e} \in K^n$ 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

$$f = g.$$

Assuming that we have established the mechanism for ensuring correct sharings as discussed above, there is a simple open protocol for this commitment scheme.

Open Protocol (*Version I*):

1. Each player P_i simply reveals his share s_i to all other players P_j.
2. Each of them individually recovers the committed value a that is uniquely defined by them. This can be done by exhaustive search, or by the efficient method described below.

Note that broadcast is not required here.

We now show one particular method to *efficiently* recover the committed value. In fact, we'll recover the entire polynomial f. [2]

Write

$$\tilde{\mathbf{s}} = (\tilde{s}_1, \ldots, \tilde{s}_n).$$

The method "interpolates" the points $(i, \tilde{s}_i)$ by a bi-variate polynomial Q of a special form (which from a computational view comes down to solving a system of linear equations), and "extracts" the polynomial f from Q in a very simple way.

Concretely, let $Q(X,Y) \in K[X,Y]$, $Q \neq 0$ be any polynomial such that, for $i = 1 \ldots n$,

$$Q(i, \tilde{s}_i) = 0,$$

and such that

$$Q(X,Y) = f_0(X) - f_1(X) \cdot Y,$$

for some $f_0(X) \in K[X]$ of degree at most $2t$ and some $f_1(X) \in K[X]$ of degree at most t.

Then we have that

$$f(X) = \frac{f_0(X)}{f_1(X)}.$$

Clearly, the conditions on Q can be described in terms of a linear system of equations with Q's coefficients as the unknowns.

To recover f, we simply select an arbitrary solution to this system, which is a computationally efficient task, define the polynomial Q by the coefficients thus found, extract f_0, f_1 from it by

[2] What we show is actually the Berlekamp-Welch decoder for Reed-Solomon error-correcting codes.

appropriately ordering its terms, and finally perform the division of the two, which is again a computationally efficient task.

We now show correctness of this algorithm. First, we verify that this system is solvable. For this purpose, we may assume that we are given the polynomial f and the positions A in which an error is made (thus, A is a subset of the corrupted players). Define

$$k(X) = \prod_{i \in A} (X - i).$$

Note that its degree is at most t. Then

$$Q(X, Y) = k(X) \cdot f(X) - k(X) \cdot Y$$

satisfies the requirements for Q, as is verified by simple substitution.

It is now only left to show that whenever some polynomial Q satisfies these requirements, then indeed $f(X) = f_0(X)/f_1(X)$.

To this end, define

$$Q'(X) = Q(X, f(X)) \in K[X],$$

and note that its degree is at most $2t$.

If $i \notin A$, then $(i, s_i) = i, \tilde{s}_i)$. Thus, for such i,

$$Q'(i) = Q(i, f(i)) = Q(i, s_i) = Q(i, \tilde{s}_i) = 0.$$

Since $t < n/3$,

$$n - |A| \geq n - t > 2t.$$

We conclude that the number of zeroes of $Q(X)$ exceeds its degree, and that it must be the zero polynomial. Therefore,

$$f_0 - f_1 \cdot f = 0,$$

which establishes the claim (note that $f_1 \neq 0$ since $Q \neq 0$).

Below we describe an alternative open protocol that is less efficient in that it uses the broadcast primitive. The advantage, however, is that it avoids the above "error correction algorithm" which depends so much on the fact that Shamir's scheme is the underlying secret sharing scheme. In fact, it can be easily adapted to a much wider class of commitment schemes, namely those based on general linear secret sharing schemes.

Open Protocol (*Version II*):

1. D broadcasts the polynomial f.
   Furthermore, each player P_i broadcasts his share.

2. Each player decides for himself by the following rule.
   If all, except for possibly $\leq t$, shares are consistent with the broadcast polynomial and its degree is indeed at most t, the opening is accepted. The opened value is $a = f(0)$.
   Else, the opening is rejected.

   This works for essentially the same reasons as used before.

Note that both open protocols allow for *private opening* of a commitment to a designated player P_j. This means that only P_j learns the committed value a. This is achieved by simply requiring that all information is privately sent to P_j, and it works because of the privacy of the commit protocol (as shown later) and because the open protocol only depends on local decisions made by the players.

We now describe the *commit protocol*. Let $F(X,Y) \in K[X,Y]$ be a symmetric polynomial of degree at most t in both variables, i.e.,

$$F(X,Y) = \sum_{k,l=0}^{t} c_{kl} X^k Y^l,$$

and

$$F(X,Y) = F(Y,X),$$

which is of course equivalent to $c_{kl} = c_{lk}$ for all $1 \leq k,l \leq t$.

We define

$$f(X) = F(X,0),$$

$$f(0) = a,$$

and, for $i = 1 \ldots n$,

$$f(i) = s_i.$$

Note that

$$\deg f \leq t.$$

We call f the *real sharing polynomial*, a the *committed value*, and s_i a *real share*.

We also define, for $i,j = 1 \ldots n$,

$$f_i(X) = F(X,i),$$

and

$$f_i(j) = s_{ij}.$$

Note that

$$\deg f_i \leq t.$$

We call f_i a *verification polynomial*, and s_{ij} a *verification share*.

By symmetry we have

$$s_i = f(i) = F(i, 0) = F(0, i) = f_i(0).$$
$$s_{ij} = f_i(j) = F(j, i) = F(i, j) = f_j(i) = s_{ji}.$$

Commit Protocol:

1. To commit to $a \in K$, D chooses a random, symmetric bivariate polynomial $F(X, Y)$ of degree at most t in both variables, such that
$$F(0, 0) = a.$$
D sends the verification polynomial f_i (i.e., its $t + 1$ coefficients) privately to P_i for each i.
P_i sets $s_i = f_i(0)$, his real share.
2. For all $i > j$, P_i sends the verification share s_{ij} privately to P_j.
3. It must hold that
$$s_{ij} = s_{ji}.$$
If P_j finds that
$$s_{ij} \neq s_{ji},$$
he broadcasts a *complaint*.
In response to each such complaint (if any), D must broadcast the correct value s_{ij}.
If P_j finds that the broadcast value differs from s_{ji}, he knows that D is *corrupt* and broadcasts an *accusation* against D, and *halts*.
A similar rule applies to P_i if he finds that the broadcast value differs from s_{ij}.
4. For all players P_j who accused D in the previous step (if any), D must now broadcast the correct verification polynomial f_j.
5. Each player P_i that is "still in the game" verifies each of the broadcast verification polynomials f_j (if any) against his own verification polynomial f_i, by checking that, for each of those, $s_{ij} = s_{ji}$.
If there is any inequality, P_i knows that D is corrupt, and broadcasts an *accusation* against D and halts.
6. If there are $\leq t$ accusations in total, D is *accepted*.
In this case, each player P_j who accused D in Step 5, replaces the verification polynomial received in Step 1 by the polynomial f_i broadcast in Step 4, and defines $s_j = f_j(0)$ as his real share.
All others stick to their real shares as defined from the verification polynomials received in in Step 1.
7. If there are $> t$ accusations in total, the dealer is deemed *corrupt*.

We sketch a proof that this commitment scheme works. For simplicity we assume that the adversary is static.

Honest D Case: It is immediate, by inspection of the protocol, that honest players never accuse an honest D. Therefore, there are at most t accusations and the commit protocol is always *accepted*.

In particular, each honest player P_i accepts $s_i = f(i)$ as defined in step 1 as his real share. This means that in the open protocol $a = f(0)$ is accepted as the committed value.

For *privacy*, i.e., the adversary does not learn the committed value a, note first that steps 2–4 of the commit protocol are designed such that the adversary learns nothing he was not already told in step 1.

Indeed, the only information that becomes available to the adversary afterwards, is what is broadcast by the dealer. This is either a verification share s_{ij} where P_i is corrupt or P_j is corrupt, or a verification polynomial f_i of a corrupt player P_i. All of this is already implied by the information the adversary received in step 1.

Therefore, it is sufficient to argue that the information in step 1 does not reveal a to the adversary.

Denote by A the set of corrupted players, with $|A| \leq t$. It is sufficient to show that for each guess a' at a, there is the same number of appropriate polynomials $F'(X, Y)$ consistent with the information received by the adversary in step 1.

By appropriate we mean that $F'(X, Y)$ should be symmetric, of degree at most t in both variables, and for all $i \in A$ we must have $f'_i(X) = f_i(X)$.

Consider the polynomial

$$h(X) = \prod_{i \in A} (\frac{-1}{i} \cdot X + 1) \in K[X]$$

Note that its degree is at most t, $h(0) = 1$ and $h(i) = 0$ for all $i \in A$.

Now define

$$Z(X, Y) = h(X) \cdot h(Y) \in K[X, Y].$$

Note that $Z(X, Y)$ is symmetric and of degree at most t in both variables, and that it has the further property that $Z(0, 0) = 1$ and $z_i(X) = Z(X, i) = 0$ for all $i \in A$.

If D in reality used the polynomial $F(X, Y)$, then for all possible a', the information held by the adversary is clearly also consistent with the polynomial

$$F'(X, Y) = F(X, Y) + (a' - a) \cdot Z(X, Y).$$

Indeed, it is symmetric, of degree at most t in both variables, and, for $i \in A$,

$$f'_i(X) = f_i(X) + (a - a') \cdot z_i(X) = f_i(X),$$

and

$$f'(0) = F'(0, 0) = F(0, 0) + (a' - a) \cdot Z(0, 0) = a + (a - a') = a'.$$

This construction immediately gives a one-to-one correspondence between the consistent polynomials for committed value a and those for a'. Thus all values are equally likely from the point of view of the adversary.

Corrupt D Case: Let B denote the set of honest players, and let s_i, $i \in B$, be the real shares as defined *at the end* of the protocol. In other words, $s_i = f_i(0)$, where f_i is the verification polynomial as defined at the end of the protocol.

We have to show that if the protocol was accepted, then there exists a polynomial $g(X) \in K[X]$ such that its degree is at most t and $g(i) = s_i$ for all $i \in B$.

It is important to realize that we have to argue this from the acceptance assumption alone; we cannot make any apriori assumptions on how a corrupt D computes the various pieces of information.

Write C for the set of honest players that did not accuse D at any point. Note that

$$|C| \geq n - \#\text{Accusations} - \#\text{Corruptions} \geq n - 2t > t.$$

Furthermore, there is consistency between the players in C on the one hand, and the players in B on the other hand. Namely, for all $P_i \in C$, $P_j \in B$, it follows from the acceptance assumption that

$$f_i(j) = f_j(i),$$

where the verification polynomials are defined as at end of the protocol.

Indeed, let $P_i \in C$ be arbitrary and let $P_j \in B$ be an arbitrary honest player who did not accuse the dealer before step 5. Then their verification polynomials f_i, f_j as defined at the end are the ones given in step 1. If it were so that $f_i(j) \neq f_j(i)$, then at least one of the two would have accused D in step 3.

On the other hand, if P_j is a player who accused D in step 3, and if the *broadcast* polynomial f_j is not consistent with P_i's verification polynomial, P_i would have accused D in step 5.

Let $r_i, i \in C$, be the coefficients of the recombination vector for C. Define

$$g(X) = \sum_{i \in C} r_i \cdot f_i(X).$$

Note that its degree is at most t.

We now only have to verify that for all $j \in B$, we have $s_j = g(j)$.

Indeed, we have that

$$g(j) = \sum_{i \in C} r_i \cdot f_i(j) = \sum_{i \in C} r_i \cdot f_j(i) = f_j(0) = s_j.$$

The first equality follows by definition of $g(X)$, the second by the observed consistency, the third by Lagrange and the fact that $|C| > t$ and that the degree of g is at most t, and the final equality follows by definition of the real shares at the end of the protocol.

This concludes the analysis of the commit protocol. Note that both the commit and the open protocol consume a constant number of rounds of communication.

So this commitment scheme works with no probability of error, if $t < n/3$. If instead we have $t < n/2$, the commit protocol can be easily adapted so that the proof that all honest players have consistent shares still goes through; basically, the process of accusations with subsequent broadcast of verification polynomials as in step 5 will be repeated until there are no new accusations (hence the commit protocol may no longer be constant round).

However, the proof that the opening always succeeds fails. The problem is that since honest players cannot prove that the shares they claim to have received are genuine, we have to accept up to $n/2$ complaints in the opening phase, and this will allow a corrupt D to open a commitment any way he wants. Clearly, if D could digitally sign his shares, then we would not have to accept any complaints and we would be in business again. Of course, digital signatures require computational assumptions, which we do not want to make in this scenario. However, there are ways to make unconditionally secure authentication schemes which ensure the same functionality (except with negligibly small error probability, see [12]).

Finally, this commitment scheme generalizes nicely to a scenario in which the underlying secret sharing scheme is not Shamir's but in fact a general linear secret sharing scheme.

We now show a *Commitment Multiplication Protocol (CMP)* that works without error if $t < n/3$.

<u>CMP:</u>

1. Inputs are commitments $[a]_i, [b]_i, [c]_i$ where P_i claims that $ab = c$.
   First P_i performs CSP on commitments $[a]_i, [b]_i$ to get committed shares $[a_1]_1, ..., [a_n]_n$ and $[b_1]_1, ..., [b_n]_n$.
2. P_i computes the polynomial $g_c = f_a \cdot f_b$, where f_a (f_b) is the polynomial used for sharing a (b) in the previous step.
   He commits to the coefficients of g_c.
   Note that there is no need to commit to the degree 0 coefficient, since this should be c, which is already committed to.
3. Define $c_i = g_c(i)$.
   ¿From the commitments made so far and $[c]_i$, the players can compute (by linear operations) commitments $[c_1]_i, ..., [c_n]_i$, where of course P_i claims that $a_j b_j = c_j$, for $1 \leq j \leq n$.
4. For $j = 1, \ldots, n$, commitment $[c_j]_i$ is opened privately to P_j, i.e. the shares needed to open it are sent to P_j (instead of being broadcast).
5. If the value revealed this way is not $a_j b_j$, P_j broadcasts a complaint and opens (his own) commitments $[a_j]_j, [b_j]_j$. In response, P_i must open $[c_j]_i$ and is disqualified if $a_j b_j \neq c_j$.

We argue the correctness of this protocol.

Clearly, no matter how a possible adversary behaves, there is a polynomial g_c of degree at most $2t$ such that $c = g_c(0)$ and each $c_j = g_c(j)$.

Consider the polynomial $f_a \cdot f_b$, which is of degree at most $2t$ as well.

Suppose that $c \neq ab$. Thus $g_c \neq f_a \cdot f_b$. By Lagrange Interpolation, it follows that for at most $2t$ values of j we have $g_c(j) = f_a(j) \cdot f_b(j)$, or equivalently, $c_j = a_j b_j$.

Thus at least $n - 2t$ players P_j have $c_j \neq a_j b_j$, which is at least one more than the maximum number t of corrupted players (since $t < n/3$).

Therefore, at least one honest player will complain, and the prover is exposed in the last step of the protocol.

CSP:

Although CSP can be bootstrapped in a generic fashion from homomorphic commitment and CTP as explained earlier, we now argue that in the information theoretic scenario with $t < n/3$, there is a much simpler and more efficient solution: a slightly more refined analysis shows that the commit protocol we presently earlier is essentially already a CSP!

Consider an execution of the commit protocol, assuming D is honest. It is immediate that, for each player P_i (honest or corrupt!), there exists a commitment $[s_i]_i$ to his share s_i in the value a that D is committed to via $[a]_D$. The polynomial underlying $[s_i]_i$ is of course the verification polynomial $f_i(X)$ and each honest player P_j obtains $f_i(j)$ as $f_j(i)$.

Therefore, if each honest player holds on to his verification polynomial for later use, *each player* P_i is committed to his share s_i in the value a via $[s_i]_i$.

Apart from handling the corrupt D case, the only thing to be settled is that, by definition, CSP takes as input a commitment $[a]_D$. This, however, can easily be "imported" into the protocol: D knows the polynomial f that underlies $[a]_D$, and the players know their shares in a. We simply modify the commit protocol by requiring that D chooses this particular f as the real sharing polynomial. Also, upon receiving his verification polynomial in the first step of the commit protocol, each player checks that his real share is equal to the share in a he already had as part of the input. If this is not so, he broadcasts an accusation. If there are at most t accusations, the commit protocol continues as before. Else, it is aborted, and D is deemed corrupt. It is easy to see that this works; if D is honest it clearly does, and if D is corrupt and uses a different real sharing polynomial, then, by similar arguments as used before, there are more than t accusations from honest players.

As for the case of a possibly corrupt D, the discussion above shows that it is sufficient to prove the following. If the commit protocol is accepted, then there exists a unique symmetric bi-variate polynomial $G(X,Y) \in K[X,Y]$, with the degrees in X as well as Y at most t, such that for an honest player P_i, $f_i(X) = G(X,i)$ is the verification polynomial held by him *at the end of the protocol.* In other words, if the protocol is accepted, then, regardless whether the dealer is honest or not, the information held by the honest players is "consistent with an honest D."

We have to justify the claim above from the acceptance assumption only; we cannot make any a priori assumptions about how a possibly corrupt D computes the various pieces of information.

Let C denote the subset of the honest players B that do not accuse D at any point. As we have seen, acceptance implies $|C| \geq t+1$ as well as "consistency," i.e., for all $i \in C$ and for all $j \in B$, $f_i(j) = f_j(i)$. Without loss of generality, we now assume that $|C| = t+1$.

Let $\delta_i(X) \in K[X]$ denote the polynomial of degree t such that for all $i, j \in C$,

$$\delta_i(j) = 1 \text{ if } i = j \text{ and } \delta_i(j) = 0 \text{ if } i \neq j,$$

or, equivalently,

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

Recall that the Lagrange Interpolation Theorem may be phrased as follows. If $h(X) \in K[X]$ has degree at most t, then $h(X) = \sum_{i \in C} h(i)\delta_i(X)$.

Consider the polynomial

$$G(X, Y) = \sum_{i \in C} f_i(X)\delta_i(Y) \in K[X, Y].$$

This is clearly the unique polynomial in $K[X, Y]$ whose degree in Y is at most t and for which $G(X, i) = f_i(X)$ for all $i \in C$. This follows from Lagrange Interpolation applied over $K(X)$, i.e, the fraction field of $K[X]$, rather than over K. Note also that its degree in X is at most t.

We now verify that $G(X, Y)$ is symmetric:

$$G(X, Y) = \sum_{i \in C} f_i(X)\delta_i(Y) = \sum_{i \in C} \left(\sum_{j \in C} f_i(j)\delta_j(X) \right) \delta_i(Y) =$$

$$\sum_{i \in C} f_i(i)\delta_i(X)\delta_i(Y) + \sum_{i,j \in C, i \neq j} f_i(j)(\delta_i(X)\delta_j(Y) + \delta_j(X)\delta_i(Y)),$$

where the last equality follows from consistency.

Finally, for all $j \in B$, we have that

$$f_j(X) = \sum_{i \in C} f_j(i)\delta_i(X) =$$

$$\sum_{i \in C} f_i(j)\delta_i(X) =$$

$$\sum_{i \in C} G(j, i)\delta_i(X) =$$

$$\sum_{i \in C} G(i, j)\delta_i(X) = G(X, j),$$

as desired.

3.4 Realization of F_{Com}: Cryptographic Scenario

As commitments in this scenario, we can use, e.g., the commitments from [11] 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 $GF(q)$. A simple example is if we have primes p, q, where $q|p-1$ and g, h, y are elements in Z_p^* of order q chosen as public key by player P_i. Then $[a]_i$ is of form $(g^r, y^a h^r)$, i.e. a Diffie-Hellman (El Gamal) encryption of y^a under public key g, h. In [11], 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 w.r.t. different public keys. It is trivial to derive a CTP from this: P_i privately reveals the contents and random bits for $[a]_i$ to P_j. If this is not correct, P_j complains, otherwise he makes $[a]_j$ and proves it contains the same value as $[a]_i$. Finally, [11] also show a CMP protocol.

We note that for technical reasons, in order to be able to do a simulation-based proof of security of the overall MPC protocol using these commitments, each player must prove knowledge of his secret key initially, as well as prove that he knows the contents of each commitment he makes.

These commitments, together with the VSS and main MPC protocols described earlier, lead to quite efficient MPC protocols. However, one can also build protocols in this scenario based on a more general assumption, at the expense of some efficiency, namely if any one-way trapdoor permutation exists. This assumption is sufficient to do both secure public-key encryption, and secure commitment schemes.

These will of course not be homomorphic in general, but using the fact that any NP-statement can be proved in zero-knowledge using any commitment scheme, we can still use basically the same framework as before. The idea is to start with the passively secure protocol and have players commit to all their private data. Giving a share in private to a player is implemented by encrypting it under the receivers public key. And whenever P_i sends a message, he proves in zero-knowledge that it was correctly computed. This means that a corrupt player must either play correctly or be excluded. Since we have $t < n/2$ in this scenario, there are always enough honest players to complete the computation.

3.5 Formal Proofs for Active Adversaries

When giving formal proofs of security of the protocols sketched above, we need to produce appropriate simulators, as we said in the introduction. The main new ingredient in the actively secure protocol compared to the passive solution is the commitment scheme. One finds that in all cases where we are in the i.t. scenario, we can make simulators that can handle commitments even if the adversary is adaptive (and infinitely powerful). This is based on the following two observations:

- If player P_i is honest and commits to for instance his input x_i, then since the commitment is based on secret sharing, this only results in the adversary seeing a set of random shares. This

is of course easy to simulate even if x_i is not known. 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.
– 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 (which may or may not be equal to x_i). So then the simulator can in the ideal process send x'_i on behalf of P_i.

However, the situation is different in the cryptographic model. Here, the protocols we described can only be proved secure against static adversaries. The reason is as follows: suppose player P_i has not yet been corrupted. Then in the crypto model, 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 random contents 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 can't open its simulated encryptions the right way. Using a technique known as non-committing encryption, however, there are ways around this problem, at the expense of efficiency (see [9]).

References

1. D. Beaver: *Foundations of Secure Interactive Computing*, Proc. of Crypto 91.
2. 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.
3. J. Benaloh, J. Leichter: *Generalized Secret Sharing and Monotone Functions*, Proc. of Crypto '88, Springer Verlag LNCS series, pp. 25–35.
4. M. Ben-Or, S. Goldwasser, A. Wigderson: *Completeness theorems for Non-Cryptographic Fault-Tolerant Distributed Computation*, Proc. ACM STOC '88, pp. 1–10.
5. E. F. Brickell: *Some Ideal Secret Sharing Schemes*, J. Combin. Maths. & Combin. Comp. 9 (1989), pp. 105–113.
6. 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).
7. R.Canetti, U.Fiege, O.Goldreich and M.Naor: *Adaptively Secure Computation*, Proceedings of STOC 1996.
8. R.Canetti: *Universally Composable Security*, The Eprint archive, www.iacr.org.
9. R. Canetti, U. Feige, O. Goldreich, M. Naor: *Adaptively Secure Multi-Party Computation*, Proc. ACM STOC '96, pp. 639–648.
10. D. Chaum, C. Crépeau, I. Damgård: *Multi-Party Unconditionally Secure Protocols*, Proc. of ACM STOC '88, pp. 11–19.
11. 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.
12. Cramer, Damgård, Dziembowski, Hirt and Rabin: *Efficient Multiparty Computations With Dishonest Minority*, Proceedings of EuroCrypt 99, Springer Verlag LNCS series.

13. Cramer, Damgård and Maurer: *Multiparty Computations from Any Linear Secret Sharing Scheme*. In: Proc. EUROCRYPT '00.
14. Ronald Cramer. Introduction to Secure Computation. Latest version: January 2001. Available from `http://www.brics.dk/~cramer`
15. C. Crepeau, J.vd.Graaf and A. Tapp: *Committed Oblivious Transfer and Private Multiparty Computation*, proc. of Crypto 95, Springer Verlag LNCS series.
16. D. Dolev, C. Dwork, and M. Naor, *Non-malleable cryptography*, Proc. ACM STOC '91, pp. 542–552.
17. M. Fitzi, U. Maurer: *Efficient Byzantine agreement secure against general adversaries*, Proc. Distributed Computing DISC '98.
18. 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.
19. 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.
20. M. Hirt, U. Maurer: *Complete Characterization of Adversaries Tolerable in General Multiparty Computations*, Proc. ACM PODC'97, pp. 25–34.
21. M. Karchmer, A. Wigderson: *On Span Programs*, Proc. of Structure in Complexity, 1993.
22. 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.
23. S. Micali and P. Rogaway:*Secure Computation*, Manuscript, Preliminary version in Proceedings of Crypto 91.
24. T. P. Pedersen: *Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing*, Proc. CRYPTO '91, Springer Verlag LNCS, vol. 576, pp. 129–140.
25. P. Pudlák, J. Sgall: *Algebraic Models of Computation and Interpolation for Algebraic Proof Systems* Proc. Feasible Arithmetic and Proof Complexity, Springer Verlag LNCS series.
26. T. Rabin: *Robust Sharing of Secrets when the Dealer is Honest or Cheating*, J. ACM, 41(6):1089-1109, November 1994.
27. T. Rabin, M. Ben-Or: *Verifiable Secret Sharing and Multiparty Protocols with Honest majority*, Proc. ACM STOC '89, pp. 73–85.
28. A. Shamir: *How to Share a Secret*, Communications of the ACM 22 (1979) 612–613.
29. M. van Dijk: *Secret Key Sharing and Secret Key Generation*, Ph.D. Thesis, Eindhoven University of Technology, 1997.

A Formal Details of the General Security Model for Protocols

In this section we propose a notion of universally composable security of synchronous protocols.

A.1 The Real-Life Execution

A real-life protocol π consists of n parties $P_1, \ldots, P_n$, all PPT interactive Turing machines (ITMs). The execution of a protocol takes place in the presence of an environment $\mathcal{Z}$, also a PPT ITM, which supplies inputs to and receives outputs from the parties. Following Definition 4 from [?] $\mathcal{Z}$ also models the adversary of the protocol, and so schedules the activation of the parties, corrupts parties adaptively and controls corrupted parties. We assume that the parties are connected by open authenticated channels.

To simplify notation we assume that in each round r each party P_i sends a message $m_{i,j,r}$ to each party P_j, including itself. The message $m_{i,i,r}$ can be thought of as the state of P_i after round r. To further simplify the notation we assume that in each round $\mathcal{Z}$ inputs a value $x_{i,r}$ to

P_i and receives and output $y_{i,r}$. A protocol not following this convention can easily be patched by introducing some dummy value $\epsilon = \texttt{not at value}$. Using this convention we can write the r'th activation of P_i as $(m_{i,1,r}, \dots, m_{i,n,r}, y_{i,r}) = P_i(k, m_{1,i,r-1}, \dots, m_{n,i,r-1}, x_{i,r}; r_i)$, where k is the security parameter and r_i is the random bits used by P_i. We assume that the parties cannot reliably erase their state. To model this we give r_i to $\mathcal{Z}$ when P_i is corrupted. Since $\mathcal{Z}$ knows all the inputs of P_i this will allow $\mathcal{Z}$ to reconstruct the entire execution history of P_i. In detail the real-life execution proceeds as follows.

Init: The input to an execution is the security parameter k, the random bits $r_1, \dots, r_n \in \{0,1\}^*$ used by the parties and an auxiliary input $z \in \{0,1\}^*$ for $\mathcal{Z}$.
Initialize the round counter $r = 0$ and initialize the set of corrupted parties $C = \emptyset$. In the following let $H = \{1, \dots, n\} \setminus C$.
Let $m_{i,j,0} = \epsilon$ for $i, j \in [n]$.
Input k and z to $\mathcal{Z}$ and activate $\mathcal{Z}$.

Environment activation: When $\mathcal{Z}$ is activated it outputs one of the following commands: $(\texttt{activate}\ i, x_{i,r}, \{m_{j,i,r-1}\}_{j \in C})$ for $i \in H$ or $(\texttt{corrupt}\ i)$ for $i \in H$ or (`end round`) or $(\texttt{guess}\ b)$ for $b \in \{0,1\}$.
We require that no two $(\texttt{activate}\ i, \dots)$ commands for the same i are issued without being separated by an (`end round`) command and we require that between two (`end round`) commands an $(\texttt{activate}\ i, \dots)$ command was issued for $i \in H$, where H denotes the value of H when the second of the (`end round`) commands were issued.
When a $(\texttt{guess}\ b)$ command is given the execution stops. The other commands are handled as described below. After the command is handled the environment is activated again.

Party activation: Values $\{m_{j,i,r-1}\}_{j \in H}$ were defined in the previous round; Add these to $\{m_{j,i,r-1}\}_{j \in C}$ from the environment and compute

$$(m_{i,1,r}, \dots, m_{i,n,r}, y_{i,r}) = P_i(k, m_{1,i,r-1}, \dots, m_{n,i,r-1}, x_{i,r}; r_i) .$$

Then give $\{m_{i,j,r}\}_{j \in [n] \setminus \{i\}}$ to $\mathcal{Z}$.

Corrupt: Give r_i to $\mathcal{Z}$. Set $C = C \cup \{i\}$.

End round: Give the values $\{y_{i,r}\}_{i \in H}$ defined in **Party activation** to $\mathcal{Z}$ and set $r = r + 1$.

The result of the execution is the bit b output by $\mathcal{Z}$. We denote this bit by $\texttt{REAL}_{\pi,\mathcal{Z}}(k, r_1, \dots, r_n, z)$. This defines a random variable $\texttt{REAL}_{\pi,\mathcal{Z}}(k, z)$, where we take the r_i to be uniformly random, and in turn defines a Boolean distribution ensemble $\texttt{REAL}_{\pi,\mathcal{Z}} = \{\texttt{REAL}_{\pi,\mathcal{Z}}(k, z)\}_{k \in \mathbf{N}, z \in \{0,1\}^*}$.

A.2 The Ideal Process

To define the security of a protocol an ideal functionality $\mathcal{F}$ is specified. The ideal functionality is a PPT ITM with n input tapes and n output tapes which we think of as being connected to n parties. The ideal functionality defines the desired input-output behaviour of the protocol and defines the desired secrecy by keeping the inputs secret. In the execution of an ideal functionality in an environment $\mathcal{Z}$, the inputs to P_i from $\mathcal{Z}$ is simply handed to $\mathcal{F}$ and the outputs from $\mathcal{F}$ to P_i is handed to $\mathcal{Z}$. To be able to specify protocols which leak some information about the inputs of the parties $\mathcal{F}$ has a special tape. To model protocols which are allowed to leak some specified information about the inputs of the parties the functionality simply outputs this information on the special tape. An example could be the following functionality modelling secure communication: It is connected to two parties S and R. If R inputs some value $m \in \{0,1\}^*$, then $|m|$ is output on the special tape and m is output to R.

The ideal functionality also has the special input tape on which it receives two kinds of messages. When a party P_i is corrupted it receives the input (`corrupt` i) in response to which it might produce some output which is written on the special output tape. This behaviour can be used when modelling protocols which are allowed to leak a particular information when a given party is corrupted. In can also receive the input (`activate` v) on the special tape in response to which it writes a value on the output tape for each party. The rules of the ideal process guarantees that $\mathcal{F}$ will have received exactly one input for each honest party between consecutive (`activate` v) commands. The value v can be thought of as the inputs to $\mathcal{F}$ from the corrupted parties, but can be interpreted by $\mathcal{F}$ arbitrarily, e.i. according to its specification.

We then say that a protocol π securely realizes an ideal functionality $\mathcal{F}$ if the protocol has the same input-output behaviour as the functionality (this captures correctness) and all the communication of the protocol can be simulated given only the inputs and the outputs of the corrupted parties and the values on the special tape of $\mathcal{F}$ (this captures secrecy of the honest parties' inputs). When $\mathcal{F}$ is executed in some environment $\mathcal{Z}$ the environment knows the inputs and the outputs of all parties, so $\mathcal{Z}$ cannot be responsible of simulating. We therefore introduce a so-called interface or simulator $\mathcal{S}$ which is responsible for the simulation. The interface is put between the environment $\mathcal{Z}$ and the ideal-process. The job of $\mathcal{S}$ is then to simulate a real-life execution by giving the environment correctly looking responses to the commands it issues. In doing this the interface sees the outputs from $\mathcal{F}$ on the special output tape (to model leaked information) and can specify the value v to $\mathcal{F}$ on the special input tape (to specify inputs of the corrupted parties or e.g. non-deterministic behaviour, all depending on how $\mathcal{F}$ is defined to interpret v). We note that $\mathcal{S}$ does not see the messages sent between $\mathcal{F}$ and $\mathcal{Z}$ for honest parties (which is exactly the purpose of introducing $\mathcal{S}$). In detail the ideal process proceeds as follows.

Init: The input to an ideal process is the security parameter k, the random bits $r_{\mathcal{F}}$ and $r_{\mathcal{S}}$ used by $\mathcal{F}$ and $\mathcal{S}$ and an auxiliary input $z \in \{0,1\}^*$ for $\mathcal{Z}$.
Initialize the round counter $r = 0$ and initialize the set of corrupted parties $C = \emptyset$.
Provide $\mathcal{S}$ with $r_{\mathcal{S}}$, provide $\mathcal{F}$ with $r_{\mathcal{F}}$ and give k and z to $\mathcal{Z}$ and activate $\mathcal{Z}$.

Environment activation: $\mathcal{Z}$ is defined exactly as in the real-word, but now commands are handled by $\mathcal{S}$, as described below.

Party activation: The values $\{m_{j,i,r-1}\}_{i \in C}$ are input to $\mathcal{S}$ and the value $x_{i,r}$ is input to $\mathcal{F}$ on the input tape for P_i and $\mathcal{F}$ is run and outputs some value $v_{\mathcal{F}}$ on the special tape. This value is given to $\mathcal{S}$ which is then required to compute some values $\{m_{i,j,r}\}_{j\in[n]\setminus\{i\}}$ and return these to $\mathcal{Z}$.

Corrupt: When $\mathcal{Z}$ corrupts P_i, $\mathcal{S}$ is given the values $x_{i,0}, y_{i,0}, x_{i,1}, \ldots$ exchanged between $\mathcal{Z}$ and $\mathcal{F}$ for P_i. Furthermore ($\texttt{corrupt}\ i$) is input to $\mathcal{F}$ in response to which $\mathcal{F}$ returns some value $v_{\mathcal{F}}$ which is also given to $\mathcal{S}$. Then $\mathcal{S}$ is required to compute some value r_i and return it to $\mathcal{Z}$. Set $C = C \cup \{i\}$.

End round: When a ($\texttt{end round}$) command is issued $\mathcal{S}$ is activated and produces a value v. Then ($\texttt{activate}\ v$) is input to $\mathcal{F}$ which produces outputs $\{y_{i,r}\}_{i\in[n]}$. The values $\{y_{i,r}\}_{i\in C}$ are then handed to $\mathcal{S}$ and the values $\{y_{i,r}\}_{i\in H}$ are handed to $\mathcal{Z}$. Set $r = r + 1$.

The result of the ideal-process is the bit b output by $\mathcal{Z}$. We denote this bit by $\texttt{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(k, r_{\mathcal{F}}, r_{\mathcal{S}}, z)$. This defines a random variable $\texttt{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(k, z)$ and in turn defines a Boolean distribution ensemble $\texttt{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}} = \{\texttt{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(k, z)\}_{k\in\mathbf{N},z\in\{0,1\}^*}$.

Notice that the interaction of $\mathcal{Z}$ with the real-world and the ideal process has the same pattern. The goal of the interface is then to produce the values that it hands to $\mathcal{Z}$ in such a way that $\mathcal{Z}$ cannot distinguish whether it is observing the real-life execution or a simulation of it in the ideal process. Therefore the bit b output by $\mathcal{Z}$ can be thought of as a guess on which of the two it is observing. This gives rise to the following definition.

Definition 2. *We say that π t-securely realizes $\mathcal{F}$ if there exists an interface $\mathcal{S}$ such that for all environments $\mathcal{Z}$ corrupting at most t parties it holds that $\mathit{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}} \stackrel{c}{\approx} \mathit{REAL}_{\pi,\mathcal{Z}}$.*

A.3 The Hybrid Models

We now describe the $\mathcal{G}$-hybrid model for a synchronous ideal functionality $\mathcal{G}$. Basically the $\mathcal{G}$-hybrid model is the real-life model where in addition the parties have access to an ideal functionality $\mathcal{G}$ to aid them in the computation. In each round r party P_i will receive an output $t_{i,r-1}$ from $\mathcal{G}$ from the previous round and will produce and input $s_{i,r}$ for $\mathcal{G}$ for round r. This means that the r'th activation of P_i now is given by $(m_{i,1,r}, \ldots, m_{i,n,r}, y_{i,r}, s_{i,r}) = P_i(k, m_{1,i,r-1}, \ldots, m_{n,i,r-1}, x_{i,r}, t_{i,r-1}; r_i)$. In the hybrid model, still $\mathcal{Z}$ models the adversary. Therefore, the output from $\mathcal{G}$ on its special tape, which models public information, is given to $\mathcal{Z}$, and the inputs to $\mathcal{G}$ on its special input tape, which can be thought of as modelling the inputs from corrupted parties, is provided by $\mathcal{Z}$. In detail the hybrid execution proceeds as follows.

Init: The input to an execution is the security parameter k, the random bits $r_1, \ldots, r_n \in \{0,1\}^*$ used by the parties, the random bits $r_{\mathcal{G}}$ for $\mathcal{G}$ and an auxiliary input $z \in \{0,1\}^*$ for $\mathcal{Z}$.
Initialize the round counter $r = 0$ and initialize the set of corrupted parties $C = \emptyset$.
Let $m_{i,j,0} = \epsilon$ for $i, j \in [n]$ and let $t_{i,-1} = \epsilon$.
Provide $\mathcal{G}$ with $r_{\mathcal{G}}$ and input k and z to $\mathcal{Z}$ and activate $\mathcal{Z}$.

Environment activation: $\mathcal{Z}$ is defined exactly as in the real-word except that the ($\texttt{end round}$) command has the syntax ($\texttt{end round}\ v$) for some value v and that $\mathcal{Z}$ receives some extra values in response to the commands as described below.

Party activation: Values $\{m_{j,i,r-1}\}_{j\in H}$ and $t_{i,r-1}$ were defined in the previous round. Add these to $\{m_{j,i,r-1}\}_{j\in C}$ from the environment and compute

$$(m_{i,1,r},\ldots,m_{i,n,r},y_{i,r},s_{i,r}) = P_i(k,m_{1,i,r-1},\ldots,m_{n,i,r-1},x_{i,r},t_{i,r-1};r_i)\ .$$

Then the value $s_{i,r}$ is input to $\mathcal{G}$ on the input tape for P_i and $\mathcal{G}$ is run and produces some value $v_{\mathcal{G}}$ on the special tape. Then $v_{\mathcal{G}}$ is given to $\mathcal{Z}$ along with $\{m_{i,j,r}\}_{j\in[n]\setminus\{i\}}$.

Corrupt: Give r_i to $\mathcal{Z}$ along with the values $s_{i,0},t_{i,0},s_{i,1}\ldots$ exchanged between P_i and $\mathcal{G}$, see below in **End round**. Furthermore (`corrupt` i) is input to $\mathcal{G}$ in response to which $\mathcal{G}$ returns some value $v_{\mathcal{G}}$ which is also given to $\mathcal{Z}$. Set $C=C\cup\{i\}$.

End round: Give the values $\{y_{i,r}\}_{i\in H}$ defined in **Party activation** to $\mathcal{Z}$. Furthermore, input (`activate` v) to $\mathcal{G}$ and receive the output $\{t_{i,r}\}_{i\in[n]}$. The values $\{t_{i,r}\}_{i\in C}$ are then handed to $\mathcal{Z}$ and the values $\{t_{i,r}\}_{i\in H}$ are used as input for the honest parties in the next round. Set $r=r+1$

The result of the hybrid execution is the bit b output by $\mathcal{Z}$. We denote this bit by $\mathtt{HYB}^{\mathcal{G}}_{\pi,\mathcal{Z}}(k,r_1,\ldots,r_n,r_{\mathcal{G}},z)$. This defines a random variable $\mathtt{HYB}^{\mathcal{G}}_{\pi,\mathcal{Z}}(k,z)$ and in turn defines a Boolean distribution ensemble $\mathtt{HYB}^{\mathcal{G}}_{\pi,\mathcal{Z}}$.

As for an interface $\mathcal{S}$ simulating a real-life execution of a protocol π in the ideal process for ideal functionality $\mathcal{F}$ we can define the notion of a hybrid interface $\mathcal{T}$ simulating a hybrid execution of a hybrid protocol $\pi[\mathcal{G}]$ in the ideal process for ideal functionality $\mathcal{F}$. This is defined equivalently. The only difference is that an ideal interface $\mathcal{T}$ has to return more values to $\mathcal{Z}$ to be successful. For completeness we give the ideal process with a hybrid simulator in detail.

Init: The input to an ideal process is the security parameter k, the random bits $r_{\mathcal{F}}$ and $r_{\mathcal{T}}$ used by $\mathcal{F}$ and $\mathcal{T}$ and an auxiliary input $z\in\{0,1\}^*$ for $\mathcal{Z}$.
Initialize the round counter $r=0$ and initialize the set of corrupted parties $C=\emptyset$.
Provide $\mathcal{T}$ with $r_{\mathcal{T}}$, provide $\mathcal{F}$ with $r_{\mathcal{F}}$ and give k and z to $\mathcal{Z}$ and activate $\mathcal{Z}$.

Environment activation: $\mathcal{Z}$ is defined exactly as in the hybrid world, but now the commands are handled by $\mathcal{T}$, as described below.

Party activation: The values $\{m_{j,i,r-1}\}_{i\in C}$ are input to $\mathcal{T}$ and the value $x_{i,r}$ is input to $\mathcal{F}$ on the input tape for P_i and $\mathcal{F}$ is run and outputs some value $v_{\mathcal{F}}$ on the special tape. This value is given to $\mathcal{T}$ which is then required to compute some values $\{m_{i,j,r}\}_{j\in[n]\setminus\{i\}}$ and a value value $v_{\mathcal{G}}$ and return these to $\mathcal{Z}$.

Corrupt: When $\mathcal{Z}$ corrupts a party $\mathcal{T}$ is given the values $x_{i,0},y_{i,0},x_{i,1},\ldots$ exchanged between $\mathcal{Z}$ and $\mathcal{F}$ for P_i. Furthermore (`corrupt` i) is input to $\mathcal{F}$ in response to which $\mathcal{F}$ returns some value $v_{\mathcal{F}}$ which is also given to $\mathcal{T}$. Then $\mathcal{T}$ is required to compute some value r_i, some value $s_{i,0},t_{i,0},s_{i,1},\ldots$ and some value $v_{\mathcal{G}}$ and return it to $\mathcal{Z}$. Set $C=C\cup\{i\}$.

End round: When a (`end round` v) command is issued $\mathcal{T}$ is activated with input (`end round` v) and produces a value v'. Then (`activate` v') is input to $\mathcal{F}$ which produces outputs $\{y_{i,r}\}_{i\in[n]}$. The values $\{y_{i,r}\}_{i\in C}$ are then handed to $\mathcal{T}$ which produces an output $\{t_{i,r}\}_{i\in C}$ and the values $\{t_{i,r}\}_{i\in C}$ and $\{y_{i,r}\}_{i\in H}$ are handed to $\mathcal{Z}$. Set $r=r+1$.

Notice that the interaction of $\mathcal{Z}$ with the hybrid model and the ideal process has the same pattern. The goal of the interface $\mathcal{T}$ is then to produce the values that it hands to $\mathcal{Z}$ in such a way that $\mathcal{Z}$ cannot distinguish whether it is observing the hybrid execution or a simulation of it in the ideal process.

Definition 3. *We say that π t-securely realizes $\mathcal{F}$ in the $\mathcal{G}$-hybrid model if there exists an hybrid interface $\mathcal{T}$ such that all environments $\mathcal{Z}$ corrupting at most t parties it holds that $\mathtt{IDEAL}_{\mathcal{F},\mathcal{T},\mathcal{Z}} \stackrel{c}{\approx} \mathtt{HYB}^{\mathcal{G}}_{\pi,\mathcal{Z}}$.*

A.4 Composing Protocols

Assume that we are given two protocols $\gamma = (P_1^\gamma, \ldots, P_n^\gamma)$ for the real-life model and $\pi[\cdot] = (P_1^\pi[\cdot], \ldots, P_n^\pi[\cdot])$ for a hybrid model. We describe how to composition such protocol to obtain a real-life protocol $\pi[\gamma] = (P_1^\pi[P_1^\gamma], \ldots, P_n^\pi[P_n^\gamma])$, which is intended to be the two protocols run in lock-step while replacing the ideal functionality access of $\pi[\cdot]$ by calls to γ. The messages send by the parties $P_i = P_i^\pi[P_i^\gamma]$ will consist of a message from each of the two protocols. For this purpose we fix some bijective encoding $(\cdot,\cdot) : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ which can be computed and inverted efficiently.

The activation $(m_{i,1,r}, \ldots, m_{i,n,r}, y_{i,r}) = P_i(k, m_{1,i,r-1}, \ldots, m_{n,i,r-1}, x_{i,r}; r_i)$ is computed as follows. If while running $P_i^\pi[\cdot]$ and P_i^γ these machines request a random bit, give them a fresh random bit from r_i. For notational convenience we let r_i^π and r_i^γ denote the bits used by $P_i^\pi[\cdot]$ respectively P_i^γ. For $j \in [n] \setminus \{i\}$ let $(m^\pi_{i,j,r-1}, m^\gamma_{i,j,r-1}) = m_{i,j,r-1}$ and let $((m^\pi_{i,i,r-1}, m^\gamma_{i,i,r-1}), t_{i,r-1}) = m_{i,i,r-1}$. Then compute $(m_{1,i,r}, \ldots, m_{n,i,r}, y_{i,r}, s_{i,r}) = P_i^\pi(k, m^\pi_{1,i,r-1}, \ldots, m^\pi_{n,i,r-1}, x^\pi_{i,r}, t_{i,r-1}; r_i^\pi)$ and then compute $(m_{1,i,r}, \ldots, m_{n,i,r}, t_{i,r}) = P_i^\gamma(k, m^\gamma_{1,i,r-1}, \ldots, m^\gamma_{n,i,r-1}, s_{i,r}; r_i^\gamma)$. Then for $j \in [n] \setminus \{i\}$ let $m_{i,j,r} = (m^\pi_{i,j,r}, m^\gamma_{i,j,r})$ and let $m_{i,i,r} = ((m^\pi_{i,i,r}, m^\gamma_{i,i,r}), t_{i,r})$.

The following composition theorem follows directly from Lemma 2 in the below section.

Theorem 3. *Assume γ t-securely realizes $\mathcal{G}$ and that $\pi[\cdot]$ t-securely realizes $\mathcal{F}$ in the $\mathcal{G}$-hybrid model. Then $\pi[\gamma]$ t-securely realizes $\mathcal{F}$.*

A.5 Composing Interfaces

We now describe how to compose two interfaces. Assume that we are given a real-life interface $\mathcal{S}$ and a hybrid model interface $\mathcal{T}[\cdot]$. We now describe how to construct a new real-life interface $\mathcal{T}[\mathcal{S}]$. The idea behind the composition operation is as follows. Assume that $\mathcal{T}[\cdot]$ simulates a protocol $\pi[\mathcal{G}]$ while having access to the ideal functionality $\mathcal{F}$, and assume that $\mathcal{S}$ simulates a protocol π while having access to $\mathcal{G}$. We then want $\mathcal{U} = \mathcal{T}[\mathcal{S}]$ to simulate the protocol $\pi[\gamma]$ while having access to $\mathcal{F}$. This is done as follows. First of all $\mathcal{U}$ runs $\mathcal{T}[\cdot]$ using $\mathcal{U}$'s access to $\mathcal{F}$. This provides $\mathcal{U}$ with a simulated version of $\pi[\mathcal{G}]$ consistent with $\mathcal{F}$, which in particular provides it with a simulated access to $\mathcal{G}$. Using the simulated access to $\mathcal{G}$ it then runs $\mathcal{S}$ and gets a simulated version of γ consistent with $\mathcal{G}$ from the simulated $\pi[\mathcal{G}]$ consistent with $\mathcal{F}$.

It then merges the values of the simulated version of $\pi[\mathcal{G}]$ and the simulated γ as defined by the composition operation on protocols and obtains a simulated version of $\pi[\gamma]$ consistent with $\mathcal{F}$. The notation used to describe the composition operation will reflect the above idea. The composed interface works as follows.

Init: $\mathcal{U}$ receives k and random bits r. When $\mathcal{S}$ or $\mathcal{T}[\cdot]$ request a random bit $\mathcal{U}$ gives them a random bit from r.

Party activation: $\mathcal{U}$ receives $\{m_{i,j,r-1}\}_{i\in C}$ from $\mathcal{Z}$ and $v_{\mathcal{F}}$ from $\mathcal{F}$ and must provide outputs $\{m_{i,j,r}\}_{j\in[n]\setminus\{i\}}$. This is done as follows.

1. For $i \in C$ compute $(m^{\pi}_{i,j,r-1}, m^{\gamma}_{i,j,r-1}) = m_{i,j,r-1}$.
2. Input $\{m^{\pi}_{i,j,r-1}\}_{i\in C}$ and $v_{\mathcal{F}}$ to $\mathcal{T}[\cdot]$ which generates values $\{m^{\gamma}_{i,j,r}\}_{j\in[n]\setminus\{i\}}$ and $v_{\mathcal{G}}$.
3. Input $\{m^{\gamma}_{i,j,r-1}\}_{i\in C}$ and $v_{\mathcal{G}}$ to $\mathcal{S}$ which generates values $\{m^{\gamma}_{i,j,r}\}_{j\in[n]\setminus\{i\}}$.
4. Output $\{m_{i,j,r}\}_{j\in[n]\setminus\{i\}}$, where $m_{i,j,r} = (m^{\pi}_{i,j,r}, m^{\gamma}_{i,j,r})$.

Corrupt: $\mathcal{U}$ receives $x_{i,0}, y_{i,1}, x_{i,1}, \ldots$ and $v_{\mathcal{F}}$ and must provide an output r_i. This is done as follows.

1. Input $x_{i,0}, y_{i,1}, x_{i,1}, \ldots$ and $v_{\mathcal{F}}$ to $\mathcal{T}[\cdot]$ which generates values r^{π}_i and $s_{i,0}, t_{i,1}, s_{i,1}, \ldots$ and $v_{\mathcal{G}}$.
2. Input $s_{i,0}, t_{i,1}, s_{i,1}, \ldots$ and $v_{\mathcal{G}}$ to $\mathcal{S}$ which generates a value r^{γ}_i.
3. Outputs $r_i = [r^{\pi}_i, r^{\gamma}_i]$.

End round: $\mathcal{U}$ is given (`end round`) and must produce an output for $\mathcal{F}$ in response to which it receives $\{y_{i,r}\}_{i\in C}$. To run $\mathcal{S}$ and $\mathcal{T}[\cdot]$ as they expect this is done as follows.

1. Activate $\mathcal{S}$ on input (`end round`) and receive as output a value v.
2. Activate $\mathcal{T}[\cdot]$ on input (`end round` v) and receive as output a value v'.
3. Outputs v' and receive $\{y_{i,r}\}_{i\in C}$.
4. Hand $\{y_{i,r}\}_{i\in C}$ to $\mathcal{T}[\cdot]$ and get the output $\{t_{i,r}\}_{i\in C}$.
5. Then input $\{t_{i,r}\}_{i\in C}$ to $\mathcal{S}$.

Using the proof techniques from [**?**] it is straight forward to construct a proof for the following lemma. The proof contains no new ideas and have been excluded for that reason and to save space.

Lemma 2. *Assume that all environments $\mathcal{Z}$ corrupting at most t parties it holds that $\texttt{IDEAL}_{\mathcal{G},\mathcal{S},\mathcal{Z}} \stackrel{c}{\approx} \texttt{REAL}_{\gamma,\mathcal{Z}}$, and assume that for all hybrid environments $\mathcal{Z}$ corrupting at most t parties it holds that $\texttt{IDEAL}_{\mathcal{F},\mathcal{T},\mathcal{Z}} \stackrel{c}{\approx} \texttt{HYB}^{\mathcal{G}}_{\pi,\mathcal{Z}}$, then for all environments $\mathcal{Z}$ corrupting at most t parties it holds that $\texttt{IDEAL}_{\mathcal{F},\mathcal{T}[\mathcal{S}],\mathcal{Z}} \stackrel{c}{\approx} \texttt{REAL}_{\pi[\gamma],\mathcal{Z}}$.*